



1986 WINTER
USENIX
TECHNICAL
CONFERENCE

Denver, Colorado

CONFERENCE PROCEEDINGS

XEROX PARC
TECHNICAL INFORMATION CENTER

QA
76.8
U65
U84
1986w

USENIX Association

Winter Conference Proceedings

**January 15-17, 1986
Denver, Colorado USA**

For additional copies of these proceedings, write:

USENIX Association
P.O. Box 7
El Cerrito, CA 94530 USA

Price: \$20.00

© Copyright 1985 by The USENIX Association
All rights reserved.
This volume is published as a collective work.
Rights to individual papers remain
With the author or the author's employer.

UNIX is a trademark of AT&T Bell Laboratories.
Other trademarks are noted in the text.

ACKNOWLEDGEMENTS

Program Chairs and Committee Members

Window Environments & UNIX

Sam Leffler, Chair

Mike Hawley, Co-Chair

Jim Gettys

James Gosling

Rob Pike

Lucasfilm

The Droid Works

Digital Equipment Corp.

Sun Microsystems

Bell Laboratories

UNIX on Big Iron

Peter Capek, Chair

Jim Lipkis

Eugene Miya

IBM Research

New York University, Courant Institute

NASA Ames Research Center

Ada and the UNIX System

Charles Wetherell, Chair

Donn Milton

Tucker Taft

Larry Yelowitz

AT&T Information Systems

Verdix Corporation

Intermetrics

Ford Aerospace

SPONSORED BY:

USENIX Association, P. O. Box 7, El Cerrito, CA 94350

TUTORIAL COORDINATOR

Michael Tilson, Human Computing Resources

USENIX CONFERENCE COORDINATOR

John Donnelly

USENIX MEETING PLANNER

Judith F. Desharnais

CONFERENCE HOST

UNIVERSITY OF COLORADO

Evi Nemeth, Computer Science Department

1986 Winter USENIX Technical Conference

Window Environments and UNIX

Chair: Sam Leffler and Mike Hawley

Denver, Colorado

Wednesday, January 15, 1986

Wednesday (8:30-10:10) Hardware and Hardware Issues

Opening Remarks

Conference Organizers and USENIX Board

Galadriel: A Display List-Based Window Manager 1
Bob Lewis, Tektronix

Next-Generation Hardware for Windowed Displays 11
S. McGeady,

Real-Time Resource Sharing for Graphics Workstations 23
Mark S. Grossman and Glen E. Williams, Silicon Graphics, Inc.

Wednesday (10:30-12:00) Applications

GLO - A Tool for Developing Window-Based Programs 34
Thomas Neuendorffer, Carnegie-Mellon University

A Workstation-Based Inpatient Clinical System in the 45
Johns Hopkins Hospital
*S. N. Kahane, S. G. Tolchin, M. J. Schneider,
D. W. Richmond, P. Barta, M. K. Ardolino,
H. S. Goldberg, Johns Hopkins University Hospital*

The Feel of Pi 62
T. A. Cargill, AT&T Bell Laboratories

Wednesday (1:30-3:30) Systems and System Issues

Flamingo: Object-Oriented Abstractions for User Interface Management 72
Edward T. Smith, David B. Anderson, Carnegie-Mellon University

A Proposal for Interwindow Communication and Translation Facilities 79
Daniel P. Gill, Exxon Research and Engineering

Problems Implementing Window Systems in UNIX 89
James Gettys, Massachusetts Institute of Technology

SUNDEW: A Distributed and Extensible Window System 98
James Gosling, Sun Microsystems

Wednesday (4:00-5:00) Panel Discussion

"Color? Do We Need It? How Can We Use It? How Do We Deal With It?..."

1986 Winter USENIX Technical Conference

UNIX on Big Iron

Chair: Peter Capek

Denver, Colorado

Thursday, January 16, 1986

Thursday (8:30-10:00) Applications and Requirements

Opening Remarks

Peter Capek, IBM Research

User Requirements for Future-nix

104

Eugene Miya, NASA Ames Research Center

Experience with Large Applications on Unix

110

Bob Bilyeu, McNeill-Schwindler, Inc.

UNIX Scheduling for Large Systems

111

*Jeffery H. Straathof, Ashok K. Thareja,
and Ashok K. Agrawalal, University of Maryland*

Thursday (10:30-12:00) Real Systems I

A Straightforward Implementation of a 4.2BSD on a High
Performance Multiprocessor

140

Dave Probert, Culler Scientific Systems Corporation

Porting UNIX to the System/370 Extended Architecture

157

Joseph R. Eykholt, Amdahl Corporation

Full Duplex Support for Mainframes

165

Don Sterk, Amdahl Corporation

Thursday (1:30-3:10) Real Systems II

- Concentrix -- A Unix for the Alliant Multiprocessor 172
Jack Test, Alliant Computer Corporation
- A User-Tunable Multiprocessor Scheduler 183
Herb Jacobs, Alliant Computer Corporation
- High Performance Enhancements of C-1 Unix 192
Rob Kolstad, Convex Computer Corporation

Thursday (3:30-5:30) Real Systems III

- Considerations for Massively Parallel Unix Systems on the NYU
 Ultracomputer and the IBM RP3 193
*Jan Edler, Allan Gottlieb, Jim Lipkis,
 New York University - Courant Institute*
- Unix of CTSS for the Cray-1, Cray X-MP and Cray-2 Supercomputers 211
*Karl Auerbach, ZERO-ONE Systems and NASA Ames
 Research Center; Robin O'Neill, National Magnetic
 Fusion Energy Computer Center*
- Experience Porting System V to the Cray 2 219
Tim Hoel, Cray Research

1986 Winter USENIX Technical Conference

ADA and the UNIX System

Chair: Charles Wetherell

Denver, Colorado

Friday, January 17, 1986

Friday (9:00-10:30)

Introductory Remarks

Ada and UNIX

Robert Firth, Tartan Labs

UNIX, C and Ada

Herman Fischer, Mark V Business Systems

225

Friday (11:00-12:00)

Revision Control Tools and the Ada Program Library

Dick Schefstrom, TeleLOGIC AB

241

Managing Separate Compilation in the AT&T Ada Translator System

G. W. Elsesser, M. S. Safran and T. Tieger, AT&T

252

Friday (1:30-3:00)

Targeting Ada to 68000/UNIX

Mitchell Gart, Alsys Inc.

261

A Comparison of UNIX and CAIS System Facilities

*Helen Gill, Rebecca Bowerman, and Chuck Howell,
MITRE Corporation*

275

SVID as an Interim Basis for CAIS

Herman Fischer, Mark V Business Systems

294

Friday (3:30-4:30)

An Overview of the Ada Shell	302
<i>Lisa Campbell and Mark Campbell, NCR Corporation</i>	
Implementing Curses in ADA	314
<i>Karl Nyberg, Verdix Corporation</i>	

1986 Winter USENIX Technical Conference

**Author Index
Denver, Colorado**

January 15-17, 1986

Author	Page	Author	Page
P.Barta	45	Chuck Howell	275
Ashok Agrawalal	111	Herb Jacobs	183
David Anderson	72	S.N. Kahane	45
M.K. Ardolino	45	Rob Kolstad	192
Karl Auerbach	211	Bob Lewis	1
Bob Bilyeu	110	Jim Lipkis	193
Rebecca Bowerman	275	S. McGeady	11
Lisa Campbell	302	Eugene Miya	104
Mark Campbell	302	Thomas Neuendorffer	34
T.A. Cargill	62	Karl Nyberg	314
Jan Edler	193	Robin O'Neill	211
G.W. Elsesser	252	Dave Probert	140
Joseph Eykholt	157	D.W. Richmond	45
Herman Fischer	225	M.S. Safran	252
Herman Fischer	294	Dick Schefstrom	241
Mitchell Gart	261	M.J. Schneider	45
James Gettys	89	Edward Smith	72
Helen Gill	275	Jeffery Staathof	111
Daniel Gill	79	Don Sterk	165
H.S. Goldberg	45	Jack Test	172
James Gosling	98	Ashok Thareja	111
Allan Gottlieb	193	T. Tieger	252
Mark Grossman	23	S.G. Tolchin	45
Tim Hoel	219	Glen Williams	23

1986 Winter USENIX Technical Conference

Tutorials

Denver, Colorado

January 15-17, 1986

Wednesday, January 15, 1986

1. Design Considerations for SNA Communications Under UNIX
Daniel Fisher, System Strategies, Inc.
2. UNIX Device Driver Design (4.2BSD)
Daniel Klein, Consultant
3. System V Interprocess Communication Application Programming
Dr. Jon H. LaBadie, AUXCO
4. ADA - From the Top: An Introduction
Putnam P. Texel, Texel & Company
5. UNIX System V Internals
Maury Bach and Steve Buroff, AT&T Information Systems

Thursday, January 16, 1986

6. Introduction to 4.2BSD Internals
Dr. Thomas W. Doepfner, Jr., Brown University
7. Windowing Systems Implementations
David Rosenthal, Sun Microsystems, Inc.
8. Language Construction Tools on the UNIX System
Stephen C. Johnson, AT&T Bell Laboratories
9. Advanced C Programming
William C. Steward, AUXCO

Friday, January 17, 1986

10. Advanced Topics on 4.3BSD Internals

Mike Karels and Marshal Kirk McKusick, University of California, Berkeley

11. UNIX Networking

Bruce Borden, Silicon Graphics, Inc.

12. Managing a Local Area Network

Evi Nemeth and Andy Rudoff, University of Colorado, Boulder

13. Introduction to UNIX System Administration

Ed Gould, mt Xinu

14. Writing Portable C Programs

Dr. Tom Plum, Plum Hall, Inc.

Window Environments and UNIX

January 15, 1986

Denver, Colorado

Galadriel: A Display List-Based Window Manager

Bob Lewis

Graphics Workstation Division
Information Display Group
P.O. Box 1000, MS 61-277
Tektronix, Inc.
Wilsonville, OR 97070
tektronix!tekecs!bobl

ABSTRACT

This paper presents an architectural description of Galadriel, a window management system that provides both text and graphics services to client processes. Unlike most other window managers, Galadriel runs under UNIX[†] on a hosted, display list terminal instead of a bitmapped workstation. It discusses the advantages and disadvantages of this approach as well as areas for further development.

1. Introduction

Window managers are gaining increasing acceptance as a part of many engineering environments. Usually, however, they use a display connected directly to a single-user workstation over a high-bandwidth communications line such as a memory bus.

This paper discusses a window manager called Galadriel¹ which shares many characteristics with previous window managers, but runs on a substantially different hardware configuration.

Galadriel runs under the UNIX operating system residing on a host computer and interfaces to several different terminal models over an RS-232 communications line. All of the models supported are display list-based; in addition to conventional ASCII text (alphatext), output to them may include encoded graphics primitives such as polylines, polymarkers, filled polygons, and graphic text (graphtext) which can be sent directly to the display processor or put in terminal RAM as a segment. The host can then change various attributes of the segment, redisplay it, and replicate it without retransmitting the primitives.

Section 2 describes the target hardware and software constraints. Subsequent sections deal with the effects of these constraints on traditional window management capabilities. Section 3 covers the ways Galadriel organizes windows and related objects from the application's and the user's points of view. Section 4 briefly discusses the interfaces to these objects that Galadriel presents. Section 5 goes into more detail on the internal operation of the window manager process itself. Finally, Section 6 discusses possible directions for future development.

[†] UNIX is a trademark of AT&T Bell Laboratories.

1. From J. R. R. Tolkien's *Lord of the Rings*, owner of magic mirror is about as close as we could come to a mythological window manager.

2. Hardware and Software Constraints

Galadriel was originally implemented on a Tektronix 4115B terminal connected over an RS-232 line to a VAX† (780) running an internal Tek version of (relatively standard) 4.2bsd UNIX. So far, it's been ported to a different 4.2bsd variant (Tektronix's UTek^{2‡}), two other host compute engines (Tek 6030 and 6130), and four other terminals (Tek 4111, 4125, 4128, and 4129).

2.1. Hardware Characteristics

Table 1 gives the relevant differences between the various terminals that affected system design.

model(s)	4111	4115B, 4125, 4128, 4129 ³
display size (pixels)	1024 x 768	1280 x 1024
character size (pixels)	8 x 16	8 x 16 or 16 x 30
#rows x #columns	48 x 132 ⁴	64 x 160 or 34 x 80
number of bit planes	4	2 ⁵ , 4, 6, or 8
display list RAM	1.2MB	768KB

Table 1 -- Display Characteristics

All terminals⁶ accept the same set of over 200 commands. Other design-relevant features common to all of the terminals are:

- 64 viewports (i.e., clipping rectangles)
- 64 dialog areas (independent alphanet output areas, each emulating most of a VT100†)
- signed 32-bit integer display list coordinate space
- raster operations between screen, display list RAM, and host
- a 24-bit color map, indexed by the value of each pixel
- baud rates of up to 19.2 kbaud
- the ability to make any segment the cursor
- mouse or puck/tablet

2.2. Software Considerations

By far the most serious constraints on the design and implementation of Galadriel arose from two factors that set it apart from most other window managers now available. The first is that communication between the host and the terminal must take place over a serial RS-232 line. The second is that Galadriel's *modus operandi* must be acceptable on a host shared with several other users.

The RS-232 communications bottleneck means that the design must use whatever resources it can on both host and terminal ends to minimize the number of bytes that need to be transferred to get things done. Having a display list with definable segments is a big help with this, because once graphic primitives are put into a segment, that segment can be transformed and redisplayed as a unit — the primitives never have to be sent again.

The shared host requirement limits the desirability of tight interaction loops. For such a loop, the nature of the interface would require the window manager to poll the terminal continually for the cursor position. At 9600 baud, the transit time for the 20 or so characters involved in the poll is <

† VAX and VT100 are trademarks of Digital Equipment Corporation.

2. The only change needed in the source code was caused by UTek's allowing 64 open file descriptors (cf. 4.3bsd) instead of the usual 20.

‡ UTek is a trademark of Tektronix, Inc.

3. The 4125 is an upgrade of the 4115B. The 4128 and 4129 include three-dimensional capabilities that Galadriel does not currently use.

4. Galadriel only uses the 128 columns that have pixels under them.

5. The 4115B has a minimum of 4 bit planes.

6. We'll use the somewhat inexact term 412x hereafter to refer generically to all of the terminals in Table 1.

0.02 second, so the communications bottleneck is not the culprit here. The problem is that the window manager would be almost constantly writing to and reading from the terminal. Because input from both of them goes in through the tty driver, the operating system has no way to distinguish responses to polling from (asynchronous) user input. The window manager would look like a highly interactive process, even though there might be no input from the user taking place. The effect is that a process in a polling loop hogs as much system time as several conventional interactive processes. Galadriel should therefore avoid polling as much as possible.

The window manager process *windman* was designed to run as a normal user process and require no modifications to the 4.2bsd kernel. It has met these goals, although some speedups in the pseudo-tty driver have improved performance.

Existing programs should work without recompilation. This included not only glass tty programs like *ls(1)* and *ed(1)*, but screen-oriented ones like *vi(1)*, *emacs(1)*, and programs using *curses(3t)*⁷.

2.3. System Performance Goals

Galadriel was designed to support CAD systems. Its original clients included VLSI layout and schematic capture editors. As usual for such systems, performance is a critical requirement.

Galadriel maximizes performance by several strategies:

- Minimize the number of bytes sent to the terminal.
- Reduce the number of times scalars are encoded for transmission (most of the time, they are now encoded only once).
- Reduce the number of byte copies.
- Use 412x features (e.g., macros) to speed things up.
- Use efficient programming techniques (loop unrolling, etc.) in critical areas.

Benchmarks of the layout editor (Tek's *Leia*) on a Tek 6130, comparing a version running under Galadriel with one that outputs directly to the terminal using a compatibility library, show less than 5% overhead (clock time) of the window managed over the non-window managed version.

3. Window Organization

The window manager process exists as a separate process servicing requests from you, sitting at the terminal, and one or more client processes (see Figure 1).

3.1. Windows and Window Ttys

Each of these clients owns one window tty (*wintty*) and one or more possibly overlapping, rectangular windows. Each window has a title bar, which displays the name of its *wintty*, which is of the form */dev/tty??*.

Internally, each *wintty* is implemented as a pseudo-tty (*pty(4)*), so its name is that of the slave end of the pseudo-tty whose master end Galadriel is watching.

3.2. Panes

Clients may subdivide windows into panes, rectangular areas contained within windows which display output. Most of a window's area is devoted to panes. Each window starts out with a single pane, called its primary pane.

The terminal hardware has no notion of windows or panes. The software treats windows as groups of panes that get created, deleted, reframed, etc. together, associated with a title bar. It then maps each pane to a list of virtual viewports.

This list may have no members, one member, or several members depending on whether the pane is completely obscured or collapsed, completely unobscured, or partially obscured, respectively.

⁷ Including, of course, *rogue(6)*.

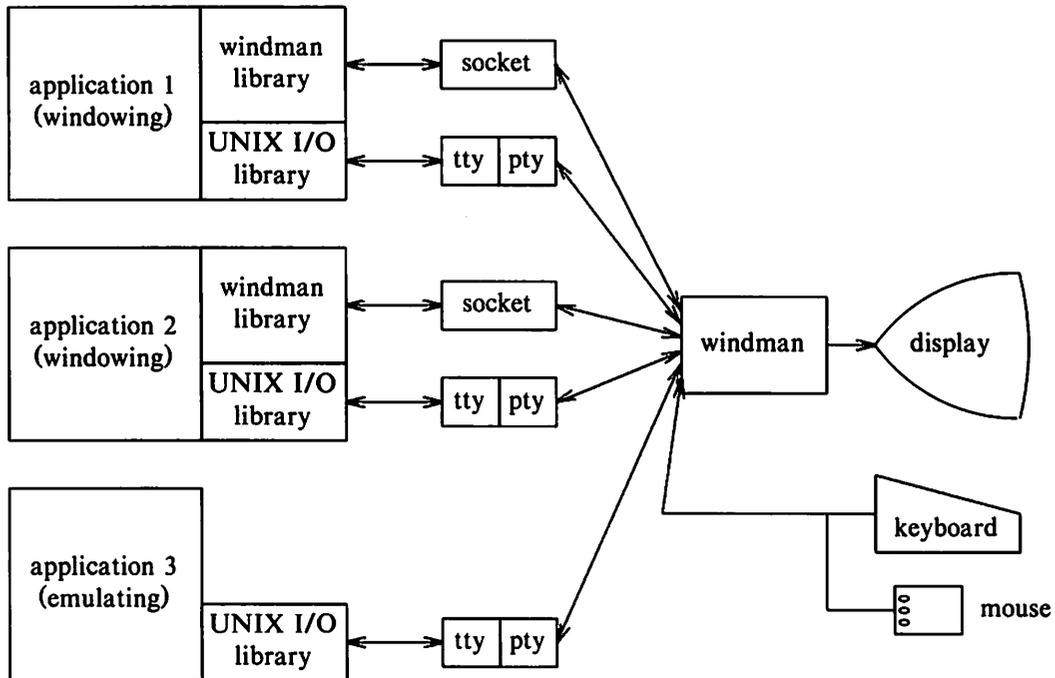


Figure 1 -- The Window Manager Process

Actions affecting this pane, such as making a new segment visible in it, require a traversal of the list.

Galadriel maps the 64 most recently used virtual viewports to the 64 viewports that the 412x supports in hardware, so the window manager must maintain a relation between panes and segments to be able to actualize any virtual viewport at any time. For bookkeeping and memory reasons, however, each application is arbitrarily limited to 128 panes.

The management of viewports is the counterpart to the management of on-display and off-display bitmaps that the Blit (see [Pike84]) performs.

3.3. The Client Models

The window manager looks at clients in one of two ways; emulating or windowing, depending on the functionality the client requires.

3.3.1. Emulating Clients

Galadriel supports terminal emulation for conventional UNIX programs such as *mail(1)*, *vi(1)*, *ls(1)*, etc. Emulating clients do not require recompilation to run under the window manager. Galadriel treats new clients as emulating by default.

By using pseudo-ttys along with the 412x's dialog areas, each emulating client gets its own VT100-like screen. *Pty(4)* lets clients believe that they're talking to a *tty(4)*-like device. Hardware support allows fast scrolling of even partially obscured windows and an emulating client never has to regenerate a window as a result of a window manager command (e.g. **Move**).

3.3.2. Windowing Clients

As Figure 1 shows, windowing clients are compiled with an additional library of window management functions. Section 4.2 will discuss these.

Only windowing clients can own more than one window or more than one pane. They can specify that one or more of those panes emulate a VT100 as above. They then select one of them to be the

text pane.

Most windowing clients, however, are more concerned with graphics output. Galadriel does not use pseudo-ttys for this, but sockets. This allows greater speed⁸ and reduces the need for Galadriel to demultiplex emulated text from window manager commands.

The graphics that windowing clients can perform is modeled after the Graphical Kernel System (GKS). Unlike text output, this allows for simultaneous graphics output to multiple panes.

3.4. The Active Process

In the case of mouse⁹ input, the cursor presence in a particular window implies that the recipient of the input should be that window's owning process, but keyboard and function key input has no such implied destination. Galadriel therefore designates one process as active and takes it to be the current focus of the user's attention. The active process gets keyboard and function key input.

There are also certain terminal-wide resources that can have only one controlling process. These include:

- rubberbanding
- the current color map¹⁰
- the cursor segment

It's convenient to let the active process also take charge of these, so that whenever a process is activated, Galadriel takes care of restoring these resources.

4. Window Manager Interface Implementation

Galadriel provides most of the features commonly found in window managers. This section will concentrate on how it does so within its design constraints. The window manager has three interfaces: user, shell, and programmer. The shell interface, however, doesn't use anything that the user and programmer interfaces don't use as well, so it won't be discussed here.

4.1. User Interface

4.1.1. Mouse Input and the Cursor

The 412x provides the ability to define a particular segment to be the cursor, tracking mouse movement without host intervention (i.e., polling). Whenever the user presses a mouse button, the terminal sends a GIN (graphic input) report to the host. It is up to Galadriel to decide what to do with the report, depending on what the cursor was pointing at as well as its own internal state.

4.1.2. Pop-Up Menus

One of the buttons on the mouse is designated the window manager button. Pressing it brings up the system pop-up menu. With it, you can perform all of the basic window manager commands: **Activate**, **Bury**, **Collapse**, **Create**, **Delete**, **Expand**, **Move**, **Reframe**, and **Uncover**. **Reframe** is the only command that requires polling, since the 412x doesn't have a rubber box cursor to show the new frame of the window.

All of these commands are postfix or object-verb. This means that you first point at the window you want to act on, request the system pop-up menu, choose a command, and the window manager acts¹¹.

8. On a 6130, for large (4K byte) block sizes, pseudo-tty's are typically a factor of 10 slower than sockets, which makes the latter preferable for large segment definitions.

9. Galadriel works with either a mouse or a puck/tablet. We'll use mouse hereafter to refer to both.

10. Galadriel gives each process control of all 2^{# of planes} colors, although restraint is recommended.

11. One exception to this: because it's so final, the **Delete** command requires you to confirm the window you want to delete.

Of course, application programs can define pop-up menus as well.

Pop-up menus must appear and disappear quickly. They could be implemented as titleless windows, i.e., viewports, but this causes problems when you pop-up a menu over a window containing many segments. Even though the window would be refreshed at the rate of tens of thousands of vectors per second, a few seconds is an unacceptable time for a menu to go away.

An alternative is to use the 412x pixel save and restore commands. These copy an area of the screen to and from display list RAM. Galadriel saves what's underneath a menu until the selection is made and then restores it before further output takes place. There are two drawbacks here:

- There can be no output to the terminal while the menu is up, since, unlike the Blit [Pike84], no output can go to the saved rectangle.
- The size of the menu is restricted, since there are 1.2MB of frame buffer and available display list RAM is usually much smaller than that.

The first drawback turns out to not be particularly annoying, and Galadriel satisfies the second by limiting the use of this technique to pop-up menus no larger than 32 characters wide by 16 characters high. It treats any pop-up larger than that with the previous window creation approach.

4.1.3. Scroll Bars

Clients can specify that any or all of their panes have scroll bars associated with them. Scroll bars on a pane allow a consistent and convenient way to browse its design space, assuming that that design space is larger than you can see all at once. Depending on the application, you can pan and zoom both horizontally and vertically¹². There is also an overview button which toggles a view of the entire design space with the previous view.

Galadriel performs pans and zooms by sending a short sequence of commands to the terminal to change the pane's viewport transformation(s) and redraw the viewport(s). It does not need to resend primitives contained in segments. Unless the application has requested notification (see Section 4.2.2), it won't know that the pan or zoom has taken place.

Unlike those of other window managers (cf. Smalltalk-80† [Goldberg84]) Galadriel scroll bars are static. This means that a pane is created (optionally) with scroll bars and the bars stay with the pane until it is deleted. This has the disadvantage that the scroll bars, if present, always take up screen area, but this is outweighed by the advantage gained by not requiring the host to poll for cursor presence.

4.2. The Programmer Interface

This interface allows applications to control windows and output to them, as well as receive input from you. It has three layers: UNIX I/O, window (WL), and graphics (GL). Within this interface, a program can only control the windows and panes that it owns, but it has a greater measure of control over those.

4.2.1. UNIX I/O Layer

This is the only layer available to emulating clients. As we've said before, output is directed to panes. A pane is created with an attribute that says whether or not it can contain alphanetext, i.e., whether or not it needs to be able to emulate a VT100. The reason for this is to conserve the 412x's 64 dialog areas. Each pane that can contain alphanetext takes up one dialog area, and, unlike viewports, dialog areas cannot be virtualized.

To an application, such a text pane appears as a virtual terminal. The most commonly used ANSI X3.64 commands work in it, and under UNIX it has a TERMCAP entry describing these commands. Additionally, for emulating clients the TERMCAP includes the proper window dimensions¹³.

12. Zooming maintains aspect ratio, so a vertical zoom also causes a horizontal zoom, and vice versa.

Smalltalk-80 is a trademark of Xerox Corporation.

13. This also applies to windowing clients that don't split their primary panes.

The **Reframe** command causes problems, though. Although the window manager sets the **TERMCAP** environment variable to a temporary file containing an up-to-date *termcap(5)*-like description, there is no standard way to inform screen editors and the like that this has happened while they're running¹⁴.

Input is slightly more complicated. Following GKS, GL considers keyboard input to be part of graphic input (GIN). There are two keyboard input models corresponding to emulating and windowing clients. Emulating clients have a control tty (*/dev/tty*) that behaves like a normal tty. They can call *ioctl(2)* to control echoing, interrupt characters, line editing, and other *tty(4)* features.

Windowing clients get all of their input from the GL GIN routines, and any attempt to read from their control ttys will block because GIN doesn't travel that way¹⁵.

4.2.2. The Window Layer

There are 24 functions in this layer. WL duplicates much of the functionality of the user and shell interfaces for the programmer. In addition, the programmer has greater control over panes, including the ability to create and delete individual panes. The application can create non-primary (see above) panes by splitting existing panes.

A significant departure here from most window managers is the association of two boolean flags with each of several window manager and user actions. One flag is for permission and the other is for notification. The permission flag allows the associated action to be carried out on the entity to which it is bound. If the notification flag is set, after the action is performed (or attempted, if the permission flag is not set), the owning process learns about it in the form of a detailed GIN report.

Permission/notification flags are bound to both windows and panes. For windows, they control the actions the user attempts from the system pop-up menu. For panes, they determine the presence and function of scroll bars.

4.2.3. The Graphics Layer

GL is a set of 102 graphics routines. Although not entirely GKS-compatible (usually for performance reasons), about 90% of these routines map into a Level 2b GKS implementation with enhancements.

5. The *windman* Process Architecture

Figure 3 shows the flow of data within *windman*, the window manager process.

Basically, *windman* is a command-driven finite state machine. Every command, including ANSI text output and escape codes, WL and GL requests, keystrokes, and mouse button presses, are awaited by a single *select(2)* call in the queue manager. Once a command is received, the queue manager decides which of the four parsers; ANSI, GL, WL, or GIN; to pass the command to. Apart from saving and restoring some context information, the queue manager is the only part of the *windman* process that knows (or cares) that there may be multiple clients.

The ANSI parser is a finite state machine itself. All pseudo-tty output from applications goes through this parser. It sees that only complete ANSI commands get sent to the terminal and prevents certain other sequences (such as **RESET TERMINAL** and non-ANSI commands) from getting there.

The GL parser is responsible for maintaining client context, so that each client can have the attributes it sets present while it's writing primitives and defining segments. This parser also queues segment definitions until the client closes the segment. Both of these features are necessary because the terminal permits only one set of attributes and one open segment, at most, at a time. It also manages terminal memory and sends responses to clients when needed.

14. The **SIGWINCH** and *tty(4)* enhancements made to 4.3bsd will be helpful here.

15. A way to permit clients to do this is under consideration.

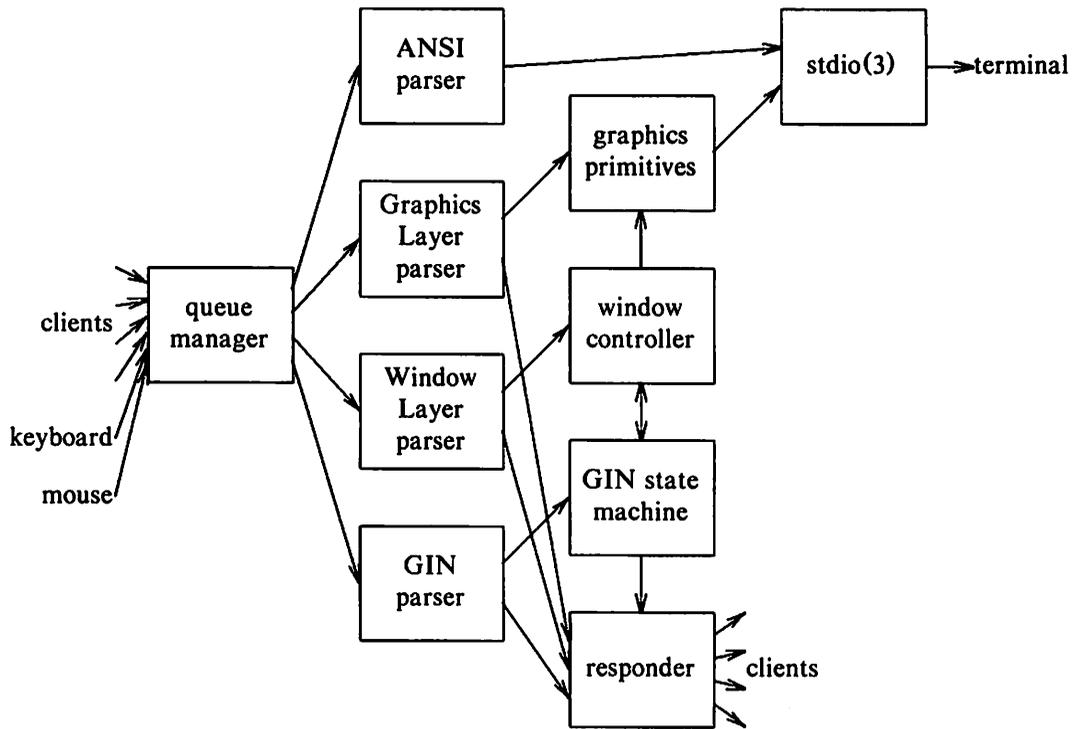


Figure 3 -- Dataflow Within *windman*

The WL parser deals with the higher-level commands that make up WL and the shell interface as well. Most of the time, about all it does is decode the arguments and pass them on to the window controller, which does the real work.

Both WL and GL parsers converse with the window manager library in each application over sockets using a syntax that resembles a remote procedure call, but is buffered and otherwise optimized for Galadriel.

The main purpose of the GIN parser is to see who gets keyboard and mouse input. Under most circumstances, Galadriel permits GINahead – you can perform mouse or keyboard input before it is requested¹⁶.

The GIN state machine handles what tight interaction loops Galadriel provides: rubberbanding, rubberboxing, and window moving. It also exerts some control over the queue manager. It can restrict the *select(2)* call to watch only the terminal when, for example, the user is making a selection from a pop-up menu.

The responder is called whenever it's necessary to send a GIN report or a command return to a client. The latter corresponds to the return mechanism from a remote procedure call, except that no ANSI and very few GL commands require a return.

All graphics output takes place through *stdio(3)* via a package of primitives which is the only part of Galadriel that knows 412x escape codes.

16. Implementing this is not as easy as it sounds.

6. Future Developments

6.1. Faster Communications

As discussed in Section 2, the major bottleneck is the RS-232 communications line. Obvious possibilities are such things as Ethernet† or DMA communications between the host and the terminal. In fact, there already is a Unibus‡ interface card for the 412x, and the Galadriel project is looking at the capabilities it offers very closely.

As the communication speed increases, the distinction between host/terminal and workstation blurs. The terminal becomes more like a workstation with an independent display engine and a shared compute engine (the host).

6.2. Increased Hardware Support

The 412x was not originally designed for window management. Galadriel has indicated some likely functions to migrate from software to firmware and hardware. These include:

- the pane → virtual viewport → hardware viewport mapping
- output simultaneously to several panes
- better support for tight interaction loops
- context settings
- retain non-segment output during moves and occultations

6.3. Improved Pseudo-Tty Support

Under 4.2bsd, the number of pseudo-ttys is fixed in the kernel and must be created by /dev/MAKEDEV. This is rather artificial. It would be preferable to invoke *open(2)* on /dev/pty or whatever and have that create a new pseudo-tty automatically. An *ioctl(2)* call would return the actual names of the pair. The device should go away whenever the opener of the master end closed it, treating it as a hangup (i.e., SIGHUP) on the slave end.

Alternatively, Ritchie's streams (see [Ritchie84]) would also provide a better way to implement pseudo-ttys.

6.4. Bitmaps

Although the 412x has several commands to copy between host, screen, and display list RAM, they are neither orthogonal nor functionally complete. These should be cleaned up, since as communications become improved, it will be feasible to treat the 412x as both a bitmapped and display list device, and let the clients choose accordingly.

6.5. Distributed Windows

Any emulating client can run *rlogin(1)*, so this feature is already partially present. For windowing clients, intermachine sockets and a naming server (to get the name of the local *windman*'s socket to the remote client and vice versa) are needed.

6.6. More Segment Memory

Large applications can use up the 3/4 megabyte or so of display list RAM in the 412x with relative ease. Two ways to alleviate this are (1) more display list RAM (obviously) and (2) virtual storage in the terminal. If there were greater bandwidth between the terminal and the host, it might be feasible for the terminal to use the host's virtual memory.

† Ethernet is a trademark of Xerox Corporation.

‡ Unibus is a trademark of Digital Equipment Corporation.

Acknowledgements

Paula Mossaides is Galadriel's project manager. In addition to the author, designers, implementers, and maintainers are/were: Scott Hennes, Donna Nakano, Karen Palmer, Rob Reed, and Bob Toole. Evaluators were: Keith Koplitz, Larry Jones, and Max Miller.

References

- [Goldberg84] Goldberg, A., *Smalltalk-80 The Interactive Programming Environment*, Reading, MA: Addison-Wesley, 1984.
- [Pike84] Pike, R., The Blit: A Multiplexed Graphics Terminal, AT&T Bell Lab. Tech. J., 63, No. 8 (October, 1984).
- [Ritchie84] Ritchie, D. M., A Stream Input-Output System, AT&T Bell Lab. Tech. J., 63, No. 8 (October, 1984).

Next-Generation Hardware for Windowed Displays

S. McGeady

Intel Corporation
Hillsboro, Oregon

ABSTRACT

Hardware support for windowing on bitmapped displays has historically been minimal. Indeed, there is a widely-held belief that window management is an aspect of graphical image generation, to be treated with the same tools. This belief has been encouraged by the wide use of the `bitblt` operator in bitmapped systems for both image generation and windowing. A reliance on the use of `bitblt` for windowing has hindered the development of windowed color displays, and the performance problems involved in moving large amounts of data have made real-time scrolling and panning an impossibility in `bitblt`-based systems.

The creation of a distinction between *imaging*, the drawing of images in bitmaps, and *windowing*, the tiling of those images onto displays, makes it easier to think clearly about how to apply hardware to both aspects of display systems.

This paper presents descriptions of three existing systems to demonstrate how hardware can be applied to enhance windowing performance: one is a combined hardware/software approach using traditional hardware; another is a hardware system for windowing on a character-mapped terminal; and the third is a fully hardware-windowed bitmap display.

1. Introduction

Much effort has been expended in recent years in the application of new hardware to enhance the performance and flexibility of graphics display devices, especially *bitmapped* displays [1,2,3,4,5]. Numerous hardware innovations have recently become available that greatly increase the capability, performance, and flexibility of these displays. During the same period the idea of dividing a display's viewable area into independent, overlapping rectangular regions, or *windows*, has become popular. Despite the growing popularity of windowed displays, very little effort has been directed toward hardware for enhancing windowing capabilities and performance. The effort to move window environments from existing monochrome, medium-resolution displays to high-resolution multi-planed color displays will make necessary the development of new types of hardware designed specifically and solely to support windowing.

The rapidly falling cost of bitmap display hardware is bringing windowed-display development efforts onto an equal footing with graphics image-generation work, yet windowing is still approached with traditional graphics tools that are poorly suited to window management. This paper shows that the problem of window management becomes much simpler when it is separated from graphics image-generation. This separation allows the application of very powerful, but not overly expensive, hardware to improve windowing performance and simplify window management software while losing none of the flexibility of current schemes.

2. Imaging versus Windowing

To fully understand the essence of windowing, one must carefully and clearly distinguish it from *imaging*. *Imaging*, used in this context, is the broadest possible term for drawing pictures, and comprises every manner of line-drawing, area-filling, surface rendering, and other rasterizations. Imaging systems tend to deal internally in coordinate systems of their own choosing, producing bit-maps in the resolution of the display device only at their final stage. A traditional model of graphics is useful here: an imaging system expects to produce a picture, or series of pictures, on a single display device, for the user to view and interact with. The application driving the imaging system does not want to deal with clipping boundaries which are not part of its abstraction, with the possibility that the color map may be changed from underneath it, or with the idea that it may be called upon to redraw some random section of the display which it has already drawn, and it seldom wants to directly consider transformations into device coordinate space. This view is justified in more detail in [6] and [7].

On windowing bitmapped systems, the imaging system can be considered to be writing in a *virtual frame-buffer* for which it alone is responsible. An application must be presented (barring active intervention on its part) with the illusion that it is the only application using this virtual display, in much the same way that the program itself is presented by the underlying operating system with the illusion that it alone is running on the processor. The window manager is responsible for supporting this illusion on a display system, in the same way that the operating system supports the virtual machine on a timesharing system [7].

Windowing, distinguished from *imaging*, is the process of layering or *tiling* these virtual frame-buffers into a physical frame-buffer or onto a physical display. Windowing systems need not and should not deal with the content of these virtual frame-buffers, only with their presentation on the screen.

This segregation of imaging and windowing functions has benefits for both areas: the windowing software (and hardware) support is concentrated in a single, shared unit consisting of the window management process and any supporting hardware; and the imaging systems are simpler, because they needn't worry about redrawing themselves at random times, or about clipping to window boundaries, and because existing, non-window-cognizant applications run without modification.

3. The Failure of Existing Imaging Hardware

New developments in graphics image-generation hardware are coming at such a fast pace that it is becoming difficult to keep up with them. A comprehensive list of imaging hardware would be far too lengthy to include, but a few of the more significant developments are:

- Jim Clark's Geometry Engine [8], a multi-chip processor allowing extremely fast three-dimensional transformations, clipping, and scaling;
- the NEC 7220 [9] graphics controller chip and other manufacturer's equivalents, popular because of hardware implementation of moderately fast line-drawing primitives;
- several hardware **bitblt** implementations: Silicon Compilers' chip for Sun Microsystems, Inc.; Apollo's proprietary bitmover; Texas Instruments', Motorola's, and National Semiconductor's, about-to-be-announced mass-market **bitblt** chips; and others; and
- very-high-performance rendering engines, typified by the Lucasfilm Pixar processor [10].

Several of the low-cost all-in-one CRT controllers make attempts at providing windowing, often allowing horizontal or vertical split screens, or a small number (typically one) of non-overlapped hardware windows. In all cases there are restrictions of such severity that this special hardware is useful only in special-purpose windowing environments.

Manufacturers with high-speed vector- and polygon-drawing engines, such as Tektronix and Silicon Graphics [11], tend to take the view that windowing can be accomplished by redrawing the entire screen from a stored display list when window changes are made, using the clipping and transform capabilities of the vector-oriented hardware. These techniques can be made quite effective, although the high-speed drawing engines are often too expensive for those who want windowing

and bitmapped graphics without sophisticated and expensive imaging systems. Furthermore, it is clear that window management is an inefficient and inappropriate use of such hardware.

Manufacturers of color displays who do not use the display-list redraw technique have particular problems with windowing, due to the extremely high overhead involved in **bitblt** when more than one image plane is involved. Moving bitmaps on one plane at a time produces unpleasant color effects if the transfer cannot occur entirely during the Vertical Retrace Interval of the monitor, an uncomfortably short period of time. In all color systems, extreme havoc will ensue if an application running in one window changes the display's color map. Since the color map controls the entire display, an application running in a window cannot change it without affecting all the windows on the screen.

4. Windowing Hardware and the Tyranny of Bitblt

Although **bitblt** chips have occasionally been used to good advantage in windowing systems, and some hardware exists that provides limited split-screen features, little hardware has been applied to windowing. Because the **bitblt** operator has been seen as the primary, if not the only implementation vehicle for windowing systems, most attention by both software and hardware developers has been toward speeding it up by the use of sophisticated and highly-tuned algorithms, or with specially-constructed memory arrays which reduce processor/display memory access collisions.

Bitblt's usefulness as the basic implementation tool for windowing is limited because it is ultimately the wrong approach. The historical use of **bitblt** for windowing is closely tied to the monochrome bitmapped display hardware on which windowing was first developed [2]. First used for painting small images in this frame-buffer memory, **bitblt** came to be used as the basis for the notion of windows, and has been used that way for over 10 years. Today the whole idea of copying blocks of data to arrange regions on the screen is as artificial as the ornate program-overlay systems developed in the absence of virtual-memory hardware on the processors of the 1960's.

There are performance problems with **bitblt** which can never be completely resolved, since fundamental window operations involve the transfer of such large amounts of data. Performance is a moderate problem in monochrome systems, and an extremely severe problem in multi-plane color systems. More importantly, the ALU operations of **bitblt** such as XOR have no clear correlate in color systems, where one wants more complex operations like *over* and *under* [12], and *textures*, used to good advantage in monochrome systems, don't work on color systems, where one wants shading. **Bitblt** forces programs to work in device coordinates in a world that is rapidly moving to device-independent coordinates. Windows and per-device color maps are ill-suited to one another, not providing the abstraction desired by the application.

Few have thought to look beyond **bitblt** at the deeper needs of a windowing system. The problem of windowing is not one of image generation, it is one of memory management, particularly memory mapping and access multiplexing. Once we throw away the idea that image-drawing technology has anything other than historical and incidental use in windowing, it becomes much easier to apply new ideas to the problem.

5. Advantages of Hardware Windowing

Hardware windowing has many advantages over software approaches, either display-list or **bitblt** oriented. The advantages spring partly from the purer embodiment of windowing that is enforced by the hardware, and partly from the hardware itself.

- **Applicability to Color** — the hardware windowing makes moderate-cost windowed color displays possible: per-window color maps eliminate a major problem area with existing windowing color displays; Window management software written for hardware-windowed monochrome systems is portable to color systems; and Color bitmaps (multi-plane or multi-valued pixel frame buffers) can be operated on without color swim.
- **Scrolling and Panning** — hardware windowing allows fast and simple vertical scrolling and

horizontal panning of bitmap images which are represented in a frame buffer without moving large amounts of data, and this in turn frees the imaging hardware to fill the newly revealed area, if it does not already reside in the frame buffer.

- **Simplification of Graphics Applications** — hardware windowing simplifies user-level graphics applications by: freeing applications from the need to unnecessarily redraw or clip their output, and thus of the need to maintain a display list; by allowing porting and development of non-window-cognizant applications; and by freeing applications from the need to use device-dependent coordinates.
- **Simplicity of Window Management Software** — Window Managers no longer need to multiplex access to the frame-buffer, or have any knowledge of the content of the frame buffer, and are generally smaller and simpler.
- **Speed** — in addition to scrolling speed, creation, deletion, and movement of windows is much faster than in a **bitblt** system, since no data copies are required.

The only disadvantage of hardware windowing is its current cost. In the absence of inexpensive VLSI solutions, windowing hardware is too expensive for low- and moderate-cost systems. The need for a frame buffer that is substantially larger than the total display area is a disadvantage, although as memory prices continue to fall this will become less of an issue. While a 4k-by-4k bit frame buffer plane costs \$64 today, this is expected to drop to \$16 or less in the next few years. Processing power will continue to cost a great deal more than memory, and hardware windowing substantially reduces demands on display processors for windowing.

6. Example Systems

In order to illustrate some of the ideas discussed above, and as an existence proof for windowing hardware, I am going to briefly discuss four hardware windowing systems, three bitmapped displays, and an alphanumeric display. Only the two alphanumeric displays are currently available or likely to become available in the near future.

6.1. Bitmap Displays — Background

Some numbers will be needed in our discussion to relate these ideas to reality, and will be presented here for future reference. A typical medium resolution (1024x768) monochrome display, running at 60Hz refresh (non-interlaced), paints a new screen every 16 milliseconds, paints an individual line every 15 microseconds, and thus paints a pixel every 10-20 nanoseconds. There is a 4-8 microsecond horizontal interval between each scan line, and a 200-600 microsecond vertical interval between each frame. A typical bitmap system is represented in Fig. 1.

In such a system, the *sequencer*, a simple address generator produces a stream of addresses that sweep across the frame buffer, successively transferring each word to a shift register that outputs one pixel at a time. Hardware window systems simply replace this trivial address sequencer with a more complex, table-driven address generator.

The window controller can be conveniently thought of as a memory management unit for the frame buffer. The window controller must translate between screen addresses, represented by a combination of vertical (scan line) and horizontal (per-line beam position) components, and frame buffer addresses, which consist of word-offset and pixel-offset components.

6.2. The Tektronix 6200 Display

The first example is a display system designed at Tektronix, Inc. as part of the Engineering Computer Systems (ECS)* Division workstation project. The hardware for this system was designed in 1983, and built in 1984 and 1985. The software architecture described here (and in more detail in [7]) was designed in 1983. A slightly different version of the architecture, implementing the virtual frame-buffer concept at a higher hardware level, was implemented in 1984 and 1985.

* Now Graphics Workstation Division (GWD).

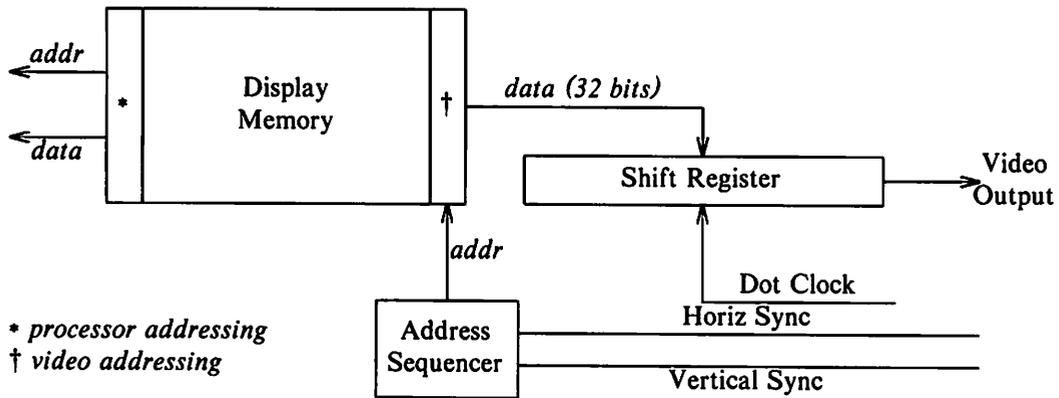


Fig. 1 – Typical Bitmap Hardware

The hardware consists of two processors: the Display Processor (DPU), a general-purpose processor which runs processes implementing the imaging and user interaction for each window, and the overall control and communication code for the display system; and a microprogrammed bit-slice processor (the *MicroEngine*) which is responsible for low-level windowing operations and graphics primitives. While the hardware for this system was powerful, it does not directly support windowing operations. This paper, however, takes the view that the MicroEngine and its microcode together comprise a hardware system.

The MicroEngine hardware is similar to analogous drawing processors in other Tektronix terminals, but is used in a much different way. Most terminals use these engines to traverse display lists of graphics primitives (line-draws and area-fills of sundry types, some transformations, and character drawing), to generate an image in the frame-buffer. The architecture sought to provide in addition a transparent *virtual frame-buffer* abstraction to processes running in the Display Processor (and by extension, to applications running on the host) by implementing a set of graphics primitives which understood the window structure of the display, and which automatically transformed and clipped their output to the appropriate window, while writing concealed parts of the image into non-frame-buffer memory. Thus, portions of the virtual frame-buffer which are not represented in the physical frame-buffer are cached in main memory. This implementation was formalized in [6] which presents the *layerop* concepts, specifically the idea of an automatically clipped *bitblt*, and the restartable DDA line algorithm. The MicroEngine is effectively a hardware *layerop* processor, automatically clipping images into on-screen and off-screen sections. The principal data structures of the 6200 are shown in Fig. 2.

A Window Manager process in the Display Processor maintains the *Window Descriptions* data structure representing the locations and extents of each window, the location (in physical Display Processor memory) of the display list (if any) associated with each window, and the locations (also in physical DPU memory) of any pieces of the window that are cached in the *Layer Pool*. The Display List may contain invocations of any graphic primitives implemented by the MicroEngine, from *lbitblt* calls to polygon fills. The Display List with which the Window Manager is concerned contains only image elements that have not yet been drawn. The Window Process is not required to retain previously drawn image elements. The Window Manager organizes the Window Description data by window priority and flags windows which are in need of update. The MicroEngine traverses the window list, executes the Display List for each window that needs updating, automatically clips the output of each primitive it executes to the visible area of the window on the screen, and directs output destined for obscured portions to the cached area in the *Layer Pool*. If a window extent is changed to reveal a previously obscured section of frame-buffer, the Window Manager appropriately

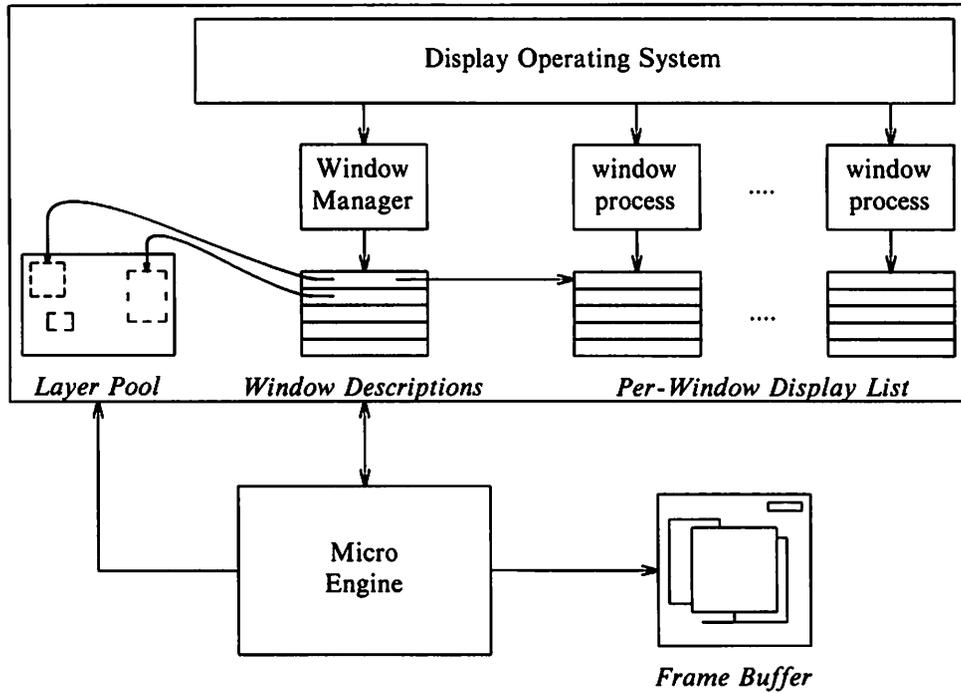


Fig. 2 — Principal Data Structures of original ECS Display Architecture

modifies the tiling information (represented in this case by a list of clipping rectangles) and signals the MicroEngine, which then copies the cached layer section into position.

Although it uses wholly traditional techniques for both windowing (**bitblt**) and for imaging (display list traversal), this architecture keeps these two operations separate throughout the generally-visible portion of the system. The illusion of the virtual frame-buffer breaks down only when the system runs out of memory for the Layer Pool, which was expected to be an unlikely event.

The 6200 Display gained great advantage from providing orthogonal imaging and windowing interfaces to application programs. Applications could be easily generated which used the imaging (graphics) interface, but did no windowing, or vice versa. Application code did not have to worry about having its window moved around the screen by the operator, about the current clipping boundary of the window, or about redrawing newly unobscured portions of the window. This approach did, however, lack the ability to scroll or pan without moving large volumes of data, and performed no better than equivalent hardware in this regard.

6.3. The VXL Window Hardware

The VXL*, despite the fact that it is character-mapped rather than bitmapped, is an interesting example of windowing hardware. The VXL is the first generally-available terminal to implement overlapping, scrollable windows in hardware. The hardware used to provide this capability is similar in principle to hardware which would be used to perform a similar function on a bitmap

* VXL is a trademark of Ann Arbor Terminals, Inc., the manufacturer of the terminal. The hardware was designed by Jim Russo, Michael Sleator, and Marc Schuman, and the software was designed and implemented by the author and Ken Rhodes.

terminal, though somewhat simpler and easier to examine.

The VXL provides a bank of *Character Memory* (28Kb) which contains a number of *virtual screens* (which would be *virtual frame-buffers* if this were a bitmap). These screens are logically (though not necessarily physically) contiguous strings of characters which, taken as a whole would represent the entire viewable area of a window. Thus, a 60-line by 80-column screen is represented by $60 \cdot 80 = 4800$ bytes of character memory. The terminal processor treats this memory in the expected way, as a two-dimensional array of characters, and inserts, deletes, or overwrites characters in the expected fashion. There may be any number of virtual screens in the character memory. All potentially displayable characters are necessarily contained at all times in this memory. There is also one special area that contains a long line of background characters (typically spaces).

The hardware provides an additional, distinct area of memory, called *Mapping Memory*, which contains a map of which positions in character memory are to be displayed on the physical screen, and in what order. Mapping memory is organized as a list of records, one for each line of characters on the screen. Each field in these records corresponds to a successive horizontal region on the corresponding line. The field encodes the length of this region and an address in Character Memory from which strings of characters are displayed. As an optimization, a field may also contain literal characters to be displayed.

The VXL Window Controller (represented in Fig. 3), traverses this list every frame, starting at the beginning every time a vertical sync signal is received. It fetches a word of line-control attributes and the first field of the mapping record, and emits the address, decrements the duration count, and increments the address. When the duration count reaches zero, the next field is fetched and the process continues. In this way the Window Controller emits a stream of addresses, one for each character to be displayed. This stream addresses the character memory in the expected way, and the resultant stream of character data is directed to a fairly typical alpha-terminal character generator and video back-end.

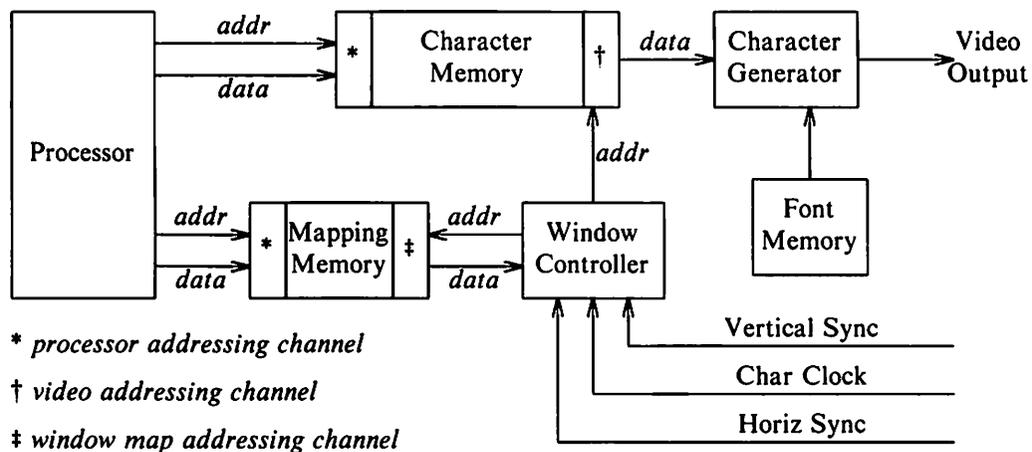


Fig. 3 — Ann Arbor Terminals VXL Hardware

To scroll a window vertically using this scheme, the terminal processor iterates over the window map replacing the appropriate field in the each record with the analogous field from the previous record. To scroll a window horizontally, the address in the appropriate field of each record is incremented or decremented. If these modifications to the display list occur wholly during the vertical retrace interval, the change is effectively instantaneous. Since there can be no empty (zero-duration) fields in the records, creation and deletion of windows involves some fancy footwork, specifically the insertion or deletion of the new fields in the record, moving any fields to the right to

create or fill the gap. Window borders are handled using the literal-character inclusion facility of the mapping memory (not otherwise discussed here), and thus do not occur in character memory. The VXL accomplishes all windowing operations, along with accepting input and decoding commands from four serial lines, with a 6-MHz Intel 8088 processor. The VXL windowing software consists of about 1000 lines of C code.

The VXL hardware was designed in 1982. In late 1984, both AMD [13] and Intel [14] announced low-cost alphanumeric CRT controllers that can be used to implement windows in much the same way and with very similar data structures. The presence of these chips in the market makes it likely that more windowed alphanumeric terminals will come into existence in the near future.

6.4. Bitmap Window Controllers

Hardware which applies a control-list based memory-mapping scheme of this nature has been developed at least twice, independently by John Providenza and Mike Zuhl of Tektronix, Inc, in 1981 and 1982, and by Michael Sleator of Ann Arbor Terminals, Inc, between 1981 and 1985. Both systems provide similar capabilities, though the two implementations differ significantly. I will concentrate on the latter system, dubbed the *Tessera**[15].

Windowing hardware for a bitmapped terminal is much like that of an alphanumeric terminal, except that *words* of frame buffer memory are mapped onto individual scan-lines, rather than mapping characters into rows. Two difficulties arise, the first related to the increase of in the number of potential vertical divisions (rows or scan lines) from fewer than 100 to more than 1000, and the second the limitation of mapping only word-width units onto the display.

6.4.1. Bitmap Windowing Implementation

A data structure (like the VXL's) that for each horizontal scan line contains a series of address/duration pairs will adequately map every 32 pixels on the screen into a word in the frame buffer. The video-generation hardware clocks the window controller with three signals:

- 1) a *dot clock* (one cycle for each pixel), that is divided by the word width, and to which the window controller responds by emitting an address from which the next displayed word is read. At the beginning of the scan line an address/duration pair is loaded into registers, and each time an address is required the contents of the address register is emitted and incremented by one and the duration register is decremented by one. When the duration counter reaches zero, a new address/duration pair is loaded from mapping memory.
- 2) a horizontal retrace signal (one cycle for each scan line), which generates no address, but instructs the window controller to move to the mapping data for the next scan line.
- 3) a vertical retrace signal (once per screen refresh), again generating no address, but signaling the window controller to start again from the top of the mapping data, at the first scan line.

Since, for a 1024 by 768 pixel system, up to 192Kb of mapping information would be required (assuming the naive format mentioned above), an unreasonably large investment in very fast memory would still be required. Once we realize that window boundaries seldom occur on every scan line of the display, or rather than many scan lines will contain exactly the same information as the previous line, we can restructure our mapping information to take advantage of this. If a vertical duration count is placed in front of each line's worth of address/data pairs, the controller can repeat those pairs until the that count is exhausted, then proceed to the next vertical section. The data structure looks like this:

* *Tessera* is a trademark of Ann Arbor Terminals, Inc.

```
struct memmap {
    int     vrepeat;      /* vertical repeat count */
    struct mapfield {
        word *addr; /* address of this section */
        int   hdur; /* horizontal duration */
        int   hscroll; /* horizontal scroll info */
        int   color; /* color-map information */
    } horiz[†];
} vert[†];
```

† — there are a variable number of these fields

For typical mappings this reduces the total amount of mapping memory by an order of magnitude. The total size for a worst-case scenario where a window boundary occurred at every 10 vertical lines and every 32 horizontal pixels would be somewhat less than 2500 address/horizontal duration pairs and about 100 vertical duration counts, or less than 20Kb. The *Tessera* realizes further reductions in mapping memory size by clever packing of address fields pairs, and in practice uses less 4Kb of mapping memory.

6.4.2. The Zuhl-Providenza Hardware

The Zuhl-Providenza machine is implemented much differently. It does not require supporting software to pre-tile the display into most-horizontal regions, but rather accepts a list of window locations (upper-left-hand corner) and extents (width and height), and a coverage (z-axis) value, to wit:

```
struct memmap {
    int *addr; /* address of ULC of window in frame-buf */
    Point scrn_ulc; /* screen address of window */
    int height; /* height of window */
    int width; /* width of window */
    int depth; /* z-axis (depth) of window */
};
```

Many simple engines are each loaded with a individual window descriptions. Each engine tracks the current horizontal and vertical beam address, and compares it against the bounds of its assigned window. They then, for each address to be generated, vote on whether their particular window is to be displayed. In the case of multiple *yes* votes, the engine with the lowest z-axis value prevails, and its address is emitted. The engines are straightforward, consisting of only a few counters and comparators each, and a complete window controller can be implemented by replicating a large number of them on a chip together with a single z-axis priority resolution circuit. Unlike the more general (and more expensive to implement) scheme of the *Tessera*, this technique is limited in the number of windows it can handle not by the total amount of mapping memory, but by the number of mapping engines provided.

6.4.3. Boundary Restrictions

In both systems, windows must begin on word boundaries in memory and on the screen. This restriction can be lifted with the addition of an output buffer on the video stream. If the window controller, which normally sits idle during horizontal and vertical retrace, can get ahead on the translation during these periods, buffering up output later clocked out by the video circuitry, it can access many more words of frame-buffer memory during a single scan line, allowing it, for example, to fetch extra words when window boundaries are broken across word boundaries. An drawing of this type of hardware is presented in Fig. 4.

6.4.6. Implementation

The Tessera hardware has been prototyped in random logic using a bit-slice microprocessor, some special-purpose hardware, a single 2k by 2k frame-buffer and a monochrome screen, and works well. Window creation and deletion are instantaneous, and users can scroll the internal frame-buffer across the screen as fast as desired, all without any visible tearing, inchworming, or other effects. Virtually no processing power is consumed by windowing operations. This implementation, unfortunately, is too expensive for inclusion in a Blit-style personal display, and at present, no implementations of either the Tessera or the Zuhl-Providenza display are available. Despite the cost, it is clear that hardware such as that described here is a necessity if windowing systems are to make the leap to color, and if adequate performance is to be had in low-cost windowed displays.

7. Conclusion

The only thing that makes this technology 'next generation' is the fact that there are no existing implementations. It should be possible for a clever hardware designer to create a chip implementing a system similar to those described above. Barring another unforeseen breakthrough in windowing technology, the next generation will have come when it is as common for a windowed display to use this style of hardware as not. It is hoped that that day will come soon.

In [6] reference is made to [4], in which the statement is made: "Research needs to be done to develop a way in which to conveniently store and manipulate graphics data in the context of a window manager." While [6] provides an elegant software solution to this challenge, the hardware solution, when fully realized, will change the way we implement windows as radically as virtual-memory changed the way we wrote programs.

Acknowledgements

Thanks to Michael Sleator, who convinced me that hardware isn't all bad; to Jim Valerio, who reviewed several early versions of this paper; and to Rob Pike who, as referee, provided helpful and timely criticism.

1. R.F. Sproull, "Raster Graphics for Interactive Programming Environments," *Computer Graphics* 13(2), (Originally, Xerox PARC CSL-79-6) (August, 1979).
2. D.H.H. Ingalls, "The Smalltalk Graphics Kernel," *Byte* 6(8) (August, 1981).
3. L. Tesler, "The Smalltalk Environment," *Byte* 6(8) (August, 1981).
4. N. Meyrowitz and M. Moser, "BRUWIN: An Adaptable Strategy for Window Manager/Virtual Terminal Systems," *ACM 8th Symposium on Operating System Principles* 15(5) (December, 1980).
5. K.A. Lantz and R.F. Rashid, "Virtual Terminal Management in a Multiple Process Environment," *Proc. of the 7th Symposium on Operating Systems Principles*, ACM.
6. R. Pike, "Graphics in Overlapping Bitmap Layers," *ACM Transactions on Graphics* 2(2) (April, 1983).
7. S. McGeedy, "Window Managers are Operating Systems: Software for a Distributed Graphics System," *Proc. of the 1st Symposium on UNIX and Graphics*, USENIX Assoc., Monterey, CA, December 1983 (forthcoming).
8. J. Clark, "The Geometry Engine: A VLSI Geometry System for Graphics," *SIGGRAPH 1982 Proceedings* (July, 1982).
9. *NEC 7220 CRT Controller Data Sheet*.
10. A. Levinthal and T. Porter, "CHAP — A SIMD Graphics Processor," *SIGGRAPH 1984 Proceedings* (July, 1984).
11. R. Rhodes, Haerberli, and Hickman, "MEX — A Window Manager for the IRIS," *Proceedings of 1985 USENIX Conference* (June, 1985).
12. T. Porter and T. Duff, "Compositing Digital Images," *SIGGRAPH 1984 Proceedings* (1984).

13. *Advanced Alphanumeric Display Products Specifications*, (Am8052 Data Sheet), February, 1984.
14. *Intel 82720 Data Sheet*.
15. M. Sleator, *Methods and Apparatus for Computer Display with Windowing Capability*, U.S. Patent Office (Sept. 18, 1985). Patent Application

Real-Time Resource Sharing for Graphics Workstations

*Mark S. Grossman
Glen E. Williams*

Silicon Graphics Inc.
2011 Stierlin Road
Mountain View, California 94043
(415) 960-1980

ABSTRACT

The IRIS is a real-time graphics workstation that supports object-space 3-D graphics. In order to achieve concurrent graphics in windows, we have had to address several issues involving resource allocation and synchronization. This requires sharing special-purpose hardware among competing tasks. This paper discusses solutions that involve UNIX* kernel and hardware cooperation.

1. Introduction

In less than ten years real-time systems have evolved from multi-rack behemoths to deskside workmates. 32-bit microprocessors and custom VLSI contributed to the creation of color workstations such as the Silicon Graphics IRIS. All of the power of such a device is meant to be used by a single user performing multiple tasks with realistic response times. Although this device can be used in a local area network, all the work for generating real-time displays is performed within the device.

The graphics workstation emerged as a productivity tool because it offers a general programming box and the perceptual impact of realistic pictures. In the era of the giant simulator, development and modeling were off-line tasks performed under a different operating environment. A workstation is particularly valuable because it serves both as the prototyping/development tool and as the end product itself.

Another outgrowth of technology evolution is that users now demand more task handling complexity. Many workstations are thus now providing some kind of window management capability, so numerous views of a single project or multiple projects can be multiplexed.

The IRIS is based on the UNIX operating and provides a window manager that supports real-time 3-D displays. One might argue that "real-time", "multi-window" and "UNIX" are incompatible terms. *Real-time* implies that any given display process has to be able to respond instantaneously to changes in some data base or input event. *Multi-window* implies that there are multiple graphics processes, each of which has independent control over its universe of resources. UNIX has traditionally meant unprioritized interrupts, unpredictable scheduling and substantial interrupt latency. This paper discusses some ways to resolve these conflicting worlds in a workstation.

2. Some Criteria

A real-time system can be characterized by response time and data rates. System response time has been defined as "the time within which a system must detect an internal or external event and respond with an action." [6] An external event can be as diverse as a mouse movement or the

* UNIX is a trademark of AT&T Bell Laboratories.

arrival of a real aircraft's coordinates coming from a network node.

Vendors of every sort of graphics machine from PCs to visual simulators claim a right to the "real-time" tag. Given a criterion of adequate response time, they do this by constraining the system's generality or scene realism. Even the response time issue may be compromised by reducing frame update rate for complex scenes. Newman and Sproull define real-time graphics as the ability to scan-convert a 30 or 60 Hertz monitor with a changing picture at that rate. In these respects, a \$50 video game is a real-time graphics device.

For there to be a fair metric for comparing real-time graphics workstations, system data rates must be measured at various levels. For example, it is equally important to measure the matrix transformation rate of high-level database objects as it is to measure the time needed to scan convert graphical primitives. Moreover, the solution to the data rate problem requires that attention be focused not only on the performance of each component (such as a matrix multiplier), but the way each is connected to its neighbors (system tuning).

It is regrettable that criteria in this area are so ill-defined, because response time directly affects user productivity. For example, in a study performed by IBM [4], sub-second response to human input was shown to be pivotal in the productivity of users with varying skill levels.

3. The IRIS

The IRIS is a system intended to address the above issues and constraints, yet deliver acceptable functionality.

The graphics hardware of the IRIS is divided into three pipelined components (Figure 1): the applications/graphics processor, the Geometry Pipeline, and the raster subsystem. The applications/graphics processor runs the applications program, and controls the Geometry Pipeline and the raster subsystem. The key component of the Geometry Pipeline is the Geometry Engine, a configurable VLSI processing element for graphics [2]. Twelve Geometry Engines are used to build the Geometry Pipeline subsystem that can map graphic primitives from user coordinate space to some region of screen space.

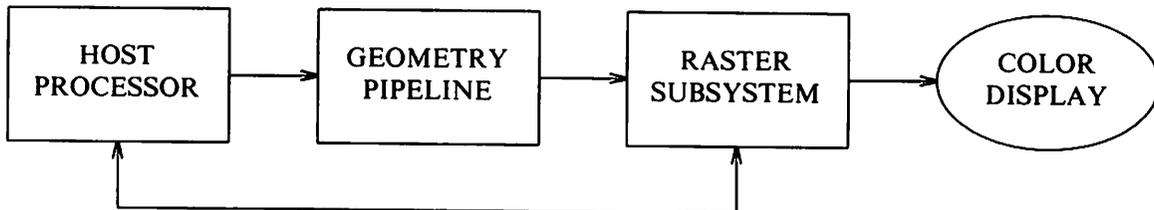


Figure 1.

Graphics commands are sent through the Geometry Pipeline, which performs matrix transformations on the coordinates that can be expressed as 2-D or 3-D in user space, clips the coordinates to normalized eye space, and scales the transformed, clipped coordinates to screen space. The output of the Geometry Pipeline is then sent to the raster subsystem. The raster subsystem fills in the pixels between the endpoints of the lines, fills the interiors of polygons, converts character codes into bit-mapped characters, and performs shading, depth-cuing, and hidden surface removal. A color value for each pixel is stored in the system's bitplane memory. The values contained in the bitplanes are then used to display an image on the monitor, either through a color map or directly to the red, green, and blue CRT guns.

Users write their applications in C, FORTRAN, or PASCAL. The IRIS Graphics Library provides a procedural interface to the graphics capabilities of the IRIS. Most graphics primitives translate into commands and coordinate data that are sent directly to the Geometry Pipeline from the user's process. A subset of the graphics primitives is handled by the kernel graphics library (KGL), which is an augmentation of the UNIX kernel. The KGL handles traditional kernel duties, such as keyboard and mouse I/O. It is also responsible for directing events from such devices to the proper window using "event queues". There is one event queue for each graphics process.

4. Design Considerations

We chose the architecture outlined above because of a few compelling principles.

- Generality** The user should not need to worry about the physical limitations of the display medium. Rather, he should be allowed to construct models in a high-precision representational space. The Geometry Engines allow this level of conceptualization. Accordingly, movement of a three-dimensional object produced in this space should be specifiable in an efficient way. The Geometry Engine provides the means to do this using no more than five "short" integers (16 bits).
- Speed** We wanted to provide a data rate high enough to be able to display dynamic scenes with a reasonably high level of complexity (on the order of 4000 graphic primitives per frame). This is achieved by offloading the geometric computing and the rendering tasks to the Geometry Pipeline and the raster subsystem.
- Realism** We wanted to produce smooth, realistic images. Double buffering allows the system to generate the next scene without disturbing the displayed scene. In addition, it provides true rate buffering between the update and refresh tasks. The IRIS hardware supports smooth shading and depth-cuing of colored objects, which allow generation of scenes with realistic lighting models and with enhanced 3-D perception.

5. Questioning Bitblt

These considerations taken together: display space independence, fast scene rendering, and color us away from the BITBLT approach to graphics.

BITBLT mainly copies pixels; optionally it can perform special logical operations between source and destination pixels (RasterOp, [8]). We felt we wouldn't use the features of BITBLT for three reasons.

1. Using logical operations on bits that represent colors would result in amusing but unacceptable results.
2. Using BITBLT merely to move bits was simply not useful. A common use of BITBLT is moving windows. However, a window's aspect ratio can change when it is moved. On the IRIS, the contents are resynthesized, taking the new aspect ratio into consideration. Since the objects in the window could have new shapes or sizes, BITBLT (i.e., a straightforward copy of pixels) would not be appropriate.
3. Moving all those bits uses bandwidth. The efficiency of the bitplane bus is cut in half by reading the bits before writing them. In the IRIS, the bits are generated and written by the scan conversion hardware.

6. Evolution to UNIX

Given all of our hardware resources, the question of how to provide a usable tool remained. Motivated primarily by the desire to deliver fast graphics, we produced a terminal that is display-list oriented. The user is provided a graphics library, an interface with which to create and manipulate display lists in the terminal. The interface resides on the user's host machine; calls to it generates tokens that are transmitted to the terminal (the media supported included serial line, IEEE-488, IP/TCP and Ethernet.) The tokens are parsed in the terminal by an interpreter that has been downloaded at boot time. The software in the terminal included the graphics library interpreter and a real-time executive, the V kernel [1], which supplies us with a basic set of resource management facilities.

The terminal met our expectations: it was fast and could incorporate data from valuator directly into executing display lists.

However, display lists have limitations. Applications must often maintain two sets of representations of the same data: the display lists and the application's data structures. For some applications, the mapping is direct. However, some applications require that the user's data structures be

traversed, issuing graphics commands in passing. That is, rather than draw 100 points by constructing a display list containing 100 sets of coordinates, one could instead create a function that generates the "point" commands and coordinates directly. Display lists are also difficult to edit. Again, every change in the user's data structure must be reflected in the display list. Special tagging and pointer manipulation tools must be invoked to change any entry, especially if the internal structure of the display list is not known to the programmer. But using the capabilities of a general-purpose programming language, changes to graphics parameters can be made as part of the general application maintenance.

Our company's solution was to create a workstation that extended and optimized the operating system for faster graphics execution without requiring the user to build display lists. The workstation by its very nature more tightly couples the general purpose computing engine and the graphics resources. We were interested in choosing an operating system that could easily accept new peripherals. We wanted an operating system that already had a customer base, and finally, one that was not too hard to port. We chose UNIX.

Having chosen UNIX, we then set out to minimize what we found to be deficiencies from a real-time graphics perspective. We also designed a window manager that allows processes to run real-time graphics simultaneously. In the workstation, there is still the concept of the "host" processor, but in this case, it is a Motorola 68020 CPU that both runs UNIX and feeds graphics commands to the Geometry Pipeline.

In designing the workstation, the numerous problems we encountered fell generally into five categories: hardware ownership, processes and contexts, scheduling and synchronizing, load balancing, and sharing hard resources.

7. Hardware Ownership

There are numerous ways to connect a piece of hardware to UNIX. One is to use "device" protocols, whereby a client can allocate the resource through the *open* and *close* system calls, and access it through *read* and *write* calls. In our analysis, this virtually precludes multi-client real-time use, due to the number of kernel software layers needed to perform the underlying I/O operation.

Another connection mechanism is to have the hardware "owned" by a single process that handles requests from other clients and manages the hardware I/O. This requires at least two context switches with an intervening scheduler call. The IRIS instead gives direct control over the hardware to a custom part of the kernel itself, the Kernel Graphics Library. Messages and status information are communicated to clients through a small section of shared memory. The kernel fields hardware interrupts from the graphics subsystem, comprising coordinate feedback, vertical retrace, and other events. In addition, a process that wants to do graphics is given direct write access to the Geometry Pipeline through its memory space. This is the path over which most drawing primitives, such as point, line and polygon drawing commands, are sent.

A special advantage in system bandwidth is gained from the pipeline structure of the graphics hardware. Many commercial architectures treat graphics processing units—geometric computing engines or rendering engines—as separate peripherals, connected to the system bus but not to each other. In the IRIS, each unit communicates directly to the its neighbor, freeing the system bus from the intermediate transfers.

8. Processes and Contexts

SGI conducted a special study on the UNIX System V scheduler in the interest of achieving an interactive "feel" to the graphics [5]. A notable artifact, eliminated in the Berkeley Version 4.2 UNIX implementation, was the switch of the rescheduler into process 0 before it ran. A less obvious problem was deciding just how to assign time slots to graphics users. We postulated that if a user wanted to run some number of different windows, he or she wanted each window to have the same responsiveness. The most-unused, round-robin scheme was maintained, but two changes were made. First, the time slice given to each processes decreases according to the number of processes in the queue, insuring adequate response to external events. Second, since the rescheduler algorithm was left intact, the real time it takes for any process to decay to inactivity remains invariant with the

number of competing processes. This means that the user sees consistent behavior no matter how many windows are on the screen.

8.1. Switching Overhead

One of the drawbacks of scheduling responsive processes under UNIX is the overhead of context switching. Many general microprocessor design tricks have been devised to expedite UNIX's job. A common focus is the memory address-mapping hardware. Most schemes embody a context-segment-page map structure in some form. The page map, subdivided into text, data, and stack segments, translates from virtual page addresses to physical memory addresses. Several contexts, representing UNIX processes, own a set of segments coexisting simultaneously in a dedicated map memory. Additional hardware detects limit violations to implement protection and demand-paging schemes. The result is a system that can efficiently switch among processes and maintain high memory bandwidth.

But in the multi-windowed IRIS, a considerable amount of state data used in the graphics hardware must be changed as the active window switches. This information—including transformation matrices, drawing attributes, and color maps—can be considered the *graphics context* for the current *graphics process*. There is a time lag between the UNIX process and the associated graphics process, and there is currently insufficient storage in the graphics hardware for its own inactive contexts. So two problems in swapping arise: the *linkage* between the two processes and the *overhead* required to move graphics contexts.

Parts of the graphics context are sufficiently compact to be treated dynamically. Of these many are shadowed in a data structure associated with the UNIX process, from which they are rapidly sent to the hardware as an initialization step. Examples of these are the current color code, linestyle, and shading mode. The other dynamic attributes—current graphics position and matrix stack—must be retrieved from the hardware and saved in the data structure of the host processor. Thus arises the linkage problem: the two parts of the system must cooperate synchronously in order to perform the context-saving operation. The flow of state information across this graphics-UNIX interface bears consideration as the secondary problem; it is sufficient to increase the switching time by several hundred microseconds.

8.2. Context Reservoir

A key to solving the swapping problem is to view the graphics device as a true multi-user resource. Optimum performance, at the expense of duplicate structures, could be provided in hardware by placing a process list at every point where a dynamic variable (e.g., line style, viewport) exists. This way a number of concurrent graphics processes can access their own attributes by means of a simple pointer movement, but this number has a fixed upper limit.

A more flexible solution takes advantage of the directional flow of graphics commands through the pipeline. By adding a parallel path from the last processing element to the first, a mechanism can be developed for managing save and restore commands independently of the UNIX processor (see Figure 2). For a save operation, each element of the pipe passes its state information forward to a reservoir placed at the end. For a restore, the appropriate command is passed to the “reservoir reader”, which is capable of issuing data for the new context to the upstream elements. Depending on the size of the reservoir it can be viewed either as a cache for most-recently-used window states or as complete storage for windows not currently updating. This enhancement would allow the switching work to proceed concurrently in the host and in the graphics processors. This concurrency and information decoupling can facilitate future development of schemes involving multiple processes per window and multiple windows per process.

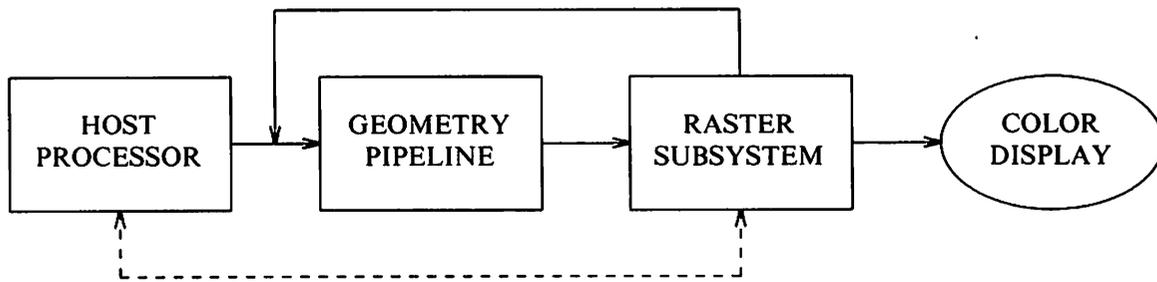


Figure 2.

8.3. Feedback

The Geometry Pipeline can be used in other ways than displaying graphics. Part of making graphics interactive is to be able to query the current state of the graphics scene. As we will show, this kind of feedback poses problems in a multiple-graphics window system. The most interesting uses of feedback are bounding box testing and picking. Here a set of clipping and/or scaling operations results in the selection of a graphical object or the culling of objects from a display list. The bounding box command sends a set of world-space coordinates down the pipe in a special operating mode. In the process of mapping the coordinates to screen space, the results of comparison against each of the clipping planes are recorded. The "or" combination of these results indicates how a partially visible object falls into the current viewport. The differences between the maxima and minima of the coordinates indicate the size of the object in pixels. A bounding box that falls completely outside the viewport will not produce any transformed output at all, and a message of this fact must be produced. For a picking operation, the coordinates are discarded and only the object names and clipping results are collected. Judicious use of local intelligence in the Geometry Pipeline can help reduce the number of tokens necessary to send back to the requesting process; nevertheless, some information must be transmitted.

The current scheme for any kind of feedback to the host processor involves the following steps. The bitslice controller in the raster subsystem generates a hardware interrupt to the host processor when it has collected the data to be fed back. The kernel, based on a message type code read from the bitslice, understands the destination and size of the packet. Some messages, such as errors and matrix stack contents, are directed to other parts of the kernel itself. Bounding box coordinates or object names are copied directly into the memory space of the requesting process, which must be locked in memory for the duration of the entire transaction. This is necessary because as long as the Geometry Pipeline holds data to be fed back it cannot effectively perform any graphics tasks for other users. Thus, not only must the application block for the interrupt servicing and pipeline latency, but a special priority demand is placed on the scheduling mechanism.

An aggressive attempt to reduce the various latency times would have payoffs here. But some amount of power and flexibility can be gained if the *request* for feedback can be decoupled from the *reception* of feedback data. A single-task real-time system might not want to do this, but an optimized multi-task one might, especially if process-switching has been optimized in the system. The freedom to swap in a higher priority competing process during a lengthy feedback operation may be valuable. Further, the user may benefit from the ability to perform his own alternative task between the request and reception.

An alternative to the current scheme which allows normal scheduling is to treat feedback packets as messages placed in users' event queues. The kernel would have to perform the buffering of the data for any inactive process. This buffering would halve the hardware-to-host bandwidth, and incur additional overhead for the message-passing protocol.

Another alternative involves providing the same sort of reservoir facility in hardware for context-specific buffering as was shown useful for the save-and-restore function. In this case the switchability allows the creation of virtual feedback channels. It requires that a given feedback process be interruptible at any time at the hardware level. The end-of-pipeline processing element would manage some number of FIFOs—either hard or soft—that would be allocated to graphics processes performing feedback. The same commands that initiate state saving and restoring could also control the activation and de-activation of the FIFOs. As with DMA, fed-back data is only

moved once: directly from hardware to the requesting process.

9. Scheduling and Synchronizing

This section concerns allocating fixed resources in a time-varying manner. Various graphics subsystem resources must be shared by competing processes: the Geometry Pipeline, the color map and the frame buffer memory.

9.1. Geometry Pipeline

The Geometry Pipeline is a device that is allocated dynamically. It is connected to the host processor by means of a hard-connected port addressable by any process. An early problem was reserving the hardware port long enough for a process to send a complete, atomic multi-word command/data stream. We devised a "free/busy" token that provided synchronization without incurring a great deal of switching overhead. As described in the Mex implementation paper [7], the token bit is manipulated directly by means of a subdivision in the port's address space: sending the first word of a command to one address sets the "busy" indicator; sending the last word to the other address clears it.

When a graphics process is about to be activated, if the synchronization bit is in the "busy" state, the scheduling of the new process must be delayed and the old one resumed. The time cost of this, while less than a full context switch, still involves an interrupt period and a kernel startup.

This requirement arose out of the development of an economical data path that makes no explicit distinction between commands and data, and of an economical state machine in the pipeline processing elements that does not recognize command interrupts. A proposed enhancement would provide fully interruptable processors that could respond to state-change requests at any time. Then streams of commands from different processes could be sent with a compact change command as a separator. As soon as the first pipeline processing element is through switching, processing the new commands resumes.

9.2. Color Map Editing

Another interesting example of scheduling behavior concerns the color map. As with most implementations, a single, shared address port to the color map memory means that host access must occur when the color map is not being used for displaying visible portions of the screen. Thus the most logical time to make changes is during the vertical retrace interval, when a relatively large time window is available. An early solution was to force a process to block until the interval arrived. In the current IRIS, map changes are sent as requests from the application to the kernel and are placed in a software queue; when the retrace period arrives, a hard interrupt causes the kernel to send as many of the queue entries to the map hardware as it has time for. Meanwhile, the application is free to make changes in the current color attribute used for updating the frame buffer. This freedom must be carefully considered when using a single-buffered display to avoid flashing items on the screen.

Again, the best solution seems to lie in offloading the queuing function to hardware. An engine in the graphics subsystem would be given direct control over the writing of the color map, and would receive the vertical retrace signal as an interrupt. This way a map-writing daemon without the encumbrances of a UNIX process would be activated.

9.3. Frame Buffers

The constraints placed on synchronization by the retrace event extend to the sharing of frame buffers. The raster subsystem, and hence the window manager, are capable of operating in either single- or double-buffered mode. In double-buffered mode, a hardware flag determines which of two sets of bitplanes is available to the update controller and which is actively refreshing the display. In order to avoid "tearing" a dynamic image, the swap between the two is made only after a complete field sweep is made by the video beam. In an early implementation, when a particular window had finished producing a new image in the update buffer, it sent a swap request to the kernel and went to sleep until after the vertical retrace begins. The kernel at that time toggled the display and

reactivated any sleeping graphics processes. More recently, a semaphore was added to the shared memory space that allowed the kernel to announce the retrace event to the clients. Graphics users then could not only avoid the mandatory sleep/wakeup overhead, but could opt to schedule interim subtasks with a simple polling mechanism to signal completion. With the further addition of a field-locked timer, even more intelligent scheduling decisions could be made, either by the user or by the system scheduler.

With the color map and buffer swap problems out of consideration, it seems attractive to free the host of any concern over the display retrace event. By fielding all retrace-related activity and allowing the buffer mode to be part of the graphics context, a powerful simplification of the software is made possible.

10. Load Balancing

We have spent time considering how to make each subsystem resource perform graphics tasks optimally. This section addresses the following concern: how can the user schedule resources in order to achieve the fastest action possible with as complex a scene as possible?

10.1. Hardware Assistance

Hardware FIFOs have been heavily used as a solution to differential rate problems in real-time systems. If the data rate distributions can be adequately predicted at the system design phase, a FIFO of appropriate depth can offer a straightforward tradeoff between the upper limits of throughput and latency. For example, a longword FIFO between the processor and the Geometry Pipeline smooths the transmission rate of commands by the graphics library code. In dedicated flight training simulators built by Evans & Sutherland, a much larger FIFO, combined with careful ordering of graphic primitives by software, minimized wait time between the geometric and display processing subsystems.

10.2. Software Optimizing Filter

Certain classes of load predicting can be anticipated and handled with the aid of algorithmic tools. The appearance of the directives would resemble predicates in theorem-proving languages. Such directives can be generated by some of the more common latency scenarios in the system. For example, certain commands bog down the Geometry Pipeline (the actual Geometry Engines), while others become lodged in the raster subsystem. Currently, the programmer must balance the load if the graphics is to run optimally fast.

Let's say a *draw point* command takes 10 *usec* to clear the Geometry Pipeline, followed by 2 *usec* in the raster subsystem. On the other hand, a *clear screen* command (which completely paints the current viewport with the current color) takes 2 *usec* in the Geometry Pipeline and 15 *msec* in the raster subsystem. Filled polygons fall between these limits. If 1000 polygons and 3000 points are to be drawn, it makes sense to intersperse some of the polygons with points. Depending on the depth of FIFOs within the system, the filter may suggest breaking up the sequence of commands by sending 25 points for every 10 polygons for a substantial gain in throughput.

Lacking enough disparate commands to decluster (disparate in transformation/imaging speeds), the filter may generate a suggestion that the application do computation at certain command intervals. This can be quite useful in applications involving real-time motion, as some amount of computation is required to guide objects through the scene. One implementation might link the filter to the program by the filter's putting the main process to sleep and triggering a subprocess to perform computation.

The filter can be running concurrently to the graphics process or it can be a preprocessor that generates "good taste" programming suggestions to the graphics programmer. A subsequent version of the filter could be a filter that actually runs parallel to the graphics process as a UNIX-style filter. This filter would look for clusters of commands that can be reordered and feed them to the Geometry Pipeline in a sequence driven by the recent history of what went down the pipe.

11. Sharing Hard Resources

Certain resources in the graphics subsystem, such as the color map, are too large to be replicated or to be dynamically updated on a time-slice basis. These hard resources must be subdivided and allocated among competing users, or shared in some cooperative scheme.

A traditional problem for multi-window systems is how to share screen space. A number of elegant solutions exist in the literature, lying mostly in optimizations of display hardware and BITBLT algorithms. [9,10]. In a system such as the IRIS, whose strength lies in its ability to rapidly recreate window contents from a high-level description, the monolithic x-y addressed screen space is broken into pieces that are managed by the window manager process [7]. It is up to the bitslice processor in the raster subsystem to control the clipping of each graphics primitive to a list of rectangular pieces that make up a viewport. Performance degrades gracefully as complexity of window overlap increases and pieces are added.

Many commercial systems reserve one or more overlay bitplanes to sidestep the plane-sharing problem for high-priority displaying purposes. Examples are pop-up menus and special-purpose windows. These guarantee that response to user input is immediate, and the ports can be given highest visual priority without requiring redraw of the background after erasure. These special planes can also be given their own color map space, avoiding that particular allocation issue as well.

The color map is also considered a hard resource. Bitplane area has been divided up among the graphics users, but the planes themselves ordinarily form a monolithic input bussed to a single color map. Due to the size constraints of map memory (which are costly due to the very high pixel rates they must accommodate), most systems limit the palette to 256, 1024, or 4096 colors. Allocating these few among applications that are unable or unwilling to share can frustrate users.

Here the idea of overlay planes can come to the rescue, especially when generalized somewhat to the idea of identity (ID) planes. We propose reserving a small set of bitplanes that can act as a guide to the use of the remaining planes without serving as color map inputs. Rather than giving a single overlay highest viewing priority, the ID planes hold both the priority level and the palette selection for each viewport on the screen.

We start by assuming that a user may want to see activity in overlapping and non-overlapping viewports, but may not demand a great depth of viewport overlap. Thus we can assign the same ID to all non-overlapping viewports at the same depth (Figure 3). So n ID bits translates to 2^n layers of non-overlapping viewports on the screen. Viewports are moved or re-prioritized by erasing and filling a suitable area in the ID planes with the desired shape or ID number.

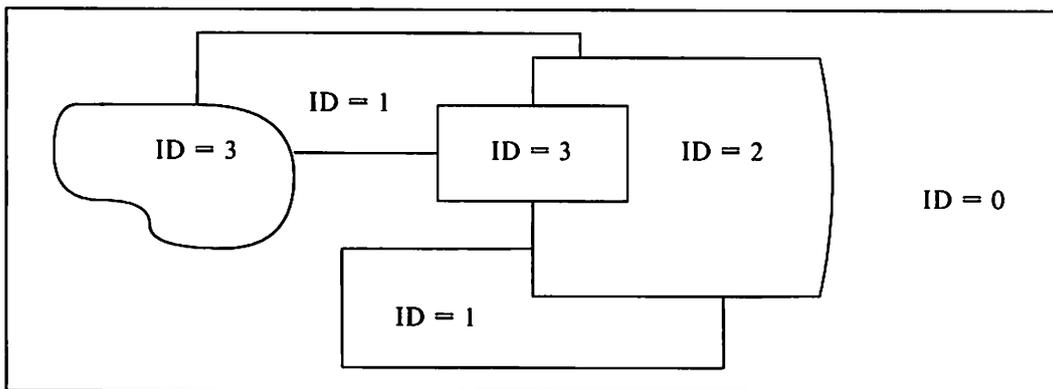


Figure 3.

Having entire bitplanes' worth of memory for IDs may eat up savings made in the color maps, but certain unique advantages accrue. Window priority level can apply on a pixel-by-pixel basis. The raster update engine would compare the ID value stored at a pixel location with the current window layer number before updating that pixel. This permits arbitrary shape viewports to be

drawn into, trading one serial task for the bitslice processor per rectangular piece for perhaps many parallel tasks at the pixel level. Second, having decided on a maximum color map size, the ID planes can be used to select an alternate map of equal or smaller size. As IDs increase linearly, the number of maps can also increase linearly, rather than by powers of two. Finally, the ID planes can be used to make per-viewport buffer or bitplane set selections. In conjunction with a set of multiplexers or crossbar-type circuitry, a great deal of independence in hardware-to-window assignments can be achieved.

11.1. Font Management

In the IRIS, the font store is a sharable hard resource. The fonts are stored at the “end” of the system as two-dimensional masks. The masks are kept in a special font memory integrated with the bitplane update hardware. Commands to paint characters into the frame buffer are passed through the pipe; characters are rendered by writing the current color into the bitplanes at pixels corresponding to “ones” in the masks. This scheme is used because it is compact. Even so, it is not feasible to swap font sets with processes. The bandwidth required is too large: the font memory is loaded by passing the font information through the pipe.

We chose not to represent characters as spline outlines that can be scan-converted for two reasons. First, the customers do not yet need a large font mix: the current font store is large enough. Second, at this resolution, the fonts need to be tuned. While there are methods to tune fonts algorithmically, such a procedure does not lend itself to real-time operation.

To add a larger font set, we have considered providing a character-mask cache with DMA support. This would allow many more fonts at higher volume. This is also useful when considering grayscale fonts, which take more space. In either case, storing the fonts near the bitplanes is attractive for bandwidth considerations. Note that even with the DMA approach, there is no concept of pairing graphics processes with font sets and that this must be managed through the cooperation of the concurrently running processes. But given adequate hardware, we could meet the goal of providing context-sensitive DMA capability. This can be done by maintaining hardware address tables for fetching fonts associated with the current process.

12. Function Migration

A pattern of migration is emerging in graphics devices—more and more complex functions are moving to hardware. This continues the trend that started when the original graphics controllers were introduced in the marketplace. [3] The migrations discussed here suggest that a multi-window graphics system should be complemented by hardware possessing multiprocess capabilities. Developing faster and faster rasterizing techniques will of course improve real-time performance; but more important is the process of adding moderate amounts of intelligence to provide virtualization of the resources associated with a window (i.e., each user thinks he owns the machine completely). The pitfall with adding this intelligence is that someone will eventually want to program it in order to customize it. The proper way to offer this power is to provide a layered procedural interface to the system. The lowest level might contain a few domains that present their interfaces to the next higher level. When a user needs to access or modify the system at a given layer, it is his responsibility to leave the interface intact.

13. Conclusion

General purpose real-time graphics systems are not cheap to build. Demands of a wide variety of applications create problems that must be handled flexibly by an architecture, meaning a larger system. User understanding and involvement in the implementation of the real-time aspects of the package are key to optimal performance. A system that provides comprehensible tools for dealing with these aspects is a key to users' success.

Acknowledgements

Jim Clark and Marc Hannah invented the Geometry Engine. Kipp Hickman, Paul Haeberli, Peter Broadwell, Rocky Rhodes, Henry Moreton, and Tom Davis implemented the IRIS UNIX and graphics software. These people and many others contributed to our thinking in this paper.

Bibliography

- [1] Cheriton, D.R., and Zwaenepoel, W., "The distributed V kernel and its performance for diskless workstations", SIGOPS Operating Systems Review (ACM), 17(5) July 1983.
- [2] Clark, J.H., "The Geometry Engine: A VLSI Geometry System for Graphics", Computer Graphics, pp. 127-133, 16(3) July 1982.
- [3] Clark, J.H., "The Wheel of Reincarnation", Panel, "Fundamental Algorithms: Retrospect and Prospect", proc. ACM SIGGRAPH, July 1985.
- [4] Doherty, W.J., and Thadhani, A.J., The Economic Value of Rapid Response Time. 1982; International Business Machines Corporation, White Plains, New York, 10604.
- [5] Hickman, K., "Some Enhancements to the System V Scheduler", unpub. memo, 1985.
- [6] Hindin, H. J, and Rauch-Hindin, W.B., "Real-Time Systems", Electronic Design, pp. 288-318, January 6, 1983.
- [7] Rhodes, R., Haeberli, P., and Hickman, K., "Mex - A Window Manager for the IRIS", proc. Portland USENIX conference, 1985.
- [8] Newman, W.M., and Sproull, R.F., Principles of Interactive Computer Graphics, 2nd ed. McGraw-Hill, 1979.
- [9] Pike, R., "Graphics in Overlapping Bitmap Layers", ACM Transactions On Graphics, 2(2) April 1983.
- [10] Wilkes, A.J., et. al., "The Rainbow Workstation", University of Cambridge Computer Laboratory, Cambridge, UK. August 1983.

GLO – A Tool for Developing Window-Based Programs

Thomas Neuendorffer

Carnegie-Mellon University

CDEC, 202 UCC

5000 Forbes Ave.

Pittsburgh, Pa. 15213

tpn%cmu-itc-linus@@pt.cs.cmu.edu.ARPA

1. Introduction

GLO (the Graphic Layout Organizer) is an application builder's tool designed to provide easy access to the facilities of Carnegie-Mellon's window-oriented Andrew system¹. It allows one to arrange a set of layouts (rectangular sub-windows) within a window, define their actions, create a working prototype, and ultimately turn that prototype into a final application. Certain built-in layout types are predefined to provide the application's builder with tools to do text-editing, program interfaces, control buttons, and simple animated graphics. Applications using only predefined layouts may be created with no programming whatsoever. More complex applications can take advantage of GLO's client interface to create user-defined layouts, define interactions between the various layouts, or define actions to take place upon a menu selection or a mouse hit. There is also support for applications that use multiple layout sets.

GLO itself is a combination of three tools. The first is a special-purpose editor designed to manipulate layout set descriptions, consisting of a graphically-depicted layout set and a series of layout descriptors. The second is a prototyping tool that knows about the predefined types and can rapidly produce a mock-up of a final application. Lastly is a program library that allows the programmer to combine the layout set with application-specific functions to create a final product.

The following document describes GLO and steps through the creation of an interactive symbolic debugger like Sun's DBXTOOL, using GLO and about 100 lines of application specific code.

2. Background

A major development effort is taking place at Carnegie-Mellon University to develop software for the "3 M" workstation (one million instructions/second, one megabyte RAM, and one million pixels)². This effort is centered around Andrew, a software system developed here by the Information Technology Center (ITC) and built on top of 4.2 BSD UNIX. The system currently runs on a variety of hardware, including Suns and Vaxstations.

The ITC has produced some very powerful tools for handling the window environment in a consistent way. In addition to window manager routines, these include a base-editor library that allows the client programmer to embed text editing functions within an application, and a layout manager library to allow an application to organize and interact with various objects (both textual and graphic) within a window³. One might consider these 'level 2' tools, over and above the 'level 1' tools of BSD Unix.

Unfortunately, while this may be a 'hacker's heaven' of sorts, it does not fill the needs of many faculty and students, who are coming from the world of PC's, Mac's and Tops 20 mainframes. Our approach to this problem at CDEC (the Center for Design of Educational Computing) is to provide

a third level of tools to the applications builder. These are designed to support easier and more immediate access to many of the level 1 and 2 tools. By simplifying the program creation process, we hope to facilitate the development of discipline-specific software by faculty and students.

3. User Interface

The GLO program has three basic modes.

3.1. Draw Mode:

This is where one specifies how layouts are to be placed in the window. The mouse may be used to divide the window into layouts, either creating new divisions or moving existing ones. In this way, a tree of parent and child layouts is built. The relative position of layouts in this window will be maintained in the both the prototype and final application.

3.2. Select Mode

In this mode, clicking the mouse inside the desired layout will select it for definition. Parent layouts (i.e. layouts that contain sub-layouts) are selected by pointing to the line separating its two children. When a layout is selected, a document describing it will appear in an editor at the bottom of the window. Document attributes will be described in the form:

<attribute name> = <value>

New layouts are given the type *init*. When the user chooses a type (by replacing the word *init* with the type name), the attributes appropriate to that type will be inserted into that document. New values may then be assigned to any attribute name.

The following layout types are currently supported:

3.2.1. Base-Editor (be)

Any number of these editor layouts can be defined. They allow the application user to edit existing documents or create new documents. The GLO programmer may also take advantage of the client interface of the base-editor to interact with the user via one or more of these layouts. File names can be wired in, or a user prompt can be provided. Attributes controlling write access, file checkpointing, keymaps, and menu items may all be specified if desired.

3.2.2. Multiple Base-editor (mbe)

Works as above, except that multiple file names may be specified. A buttons layout may be associated with this layout to allow the end user to switch between several different documents within the same layout. Any number of initial files may be specified. Files may be added or selected at any time by the client program.

3.2.3. Typescript.

A typescript is an editor interface to an executable program. The default program is the cshell, but any stdio program may be specified (with or without arguments). There is also a facility for passing the application's arguments to the program running in the typescript. The client program may define filters to modify user IO to the program.

The above three layout types are all based on the Andrew Base-Editor and may have Menu-Map attributes associated with them. The MenuMap attributes have the form

MenuMap=<MenuHeader,>[Prompt]:[Function]<:KeyMap>

and any number of them may be assigned to each layout. These commands will allow the GLO programmer to customize the actions of a layout by creating menu prompts on the deck-of-cards menus, and associating these prompts with programmer-defined functions. The optional KeyMap field may be used to indicate what key-bindings should be associated with the menu action. The optional MenuHeader field may be used to indicate which menu card is to contain the prompt.

3.2.4. Buttons.

Here a set of strings may be defined which will translate into labeled control buttons to occupy the layout. In the running application, a mouse hit on one of these buttons will call a programmer defined routine specified in an attribute. Multiple buttons layouts with different controlling routines may be defined.

3.2.5. Fad.

Fad (for Frame Animation Drawing), is a drawing editor designed for animation of simple line drawings and icons. The 'artist' works by creating a 'frame' of lines and icons using the mouse. This frame may then be copied to another frame and the lines and icons moved. The program can then animate these frames by interpolating between the corresponding points. Since frames may also be displayed individually, and display may be controlled by user routines driven by other layouts, fad is a useful tool for providing illustrations. Multiple fad layouts are available.

3.2.6. User-defined

By specifying the name of routines to be called for layout redraw and mouse input, the programmer can completely define the action of any number of layouts.

3.2.7. Other layout attributes

In addition to the type-specific attributes described above, all layouts are given a unique name which client programs can use to access and alter the layout. Like other attributes, this name can be set as desired. Box and border attributes may be set to put an n-pixel black box or white border around a layout.

3.3. Prototype Mode

When all the layouts are set up to the user's satisfaction, the next step is to enter prototype mode. At this point, several things happen:

(Note: *name* represents the name of the GLO application, specified as the main argument to GLO)

1. A file (*name.GLO*) is created containing the layout descriptions. This file is interpreted by both GLO and GLO applications and contains all the information necessary to create a layout set. It is the only file read by GLO when GLO is used to modify an existing GLO application.

2. A second file (*name table.c*) is created containing information that must be compiled into the final application. This is basically just a table relating the various layout names with their associated function and keymap pointers.

3. If the -k option was specified, keyboard (.kbd) files will be written to indicate the relationship between menu hits and function calls specified in the MenuMap definitions.

4. If the -m option was specified, a makefile (makefile.*name*) will be written that is capable of compiling the C files, the keyboard files, and the necessary libraries into the final application. It assumes that user defined functions will be contained in the file *name.c*.

Once these files are written, GLO makes layout manager calls to display a prototype of the final application. This prototype will contain working versions of the predefined layouts, along with dummy layouts where the user defined layouts will be.

This prototype can be used to test out the feel of the layout set in windows of various sizes. It can also be used to create the editor files and Fad diagrams.

4. Client Interface

As stated above, the GLO programmer may not need to write any C code at all. A default main() function is even included in the GLO function library. Between GLO, the base-editor, and the layout manager however, there are quite a number of client functions that may be called to facilitate building complex applications.

The base-editor provides facilities for examining and editing documents while GLO provides

some higher level interfaces to these functions. The function *glo_TellTypescript*, for example, will insert a string into the typescript view and pass the line on the the program running in the typescript as if it were typed in by the user.

In general, the programming model that is best supported is one where an application program starts up in some steady state, and actions are initialized by menu hits, mouse clicks, or keyboard entry.

5. Multiple Layout Sets

A facility is provided to allow an application to deal with a series of layout sets. Each layout set is created separately with GLO and tested. A separate program (glocombine) exists to combine GLO table files and makefiles to produce an application that can support multiple calls to *glo_init*. This function can create or restore the layout trees described in the .glo files. The makefile produced is also capable of making any of the applications individually, so one doesn't need to deal with multiple makefiles. These 'super-applications' may also be combined or added to with additional calls to glocombine.

6. GLO-DBXTOOL, a sample application.

GLO-DBXTOOL is a window based interface to the DBX debugger⁴, functionally similar to the DBXTOOL program released by Sun Microsystems.⁵ By using Andrew and GLO I was able to capture much of the functionality of the Sun DBXTOOL by writing only about 100 lines of application-specific C code. In doing so, I also created a machine-independent application that readily recompiles on any Andrew machine. Unlike Sun, I made no changes to DBX itself, and I made a few simplifying assumptions (i.e. that the most relevant DBX files will be the .c and .h files in the current directory). None the less, the resulting program has proven to be useful both on its own merits and as a demonstration of the power of GLO. It also demonstrates a general facility for creating window and mouse interfaces to glass tty programs.

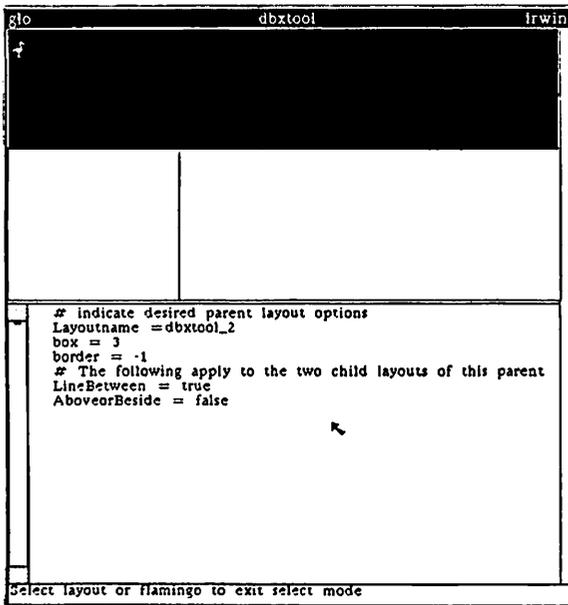


Fig. 5: This shows the attributes of a parent layout. By setting the box attribute, we put a 3 pixel box around the DBX and button layouts.

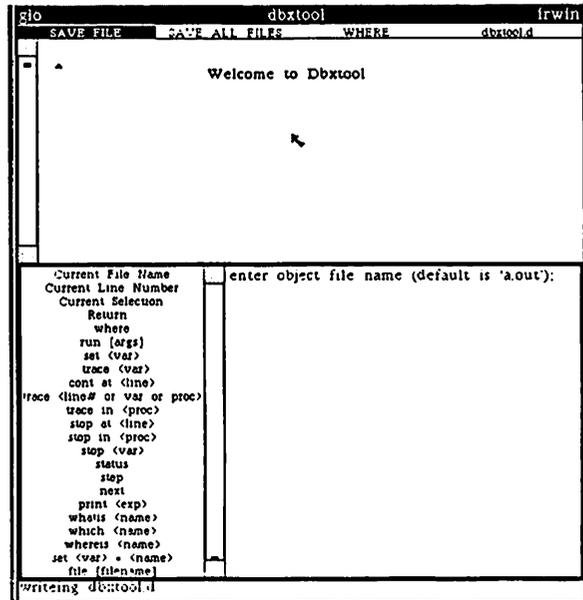


Fig. 6: Here we have entered prototype mode. The editor, the buttons, and DBX are all functional though not interacting with each other.

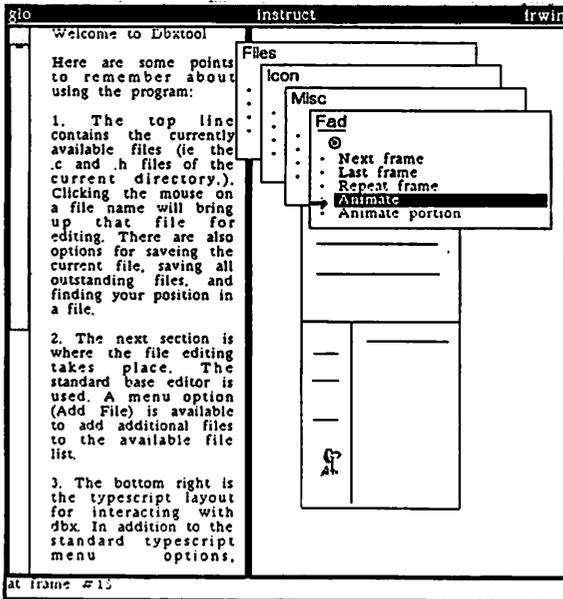


Fig. 7: GLO has been reinitialized to create the instruction layout set. We are in prototype mode to enter the documents text and set up the fad animations.

6.1. Completing the application

Combining these layout sets is done simply with the one command:

```
glocombine -o glodbxtool dbxtool instruct
```

Once the functions are written in the files dbxtool.c and instruct.c, the command "make -f makefile.glodbxtool glodbxtool" will produce the final application. The C code for dbxtool.c may be

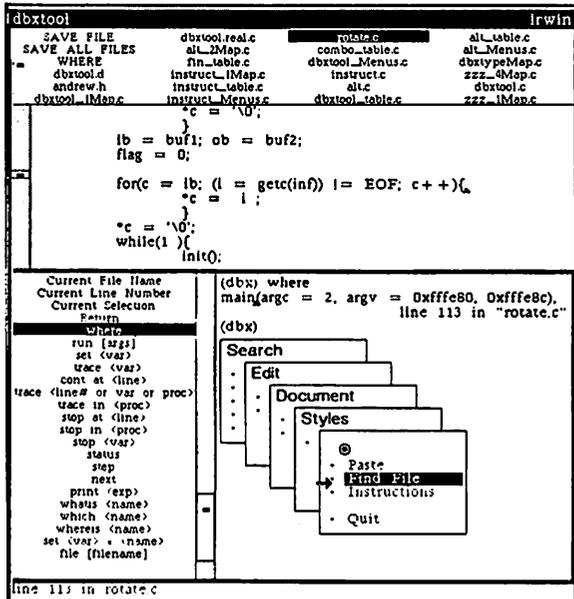


Fig. 8: The completed DBXTOOL in use.

found in the appendix.

6.2. Running GLO-DBXTOOL

The final application is called just as DBX would be called.

```
dbxtool [options] [object file] [core file]
```

The application calls `glo_init` to bring up all the functioning layouts. `Glo_init` in turn calls the application's `getfiles` function which adds the `.c` and `.h` files of the current directory to the mbe buttons layout. `Glo-init` also starts DBX running in the typescript layout and brings up the glo-defined buttons layout. At this point the user has a number of options.

Choosing the *Instruct* menu option will bring up the alternate layout set to instruct the user in the use of DBXTOOL. Instructions will be displayed in an editor window while a Fad animation provides illustrations. State information of old layout sets is saved when a new one is initialized. Thus the menu option *return to dbxtool* will return the user to the same place he left off.

To edit one of the default files, the user need only click on the button with that file name. Other mbe buttons will save the current file, save all mbe files that have been modified, or tell the user what file is being edited.

The user may use the keyboard to interact with DBX in the typescript layout. Alternately, clicking the mouse on one of the DBX-command buttons will type that command to DBX and clicking on one of the buttons prefixed with the word *Current* will enter information regarding the the file being edited. Thus DBX commands may be built up and entered without touching the keyboard. There are also a number of functions built into the typescript layout that facilitate ease of use. For example, there are key-strokes for repeating or editing previously entered commands, and menu options for accessing a cut-and-paste buffer common to all layouts.

The *locate* menu option will attempt to parse the current DBX line for a file name and location. Finding this, it will bring up that file in the editor layout and position the cursor at the specified line. See *Fig. 8*.

Note that while certain GLO information must correspond to what `dbxtool.c` expects (the format of the file information buttons for example), other information is more flexible. Thus, if the application's developer decides that a smaller set of DBX commands might make the program easier to use, or that the DBX buttons should go on top of the DBX layout, these changes can be made simply by using GLO to modify the `dbxtool.glo` description. The application does not need to be recompiled.

7. Disadvantages of GLO

GLO applications tend to be very large, using sizable portions of virtual memory. There can be difficulty in debugging applications where control flow is passed around between various tools of which the end programmer has little knowledge. And as with all high level tools, it is sometimes not clear how to perform some low level tasks that were not anticipated by the tool's designer.

One should note however, that these problems were already apparent in applications using only the tools that GLO was built on top of. The bottom line is that the amount of functionality provided seems to make the trade-offs reasonable for many applications.

8. Future

The Andrew Base-editor and Layout Manager are in the process of undergoing a complete rewrite and GLO will have to be rewritten to accommodate them. A major addition will be a dynamic linking facility which should further facilitate fast and flexible application development. Another tool to be incorporated into GLO is C-MuTutor, an Andrew implementation of the Micro-Tutor language⁶ developed for the PLATO project at the University of Illinois. C-MuTutor is an incrementally compiled authoring language, designed for writing educational materials. Within the context of Andrew, C-MU Tutor is proving to be a very useful tool for handling complex graphic displays, as well as various types of interactive computer-based instruction. It is hoped its addition

to GLO will produce a 'Swiss army knife' programming tool that will have something for most every application.

9. Conclusions

The goal of GLO was to create an environment with which faculty and students could create useful windowing applications with a minimum of Unix and Andrew experience. What was produced is a tool that supports several models of applications building.

The first model is a programmer given a project to develop on his own. With GLO as a tool-box, he can plan out where he wants to go before he gets there. A structured style develops that encourages separation of function and user-interface.

Similarly, a model where one or more program designers can prototype an application and experiment with interfaces before turning the result over to an experienced programmer is also supported.

Lastly there is the case where a program has been (or is being) developed for a glass tty environment. As in GLO-DBXTOOL, GLO supports interfaces to allow the application to take advantage of the windowing environment, without becoming useless when only a tty is available. There is also the possibility of running something like a lisp or awk application in the typescript layout and using the output to drive a graphic simulation.

In all cases, the use of GLO provides a fast prototyping tool that isn't just thrown away when it comes time to do the real application.

10. Acknowledgements

Thanks to Bruce Sherwood, David Trowbridge and Jill Larkin for their support and suggestions regarding GLO. Special thanks to James Gosling, Fred Hansen, Bruce Lucas, Andrew Palay, David Rosenthal, and all the ITC upon whose work GLO is based.

11. References

1. James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H Howard, David S. H. Rosenthal, F. Donelson Smith, "Andrew: A Distributed Personal Computing Environment", *Communications of the ACM*, in press.
2. The Task Force for the Future of Computing, Alan Newell (Chairman). The Future of Computing at Carnegie-Mellon University. Available from author.
3. James Gosling and David Rosenthal, *The User Interface Toolkit*, Proceedings PROTEXT 1 Conference, 1984.
4. "dbx(1)", *Unix Programmer's Manual - 4th Berkley Distribution*, July 18, 1983.
5. Evan Adams and Steven S. Muchnick, *DBXTOOL, A Window-Based Symbolic Debugger for Sun Workstations*, Proceedings USENIX Users Group Conference, Summer 1985.
6. B.A. Sherwood and J.N. Sherwood, *The MicroTutor Language*. Stipes Publishing Co., Champaign Il, 1985.

12. Appendix: dbxtool.c

```
#include "andrew.h"
#define DBXTOOL "/itc/itc/tpn/glo/test/dbxtool"
#define INSTRUCT "/itc/itc/tpn/glo/test/instruct"

struct view *
glo_findview(), *typescriptview;
char *glo_mbewhere();

main(argc, argv)
int argc;
char *argv[];
{
    StartTool(argv[0]); /* initialize the base editor and layout
                        * manager */
    glo_ForwardArgs(argc, argv); /* forward arguments to the program
                                * running in the Glo typescript */
    glo_init(DBXTOOL); /* create and initialize the layout tree
                       * for dbxtool */
    while (TRUE)
        Interact(); /* loop forever while interacting with the
                   * user */
}

typeinit(a, b, v)
struct layout *a;
struct view *v;
{
    /* Called by glo on initialization of the
     * typescript layout. */
    typescriptview = v;
}

instruct()
{
    /* initializes the instruction layout */
    glo_init(INSTRUCT);
}

findlinecom()
{
    /* parse a dbx line for file name and line
     * number if found, place file in mbe
     * window */
    register char *l, *c, *filenm;
    char *getviewline();
    int lnm = -1;
    filenm = NULL;
    l = getviewline(typescriptview);
    for (c = l; *c != ' '; c++) {
        if (*c == 'l' && strncmp(c, "line ", 5) == 0) {
            c += 5;
            sscanf(c, "%d", &lnm);
        }
        else if (*c == '"') {
            c++;
            filenm = c;
            while (*c != '"' && *c != ' ')

```

```
        c++;
        if (*c == ' ')
            break;
        *c = ' ';
    }
}
if (lnm < 0 || filenm == NULL) {
    TellUser("Insufficient data on line");
    return;
}
glo_mbeinfile(filenm, lnm);          /* Displays the file in the mbe
                                     * layout */
}

getfiles()
{
    /* add all relevant names in current
       * directory to the mbe file list */
    DIR      *dirpt, *opendir();
    struct direct *readdir(), *dt;
    char      *malloc();
    char      *dirname = ".";
    if ((dirpt = opendir(dirname)) == NULL)
        return (NULL);
    for (dt = readdir(dirpt); dt != NULL; dt = readdir(dirpt)) {
        if (issource(dt->d_name)) {
            glo_MbeAddFile(dt->d_name);
        }
    }
    closedir(dirpt);
    return;
}

issource(s)
char      *s;
{
    /* return TRUE if string is a .c or .h
       * file */
    register char *c;
    char      *rindex();
    return ((c = rindex(s, '.'))
            && ((*++c == 'c' || *c == 'h') && *++c == ' '));
}
char      *
getcurrent(c)
    register char *c;
{
    /* returns information about the mbe file */
    int      i, j;
    static char buf[256];
    struct view *v, *glo_mbe_view();
    struct document *d;
    switch (*c) {
    case 'L':
        /* return current line number */
        if ((c = glo_mbewhere(&i)) == NULL)
            return (NULL);
        sprintf(buf, "%d", i);
        break;
    }
```

```
case 'F':                                /* return current file name */
    if ((c = glo_mbewhere(&i)) == NULL)
        return (NULL);
    sprintf(buf, "%s", c);
    break;
case 'S':                                /* return current editor selection */
    if ((v = glo_mbe_view()) == NULL)
        return (NULL);
    i = v->dot.pos;
    d = v->document;
    if ((j = v->dot.len) == 0)
        return (NULL);
    for (c = buf; j--; i++)
        *c++ = CharAt(d, i);
    *c = ' ';
    break;
default:
    return (NULL);
}
return (buf);
}

dbxbutton(i, c, mask)
    register char *c;
{
    /* Insert the buttons string in the
     * typescript at the current carrot
     * position */
    if (*c == 'C' && strcmp(c, "Current ", 8) == 0) {
        if ((c = getcurrent(c + 8)) != NULL)
            FencedInsertString(typescriptview, c, strlen(c));
    }
    else if (*c == 'R' && strcmp(c, "Return ", 7) == 0)
        TypescriptReturnCommand(typescriptview);
    else {
        while (*c != ' ' && *c != '[' && *c != '<')
            FencedInsertString(typescriptview, c++, 1);
        if (*c == ' ')
            TypescriptReturnCommand(typescriptview);
    }
    return 1 << i;
}

addfile()
{
    /* Prompts for and adds a new file to the
     * mbe layout file list */
    register char *c;
    if ((c = AskUser("", "File Name? ")) != NULL)
        glo_MbeAddFile(c);
}
```

WINDOWS IN THE HOSPITAL

or

A WORKSTATION-BASED INPATIENT CLINICAL INFORMATION SYSTEM

in the JOHNS HOPKINS HOSPITAL

Stephen N. Kahane†, Stephen G. Tolchin†, Marvin J. Schneider**,*

Debra W. Richmond, Patrick Barta,†, Margaret K. Ardolino,

and Howard S. Goldberg††

The Johns Hopkins Hospital

(† and the Johns Hopkins University School of Medicine)

(* and the Applied Physics Laboratory)

(** and SOFTA Technologies, Inc.)

(†† and the Albert Einstein College of Medicine)

ABSTRACT

The Johns Hopkins Hospital (JHH) is developing a new, comprehensive clinical information system. This system integrates many distinct functional subsystems using a local area network. One such subsystem is a new inpatient clinical management system. The components of this inpatient system are workstations, a mini-computer and a network connecting the workstations to the Hospital Ethernet. This paper discusses the workstation component of our proposed inpatient system. The reasons for choosing workstation technology, the attributes deemed important for medical workstation development, the results of an evaluation of workstations and the plans to develop the new system are reviewed.

1. Introduction

The Johns Hopkins Hospital is a 1000 bed teaching and treatment facility located in Baltimore, MD. The Hospital currently has a collection of clinical information system components which have been developed and operated by different departments under a decentralized management structure. A pre-planned approach for the efficient sharing and transfer of data did not exist when these systems were implemented. Therefore, ad-hoc special-purpose, low-speed interfaces between systems were developed when a need existed.

Generally, duplicate data entry into the various systems occurs due to the absence of comprehensive application-level integration. This is costly and results in data inconsistencies. Furthermore, automated support for clinical functions is minimal. Therefore, in 1984, the JHH initiated development of a modern comprehensive clinical information system. A key objective of this system is to achieve functional integration of the current and future clinical systems by applying local area network technology. The Operational and Clinical Systems (OCS) Division was formed to prepare and implement the plan.

A major subsystem of the hospital-wide clinical information system is an inpatient clinical

management system that is workstation-based. The Inpatient system will support:

- admissions planning based on pre-admit notice
- inpatient care management - daily care plans, patient event scheduling, clinical data retrieval and display, generic medical information retrieval, remote order entry, clinical correlations and analyses
- access from terminal, touch-tone phone, personal computer or workstation (local and remote)
- concurrent (with hospitalization) practice monitoring and review ("concurrent review")
- discharge planning and interface to outpatient service scheduling
- production of an automated clinical resume on discharge
- patient chart abstraction and diagnosis coding to support the current reimbursement methods
- maintenance of census - support for discharge and transfer
- generation of billing information

This paper starts with a review of the current operational systems at JHH. This is followed by a description of the current methods for managing inpatient clinical information and of the architectural alternatives that are available for the development of the Inpatient system. This paper then discusses the workstation project goals, the clinical functions that need automated support, the workstation evaluation criteria, our experience developing several demonstration systems (workstation experience) and workstation project plans.

2. Review of Current Systems and Architecture

There has existed at the JHH a collection of independently developed and operated computer centers which process various, frequently duplicated clinical, financial and administrative information. Several low speed interfaces (4800 baud and less) have been established to exchange information for certain very high priority situations. The following is a description of the various computing centers which are depicted in Figure 1:

- an IBM mainframe shop consisting of a 3081 and 3083 running VM/MVS in a CICS TP monitor environment. Clinical applications include an admissions, discharge, transfer (ADT) system with many add-on functions and an Inpatient Pharmacy system. The Hospital's financial systems run on these computers. A separate Patient Identification (PID) system containing about 1.5 million records in a VSAM file runs on this computer, however most outpatient services do not have on-line access to this system, nor are other clinical systems functionally integrated with this system. Information is obtained by telephone calls to Medical Records personnel; the latter are the only individuals with on-line access.
- a Department of Laboratory Medicine (DLM) Information System which runs on three PDPP1/70 computers running InterSystems MUMPS M11+ native. Low speed interfaces connect this system with the IBM mainframe to pass admit and change information to DLM and to pass lab results back to the mainframe for display on 3278 terminals on the clinical wards.
- an Oncology Center system which runs on two PDPP1/70 computers running InterSystems MUMPS M11+ and TEDIUM. This is a sophisticated system which supports many clinical functions in the Oncology Center. It is connected by low speed lines to the DLM to transfer lab results.
- an Anesthesiology and Operating Room scheduling system which runs on a PDPP1/84 under InterSystems MUMPS and TEDIUM. This system is stand-alone.
- a VAX 11/750 computer running VMS with InterSystems M/VX MUMPS, the Wollongong Eunice UNIX emulation system for VMS and the Relational Technology, Inc. INGRES relational database management system. MUMPS is used to support the current Emergency Medicine system, which is developed in TEDIUM. The Emergency Medicine system is currently stand-alone, yet it keeps the only on-line clinical patient history in the institution.
- two PDP 11/70 computers running InterSystems MUMPS M11+ native support the Johns

Hopkins School of Medicine. These computers provide professional fee billing services and scheduling services for the Johns Hopkins Internal Medicine Associates. These systems are stand-alone.

- a VAX 11/750 running BSD 4.2 UNIX and INGRES is being developed to support the Wilmer Eye Institute. This system will be used primarily for research.
- three Pyramid 98x super-minicomputers running the OSx dual port of AT&T System V UNIX and BSD 4.2 UNIX currently support the OCS Division of the Johns Hopkins Hospital. These systems are being used for many new development projects. INGRES and both the DoD IP/TCP and Xerox XNS/SPP networking protocols run on these machines.
- several special touch-screen reporting stations which allow radiologists to compose reports and uplink them to the IBM mainframe to support the Department of Radiology. This enables radiology reports to be available on-line immediately after readings in most cases. Plans call for changing the type of reporting stations and connecting these to the Pyramid supporting the radiology information system under development. Reports would then be transferred across the network to workstations on clinical wards as well as to the mainframe (for an interim period, to allow access from 3278s currently on the clinical wards.)

Several other computer systems are also operating at the JHH for specialized purposes. None of these systems process transactions with the IBM mainframe system to obtain PID information; consequently there have been several, mutually inconsistent PID files extant. Also, functional interfaces to support a wide range of clinical and administrative needs do not exist.

An extensive Ethernet has been installed to connect these computing centers. IP/TCP and XNS/SPP protocols are used on the network. The IBM mainframes have been connected to the Ethernet using an AUSCOM 8911A block multiplexed channel adapter. This device supports the XNS/SPP protocols offloaded from the host. The Sun Microsystems Remote Procedure Call (RPC) and External Data Representation (XDR) (for integer and string types) protocols have been implemented in MVS/CICS on the IBM and over XNS/SPP on the Pyramids to enable high speed transaction processing across the UNIX and MVS/CICS environments.

The OCS Division is implementing several new systems under UNIX on Pyramid 98x machines using INGRES. These include:

- the Long Term Database (LTDB) which is accessible from all the computers across the network. It contains records on approximately 1.5 million patients, including patient identification (PID) and demographics, clinical encounter summary and some financial information. This replaces the mainframe PID system. The mainframe admitting system accesses this database using RPCs. Synchronization of replicated data is handled by an RPC-based commit protocol.
- a new Radiology Department information system, integrated into the network for scheduling, film tracking, on-line report access, resource management and other functions. A network of personal computers (MS-DOS based) connected by a 3Com Ethernet that serves the Department will also be integrated with the system.
- a new Emergency Medicine system for complete on-line support with special focus on clinical needs and urgency of care as well as administrative and financial functions.
- an Outpatient clinical information system to support many of the outpatient clinics at JHH. Initially, patient identification, registration, and appointing will be supported, but the major long-term focus includes clinical support.
- a new Inpatient system which will itself be a distributed subsystem using workstations on the clinical wards. The workstations will be connected to the hospital network and will be closely coupled with a super-minicomputer holding a complete copy of the inpatient clinical database. OCS has several workstations which it is evaluating for the development of the inpatient clinical system. These include Xerox 6085 and 8010 equipment running the Xerox Development Environment (XDE) and Viewpoint environments, Sun Microsystems equipment running BSD 4.2 UNIX, and AT&T UNIX PCs running UNIX System 5.2.

3. The Inpatient Information System

3.1. Current Method of Inpatient Information Management

On the clinical wards at JHH, patient information management is semi-automated. While requests for services such as blood tests, radiology studies, and consultations are processed manually, a subset of inpatient test result reports are available on-line.

At our institution, manual methods have been shown to have detrimental effects (Blum, 1983) in that they are associated with high transcription error rates, high communication error rates, high staff requirements and prolonged request processing times. In addition, a variety of problems limit the on-line system's (result reporting) clinical usefulness. Results are kept on-line for only three days. The login procedure is awkward and data access paths are hierarchical and cumbersome. Displays are static and display formats are not uniform. Only a single test result (or part of a test result) may be displayed at any given time. There is no support for (generic) medical information retrieval, and there are no automated (implicit or explicit) decision support tools available. There are no facilities to support document preparation. With rare exceptions, data collection is via secondary computer data entry or via ad hoc dictation and transcription.

3.2. System Requirements and Architectural Alternatives

3.2.1. Overview

The inpatient system must communicate with all ancillary services for ordering tests and procedures, scheduling patient events, and receiving results. This requires interfaces to many systems, including: Department of Laboratory Medicine, Inpatient Pharmacy, the Radiology Information System, Admission-Transfer-Discharge, various billing systems, and the Long-Term Database. The inpatient system must maintain a database of clinical information on current and recent inpatients, including test results for the duration of the encounter. Other databases of generic medical information (eg. disease or drug information) should be available.

The inpatient system should support implicit and explicit decision support tools. An example of the former is the intelligent grouping of clinical data for display (even if components of the display come from different specimens). Behind the scene cross-checks such as drug-lab test interactions based on accepted medical protocols is another example. An example of tools providing explicit decision support is a facility to provide a list of diseases worthy of consideration given some set of patient signs and symptoms entered by the user.

The system must extend to the clinical wards and support the operation of clinicians and clerical staff in ordering, requesting, entering, retrieving, displaying, and analyzing patient clinical and administrative information. This includes generation of care plans, transmission and display of graphic data, graphing (charting) of values for trend and other analyses, and local printing on the ward. Integration with the telephone system for call reception/message logging and call generation is desired. The ability to leave notes and messages electronically and to communicate via electronic mail with other hospital areas is very desirable.

The user interface must be simple to use. There must be several display/entry devices (3 or 4) per clinical unit as well as quality hard copy output (e.g., a shared laser printer.) The system should provide the ability to tailor the functional support to the specific needs of different departments or divisions. Communications and display of simple images must be supported.

3.2.2. The Alternatives

Four architectural alternatives may be identified for the general design of the inpatient system. These are:

- 1 *centralized*. This approach provides highly centralized logic which is shared by all users. Users are served by ordinary CRT terminals and, where required, by graphics terminals. This approach would use a fairly large central computer to handle a heavy interactive and computational load.

- 2 *clustered*. This distributed approach provides shared logic in clusters, with each cluster supporting a collection of user communities (several clinical wards.) Users are served by ordinary CRT terminals and, where required, by graphics terminals. This approach could use several supermicrocomputers networked together.
- 3 *highly distributed - diskless*. This approach distributes logic onto the clinical ward. Each user's computer support consists of a bit-mapped display and intelligent device with local memory. A server for hard disk support and paging across the network would be needed for every dozen or so workstations. The database support would be provided by either a distributed database across these servers, or by a single "central" networked database server, or by a combination of both schemes. The central database server could be both a backup to the other servers as well as a single location for access by CRT terminal users. Sun Microsystems' and Apollo's diskless workstations and servers are representative of this model.
- 4 *highly distributed - local disk*. This approach enables each workstation to operate from a local disk on which the operating system and possibly a portion of the database would reside. A "centralized" processor would network with these devices; both would also connect to the JHH Ethernet. This processor would contain the backup to the databases kept in all the clinical workstations to provide reliability and to protect against data loss and perhaps to perform computationally intensive tasks for the workstations. Remote network access to servers such as databases would be provided. In this scheme, the workstations provide customized support for each clinical unit and can handle many local management tasks for the ward. The "central" processor would support a regular CRT terminal community for many functions not requiring workstations. Xerox 6085 and Sun workstations with local disk are representative of this model.

3.3. Architecture Discussion

The first alternative is modelled after the classical DP center, but fails to satisfy all of the requirements and is not particularly cost effective. A networked graphics terminal on each clinical ward would cost almost as much as a workstation. It would place a very heavy I/O load on the central machine. The central machine would be a single point of failure for the system. Also, the central machine would be used for user display management and other programs as well as database access and would therefore require considerable computing capacity. Response times probably would not be as good as if data were stored local to the user on a clinical ward dedicated device and would probably not be better than remote access to a network database server from the workstations which would offload display management and user "front-end" support.

The second alternative adds complexity without satisfying the graphics and cost objectives any better than the first alternative. Perhaps 15 supermicros would be needed with each supporting 3 or 4 clinical wards. This would be expensive and would impose technical difficulties. For example, which supermicro should Lab send a result to?

The third alternative allows a collection of clinical wards to be supported by a single server remote from the workstation location. Each such collection forms a distinct subsystem. Data replication is somewhat reduced as is overall system reliability. The diskless workstations cannot operate stand-alone, thus a different approach is needed to place workstations in physicians' offices. However, this alternative does permit creation of the desired user interface on the workstation.

The fourth alternative is a hybrid solution which provides both a unique central database server, local data storage as necessary and special capabilities in the workstation for creating sophisticated user applications. Modest central computing capacity is required since display management and user "front-end" and other processing are off-loaded from any central computer. The workstations can access the "central" database system, as well as any other server system on the JHH network (this would include special AI-based expert system servers, library reference systems, as well as databases and applications on other systems.) The workstations provide improved reliability (no single point of failure), and scaling (easy to add another workstation).

The distributed approach is the solution adopted for the JHH inpatient system architecture (see Figure 2). We are evaluating vendor products and the tradeoffs between local hard disk vs.

subsystems of workstations sharing a server.

The model for distributed computing on which the system design is based is the *client - server* model. Remote procedure calls across heterogeneous systems are used to allow processes on one computer (e.g., workstation) to invoke procedures on remote machines by executing what appears to be a local subroutine call. This is done in a way that isolates applications programmers from the details and complexity of network mechanisms. We intend to use this model to allow processors and workstations to access remote databases, expert system servers and computational servers in a uniform manner. Similarly, commitment of updates to replicated data will be treated as a client application. To date, the Sun RPC and XDR protocols have been used. If we select Xerox workstations, then Courier will be implemented in the UNIX systems.

4. The Clinical Workstation

4.1. Introduction

The Clinical Workstation (CWS) is the tool that makes a complicated distributed system understandable to its users. It transforms medical information from many different sources into a form familiar and useful to health care personnel. It allows transparent access to systems that provide a host of auxiliary services. Local processing power allows sophisticated display, manipulation and analysis of this clinical information. Local processing power also facilitates the use of tools that provide a multimodal communications interface that should help with data collection and entry (Johannes and Kahane, 1985).

4.2. Goals

What is unique about the workstation is that it reflects the way health care professionals think and work; users are not constrained by conventional computer representations of information (eg. sequential, hierarchical, single-tasking). With these things in mind the specific goals of the CWS Project may be summarized as follows:

- to build a clinical workstation that allows health care professionals to interact with the hospital's computing facilities while hiding the system's distributed nature;
- to employ the desktop metaphor to provide a familiar and flexible user environment;
- to provide facilities for consistent and uniform tool manipulation throughout the system (an easy to learn and easy to use interface);
- to utilize local processing power
 - in conjunction with context dependency specifications to capture and display only required clinical information,
 - in conjunction with special purpose tools (eg. voice recognition systems) to facilitate data capture,
 - in conjunction with a local relational database to structure clinical data and support ad-hoc querying,
 - in conjunction with local intelligence and the multi-windowed environment to facilitate the intelligent display of clinical information (Polister 1984, Streveler and Harrison 1985),
 - in conjunction with medical information retrieval and decision support tools to provide an environment that helps optimize the quality and efficiency of clinical care delivery;
- To build in the hooks necessary to support communication with all other health care delivery sites. (This paper discusses only the inpatient system.)

4.3. Functions

Interfaces for the physician, nurse, clerk, and other allied health professionals are being developed and will exist in parallel. Though staff functions vary greatly, it is possible to categorize workstation functions into the following groups:

- Request Entry with Automated "Backend Processing" - Examples of facilities that will be supported include remote request for blood studies, special testing, consultations, and therapeutic interventions.
- Clinical Data Capture - Real time data collection in clinical settings has proven to be a difficult problem. Because of this, applications have relied on secondary computer data entry or ad-hoc dictation for data capture and recording. Both of these methods have significant drawbacks including the requirement for extra staff, the increased time elapsed between event and computer data capture, the errors incurred secondary to the interposition of another human process, and certainly in the case of dictation, unstructured data collection. The use of the computer as a tool for clinical data collection has been delayed because of problems with the interface and because of a lack of incentive for use. To address these long standing problems, we are studying the effects of using graphics, icons, pointing devices and voice recognition technology as tools for improving data collection and documentation for a group of procedures that report on the visual inspection of a portion of the human anatomy. We hope to develop systems that structure data collection without limiting data content flexibility. The availability of an integrated set of tools to support the practice of clinical medicine will provide the incentive to use our system.
- Clinical Information Display - Display of clinical information supporting concepts of logical grouping, reduction/emphasis, and segregation/transformation techniques will provide implicit decision support for health care professionals. Tools for explicit manipulation of clinical information will be provided as well. Demographic information will be available since there are times that such information is clinically important. An on-line patient schedule will be maintained as will the problem list and medication list.
- Medical Information Retrieval - Methods to access databases of clinical information and databases of medical literature citations will be supported. Several of the databases will be supported locally (we need laser disk technology or something comparable), while others will reside at locations outside of the Johns Hopkins Medical Institution. Efficient and easy to use interfaces will be developed where such interfaces do not already exist.
- Clinical Decision Support Tools - Conventional approaches as well as non-conventional approaches will be supported. Much of the support will be implicit (eg. drug-drug or drug-lab test interaction checking, dynamic formatting of data display, ...). Facilities for explicit use of certain tools will be supported as well. Examples of the latter include tools to support building decision trees, doing risk analyses, doing threshold analyses, doing cost-effectiveness analyses, and doing sensitivity analysis. As mentioned, tools utilizing artificial intelligence techniques will also be supported. Some tools will probably use a combination of these conventional and non-conventional approaches.
- Patient Care Plan Development - Certain databases or file browsers will contain information helpful in the construction of patient care plans. The documents will be created with information obtained from a variety of sources including:
 - patient-nonspecific information obtained from the databases described above
 - patient-specific information obtained from the on-line patient database
 - patient-specific information obtained from the individual constructing the plan

The information will be pertinent only for the patient for which it is constructed. The multi-windowed environment will help facilitate the construction of these patient-specific care plans. Related to inpatient stays, care plans will be constructed to help with critical path planning, discharge planning, and post-stay follow up care planning (discharge instructions with scheduled and yet unscheduled follow up visits).

- **Communication Capabilities** - Electronic communication facilities will support free-text mail and forms transmission. Users will have facilities to construct their own forms as well (eg. history and physical, daily rounds).

4.4. Workstation Evaluation Criteria

The workstation should be based on models of professional / machine interfaces which have been proven effective in research and actual use. Sufficient resolution to present multiple active windows and subwindows of text, graphic, tabular and form-based information must exist. Windows should be scrollable, movable, and sizable. Icons, pop-up menus, pointing devices, and subwindows should be supported. Sufficiently powerful and fast local intelligence to provide memory for the bit-mapped screen capability, simultaneous active windows with good response times, and hard disk management must exist. Functional integration of communications capabilities into the JHH Network must be provided. The ability to emulate multiple terminal types, including VT-100 and 3278 is required. Support for additional dumb terminals and a local, inexpensive laser printer is desirable. A software development environment should provide an interface to all tools, to the window manager, the pointing device and to certain application programs.

One check list of criteria being used at JHH to aid in the selection of the workstation is shown in Table 1. Issues related to the system's price, performance, communication capability, user interface, application development environment, and physical characteristics are being studied.

4.5. User Interface

4.5.1. Introduction

As mentioned, the user interface is extremely important to the goals of clinical information use and capture. Systems must be capable of displaying a complex collection of integrated clinical information in a way that is easy to use and to understand. The presentation of complex medical data in a useful, informative format leading to rapid, consistent clinical decisions is similar to the presentation of a collection of complex business information to management. Computer scientists have recognized this problem for some time and have been developing "executive workstations" as a solution. Our concept of the clinical workstation is analogous to this.

In addition to clinical information display, the problem of clinical information capture must be addressed. Historically, the clinician - computer interface has always been a problem. The problems include: keyboard data entry, difficulties in finding time to learn to use a system, reliance on dictation and handwritten notes. The challenge is in building usable human interfaces for information capture and in building tools for structuring "free text" for database entry of clinical and billing information. The best information is that entered from the clinical source; passage through layers of clerical personnel is expensive, produces costly errors (of omission and commission) and reduces the timeliness of information.

4.5.2. Existing Technology

JHH is evaluating technology suitable for constructing clinical workstations. There are three generic types of workstation product available. At the high end are very high resolution, powerful devices designed primarily for CAD/CAM/CAE applications. These devices have resolution that enables display akin to a color photograph in some cases. Examples of these systems are the MASSCOMP, Apollo and Sun high-end workstations. These systems are too expensive for use as clinical workstations at JHH, they cost from about \$10,000 to \$40,000.

At the low end are the PCs with various software products to provide windows, mouse support, etc. These PC implementations suffer from problems of single-tasking operating systems (only one active window at a time), slow response times, inadequate screen resolution, inconsistency in the interface and a non-integrated very weak subset of functionality and limited tools for new applications development.

The third class of workstations are recent products with lower cost and high performance. These products include the AT&T UNIX PC (also known as the PC 7300 or the "Safari-4"), the

Xerox 6085 Professional Computer System, lower cost workstations from Sun Microsystems and Apollo. Various other UNIX-based products with mouse and window capabilities which are expected to appear soon. Prototype clinical workstations have been developed using the Xerox and AT&T systems; work has also begun on the Sun-3.

5. Experience

5.1. The AT&T UNIX PC

5.1.1. Introduction

Our first prototype was built on the AT&T UNIX PC. As a beta test site for this machine, our organization had an interest in developing a test application which would exercise its capabilities. Also, it had the first available multi-tasking, windowing, and graphics system environment which met our price/performance requirements. Development proceeded using standard UNIX tools (C, yacc, uucp, etc.) familiar to the programming staff, plus command- and subroutine-level facilities available in the User Agent. The User Agent is the multi-window user interface provided with the UNIX PC.

5.1.2. Development of a Demonstration

In about 2 person-months of design, programming and debugging, we developed a system that demonstrated key CWS goals. The user interface that windows provide was successively refined.

Figure 3 shows what the user sees after logging in. Selecting items on this menu with the mouse allows the user to request tests, X-rays and consultations, view laboratory data or patient schedules, browse through on-line medical information, access a citation service, use decision support tools, or prescribe medications. Figure 4 shows a sample of the system as a user requests a radiology study. Pop-up windows prompt the user for information that is then used to construct a requisition in a second window. The requisition window can be reviewed and then, following some form of "electronic signature", information is conveyed to the remote site.

5.1.3. Architecture

The UNIX PC architecture was robust enough to develop a useful prototype with the important exception of adequate networking capabilities. We were able to simulate network communications using uucp to remote systems. We also implemented an ASCII subset of the Sun RPC protocol to work via the serial RS232 ports. Despite limited communications speed, this sufficed to demonstrate workstation communications in a distributed network.

5.1.4. The Development Environment

The main problems we encountered in building this prototype were deficiencies in the software development tools, in processing throughput and in screen resolution. We were handicapped by having only a set of low level tools for creating, manipulating and destroying windows and interacting with the mouse, and a much smaller set of high level tools for designing forms, menus, type fonts, icons, and customized windows. Although the low level tools (called TAM-terminal access method) were reasonably complete, they were primitive and required extensive manipulations of physical rather than the logical properties of windows. On the other hand, the high level tools directly supported the logical units that we designed into the prototype such as forms, menus and windows for displaying text. These tools were simple to use, (eg. a text file to describe the structure and content of a menu window) but, they were often inflexible.

We were astonished to find that there was no window-like equivalent to the UNIX `more` command when we wanted to display text in a window and have it scroll up and down by pointing to arrow icons in the window borders. Although another command, `uahelp`, in the UNIX PC development software solved this problem (and reformatted the text when the window changed shape, as well) it was frustrating to have to match the application to the tool (by inserting extraneous control characters into simple text files). This command also imposed unreasonable limits on the size of the

input file. Although we clearly saw that the TAM package could provide primitive building blocks for a complete windowing package, we were unwilling to develop the bulk of the necessary development tools for a windowing environment ourselves. Source code was unavailable, and we realized that it was important to have the right tools for developing window-based applications.

Many of the applications we envision involve forms, menus and windows containing text, but our access to their representation on the screen and the flexibility with which we could manipulate them with the high level tools was too restrictive in the AT&T UNIX PC environment. A significant limitation was the lack of subwindow capabilities. None of the supplied tools readily support the visual concept of tiled windows inside other windows (subwindows or "panes.") If each menu in a command sequence comes up within its own window, for example, the visual effect is often either one of clutter (if all the menus remain displayed) or similar to a flashing neon sign (if windows are alternately created and destroyed). We feel that the visual metaphor of subwindows considerably adds to the presentation of information in a logical style as was done in the "System Browser" of the Smalltalk-80 programming system, or the organization of shape and texture icons in the current microcomputer programs for freehand drawing, such as MacPaint. Also, forms with headings and scrollable text or columns need subwindows to group related items appropriately.

The processing speed of the UNIX PC was sometimes slow, especially when using shell scripts. Since the high level windowing tools are often invoked from the shell, this often led to perceptible delays in putting up the next window. If the system was loaded with 2 concurrent graphics-intensive tasks, delays in updating the screen became intolerably long. Unfortunately, the high level tools available forced us to use relatively inefficient shell scripts rather than calls to C routines. Some of the performance problems were made even worse by the very slow disk access speed.

The graphics resolution of the UNIX PC was also inadequate. The 720 x 348 pixel array (12 inch diagonal screen) proved to be too coarsely grained for our tastes. One advantage of a window-based environment is that several windows can be displayed at once. For a given screen size, the fineness of the resolution determines how many windows can be viewed without objectionable losses in clarity. The limit to how many windows that we could use in the UNIX PC was controlled not by the physical size of the windows but by the clarity of the window image. In other words, small window objects were hard to read because they were poorly resolved, not because they were too small to see.

At the end of this effort, we had a working prototype, an appreciation of what software tools were required to develop applications programs for a windowing environment and a rapidly growing shopping list of applications that we felt would be well-matched to the windowing user interface that we were designing.

5.2. The Xerox Development Environment - MESA and PILOT

5.2.1. Introduction

The Xerox Development Environment is a software development package that relies heavily on the use of icons, graphics and the mouse. The multi-windowed environment provided in XDE runs on the Xerox 8000 and 6085 series of computers. Currently, all software development is done in XDE using the MESA programming language running in the PILOT operating system. The target environment for Xerox workstations is ViewPoint, which provides a more controlled and well-defined user interface. ViewPoint, the successor to Star, provides an *office* metaphor, so users manipulate documents, folders, file drawers, in- and out-baskets, etc. on their "desktops".

5.2.2. Hardware

We initially worked on the Xerox 8010 hardware. Work has been shifted to their new, lower cost 6085 Professional Computer System which has the following noteworthy characteristics:

- a proprietary processor running at 16 MHz. (The so-called "Mesa" processor is based on the Xerox Mesa Architecture, a processor architecture designed for efficient execution of large scale modular programming systems. The architecture is also used in Xerox's 8000 and 1100 series processors.)

- a 32-bit address space
- a 697 x 880 pixel 15 inch display (80 pixels/inch) - (19 inch is optional)
- a two button optical mouse
- a 10 Mb hard disk (20, 40, or 80 Mb disks are optional)
- 1.152 Mb of main memory (expandable to 3.712 Mb)
- Ethernet interface
- a retail price of about \$5000

5.2.3. The MESA Programming Language

MESA is a highly structured and strongly-typed programming language similar to, but much more capable than PASCAL. MESA provides mechanisms for concurrent execution of multiple processes; the mechanisms are similar to those supported by the C programming language. In addition, MESA uses signals to indicate exception conditions. It is our impression that all of the problems inherent in ISO PASCAL have been worked out (eg. no variable length strings, funny evaluation of compound booleans, etc.)

It is noteworthy that three of our programmers, one experienced with PASCAL and LISP and the other two experienced with C, found MESA powerful and flexible. All three were able to write "useful" code after approximately one week of training. Several of the high-level tools, for example the form layout tool, facilitated our development of a demonstration system in less than one and one half person-months.

5.2.4. The PILOT Operating System

The operating system, PILOT, is written entirely in MESA. There is virtual memory management with process management tools and run-time support for concurrency. While there is support for multitasking, there is no support for multiple users in XDE. We will have to develop an application to handle rapid login and logoff - currently, initialization of a ViewPoint desktop requires approximately 30 seconds. JHH physicians have told us that rapid access to the system is absolutely necessary, or the physicians will not use the workstations. Issues of file ownership, access and read-write management will require some development work.

In addition, for remote procedure calls Pilot uses the Courier protocol. This may represent a problem for communication with UNIX machines. At the time of writing, it is believed that the release of BSD 4.3 UNIX will include support for Courier and SPP. We will also have to integrate Xerox mail with foreign (UNIX) mail formats.

5.2.5. Development of a Demonstration

Using the Xerox 8010 workstation and the Xerox Development Environment (XDE), we developed a demonstration system that implements a subset of the functions developed on the AT&T UNIX PC. Specifically, the following functions have been fully developed on the Xerox system and run in XDE:

- Laboratory Test Order/Entry -- A simple form is displayed from which laboratory tests (booleans) can be chosen. To signal a desire to begin the processing of a request the "Order" command is selected. For feedback to the user, a list of all the chosen tests appear in the tool's bottom subwindow. The form layout tool helped generate approximately ninety percent of the code needed to support this application, including automatic generation of pop-up menus from enumerated types.
- Laboratory Results Display -- The status (eg. pending, available, lost?) of various laboratory test results are displayed in a window. Those results which are available may be selected from this window and displayed by selecting the "Display Results" command. When test results are displayed, their names are removed from the status window (see Figure 5); when windows containing results are destroyed, test result names are redisplayed in the status window. The "Destroy" command in the status window destroys the status window only if all result windows

have been destroyed. While the form tool was helpful, we also learned a great deal about the ease with which one can handle window-to-window communication. In addition, we exercised some of the low level mouse tracking routines.

- Radiology Request/Entry -- A blank requisition form is displayed containing a header of patient demographics. From the requisition command subwindow, forms to request radiology tests can be called. Much clinical information must be collected (data input needs are high) prior to the performing of *certain* medical studies. The process of developing this application provided us with insight into approaches that allow context-dependent data collection. We wish to minimize questioning (prune a tree of questions) by using knowledge already available to the application (eg. RPC to remote database machine) and by using information collected at's runtime (eg. if the patient has no allergies to medications, don't ask which medications). We need tools to support dynamic forms and cascading menus. Xerox provides it all.

We are currently porting this demonstration to run under ViewPoint. The development of our demonstration has taught us a great deal about the toolset that might facilitate rapid prototyping of our applications.

5.2.6. The Development Environment

One of the most attractive features of the Xerox workstations is the comprehensive set of development tools. Xerox adheres to the philosophy that processes exist in a cooperative, not competing, environment. Each process is responsible for acquiring its own resources, and then releasing them when no longer needed. This cooperative approach has carried over to the set of development tools that are available. There are hundreds of tools, ranging from primitive routines to an interactive form layout tool (that generates MESA code), that can be used to speed the development process. The form layout tool is an example of a high level development tool that does not limit application flexibility. MESA code generated by the layout tool may be customized by the applications programmer. An application programmer can spend his time developing applications rather than developing the tools he needs for the application.

Following this cooperative philosophy, XDE provides an environment based on fully integrated tools that encourages rapid software development. The text editor, compiler, binder (linker), and source code debugger are tightly coupled so the edit — execution cycle is very fast.

High-level tools and low-level access coupled with powerful debugging tools, a consistent method for window-to-window and application-to-application communication, and a consistent and intuitive developer's interface make XDE an attractive development environment.

5.2.7. ViewPoint

As mentioned, while the XDE interface is consistent and flexible, it is not simple enough for application users. For this reason, Xerox provides tools that allow the development of ViewPoint applications from within XDE. The ViewPoint architecture is open and flexible. Like XDE, no assumptions are made about applications that run above it and a philosophy that the user should be in charge of the window layout at all times is encouraged.

To help support this concept, the developer is provided with tools that allow the construction of a host of intelligent window types. Terminal emulation for VT-100 and 3270 is supported. We are beta-testing a PC emulation board that facilitates the integration of applications available in the DOS environment. PC emulation can help users in the transition from PC-based systems to workstations. They have the functionality they are familiar with, plus the new capabilities of the workstation. Windows can be in an overlap or tiled mode and there is full support for tiled subwindows.

5.2.8. Summary

XDE provides a very attractive development environment to support the types of applications we have planned. High-level tools would facilitate rapid prototyping; the low-level access provided keeps the flexibility of the system high. A nineteen inch monitor is available and provides the same pixel density (80/inch). We feel that this screen size and resolution is what our system will require.

Product performance (speed and efficiency) is impressive in the XDE environment. We are in the process of assessing performance of applications running in ViewPoint. While initially the network issues (the need to support Courier and SPP) were a major concern, we are anticipating support for Courier and SPP in the BSD 4.3 UNIX release. All in all, the price/performance ratio for the 6085 is impressive; XDE and ViewPoint provide us with an attractive development environment and user environment respectively.

5.3. Sun-3 and SunView

We are anticipating the arrival of the Sun-3 and SunView for evaluation. The performance should be excellent, due in part to the MC68020; the physical characteristics (display size and resolution, optical mouse, etc.) are certainly acceptable. The communications environment (IP/TCP and NFS/XDR/RPC) fit well into our UNIX environment at the Johns Hopkins Hospital.

We need SunView to determine how well the Sun-3 will meet our development environment and user interface criteria. The documentation provided to us indicates considerable improvements over the earlier SunWindows tools.

5.4. Other Candidates

We continue to evaluate other workstations. We are talking with Apollo Computer, Inc. about present and future DOMAIN systems. We find their very highly distributed network attractive. IBM has met with us. However, they feel that at this time they cannot satisfy our price/performance requirements. Another major firm, under a non-disclosure agreement, is discussing with us a future workstation/system that may meet most of our needs.

6. Plans

The development plan for clinical workstations is organized into several phases:

- *candidate product selection*; identification of minimum hardware and software capabilities; establishment of vendor relationships for information and support;
- *development of a "demonstration" CWS*. This was done to provide a concrete example of what we are trying to accomplish so that user involvement could progress. A team of programmers, physicians, nurses and other personnel was formed to define specifically the requirements for physician, nurse and ward clerk workstation capabilities. A "pilot" CWS capability was baselined and a set of objectives to be accomplished over a longer term were identified. The demonstration system was built initially on the AT&T UNIX PC; enhanced demonstration systems were built on the Xerox Star under the XDE/MESA environment and begun on the Sun 2/120 under the UNIX operating system using various tools provided by Sun;
- *development of a "pilot" system*. This system is for actual use on a pilot clinical unit for the purposes of evaluation and validation of requirements. It is designed to implement the baseline capabilities which include integration with other hospital systems, these capabilities are a useful subset of the functions listed in the Introduction. An operational pilot is planned for second quarter 1986;
- *proliferation of a modified "pilot"*. Based on the evaluation, an enhanced and modified pilot system will be proliferated to other clinical wards for a wider range of feedback. Requirements for additional functionality will be determined and a "target" system will be developed. This work will occur during CY 1986.
- *development and full proliferation of "target" system*. The system will be expanded to all clinical wards (there are approximately 50) in the hospital.

7. Summary

Design criteria for the JHH inpatient system architecture and for the associated clinical workstations have been discussed. This system is currently under development and examples of the user interface on the potential workstations have been prototyped. Workstations will be deployed on clinical wards after pilot evaluation at selected sites. These will be networked with a

superminicomputer for database redundancy, support of functions not requiring workstation technology and cpu-intensive tasks. The supermini and workstations will be connected to the JHH Ethernet. Workstations will have connectivity with other networked computers, printers and specialized servers.

8. Acknowledgements

We wish to acknowledge the information and assistance provided by the following vendors: AT&T Information Systems, Xerox Corporation and Sun Microsystems, Inc.

9. Trademarks

UNIX is a trademark of AT&T Bell Laboratories
UNIX PC is a trademark of AT&T Information Systems
Ethernet, XNS, SPP, Courier, XDE, MESA, Star, 8010, 6085 and ViewPoint are trademarks of the Xerox Corporation
INGRES is a trademark of Relational Technology, Inc.
98x is a trademark of Pyramid Technology Corporation
VT-100 is a trademark of Digital Equipment Corporation
3270 and 3278 are trademarks of the IBM Corporation
MASSCOMP is a trademark of Massachusetts Computer Corp.
DOMAIN is a trademark of Apollo Computer Inc.
Sun-2, Sun-3, Sun Workstation, SunWindows and SunView are trademarks of Sun Microsystems, Inc.

10. References

- E. S. Bergan, et al., "Using Remote Procedure Call Protocols for a Distributed Clinical Information System", *Proceedings of UNIFORM*, February 1986
- B.I. Blum, "Information Systems at the Johns Hopkins Hospital", *Johns Hopkins APL Technical Digest* v.4, n.2, 1983.
- D. P. Connelly, et. al., "Graphical Representation of Clinical Laboratory Data", *Third Annual Symposium on Computer Applications in Medical Care*, 1980.
- R. S. Johannes, S. N. Kahane, B. Ravich, and H. P. Roth, "The Use of Voice and Graphics in an Endoscopy Data Collection System" *Proceedings of the Voice I/O Systems Application Conference*, September 1985.
- P. E. Polister, "Intelligent Display of Laboratory Data", *Eighth Annual Symposium on Computer Applications in Medical Care*, pp 402-405, November 1984.
- D. J. Streveler and P. B. Harrison, "Judging Visual Displays of Clinical Information", *MD Computing*, v.2, no.2, pp27-50, 1985.
- S.G. Tolchin, et. al., "Overview of An Architectural Approach to the development of the Johns Hopkins Hospital Distributed Clinical Information System", *Hawaii International Conference on System Sciences*, January, 1986.
- S.G. Tolchin, et. al., "Integrating Heterogeneous Systems Using Local Network Technologies and Remote Procedure Call Protocols", *Ninth Annual Symposium on Computer Applications in Medical Care*, November, 1985.
- S. G. Tolchin and W. Barta, "The Johns Hopkins Hospital Network Hawaii International Conference on System Sciences, January 1986

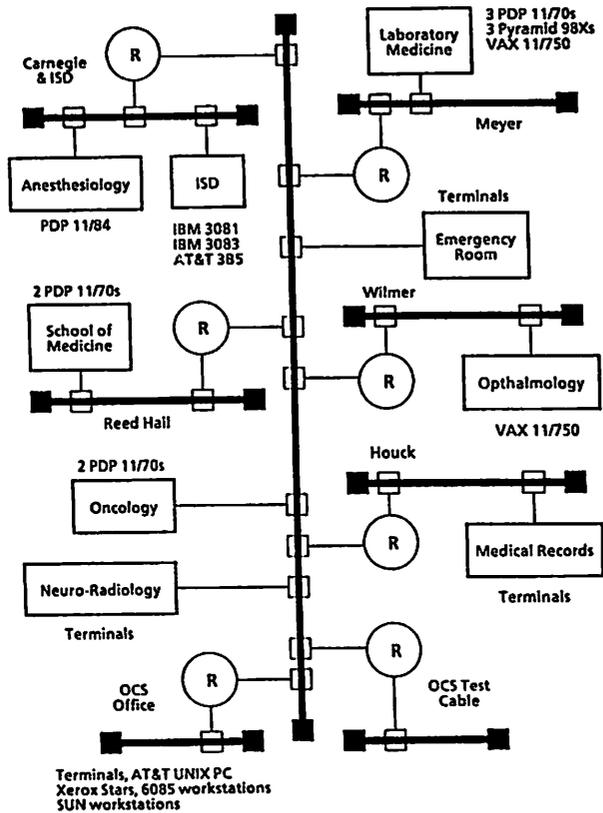


Figure 1: Johns Hopkins Ethernet Network

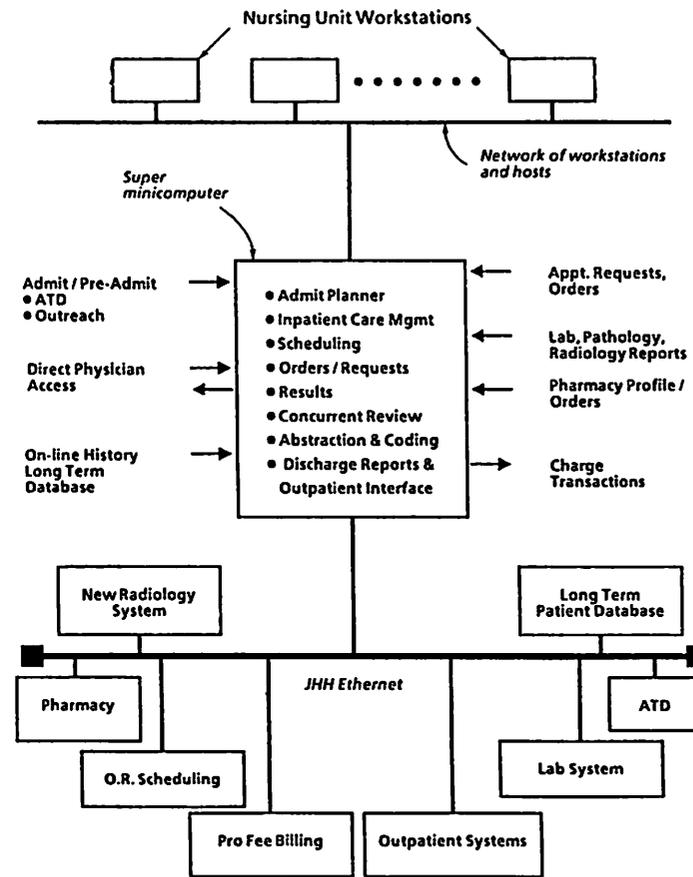


Figure 2: Architecture of the Johns Hopkins Distributed Inpatient System

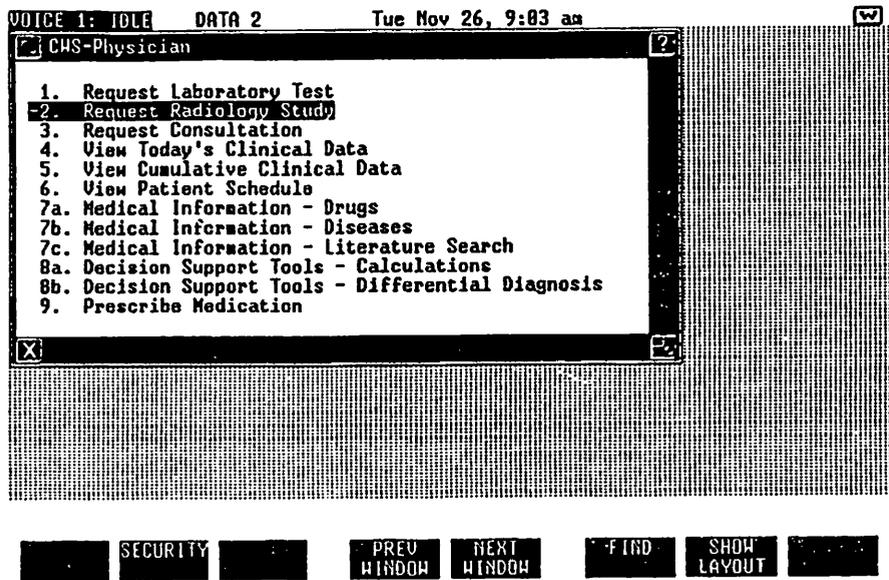


Figure 3: AT&T 7300 Screen After Login

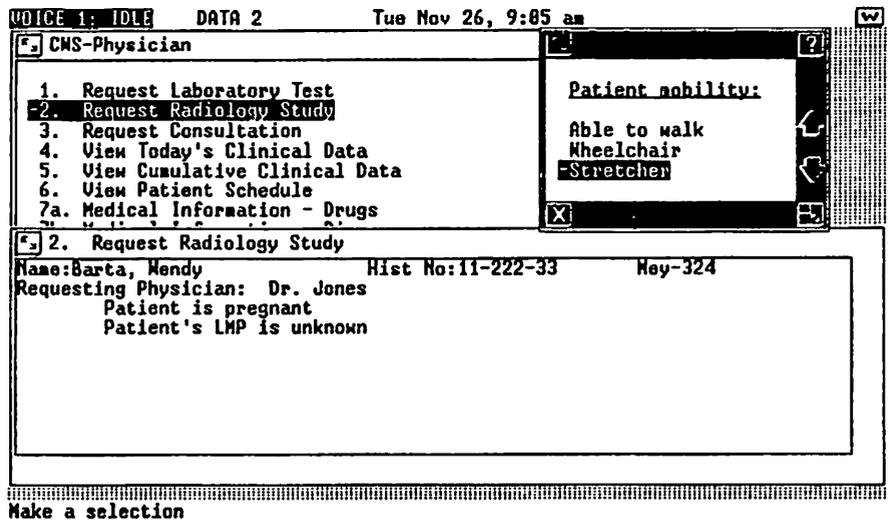


Figure 4: AT&T 7300 Screen After Selecting "Request Radiology Study"

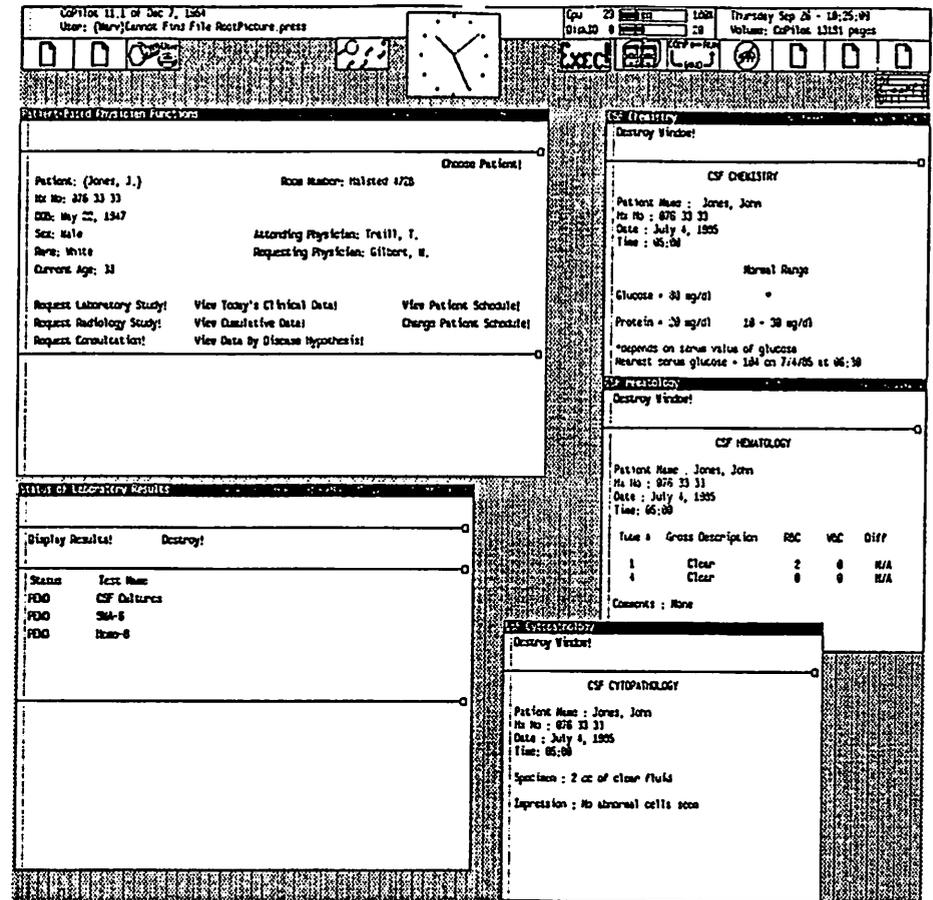


Figure 5: Xerox 8010 Screen After Selecting "View Today's Clinical Data" and Each of the Available Results

JHH CLINICAL WORKSTATION CRITERIA	
Issue	Comments
PRICE	workstation units laser printer options (eg. disks, PC emulation, etc.)
PERFORMANCE	acceptable CWS demonstration architectural design (quality, open, distributed, etc.) results from running a selection of benchmarks
COMMUNICATION	with other workstations, minis, JHH ethernet electronic mail capability with UNIX machines (uucp capability) remote access to workstations via telephone
USER INTERFACE	display size, resolution windows (multi, scrollable, movable, sizable) icons, pop-up menus, pointing devices ability to customize display (interface flexibility) context driven data capture and display emulation of terminals (VT-100, 3278) emulation of IBM-PC on the workstation support of crts hung off workstation support of remote crts
DEVELOPMENT ENVIRONMENT	array of tools for software development - edit/compile/test/debug tools - high level software design tools - tools for development of charting & graphing functions data base support source code availability or comprehensive documentation ability to interface with window manager, pointer, applications future maintenance issues training (of the developers) required how much background processing can be done effort required to create a 'foolproof' user environment ability to use standard security system tools for adhoc charting & graphing ability to send screen-print to network printer local data management artificial intelligence support (i.e. LISP)
PHYSICAL CHARACTERISTICS	reliability ruggedness ease of unit replacement power required, quietness, size
MISCELLANEOUS	solid vendor with good field support, reputation & finances product family future plans availability of sister workstations (gray scale, color)

Table 1: Workstation Issues

The Feel of Pi

*T.A. Cargill
AT&T Bell Laboratories
Murray Hill, New Jersey 07974*

ABSTRACT

Pi is an interactive debugger for C and C++ on Eighth Edition Unix[†] systems. Its user interface uses multiple windows on a DMD 5620 terminal. Pi does not feel like a debugger with a sequential command language, nor does it feel like a debugger where commands from a bitmap display are translated into a sequential command language. In contrast, Pi's multiple windows display multiple active views of its multiple subject processes, allowing the programmer to browse through a network of information. The programmer interactively wanders through a set of executing processes, probing for insight with a tool that really helps.

Each window displays a specific view of a subject process in parallel with the other windows. The contents of pop-up menus are determined by context — the current window and the line of text selected within it.

Pi is written in C++ and uses Eighth Edition's */proc* to access arbitrary live subject processes.

Introduction

Pi (Process Inspector) is an experiment in debugging with an interactive, bitmap graphics user interface. The debugging technology is conventional: breakpoints are planted in the subject process so that the states the process moves through may be examined. But the user interface is unconventional.

In a conventional debugger, the programmer inputs a sequence of commands that are interpreted by the debugger. The debugger responds with information about the subject process. Several problems arise. First, the debugger can usually accept only the subset of its commands applicable to the debugger's current state. For example, breakpoints can only be set in the current source file, or expressions can only be evaluated in the current activation record. Second, the debugger's output is passive and cannot be used to obtain further information about, or other views of, the process. For example, if a value is displayed by some command in an inappropriate format, the programmer must re-issue the command, specifying another format, or take the value and manipulate it elsewhere. The effect is that any non-trivial debugging is accomplished by combining the debugger with some of our oldest tools — pencil and paper. Third, a debugging command language must necessarily be very large, if it is to be useful. Generally, keyboard languages are complicated, and often cryptic.

The goal in writing Pi was to create a full-function interactive debugger with a good user interface: menu-driven, reactive, usable without a scratch pad or reference manual.

[†] Unix is a trademark of AT&T Bell Laboratories.

Interface Model

Pi's user interface assigns each view of a subject process to a separate window. Each window has its own menu of operations, appropriate to the view presented. Within each window are lines of text providing details of the window's view. Each line has its own menu of operations, appropriate to the information presented. Interaction is driven by the programmer selecting operations from these menus. In response to each operation, the debugger adds or removes windows, or lines, or their menus. A window or line may also choose to accept a line of input from the keyboard.

On the DMD 5620, a layer is subdivided into a set of scrolling, overlapping windows. The mechanics of the user interface are derived from Jim, a text editor by Pike[2,4]. There is a current window (with a heavy border), and within it a current line (video-inverted). Each button on the three-button mouse serves a specific role. Button number 1 is for pointing. If the cursor is outside the current window, button 1 selects a new current window. If the cursor is inside and over a line of text, that line becomes current. If inside and in the scroll zone, the window scrolls to center the proportional scroll bar over the cursor. Buttons 2 and 3 raise the pop-up menus for the current line and window, respectively. Menus also scroll and may have pop-up sub-menus, making large menus relatively easy to use.

An Example

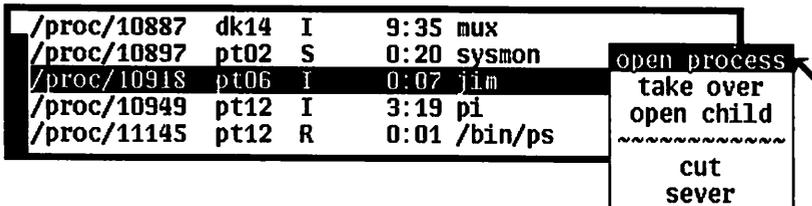
I will demonstrate Pi by examining the copy of Jim that I am using to write this paper. Jim is two processes, one in the host computer and one in the terminal. I will work with its host process. I create a new layer on my 5620's screen and simply invoke Pi:

pi

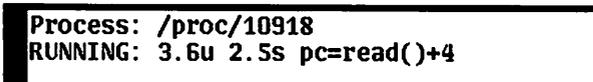
After about 20K bytes of user interface code has downloaded into the 5620, Pi's cursor icon requests me to sweep a rectangle for a new window – the "Pi" window, the master window through which Pi may be bound dynamically to processes and core dumps. I now have one (almost empty) window in Pi's layer:



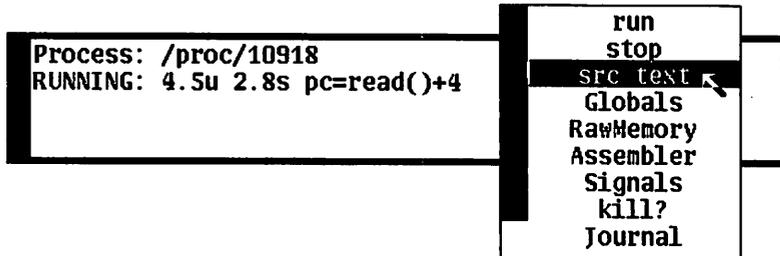
Selecting '/bin/ps' from this window's menu runs the ps command and lists the output in the window, one process per line:



It shows me with a light load – I am only editing. To examine Jim, I point to process 10918 in this list and select 'open process' from its menu. I am now requested to sweep a "Process" window. The Process window has overall control of the process and can create windows with more detailed views. The process window shows the state of the process, and a callstack if the process is stopped. The state of process 10918 is:

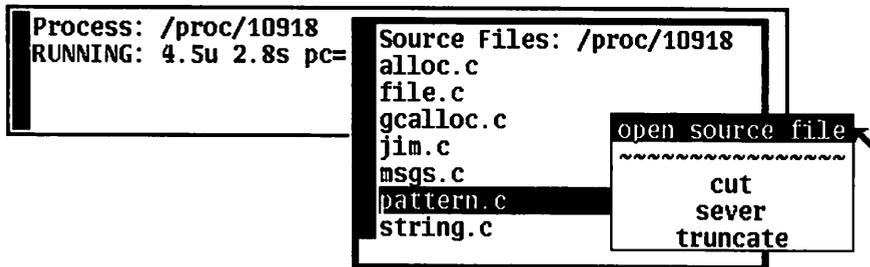


This is usual state for Jim's host process – it is blocked reading from the terminal. Pi polls the state of the process every second and updates the Process window asynchronously with respect to the user and the subject process. After some more editing the consumed processor time has increased.. I raise the Process window's menu:

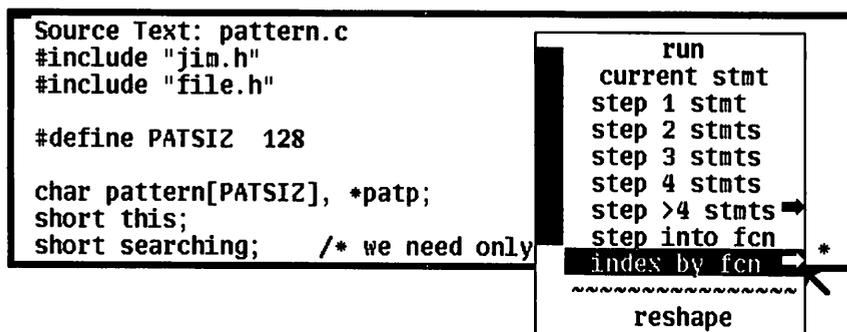


'stop' stops the process asynchronously. 'run' restarts it. 'src text' creates windows for viewing source text. 'Globals' creates a window for evaluating expressions in global scope. 'RawMemory' creates a "memory editor," in which uninterpreted memory cells may be viewed and modified. 'Assembler' creates a window that disassembles memory and provides instruction level operations. 'Signals' creates a window that monitors signals to the process. 'kill?' kills the process; the question mark calls for a confirming button hit. 'Journal' creates a window that records significant events in the process – a trace.

First, I choose to look at some source text. If there were a single source file, 'src text' would create a "Source Text" window for it. Jim has several source files; so Pi asks me to sweep a "Source Files" window that lists them:



I point to pattern.c and choose 'open source file' from its menu. I sweep a Source Text window. It fills with the first few lines of pattern.c. I raise its menu:



[I have not looked at this code before starting to write this example. I believe I will find Jim's regular expression pattern matcher here. I know no details of its implementation. It is as if I were starting from scratch to find a bug in Pike's code.]

Moving the cursor over the arrow at the right of 'index by fcn' pops up a sub-menu that is a table of contents by function (with line number) of `pattern.c`:

```

run
current stmt
step addtolist()... 394
step advpat()..... 292
step bexecute().... 533
step cclblock().... 282
step compile()..... 79
step execute().... 410
index expr()..... 205
~~~~~
feexecute().... 420
killlater()... 699
new()..... 375
newmatch().... 656
old()..... 387
optimize().... 96

```

It suggests, as I expected, that Jim compiles regular expressions into a representation from which they can be interpreted efficiently. To see some of this code, I select '`compile().....79`'. This scrolls the window so that the line with the opening brace of `compile()` is in the center:

```

int nmatch;
char *compilepat;
compile(s, save)
char *s;
{
    if(strlen(s)>=PATSI2)
        error("pattern too long\n", (char *)
            forward=1;
            startpat(compilepat=s);
            expr();
}

```

```

set bpt
trace on
cond bpt
assembler
open frame
~~~~~
cut
sever
fold

```

To set a breakpoint, I point to a line of source text, say the opening brace, and select 'set bpt' from its menu. To indicate the breakpoint, '>>>' appears at the beginning of the source line:

```

int nmatch;
char *compilepat;
compile(s, save)
char *s;
>>>{
    if(strlen(s)>=PATSI2)
        error("pattern too long\n", (char *)0);
    forward=1;
    startpat(compilepat=s);
    expr();
}

```

Note that the breakpoint was set while Jim executed asynchronously.

To force Jim to execute the breakpoint, I type (in Jim's layer) a search command whose pattern matches a non-empty sequence of 'a' followed by a non-empty sequence of 'b': `/a+b+`.

When Jim hits the breakpoint, Pi asynchronously notices its change of state and reports it in the Process window, along with as much of the callstack as fits (here, only the deepest activation record):

```
Process: /proc/10918
BREAKPOINT:
pattern.c:79 compile(s=0xDD09="a+b", save=1)
```

In the Source Text window, the breakpoint source line is selected to show the current context. To see more of the callstack I reshape the Process window, making it larger:

```
Process: /proc/10918
BREAKPOINT:
pattern.c:79 compile(s=0xDD09="a+b", save
jim.c:368+11 commands(f=0xBCAC)
jim.c:206 message()
jim.c:100+7 main(argc=0, argv=0x7FFFE55C)
```

open commands() frame
show jim.c:368
~~~~~  
cut  
sever  
fold

To see the context from which `compile()` was called, I select the `commands(f=0xBCAC)` line from the callstack and choose 'show jim.c:368' from its menu. I am prompted to sweep another Source Text window, `jim.c`, to see this context. To catch the process before it calls `execute()`, I change the selection from the line

```
compile(p, TRUE);
```

to the `if` statement four lines below and set a breakpoint:

```
break;
case '?':
case '/':
if(++p)
compile(p, TRUE);
else
dprintf("%c%s\n", c, pattern);
send(0, 0_SEARCH, 0, 0, (char *)0);
if(execute(f, searchdir=c))
moveto(f, loc1, loc2);
```

set bpt  
trace on  
cond bpt  
assembler  
open frame  
~~~~~  
cut
sever
fold

I then 'run' from the Source Text window's menu:

```
dprintf("%c%s\n", c, pattern);
send(0, 0_SEARCH, 0, 0, (char *)0);
>>> if(execute(f, searchdir=c))
moveto(f, loc1, loc2);
```

reopen jim.c
run
current stmt
step 1 stmt

When Jim reaches this breakpoint, I choose 'step into fcn' from the same menu to step the process into `execute()`. (The other source stepping commands step *over* called functions.) The source context for `execute()` is back in the first source file, `pattern.c`.

Pattern.c's Source Text window moves to the front of the screen and highlights the opening brace of execute():

<pre> } execute(f, c) File *f; int c; if(mustcompile) compile(pattern, TRUE); if(c=='/') return fexecute(f); else </pre>	<pre> run current stmt step 1 stmt step 2 stmts step 3 stmts step 4 stmts step >4 stmts step into fcn index by fcn ~~~~~ reshape move close </pre>
--	---

It appears that the real work will be done by fexecute(). I could set a breakpoint there, but I use 'step 1 stmt' from the source window's menu a few times until I get to:

```
return fexecute(f);
```

and then use 'step into fcn' again. The context shown from pattern.c changes:

<pre> return bexecute(f); } fexecute(f) register File *f; register Node *n; register long startposn=f->selloc+f->nrel; long l=length(f); register flag; register char *s; </pre>
--

fexecute() looks non-trivial. Before going further, I would like to understand the data structure driving it. I do not know what this data structure is. Looking forward through the source text of fexecute() I understand very little of the code. But three lines do make sense:

```
/* fast check for first char */
if(startchar && *s!=startchar)
  goto Continue;
```

Surely, startchar holds a literal character and s is a pointer into a scanned string. To test this I set a breakpoint on the if and 'run'. At the breakpoint I need the value of startchar. Choosing 'open frame' from the source line's menu:

<pre> >>> /* fast check for first char */ if(startchar && *s!=startchar) goto Continue; } if(flag){ nl=list[1]; tl=list[0]; </pre>	<pre> clear bpt assembler open frame ~~~~~ cut sever fold </pre>
---	--

creates a "Frame" window for the activation record of the function corresponding to the source line. A Frame window evaluates expressions with respect to its activation record. The menu contains local variables, each flagged as an argument, an automatic or a register:

pattern.c:461 fexecute(f=0xBCAC):	flag	reg
	f	arg
	i	aut
	lastwasnl	aut
	l	aut
	n	reg
	startchar	aut
	startposn	reg
	s	reg
	wrapped	aut
	registers	

Choosing startchar evaluates that expression:

pattern.c:461 fexecute(f=0xBCAC):	ca	hex	on
startchar=97	typ	unsd	dec on
	siz	sign	dec off
	fo	octal	on
	asci	on	
	float	on	
	setime	on	
	truncate		

Is that an 'a'? The value is in decimal because startchar is declared int. To override the default format, I select 'format' from the expression's menu, and 'ascii on' from the sub-menu. The expression re-displays itself:

pattern.c:461 fexecute(f=0xBCAC):
startchar='a'=97

The value of startchar looks right and probably came from the data structure I am after. Scrolling back a few lines in pattern.c I find an assignment to startchar:

```

nmatch=0;
match[0].b=0x7FFFFFFF;
match[0].e=0x7FFFFFFF;
startchar=0;
if(fstart->op<0200)
    startchar=fstart->op;
i=Fseek(f, startp)-1;
if(i<0 || f->str.s[i]=='\n')
    lastwasnl=TRUE;
Restart:

```

fstart may be the pointer I need, but it does not appear in fexecute()'s menu. It must be a global. Rather than open the global expression evaluator window and look in its menu, I enter the expression

fstart

from the keyboard, with fexecute()'s Frame window selected as the target.

The Frame window now contains two expressions:

```

pattern.c:461 fexecute(f=0xBCAC):
startchar='a'=97
fstart=0x71C0

```

```

cast $
typeof $
sizeof $
* $
$ []
$->*
$->left
$->op
$->right
format

```

What type is `fstart`? I can almost tell from its menu. Most of the entries in an expression's menu are new expressions that may be derived from it. The `$->`'s tell me that I have a pointer to a structure. (In the menu, and from the keyboard, `$` denotes the current expression.) Choosing `'typeof $'` confirms it:

```

pattern.c:461 fexecute(f=0xBCAC):
startchar='a'=97
fstart=0x71C0
typeof(fstart): *struct Node

```

```

cast $
typeof $
sizeof $
* $
$ []
$->*
$->left
$->op

```

Choosing `'$->left'`, followed by `'$->op'`, and `'$->right'` yields:

```

startchar='a'=97
fstart=0x71C0
typeof(fstart): *struct Node
fstart->right=0x71CC
fstart->op=97
fstart->left=0

```

```

cast $
ty hex on
si unsd dec on
f sign dec off
~ octal on
ascii on
sever

```

Reformatting `fstart->op` in ASCII leaves:

```

typeof(fstart): *struct Node
fstart->right=0x71CC
fstart->op='a'=97
fstart->left=0

```

```

typeof $
sizeof $
* $
$ []
$->*

```

So here is some kind of tree, where an operator code less than octal 200 is to match its own value in the scanned text. The left sub-tree is empty; the right looks promising. Dereferencing with `'* $'` yields:

```

pattern.c:461 fexecute(f=0xBCAC):
startchar='a'=97
fstart=0x71C0
typeof(fstart): *struct Node
fstart->right=0x71CC
*fstart->right={op='\202'=130,left=0x71C0,right=0x71D8}
fstart->op='a'=97
fstart->left=0

```

```

$->*
$->left
$->op
$->right
format

```

The left field of `fstart->right` is equal to `fstart` itself; maybe this is a doubly-linked list. Applying `'$->right'` to `fstart->right`, I get:

```

fstart=0x71C0
typeof(fstart): *struct Node
fstart->right=0x71CC
fstart->right->right=0x71D8
fstart->op='a'=97
fstart->left=0

```

```

cast $ →
typeof $
sizeof $
+ $
$ []
$->*

```

I already know this, but applying `'* $'` produces (showing Pi's entire layer for the first time):

<pre> Process: /proc/1 BREAKPOINT: pattern.c:461 fe pattern.c:414+10 jim.c:372+22 com jim.c:206 messag jim.c:100+ </pre>	<pre> pattern.c:461 fexecute(f=0xBCAC): startchar='a'=97 fstart=0x71C0 typeof(fstart): *struct Node fstart->right=0x71CC fstart->right->right=0x71D8 *fstart->right->right={op='b'=98, left=0, right=0x71f0} fstart->op='a'=97 fstart->left=0 </pre>
<pre> jim.c msgs.c pattern.c string.c </pre>	<pre> startchar=0; if(fstart->op<0200) startchar=fstart->op; i=Fseek(f, startp)-1; if(i<0 f->str.s[i]=='\n') lastwasl=TRUE; Restart: </pre>

Note that the value of the `op` field for the current expression is displayed in ASCII as `'b'`. The ASCII format explicitly requested for that field earlier was saved in the symbol table and is now the default. The `left` pointer is zero here. It now looks as though `left` points back to the beginning of the sub-pattern controlled by the closure operator.

Let me stop here. I have started to unravel the data structure and understand the program. I hope this paper description conveys something of the feel of Pi.

Programmer Reaction

Most programmers take somewhere from a few hours to a few days to make the transition from drowning in a sea of windows to considering Pi an indispensable tool. At the outset, they do not expect dynamic binding to subject processes and cannot see why there are so many windows. Invoking a debugger without specifying a dump or program is a foreign notion. Expectations of a debugger are very low: "I only want the value of *x* when *f()* is called — why all the windows?" With increased confidence and ambition they use Pi with more sophistication. Styles vary considerably. Each programmer uses idiosyncratic sizes, shapes and placements of windows, especially when debugging multiple processes. Some prefer to enter most of their expressions from the keyboard, others never touch it.

There are two main problems. First, binding Pi to subject processes is too complicated for novices. Experts demand many special facilities, which have been allowed to complicate what the novice encounters. Second, demand for programmable debugging is growing among the expert users. Programmability was excluded from Pi in order to concentrate on interactive behavior. Pi does have "spy" expressions, which re-display themselves if their values change, and conditional breakpoints, but it is not programmable, say, to step 10 instructions after encountering a breakpoint. It is now time to think about how programmability and interaction can be combined.

Asynchronous Multiple Processes

An arbitrary set of processes may be examined simultaneously. For each subject process there is an independent network of windows. Since all the windows are in a flat space on the screen, each successive action from the programmer may be in any window, associated with any process. Events in the set of subject processes are reported as they occur. For example, the programmer might step source statements alternately between a pair of processes while watching the changing values of spy expressions in a third process. This simplifies debugging situations that were difficult or impossible in the past. For example, it becomes straightforward to (i) compare the behavior of two similar programs; (ii) compare the effects of different inputs on a single program; (iii) observe the interaction between related processes, say child and parent.

Implementation

Pi depends on the Eighth Edition's *lproc*[1,4], and object-oriented programming in C++[3].

lproc permits Pi to bind itself dynamically to any processes, and execute asynchronously with them. For each process, Pi can tell the kernel how to handle an *exec()* by the process and signals received from other processes. A breakpoint in code executed by a child of a subject process suspends the child so that it may be opened and examined. Code sharing is managed transparently by *lproc*.

The browsing and asynchrony are driven by object-oriented programming in C++. A large host C++ program communicates with a small 5620 C program. Everything the programmer can identify on the screen is a C++ object, an instance of a class. The host program binds an object identifier (which can be thought of as a host address) and a menu of operations to each window and each line of text as it describes them to the terminal. When the programmer selects an operation from a menu associated with an object's image, the terminal sends back a remote invocation of one of the object's member functions. Generally, executing this function creates, changes or removes host objects and their images in the terminal. Host-terminal communication is asynchronous; the programmer need not wait for results to appear on the screen before issuing another operation. There is no ambiguity in this "mouse-ahead"; the identity of the object on which a menu operates is frozen when the menu is raised. A crude object registration scheme in the host detects (with high probability) and ignores operations for objects that have been destroyed.

Conclusion

Pi's easy access to information about arbitrary processes has made programmers more sophisticated in their debugging practices. Programmers working with large programs written by others are happier. Programmers who would not normally read assembly code can sometimes spot code generation bugs in the compiler. Programmers with families of interacting processes have a handle on them. In general, programmers understand their programs better.

References

1. T. Killian, "Processes as Files in Eighth Edition Unix," Proceedings of the Summer 1984 USENIX Conference, Salt Lake City, Utah
2. R. Pike, "The Blit: A Multiplexed Graphics Terminal," AT&T Bell Laboratories Technical Journal, Computing Science and Systems, October 1984
3. B. Stroustrup, "The C++ Programming Language," Addison-Wesley, 1985
4. Unix Time-Sharing System Programmer's Manual, Eighth Edition, Volume 1, AT&T Bell Laboratories, February 1985

Flamingo: Object-Oriented Abstractions for User Interface Management

Edward T. Smith and David B. Anderson

Carnegie-Mellon University

ABSTRACT

This paper describes the Flamingo User Interface System designed for use by programs running on Spice machines. Flamingo is designed to use the remote procedure call mechanism available through the various operating systems running on Spice machines to provide a flexible, robust, machine-independent interface to a variety of different machines communicating over local area networks.

Flamingo separates the abstractions of the objects used by the program to communicate with the user from the actual devices used to read or write information. A window manager is provided that makes a suitable mapping from output objects known to the program to those objects seen by the user. A user interface is provided to map input events from real devices to either window management routines or to a form suitable for input by a program.

Flamingo itself can be divided into different processes running on different machines each implementing different parts of the system. All exported objects used for communicating between users and programs are implemented with specific methods defining the operations available for those objects. New methods can be substituted for the standard ones either for an instance of a particular object or for all objects of a class in a given running Flamingo system. These mechanisms provide a flexible framework within which a variety of window managers and user interfaces can be realized and evaluated.

1. Goals

Flamingo (FLexible, Asynchronous Manager for Interactive Network Graphics Operations) is a system for building user interfaces to programs running within the Spice environment. The Spice environment¹ consists of a heterogeneous set of machines, typically large personal workstations (and a few mainframes, file servers, supercomputers, etc.), each of which supports the IPC² (inter-process communication) message passing model. This mechanism provides a transparent, language and machine independent means of communication between programs and all system resources they may need (screens, keyboards, pointing devices, file systems, other processes, and so on). A basic goal of the Flamingo effort is the creation of a system that can take advantage of this common message-passing mechanism to provide a powerful, flexible interface between programs running in the Spice environment and the human user on some Spice workstation.

A part of this goal is to address the issue of providing upward compatible user interface

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract F33615-84-K-1520.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

support to all our older software. The Spice environment currently consists of about two hundred machines representing several manufacturers. There is a large and growing base of software written on all these machines that complicates the ability to port software as new machines appear. Often in fact, some software is simply not suitable to port, since some special machine may still perform a service that other machines are not capable of performing. Thus, Flamingo should allow users to sit at any Spice machine and run programs on any other Spice machine, despite differences in the underlying input and output mechanisms of the machines being used.

A final goal, and one that has guided much of our design, has been to simplify the task of implementing and modifying user interfaces and window managers. As we surveyed the needs of the researchers in our group, we found our clients (including ourselves) wanting to be able to approach a display manager at many different levels, and to modify or replace components of the system without being required to deal with the complexity of the entire system. Our researchers need to take advantage of the Spice environment's powerful workstations, message-passing operating system, distributed file system, and large software base, but have been unable to easily modify and extend the existing user interface system to suit their particular needs. In general, buying into any particular window manager also means getting a particular style of human/computer interaction and a particular program-to-window-manager interface. We see our problem as one of wanting to define just exactly those interfaces for our various systems in such a way to support our researchers more than supporting some particular style of interaction. No particular style or set of operations has proven itself capable of answering all the desires of its users in this respect, so we set out to work on this problem for users of the Spice environment.

2. Object-Oriented Graphics Structures

The current implementation of Flamingo is written in C under Berkeley UNIX* 4.2 running on a MICROVAX.† Our version of this Unix, called *Mach*^{3,4}, has been modified at CMU to fully support the IPC mechanism in use between existing Spice machines.

To meet the criteria mentioned above, we have adopted a hierarchical, object-oriented design. The Flamingo system exports various objects related to input and output of information between the client program and the user. For input, the primary object is the *input device*, which includes input device state information plus methods for handling input events. For output (by client programs), the exported object is a *pixel array*. A pixel array (PA) is a 2-dimensional region with a shape defined by a *mask* (defined shortly), and a list of *mappings*, which map the pixel array to other PAs, to memory, or to the screen. Several *classes* of pixel arrays are provided; lower levels of the system export pixel arrays (PAs) with implementations of methods for raster operations, character drawing, and so on, while higher levels export these operations as well as window managements functions, input hooks, etc.

An object defined for use by both input and output objects is a mask. Masks are used throughout Flamingo wherever there is a need to represent shape: the shapes of pixel arrays, mouse sensitive areas, clipping regions, and the uncovered portions of overlapped windows are all represented with masks.

Each of Flamingo's graphics operations takes one or more masks as input to specify the clipping that is to be performed to the source and destination arguments. For example, our *bitblt* operation, which we call *rasterop*, has this signature:

```
mask_rasterop(srcPA, srcMask, sx, sy, x, y, dstPA, dstMask, dx, dy, op)
```

The source bits for this operation are those bits contained within the shape specified by the

* UNIX is a trademark of AT&T Bell Laboratories.

† MICROVAX is a trademark of Digital Equipment Corporation.

source mask when it is mapped to the location (sx, sy) within the source pixel array. The destination bits are determined in an analogous fashion, and the source bits are mapped to the location (x, y) within the space of the destination pixel array, where the raster operation op is performed. (One might think of the source and destination masks as 2-dimensional bitmaps that are logically ANDed into the source and destination pixel arrays, where the ones in the masks denote those bits in the source and destination that participate in the operation.)

Masks are represented internally as a list of rectangles, organized by scanline. This representation permits a compact encoding of the shapes which typically arise from overlapping rectangular windows, and also allows efficient coding of the methods for masks that we have found useful: intersection, union, difference, and the conversion of a mask to a list of rectangles. The latter is very useful: most raster graphics operations that clip to masks are implemented as a loop which performs the operation clipped to each of the rectangles comprising the mask. This approach has allowed us to take advantage of much existing graphics code to perform rectangle-clipped rasterops, and line and character drawing, etc., without having to modify this code (often written in assembly language or microcode) to clip to arbitrarily shaped regions.

3. Methods, Implementations and Classes

Flamingo operations are implemented as a set of *methods* for each object exported by the system. Each instance of a Flamingo object has a list of pointers to *implementations* for the methods available for that object, and a pointer to the object's parent (or class). Any given method, such as NewPA, RasterOP, or DrawLine, may have several implementations -- for example, the window manager has its own implementation of NewPA that creates lower-level pixels arrays for the body, sides and corners of the window, and associates these together in a private structure that represents the window. Usually, any given object inherits its implementations from several *classes* -- its parent, its parent's parent, and so on. For instance, a pixel array used with a window manager has methods associated with just being a pixel array, with being part of a window, with its ownership by a client process, and with getting input from an input device.

Each of the layers in our current system is listed below, along with the data type (data type name in parenthesis) and method implementations added by that layer. Each of the following layers that exports pixel arrays is built on top of the preceding layer, inheriting some of the methods from that lower layer, and re-implementing others. In many cases this re-implementation is accomplished through a *super* construct, borrowed from Smalltalk^{5,6}, whereby an implementation invokes the parent class' method. We give an example of how this is used in the next section.

- **mask**(*Mask*): mask intersection, union, difference, etc.
- **fliim**(*Input Device*): machine dependent input device drivers
- **flam**(*PA*): machine dependent output primitives;
rectangular raster operations, unclipped string and line drawing
- **fligraph**(*PA*): machine independent graphics;
raster operations, string and line drawing over masks
- **coverup**(*PA*): overlapping, mapped pixel arrays
- **frawd**(*PA*): window management functions
- **user interface**(*PA*): mappings from input events
to window manager functions
- **f19** (*the first Flamingo application*): provides a terminal emulator
and a shell within a Flamingo window

A special method called *SetMethod* can be used to add new methods or to replace old ones in any instance of an object. When the system is initialized, a distinguished object for each class is created, from which subsequent instances of that class inherit their methods. *SetMethod* can be used with these class objects (ClassMask, ClassInputDevice, ClassPA, ClassCoverupPA and ClassWindow) to replace a method for an entire class of objects. Presently our inheritance is done via copying: when an object is created, the methods of the parent object are copied into the new

object. What this means for replacing class methods is that the only objects to inherit the new method are those created after the method has been replaced. We are now re-thinking our inheritance mechanism; in the future we will probably delay this binding of methods to implementations.

In our current implementation these layers are all part of a single, monolithic process, but it has been our intention from the beginning to split off parts of the system into separate processes. The communication between processes will initially be done through Unix sockets, so that we can easily port Flamingo to other Unix environments. Later we will take advantage of the more powerful IPC mechanism provided throughout the Spice environment, which allows communication with non-Unix hosts. The list of procedure pointers within each object will be replaced by a list of Unix sockets and/or IPC ports, one for each method, corresponding to the location of the process that implements that method for that object. To implement methods over a network using message-passing, we will use Matchmaker⁷, a remote procedure call generator. With these interfaces, a single Flamingo system (that is, the complete collection of processes providing a single user with Flamingo services) could exist as a multitude of processes on a multitude of machines.

These remote procedure call mechanisms, coupled with the ability to substitute new methods for any object, or class of objects, will make it easy for client programs to alter or extend any of the system's default behaviors.

4. Examples

As an example of how this system structure actually works, we will consider what happens when a Flamingo application performs a raster operation (e.g. copybits) within one of the application's windows. Several different layers of the system export raster operations: PAs exported by **flam**, **fli-graph**, **coverup**, **frawd**, and the **user interface** level all have a method for rasterop. In this case, we will assume that the PA inherits its methods from the ClassWindow class, implemented by **frawd**.

The window manager, **frawd**, inherits its rasterop method from **coverup**. Initially, then, when the user level code calls rasterop, it gets the rasterop procedure within **coverup**, providing as arguments the source and destination pixel arrays and masks within those pixel arrays specifying the region to be copied and the destination's clipping region.

Coverup is responsible for maintaining mappings that connect higher-level pixel arrays to lower-level pixel arrays. These mappings allow a process other than the client to intervene easily in the management of the actual display of information created by the client program. Mappings can be as simple or as complex as necessary to implement the particular mapping abstraction at hand. For the current implementation, a set of mappings is defined for the graphics operations that takes into account the *rank* or height of a mapping over other mappings on the same lower-level pixel array. This particular process of mapping a graphic operation from a higher-level overlapped pixel array to a lower-level one involves substituting different destination masks representing the actual shape of the pixel array after all mappings "above" or "covering" the pixel array have been subtracted from the mask. The client program can continue to think of its individual pixel array as being whole, while the mapping process takes into account covered-up parts of the result when actually performing graphical operations.

In our example, for each of the pixel array's mappings, **coverup**'s rasterop method intersects the argument masks with that mapping's uncovered masks, and calls the rasterop method from **fli-graph** with these clipped masks. The **fli-graph** rasterop decomposes the masks into rectangles, and uses the machine dependent rectangle rasterop procedure in **flam** to actually move the bits.

As a further example that illustrates other aspects of the system design, we will go through the steps that are taken to apply DeletePA (the method that is used to destroy pixel arrays) to a PA obtained from ClassWindow.

Like all of our procedures that implement methods, the procedure DeletePA performs the method lookup task -- it locates the implementation of the delete function for the specified pixel

array. The default implementation of `ClassWindow` is provided by `frawd`. So assuming that the method hasn't been replaced, `DeletePA` will determine that the correct implementation of the deletion method for this PA is `frawdDeletePA`.

Our default window manager creates windows as a set of pixel arrays (supplied by `coverup`) representing the border, corners and body of the window, and keeps track of its windows through a private data structure. What `frawdDeletePA` does is to free this ancillary window data structure, call `DeletePA` to destroy each of the window's component pixel arrays, and then call `DeletePA_SUPER` on the window pixel array to delete its lower-level structures.

At this point those structures which made a window out of this pixel array have been destroyed, but its deletion method pointer still refers to `frawdDeletePA`. What we want to do is to use the implementation of `DeletePA` provided by this object's class' parent class, `coverup`, to delete the remaining parts of this pixel array. `DeletePA_SUPER` does exactly that: it chains through the pixel arrays' parent pointers to locate `coverupDeletePA`. This implementation frees each of the pixel array's mappings, calls `DeletePA` on each of the pixel arrays to which the original pixel array was mapped, and then calls `DeletePA_SUPER` to destroy lower-level pixel array structures. This time the super method is `flamDeletePA`, which frees the lowest level elements of the pixel array, including the 2-dimensional bitmap.

5. The Creation of User Interfaces with Flamingo

As a demonstration of the features of Flamingo, we have already implemented an interpreter for Andrew^{8,9} socket calls. This interpreter, currently built into Flamingo, asserts itself via the appropriate Unix socket calls as an actual instance of Andrew, and then waits for Andrew calls from Andrew programs. Note that we are running Andrew *binaries*. No changes to any of the Andrew processes are necessary. People writing Andrew code are actually writing Flamingo code! We should emphasize that Flamingo only resembles Andrew at the program interface level; what the user perceives is quite different. For example, Andrew only provides tiled windows, while Flamingo provides overlapped windows, and potentially other arrangements as well.

This interpreter demonstrates the basic functionality of the Flamingo primitives. It has also made it possible to create an entirely new operating environment for the user with numerous hooks for implementing still more features and functionality, without rendering the system unusable by the mass of software written for previous systems. (This is of course simply an argument in favor of upward compatibility, but has proven to be an equally powerful mechanism to develop the system as it evolved from just a graphics package.)

Flamingo's object-oriented architecture has provided a flexible mechanism for separating the display and graphics abstractions important to the application process from the lower level abstractions that are important to the system. Each of these Andrew processes sees only one pixel array, namely that pixel which represents its window, and more importantly, knows nothing about the details of that pixel array's method implementations. This pixel array could be a standard top-level Flamingo window, or it could just as easily be a sub-window inside of a window running someone else's window manager scheme.

6. Future Work

Flamingo was officially released within our departmental community in October 1985. Now that we have a system in place, and a growing user community, we are beginning to do some of the research for which Flamingo was originally intended.

The first issue that needs to be addressed is the creation of a programming interface to Flamingo. We have already begun work on a Unix socket interface, and have sketched out a design for an IPC/Matchmaker interface. Our first application of these interfaces will come from separating **f19** from the rest of the system. Later we intend to experiment with separating the Andrew

interpreter, and also **flim** and **flam**, the input and output device drivers. The motivation for this latter split is to provide display support for remote computers, probably Perqs or personal computers, and to take advantage of special properties of these displays, such as graphics hardware, color, etc. By porting **flim** and **flam** to such hosts, users with suitable network access will be able to connect to the full Flamingo system and use it through whatever display device they have available.

Currently a few weeks away is a Sapphire¹⁰ interpreter. This interpreter will call on Flamingo primitives from within the server half of the matchmaker remote procedure call interface for the Sapphire window manager. An interesting demonstration is planned once we have these interfaces in place: we intend to run, inside a couple of Flamingo windows, both window manager programs, Andrew and Sapphire. These window managers depend only on having some definition of an underlying graphics output device, a keyboard and a mouse for input devices, and a Unix file system. From the Flamingo windows, we can provide the virtual graphics necessary for the window managers to appear in different, overlapping areas of the screen. The Flamingo window can also provide the appropriate input events as part of its usual mapping of events to user processes. Obviously, certain input operations will have to be reserved for use by Flamingo's user interface in order to maintain control over the separate processes.

There are a number of performance issues that have yet to be addressed when considering the task of separating a Flamingo system into communicating processes. One major concern is the possibility that the interfaces between these processes will create substantial communications demands. How to structure these interfaces so as to achieve reasonable system performance is an open research problem, but this is precisely the kind of issue that Flamingo has been designed to help us investigate, and we will be looking at it more in the future.

An interesting use of mappings has been proposed that will allow high-quality images of the screen to be generated. A typical method for getting pictures of a screen is to simply dump the state of the raster memory used to generate the screen bits and display this using either a dot-matrix printer, laser printer, or other suitable, non-alphanumeric device. Such devices as laser printers have a much finer resolution than that of the screen hardware, and the resulting image is often unclear or distorted. A Flamingo mapping could be defined for PAs displayed on the screen that would map all raster operations to a generator of a file of laser printer commands. Line drawing, character drawing, and all raster operations would all be done in a scale appropriate to that of the printer's capabilities rather than to the scale of the screen.

Finally, other work needs to be done to make the system more comfortable to use; we will be adding menus, title bars, icons, etc. Another area of interest is to provide support for experimental input devices being developed by other groups within the department. Also, we are interested in examining different styles of window management and user interface; our first step in this direction will probably be to implement the constraint-based tiling algorithm developed by Cohen, Smith and Iverson.¹¹

7. Final Remarks

Flamingo addresses the problem of flexible, robust access to multiple processes running on multiple machines. Our most pressing problem is one of distributed resource management. Using the Spice Sesame¹² distributed file system, users of Spice machines have uniform access to the data located on a large number of distributed machines, but Spice users have never had uniform access to the processing power of those machines.

We gratefully acknowledge the entire Flamingo working group, which has at times included Rich Cohn, Roger Dannenberg, Dario Giuse, Mark Hjelm, Paul McAvinney, Rob MacLachlan, Randy Pausch, Rick Rashid, Walter Smith, Pedro Szekely, Avie Tevanian, and Skef Wholey, for their insights, arguments and ideas. The first running Flamingo system came up on June 5, 1985, and many subsequent versions were written during the summer and fall of 1985 by Ed Smith, David Anderson, and Walter Smith, with some help from Avie Tevanian and the entire MACH operating system crew.

8. References

1. CMU Computer Science Department, *Proposal for a Joint Effort in Personal Scientific Computing*, August 1979.
2. R. F. Rashid, G. G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel," in *Proceedings of the 8th Symposium on Operating Systems Principles* (December 1981).
3. R. V. Baron, R. F. Rashid, E. H. Siegel, A. Tevanian, M. W. Young, "MACH-1: An Operating System Environment for Large-Scale Multiprocessor Applications," *IEEE Software* (July 1985).
4. R. V. Baron, R. F. Rashid, E. H. Siegel, A. Tevanian, M. W. Young, "MACH-1: A Multiprocessor-Oriented Operating System and Environment," *SIAM Computing* (to appear).
5. D. H. H. Ingalls, *The Smalltalk-76 Programming System Design and Implementation*, Xerox PARC (1980).
6. A. Goldberg, D. Robson, *Smalltalk-80*, Addison-Wesley (1983).
7. M. B. Jones, R. F. Rashid, M. Thompson, "MatchMaker: An Interprocess Specification Language," in *ACM Conference on Principles of Programming Languages* (January 1985).
8. J. A. Gosling, D. S. H. Rosenthal, *A Window Manager for Bitmapped Displays and UNIX(tm)*, Information Technology Center, Carnegie-Mellon University (1984).
9. J. A. Gosling, D. S. H. Rosenthal, *A Network Window-Manager*, Information Technology Center, Carnegie-Mellon University (1984).
10. PERQ Systems Corporation, *User's Guide to the Sapphire Window Manager*, 1984.
11. E. S. Cohen, E. T. Smith, L. A. Iverson, "Constraint-Based Tiled Windows," in *Proceedings of the 1st International Conference on Computer Workstations* (1985).
12. M. B. Jones, R. F. Rashid, M. Thompson, *Sesame: The Spice File System*, Department of Computer Science, Carnegie-Mellon University (1982).

A Proposal for Interwindow Communication and Translation Facilities

Daniel P. Gill

Exxon Research and Engineering Company
180 Park Avenue
Florham Park, New Jersey 07932
(201) 765-6593

ABSTRACT

With the increasing levels of sophistication of windowing systems being introduced for Unix[†] today, new capabilities are now needed to allow cooperating window-based processes to communicate easily and initiate transformations on live data flowing between these cooperating processes. This paper proposes a high-level *interwindow communication* scheme whereby arbitrary window-based processes can dynamically set up communication links between one another and optionally invoke a series of *translation filters* to perform appropriate data transformations (translations) on the data flowing through the communication links between the windows.

Each of the new proposed facilities will be described together with the internal data structures that must be maintained. Also a 4.2BSD Unix implementation of the proposed interwindow communication (IWC) primitives and an interactive window-system-driven interface constructed from these primitives is discussed.

The motivation for the work described herein is a result of the author's experience with developing a window-driven integrated environment for engineering workstations at Exxon Research and Engineering Co. The work was performed on a 68010-based Sun Workstation[‡], part of which involved the customization of and addition to the Sun-provided software for control of their window system, and represents some of the ongoing software environment research at Exxon Research and Engineering Co.

Introduction

A number of Unix-based windowing systems have been introduced in the last few years and their design and capabilities have been well documented in the literature [7], [8], [9], [10], [11], [12]. In general, these systems present a powerful, general purpose operating environment, providing a wide array of multi-window facilities. However the one area which these window systems have not yet addressed is the area of window-related interprocess communication. By its omission, the burden is placed on each application developer to incorporate their own interwindow facilities.

In the context of a windowing system, mechanisms to allow processes to communicate once they are bound to windows (without any pre-conceived preparation to communicate), either are unapplicable (e.g. *pipes*) or are too low-level and cumbersome to use directly (e.g. *sockets*). Furthermore, none of these mechanisms were designed to be used in an interactive mode (the way in which one typically interacts with a window system). In other words there is no easy way to dynamically create channels of communication between processes running in windows and redirect

[†] Unix is a trademark of AT&T Bell Laboratories

[‡] Sun Workstation is a trademark of Sun Microsystems, Inc.

information flows. Moreover, it would also be desirable to be able to insert *translation filters* in the connection stream. In fact, the desire for just this type of capability has been alluded to by the designers of an existing window system [7]: "More general IPC under UNIX would be nice -- - we would like to be able to use the window manager dynamically to connect programs in building block fashion."

Window systems *should* have the facilities to allow processes to *cooperate* and thus not *force* application programs to handle these operating system interactions directly. Therefore, what is needed is a high-level set of *interwindow communication (IWC)* facilities. They should be invocable by a series of point-and-click operations (*mouse-driven*) on a graphical interface and have a reasonable high-level programmatic interface. They should be integrated and included as part of a window management system's overall functional working set (or as part of a library extension).

Implementation Approaches

The two most obvious methods of implementing these interwindow facilities would be either at the nucleus level of an operating system (e.g., kernel of the Unix operating system) or at a higher window system level (e.g., SunWindows†).

Of these two methods, arguments commonly made *for* incorporating these kinds of facilities at the kernel level raise points such as:

1. Since interwindow communication embodies many of the same concepts as an operating system, they should be implemented within the kernel.
2. Since IPC facilities vary from operating system to operating system (or even version to version of the same operating system ‡), this issue should be addressed on a per application basis using the available kernel-based IPC facilities of the particular operating system.

Arguments *against* putting interwindow communication in the kernel are:

1. There would be too much kernel overhead associated with such a set of facilities which would degrade window-system performance and also possibly increase the size of the kernel.*
2. It would be desirable to have a reasonably portable (and thus, higher level) set of interwindow interprocess tools.

The way window-bound processes communicate is most certainly an operating system issue, but a set of *high-level facilities* can be designed in such a way that most operating systems (certainly all *modern* versions of Unix) could provide the needed underlying mechanisms. In this manner a standard higher level protocol for interwindow communication can be developed with the added bonus of not having to implement new low-level, kernel-based IPC mechanisms to specifically support interwindow communication. It is the author's opinion that higher level window systems can realistically handle interwindow IPC.

Furthermore, these facilities can be built using *existing* IPC mechanisms.

† SunWindows is a trademark of Sun Microsystems, Inc.

‡ For example, System III Unix v.s. System V Unix.

* Admittedly, kernel overhead considerations raise important performance questions regarding low level implementations. However with the advent of faster processors (e.g., the Motorola 68020, the National Semiconductor 32332 and the Intel 80386) supporting these operations, it becomes less of an issue.

Existing Unix IPC Facilities

At the kernel level the existing variants of the Unix System offer a wide array of interprocess communication facilities. The ubiquitous *pipe*, common to all Unix Systems, provides a powerful mechanism for *related* processes to communicate with. However with this form of communication, processes that decide to communicate *after* they have been created (without connections having been set up in a common ancestor), are not able to do so. To remedy the limitations of the pipe, each variant of the Unix System has incorporated their own IPC extensions.

AT&T's System V [1] offers, in addition to the *unnamed pipe*, *named pipes*, *semaphores*, *shared memory* and *message queues*. These are useful for specific applications, but they are by no means what might be considered general IPC mechanisms. In addition they are extremely low-level and awkward to use for application programs.

Berkeley's 4.2BSD System offers an extremely general set of IPC mechanisms based on *sockets* [2]. They have the added bonus of allowing communication between processes residing on different machines.

The AT&T Bell Laboratories Eighth Edition *streams* [3] mechanism is probably the most general, elegant and easy to use IPC method proposed thus far. Although the Eighth Edition version of Unix is not generally available, the ideas presented are an excellent model on which to base higher level mechanisms.

Interwindow Communication Primitives

In order to describe the proposed set of *interwindow communication primitives*, some new terminology must first be introduced. In this paper we are only considering *communicating windows* which are of tty emulation window or subwindow type since this is most intuitive and we are dealing exclusively with *byte stream* communications.

An *interwindow channel* is a unidirectional connection *ultimately* between two windows (window-bound processes). In other words this channel is *capped* at either end by a window-bound process (as opposed to an arbitrary possibly non window-bound process) and forms a *virtual circuit* connection similar to a Unix pipeline. When an interwindow channel is constructed, it is assigned a unique *interwindow channel descriptor*. From then on during its existence it is referred to by that unique tag.

A *translation filter* (window-bound process or non window-bound process which transforms a byte stream) may be *inserted* dynamically at intermediate points along the interwindow channel. An *interwindow channel (node) address* is assigned upon insertion (each node gets one) and can be used to reference points of interest (processes) in the infrastructure.

What follows are the basic building block primitives to support *interwindow communication*. They can be used to easily construct interactive, window-system-driven *IWC* tools. The proposed calls are designed so as to be relatively operating system independent. The notation used is the C-like description language used in [14].

An *interwindow channel*, connecting two window-bound processes, is *initiated* by the call:

```
iwchan = iwc_request(windowid1, windowid2, iwaddr1)

result int iwchan      /* interwindow channel descriptor */
result int *iwaddr1    /* interwindow channel address of window1 */
char *windowid1       /* source window */
char *windowid2       /* sink window */
```

in the potential sending process (here *windowid1*), and *completed* by the call:

```
iwchan = iwc_accept(windowid1, windowid2, iwaddr2)  
  
    result int iwchan      /* interwindow channel descriptor */  
    result int *iwaddr2   /* interwindow channel address of window2 */  
    char *windowid1      /* source window */  
    char *windowid2      /* sink window */
```

in the potential receiving process (here *windowid2*).

This successful completion of these calls sets up a *producer-consumer* relationship between *windowid1* and *windowid2* respectively, with both processes being returned an *interwindow channel descriptor*. Also returned to each process is an *interwindow channel (node) address* which represents the particular process's position on the interwindow channel. These calls obviously provide a *distributed* interface, similar to the programming language Ada's† *rendezvous* mechanism. They are executed by processes that are currently *disjoint* and wish to establish a producer-consumer relationship. A *centralized* call:

```
iwchan = iwc_connect(windowid1, windowid2, iwaddr1, iwaddr2)  
  
    result int iwchan      /* interwindow channel descriptor */  
    result int *iwaddr1   /* interwindow channel address of window1 */  
    result int *iwaddr2   /* interwindow channel address of window2 */  
    char *windowid1      /* source window */  
    char *windowid2      /* sink window */
```

also creates an interwindow channel. This call would be used in conjunction with a *connection server*, and is functionally equivalent to the *iwc_request - iwc_accept* pair. The primary reason for including a somewhat redundant IWC mechanism is *ease of invocation*; with this primitive, information need only be supplied at one centralized location (whereas with the above mentioned distributed primitives, the required information must be supplied at two places).

To *disconnect* an interwindow channel, a window-bound process issues the following call:

```
iwc_close(iwchan)  
  
    int iwchan      /* interwindow channel descriptor */
```

This is done by/for processes on both ends of the channel.

A process may *redirect* output for display in a specified window by the call:

```
iwc_redir(windowid)  
  
    char *windowid      /* output redirection window */
```

Note, this call does not set up a producer-consumer relationship between a process and a window-bound process, it merely redirects the output flow from a process to a particular window for display

† Ada is a registered trademark of the U.S. Government - Ada Joint Program Office

puposes only.

Once an interwindow channel has been successfully set up, it may be desirable to *dynamically* insert a *translation filter* into the connection stream. This is accomplished by the call:

```
iwaddr = iwc_tfin(trfilter, iwchan, iwproc1, iwproc2)

      result int iwaddr      /* interwindow channel address */
      int iwchan             /* interwindow channel descriptor */
      char *trfilter        /* translation filter to be inserted */
      char *iwproc1        /* process on interwindow channel */
      char *iwproc2        /* process on interwindow channel */
```

which interposes *trfilter* between *iwproc1* and *iwproc2* on the interwindow channel identified by *iwchan*.

A *translation filter* can be removed from the connection stream, thereby reconnecting its *predecessor* and *successor* on the interwindow channel by the call:

```
iwc_tfout(trfilter, iwchan)

      int iwchan             /* interwindow channel descriptor */
      char *trfilter        /* translation filter to be removed */
```

This call removes connected process *trfilter* from the interwindow channel identified by *iwchan*.

An Experimental 4.2BSD Unix Implementation

A prototype implementation of the above described interwindow communication primitives was developed for a Sun Microsystems Workstation running 4.2BSD Unix. They were for the most part constructed with standard 4.2BSD file system and IPC system calls. However, some additional *IWC system structures* and manipulation routines also had to be built.

There is an *interwindow channel table* which contains one entry for each interwindow channel that has been constructed. The table is of fixed length and the indices into this table are the *interwindow channel descriptors* themselves. Each entry contains a pointer to a structure called an *interwindow node reference table*. One fixed length node reference table exists for each interwindow channel. Each table contains the same number of entries corresponding to number of available nodes for an interwindow channel. The indices into the node reference table are the *interwindow (node) addresses* themselves and each entry in the table contains either a *null* pointer (for an unallocated-uninserted node) or a pointer to an *interwindow node attribute structure* (for currently allocated-inserted nodes). Each node attribute structure contains:

- 1) a process ID
- 2) a successor pointer (to an interwindow (node) address)
- 3) a predecessor pointer (to an interwindow (node) address)
- 4) an associated *window name*[†] (if process at this node is window-bound)
or *windowless name* (if this process is non-window-bound)

An *IWC service daemon* was constructed to provide a single point of contact for requesting

[†] *Window names* were arbitrarily chosen to be a series of character strings (i.e. "win1", "win2", ..., "winN"). The particular naming convention has no inherent importance. However, taken in the context of the *node attribute*

IWC services (these services are only requested by IWC primitives). This *service server* is a continuously existing process which *listens* (at a well known address) on a *socket*. It *accepts* connections from *client* IWC processes, obtaining a message over the newly connected socket which contains (among other things) the type of service required and parameters needed to carry out the service. The service daemon then creates (via a *fork/exec* sequence) the appropriate server process (note: the *child* server process *inherits* the client-server socket connection) which finally carries out the client request. An important function of the IWC service daemon is to provide support services for IWC system structures. The general outline of the IWC service daemon is as follows:

```
create socket to detect client requests at well known address
establish queue to allow simultaneous connection requests
for(;;){
    accept client connection thereby creating socket in which
        client-server transmission will take place
    receive message from client to ascertain required service
    if (fork()==0){
        close detection socket
        exec appropriate server process
    }
    close connection socket
}
```

With the above described IWC system structures and support services as given, the implementation of the IWC primitives can now be outlined.

Here is a *rough sketch* for the *iw_c_request* and *iw_c_accept* calls:

<pre>iw_c_request(...) { create socket initiate interprocess connection via socket make interwindow channel just created - standard output update IWC structures }</pre>	<pre>iw_c_accept(...) { create socket accept interprocess connection via socket make interwindow channel just created - standard input update IWC structures }</pre>
---	---

This particular implementation of the calls uses the 4.2BSD IPC mechanism - *sockets* to create a pipe-like connection between unrelated processes (it could be viewed as a simulation of AT&T's System V *named pipes*). First, a socket is created in the *source* window-bound process (the one that invokes *iw_c_request*) and also in the *sink* window-bound process (the one that invokes *iw_c_accept*), then a process to process connection is arbitrated by the cooperating window-bound processes (the *rendezvous* is made) thereby creating an *interwindow channel*. Then *standard output* of the source window becomes the interwindow channel, and *standard input* of the sink window becomes the other end of the interwindow channel. *IWC structures* are then updated (via IWC service daemon service requests) to reflect the creation of the interwindow channel and its two *end point nodes* and the producer-consumer relationship between the two window-bound processes is consummated.

The algorithm for *iw_c_close* is as follows:

structure described above, what is provided is a process name to process ID mapping for window-bound (and non-window-bound) processes existing and being manipulated in the window system.

```

for (interwindow channel to be closed)
  for each interwindow node {
    terminate associated process
    update associated IWC.structures
  }
insert null pointer in interwindow channel table entry

```

Here, an extremely simplified approach was taken. Basically the interwindow channel is *shut down* (including all processes associated with the interwindow channel). This approach poses no real problems since everything can be reconstructed interactively in *building block* fashion.

Iwc_redir was trivial to implement. This call causes output from the invoking window to be continually redirected to the *slave* side of the *pseudo terminal*[†] associated with the target of redirection window. This in turn causes all output that normally would have went to the source window to show up in the target of redirection window.

Iwc_tfin inserts an arbitrary (*translation filter*) process (window-bound or non window-bound process) between two currently existing interwindow processes (processes currently *residing* on an interwindow channel). This *transaction* actually involves the cooperation of three processes; the *predecessor* process, the *IWC service daemon* and the *successor* process. The algorithm/process interaction is as follows:

<i>Predecessor Process</i>	<i>IWC Service Daemon</i>	<i>Successor Process</i>
create socket	create socket	create socket
initiate connection to		
IWC service daemon.....	accept connection from	
send message requesting	predecessor process	
<i>tfin service</i>	create and start <i>tfin</i> service process	
	(in <i>tfin</i> service process)	
make predecessor-to- <i>tfin</i> channel,	initiate connection to	
standard output	successor process.....	accept connection
		from service process
		make <i>tfin</i> -to-successor channel,
		standard input
	make predecessor-to- <i>tfin</i> channel,	
	standard input	
	make <i>tfin</i> -to-successor channel,	
	standard output	
	<i>exec</i> translation filter	
	(translation filter now reads	
	from predecessor process	
	and writes to successor	
	process via IWC channel)	

iwc_tfin (the primitive) is invoked from the potential predecessor process. It initiates a socket connection to the *WC service daemon* and sends a message (upon successful completion of the

[†] *Pseudo Terminals* are two part software devices that simulate the actions of hardware associated with a glass teletype. The *slave* side presents an hardware-like interface which fools programs into believing that they actually have control of dedicated display terminal. The *master* side allows a program such as a window manager to actual-ly control what is displayed on a partitioned screen and where it is displayed.

connection) containing information needed for the service daemon to carry out the *tfin service*. The service daemon accepts the connection, receives and acts on the *request-for-service message* and *forklexecs* the *tfin service* process. Concurrent with this activity, the *predecessor-to-IWC service* channel becomes the standard output channel in the predecessor process. Meanwhile in the *tfin service* process, a socket connection is initiated to establish a communication channel to the potential successor process. The successor process accepts the connection from the *tfin service* process and makes that channel it's standard input. Concurrent with this activity, the *tfin service* process makes the channel from it's predecessor process, standard input, and the channel to it's successor process, standard output. Then it *overlays* itself with the desired translation filter, which inherits the predecessor and successor connections.

Iwc tfout removes a translation filter from the interwindow channel. This primitive has not yet been implemented.

Interactive Window-System Interface

An interactive window system interface can easily be constructed out of the above primitives. As an example of this, we will look at the structure and usage of an interactive interface (that runs under SunWindows† [17]) that allows one to establish interwindow channel.

Pop-up menus are used as a selection mechanism. A user accesses IWC facilities by first pointing to the border of an existing window which exposes the *tool manager*‡ [17] pop-up menu. Then by bringing the *IWC menu* to the foreground (the menu is *stacked* behind the *tool manager pop-up menu*). This reveals a number of *IWC options* from the previously obscured pop-up menu. The options appear as follows:

IWC
<hr/>
<i>Request channel</i>
<hr/>
<i>Accept channel</i>
<hr/>
<i>Request filter out</i>
<hr/>
<i>Accept filter in</i>
<hr/>
<i>Remove filter</i>
<hr/>
<i>Disconnect channel</i>

To create an interwindow channel using this interactive interface is a two-part process and may be accomplished as follows; A user positions himself (with a pointing device) in the *border* of the potential *source* window, brings up the the *IWC menu* (as described above) and selects **Request channel**. This *event* causes a program to execute which prompts the user for the name of the *sink* window. *By inspection* (window names are displayed in the *namestripes* of open windows), the user ascertains the window name and responds to the prompting program. Once the information is successfully entered, **iw_request** is invoked which *initiates* the interwindow connection. The user must then position himself in the potential *sink* window and select **Accept channel**. This *event* causes a program to execute which prompts the user for the name of the *source* window. Once the

† SunWindows is a trademark of Sun Microsystems, Inc.

‡ The *tool manager* is Sun-provided window manipulation menu that allows a user to move, expand, shrink, open, close,...etc. existing windows in the SunWindows system.

information is successfully entered, `iw_c_accept` is invoked which *completes* the interwindow connection. The whole process can be described as follows:

on events (IWC selections - *request channel, accept channel*)
prompt for names of the windows to be connected
invoke `iw_c_request` and `iw_c_accept`
which then cooperate to consummate an interwindow channel
connection via sockets.

Other options selected work in a similar fashion; an *event* or *events* (i.e. selecting an option from a pop-up menu(s)), cause a group of *IWC processes* to interact and provide the desired end result.

Conclusions

This was a first attempt at the design and implementation of high-level interwindow communication primitives. A number of alternatives and additions have been considered. An interface extension to the 4.2BSD socket facility to allow dynamic reconnection of sockets was suggested by one of the authors of the X window system [22]. This would facilitate the easy implementation of many of the IWC primitives, especially the ones involving translation filters.

A set of interactive interwindow communication facilities has proven to be a useful add-on to a window management system. They allow users to interface with a window system in a *purely interactive* mode of operation in *building block* fashion. This alleviates the *necessity* for programmers to pre-construct *static* interprocess links between potential communicating window system processes. It is hoped that future developers of windowing systems recognize the need for these kinds of capabilities and provide them as part of their window system *toolbox*.

Future Work

Future research in this area centers around alternate ways of expressing the ideas presented, and alternate implementation strategies. For instance, using remote procedure calls [18], [19], [20], [21] to build IWC facilities or using Eighth Edition Unix System [5] IPC Primitives (when they become available) as a foundation.

Acknowledgements

I would like to thank George Lyon for his contributions and feedback. Also, I would like to thank Len Barnstone, Jay Dev, Mark Eisner, Paul Nunn, Dan Sarnowski and Jack Wiesenthal. Without their continuing support and encouragement none of this would have been possible. Drew Wright provided suggestions on some of the early drafts. In reviewing this paper, Jim Gettys supplied many useful suggestions and much food for thought. Finally, special thanks go to Mike Catolico and Frank Greco who provided valuable comments and suggestions on some of the more technical issues.

References

- [1] *UNIX System V - Release 2.0 Programmers Reference Manual*, AT&T Technologies (1984)
- [2] Leffler, S., Joy, W. and Fabry, R., *A 4.2BSD Interprocess Communication Primer*, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720 (July 1983)
- [3] Ritchie, D.M., "A Stream Input-Output System", *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, (October 1984)
- [4] Killian, T.J., "Processes as Files", *USENIX Summer Conference Proceedings*, Salt Lake City, UT (June 1984)
- [5] Presotto, D.L. and Ritchie, D.M., "Interprocess Communication in the Eighth Edition Unix

- System", *USENIX Summer Conference Proceedings*, Portland OR (June 1985)
- [6] Pike, R., "Graphics in Overlapping Bitmap Layers", *ACM Transactions on Graphics* 2, 2 (April 1983)
- [7] Rhodes, R., Haeberli, P. and Hickman, K., "Mex - A Window Manager for the IRIS", *USENIX Summer Conference Proceedings*, Portland OR (June 1985)
- [8] Evans, S.R., "Windows with 4.2BSD", *Unicom Conference Proceedings*, San Diego CA (January 1983)
- [9] Pike, R., "The Blit: A Multiplexed Graphics Terminal", *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, (October 1984)
- [10] Jacob, R.J.K., "User Level Windows for Unix", *Uniform Conference Proceedings*, Washington D.C. (January 1984)
- [11] *X Window System Protocol Specification*, MIT Project Athena (1985)
- [12] *Programmers Guide to the Window Manager: A Guide for the Uninitiated* (for Release 1 of the ITC Prototype Workstation), Information Technology Center, Carnegie Mellon University (June 1985)
- [13] *User's Manual for Release 1 of the Andrew System*, Information Technology Center, Carnegie Mellon University (June 1985)
- [14] Joy, W., Cooper, E., Fabry, R., Leffler, S., McKusick, K. and Mosher, D., *4.2BSD System Interface Overview*, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720 (July 1983)
- [15] Wulf W.A., Levin R., Harbison S.P., "HYDRA/C.mmp: An Experimental Computer System", McGraw Hill (1981)
- [16] Gehani, N., "Ada Concurrent Programming", Prentice Hall (1984)
- [17] *Programmers Reference Manual for SunWindows*, Sun Microsystems, Inc., Mountain View, CA 94043 (May 1985)
- [18] Nelson, B.J., "Remote Procedure Call", Tech. Report CSL-81-9, XEROX Palo Alto Research Center, Palo Alto, CA (1981)
- [19] *Courier: the remote procedure call protocol*, XEROX System Integration Standard X SIS-038112, XEROX Corporation, Stamford Connecticut (December 1981)
- [20] Birrell, A.D., Nelson, B.J., "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems* 2, 1 (February 1984)
- [21] *Networking on the Sun Workstation*, Sun Microsystems, Inc., Mountain View, CA 94043 (May 1985)
- [22] Gettys, J., MIT Project Athena, Personal communication, (November 1985)

Problems Implementing Window Systems in UNIX†

James Gettys

Digital Equipment Corporation
Project Athena
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

ABSTRACT

Over the last 18 months the X window system has been implemented under 4.2BSD UNIX at MIT at the Laboratory for Computer Science and Project Athena. While on the whole the resulting design is reasonably clean and pleasing, UNIX strongly limited the possible implementation strategies. This paper discusses the problems encountered, how they influenced our design decisions, and suggests issues for future study and development as UNIX evolves toward a distributed environment.

X Window System Design

While this paper is not specifically about the X window system, X will be used as an example for much of the discussion. X is best described using a client/server model. X consists of a collection of client programs which communicate with the window system server. They are implemented entirely in user code. All communications with the window system occur over a stream connection to the window system. X is completely network transparent; i.e. a client program can be running on any machine in a network, and the clients and the server need not be executing on the same machine architecture. The block diagram shown in Figure 1 describes the structure of the system.

X supports overlapping possibly transparent windows and subwindows to arbitrary depth. Client programs can create, destroy, move, resize, and restack windows. X will inform clients on request of key presses, key releases, button presses, button releases, mouse window entry and exit, mouse motion, a number of types of window exposure, unmap (removal of a window from the screen), and input focus change. Cut and paste buffers are provided in the server as untyped bytes. Graphic primitives provided include dashed and dotted multi-width lines, and splines. There is a full complement of raster operations available. The implementation supports color, though the current protocol limits the depth of a display to 16 bits/pixel.

The X window system consists of a collection of UNIX programs providing various services on a bitmap display. There is only a minimal device driver to field interrupts from mouse, keyboard, and potentially a display engine. The X server accepts connections from client applications programs. Window, text and graphics commands are all multiplexed over (usually) a single connection per client. Multiple clients may connect to the server.

The X protocol is the only way to communicate to the window system. The X server enforces clipping, does resource allocation, multiplexes output from multiple clients, performs hit detection for mouse and keyboard events, and performs graphics primitives in windows. The protocol is entirely based on a stream. The current implementation uses TCP as its stream transport layer; though it has been run experimentally using DECNET stream connections. A client program may run on any machine in a network. On a local net, performance is the same or better when run

† UNIX is a Trademark of AT&T Bell Laboratories.

remotely as when run locally given two identical unloaded processors.

The X server is a single threaded server program. Requests from clients are processed in a round robin fashion to provide apparently simultaneous output. This has proven to be sufficient, and vastly simplified the design and implementation. Single threading provides all locking and synchronization without any complex structure. The X server must therefore be very careful never to block waiting on a client, and exploits the observation that each individual graphics operation is very fast on a human time scale (though it may be slow on a systems time scale). The 4.2BSD facilities that make this easy to implement include select(2), non-blocking I/O, and the network mechanism (IPC to unrelated processes).

The current X server implementation does NOT maintain the contents of windows. Refresh of a damaged window is the responsibility of the client. The server will inform a client if the contents of a window has been damaged. This was motivated by a number of observations: 1) clients often have their own backing store, and this must be maintained by most programs when resized anyway; if the window system provides backing store, it is often duplicating existing facilities. 2) keep the window system simple and FAST. 3) the amount of data that would have to be stored for bitmap backing store on color displays is very large. Naive UNIX applications are run under a terminal emulator which provides the refresh function for them.

X delegates as much to a client as possible. It provides low level "hooks" for window management. No less than three window manager programs (a separate client program in the X design from the window system) have been written to date, and provide quite different user interfaces. Menus are left to client code to implement, using the flexible primitives X provides. There have been four different styles of menus implemented to date, including a quite sophisticated "deck of cards" menu package.

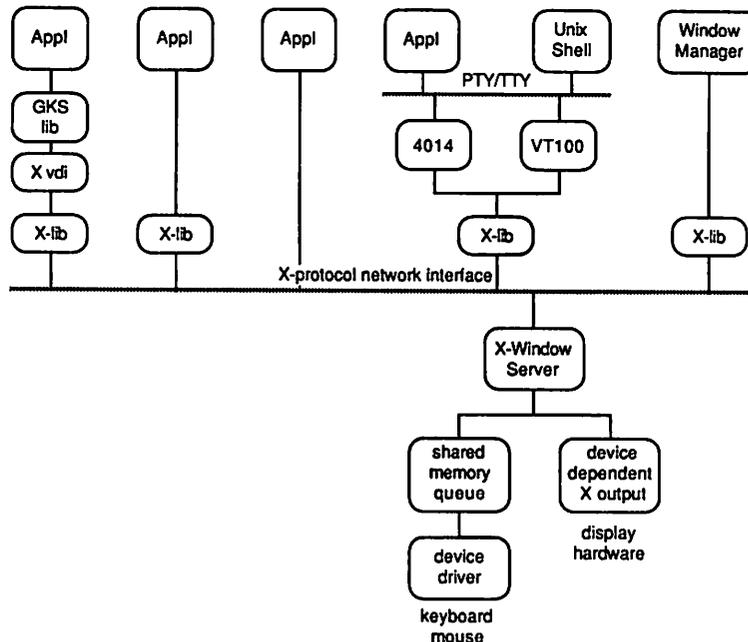


Figure 1: Block Diagram Structure of X

X runs as of this writing on four quite different types of hardware, from very intelligent to very simple. An example of a very intelligent (and reasonably well designed) piece of hardware from the programmers point of view is the DEC Vs100, though it suffers due to the nature of its interface to a VAX, which adds overhead and latencies to each operation. A QVSS display on a MicroVAX (VS1 and VS2) is at the opposite end of the spectrum, being a simple bitmap with no hardware assist. Other ports are in progress.

Alternatives to User Process Window Systems

As currently implemented on most machines, the UNIX kernel does not permit pre-emption once a user process has started executing a system call unless the system call explicitly blocks. Any asynchrony occurs at device driver interrupt level. UNIX presumes either that system calls are very fast or quickly block waiting for I/O completion.

This has strong implications for kernel window system implementations. While window system requests do not take very long (if they did, the presumptions made in X would be unacceptable), they may take very long relative to normal system calls. If a system call is compute bound for a "long period", interactive response of other processes suffers badly, as the offending process will tend to monopolize the CPU. One might argue that this is not offensive on a single user machine but it is a disaster on a multiuser machine. If graphics code and data is in the kernel for sharing, it permanently uses that much of kernel memory, incurs system call overhead for access, and cannot be paged out when not in use.

Similarly, in X as well as most other window systems, if a window system request takes too long, other clients will not get their fair share of the display. This is currently somewhat of a problem during complex fill or polyline primitives on slow displays. The concept of interrupting a graphics primitive is so difficult that we have chosen to ignore the problem, which is seldom noticeable. If such graphics primitives occur in system calls, they have a much greater impact on process scheduling.

An alternative to a strictly kernel window system implementation splits responsibility between the kernel and user processes. Synchronization, window creation and manipulation primitives are put in the kernel, and clients are relied on to be well behaved for clipping. Output to the window is then performed in each user process. This has several disadvantages (presuming no shared libraries, not available on most current UNIX implementations). Each client of the window system must then have a full copy of graphics code. This can be quite large on some hardware, replicated in each client of the window system. For example, the current bit blit, graphics and clipping code for QVSS is approximately 90kbytes, or 18000 lines of C source code. Fill algorithms may also require a large amount of buffer space.

Even worse (as the number of different display hardware proliferates with time on a single machine architecture) is that this split approach requires the inclusion in your image of code for hardware you do not currently have. Upward compatibility to new display hardware is also impossible without shared libraries, but dynamic linking is really required for the general solution.

With much existing hardware it is hard to synchronize requests from multiple processes if the hardware has not been designed to efficiently support context switching. There are sometimes workarounds for these problems by "shadowing" the write only state in the hardware. We have seen displays which incur additional hardware cost to allow for such multiprocess access. One must also then face the locking of critical sections of window system data structures if the window system is interruptible.

UNIX internal kernel structuring currently provides most services directly to user processes. It would be difficult to provide network access to the window system if it were in the kernel due to this horizontal structure but a better ability to layer one facility on another would improve this situation. Again, this is a failure of the kernel to be sufficiently modular to anticipate the evolving environment.

X finesses all of these problems: 1) X and client applications are user processes; ergo no scheduling biases. 2) There is only one copy of display code required, in the server, which can be paged since it is completely user code. This also saves swap space, in short supply on most current workstations. The resulting client code is thus small. Minimal X applications are as small as 16k bytes. No graphics code is in an application program. 3) Client code can potentially work with new hardware without relinking, as no display specific code appears in a client program image. 4) Network access to the window system comes at no additional cost, and no performance penalty (in practice, performance is often gained). 5) X avoids system call overhead by buffering requests into a single buffer and delaying writing in a fashion similar to the standard I/O library. The system call

overhead for output is therefore reduced by well over an order of magnitude per X operation. 6) User process code is easy to debug. Some complications can arise due to the distributed nature of the system. In practice, this has rarely been a problem. 7) Applications requiring a "compute server" can be run from the user's workstation.

Kernel lightweight processes could be used to solve the non-preemptable nature of system calls and would create more options for window system implementations. Since raster operations can be quite long lived, performing these in the current structure allows one process to monopolize the system to the detriment of other processes. Since all context in the system call layer of the kernel is associated with a user process, there is currently no way to divorce such operations from a process and schedule them independently.

While lightweight processes would unnecessarily complicate the X server design (requiring us to lock data structures and perform synchronization), they could be used prevent the most common X programming mistake. Programmers new to X invariably forget to flush the output buffer when testing their first program. A timer driven lightweight process in clients would be useful to guarantee automatic flushing of the buffers.

Shared Memory

On a fast display and processor, X may be performing more than one thousand operations (X requests) per second. If every access to the device requires a system call, the overhead rapidly predominates all other costs. X uses a shared memory structure with the device driver for two purposes: 1) to get mouse and keyboard input and 2) to access the device or write into a memory bitmap.

As pointed out before, X is a single threaded server. Since client programs should be able to overlap with the window system as much as possible (remember that you may be running applications on other machines), it is particularly important to send input events to the correct client as soon as possible. It is therefore desirable to test if there is input after each graphic output operation. This test can be performed in only a couple of instructions given shared memory, and would otherwise require either one system call/output operation (to check for new input) or a compromise in how quickly input would be handled.

All input events are put into a shared memory circular buffer; since the driver only inserts into the buffer, and X only removes from the buffer, synchronization is easy to provide with separate head and tail indices (presuming a write to shared memory is atomic).

Output on the QVSS is directly to a mapped bitmap. In the case of the Vs100, a piece of the UNIBUS[†] and a shared DMA buffer are statically mapped where both the driver and the X server can access them. Output requests to the Vs100 are directly formatted into this buffer, minimizing copying of data.[‡] This permits the device dependent routines to start I/O transfers without system call overhead (by directly accessing device CSR registers), and avoids UNIBUS map setup overhead that DMA from user space requires.

These changes dramatically increased performance and improved interactive feel when implemented, while greatly reducing CPU overhead. Since proper memory sharing primitives are lacking in 4.2BSD, it was implemented by making pages readable and writable in system space, where they are accessible to any process. In theory, any program on the machine could cause a Vs100 implementation to machine check (odd byte access in the UNIBUS space), though in practice it has never happened. None the less, it is the ugliest piece of the current X implementation. We are more willing to allow a server process to access hardware directly than kernel code, as it is much easier to debug user processes than kernel code.

The current X implementation uses a TCP stream both locally and remotely, though one could

[†] UNIBUS is a trademark of Digital Equipment Corporation.

[‡] Our thanks go to Phil Karlton, of Digital's Western Research Lab, for the first implementation of this mechanism.

easily use UNIX domain sockets for the local case at the cost of a file descriptor. For current applications, the bandwidth limitations (of approximately 1 million bits/second on 780 class processor) is not major, though faster devices (and image processing applications) would probably benefit from implementation of a shared memory path between the X server and client applications.

Current shared memory implementations in variants of UNIX are not sufficient. Memory sharing primitives should allow appropriately privileged programs to both share memory with other processes and map to both kernel space and I/O space. Shared libraries (available in some versions of UNIX) would also increase the options available to window system designers (see below).

File Descriptors

Andrew, the window system developed at the ITC at CMU [1] uses one connection (file descriptor) per window. While simple from a conceptual level, also allowing naive applications to do output to a window, it ties an expensive resource (file descriptor and connection) to what should be a cheap resource (a window or sub-window). It requires more kernel resources in the form of socket buffers for each file descriptor. In addition, the handshaking required for opening a connection is expensive in terms of time and will become more so once connections become authenticated. The attraction of having a simple stream interface to a window can be had by other means [2]. In addition, if a window is tied to a file descriptor, the application loses the implicit time sequencing provided by the event stream coming over a single connection.

One X application uses more than 120 subwindows, all multiplexed over a single connection. One could postulate a single connection per client for input, and a single connection per window for output; with the limited number of file descriptors in 4.2BSD and other current versions of UNIX, this was eliminated as a possibility. Sixteen client programs seems to be sufficient for most people, (this is limited by 20 file descriptors on standard UNIX, with four file descriptors needed for X; one for the display, keyboard and mouse, two to listen for incoming connections, and one for reading fonts). Sixteen is not a tolerable limit on the total number of (sub)windows, however.

4.3BSD lifts this limit to sixty four. (It can be configured to any size.) While this increase in the number of file descriptors is beneficial, it is still too expensive a resource to use one per window.

Terminal Emulation

The current terminal emulator for X (*xterm*) is a client application, in principle similar to any other application. In practice, *xterm* is probably the most complex and least graceful part of the package. Pseudo teletypes (hereafter called pty's) are used to implement this in 4.2BSD. As currently implemented, ptys consist of a device driver which presents a terminal on one side and a master controlling device on the other side. Data is looped back from one side to the other, with full terminal processing occurring (tab expansion, cooked/raw mode, etc.)

These present a number of problems: 1) pty's are a limited resource. Typical systems have 16 or 32 ptys. On a single user machine, this limit is seldom reached, but on a timesharing machine it can be inconvenient. 2) Since they appear statically in the file system, protection on the tty/pty pairs can be a problem. A previous process that terminates unexpectedly can leave the pty in an incorrect state. *Xterm* is the only application that must run set user id to root to guarantee it can make the tty/pty pair properly accessible and to set ownership on the slave to the user.

The net result is that *xterm* is the most UNIX dependent (and least likely to port between UNIX implementations) of any of the X clients currently existing. Dennis Ritchie's [3] stream mechanism appears to eliminate most of these problems by allowing stacking of terminal processing on IPC.

Window System Initialization

Most displays capable of running a window system bear little resemblance to UNIX's model of a terminal connected by a serial line. Current display hardware may require involved initialization before it is usable as a terminal, and may have an interface that looks nothing like the

conventional view of a serial device. As soon as the window system is running, however, it is easy to provide a terminal emulator to a user.

Unix currently realizes someone has logged out by the eventual termination of the process started by *init*(8). *Init* is also the only process which can detect when an orphan process terminates, so the restart of a terminal line (or window system) after logout can only be performed by *init*.

The solution taken to support X (or Andrew, which has a similar structure) was to generalize *init*. *Getty*(8) or (in X's case, *xterm*) now opens and revokes access to a terminal or pty rather than *init*. The format of the */etc/ttys* file, already extended at Berkeley, was further extended to allow the specification of an arbitrary command to be run as *getty*. For X, this would normally be the terminal emulator. *Init* will also restart an optional window system server process associated with the pty. *Init* must start this process, since *init* is the only process in UNIX that can detect its exit. The initial *xterm* can not be started from a window system server, since the server must exist all the time, and *init* has to know the process id in order for it to detect the *login* process has terminated. The X server process itself opens the display device and performs whatever initialization may be required (for example, the Vs100 requires loading with firmware stored in a file).

Once *xterm* starts execution, it exec's *getty* on the slave side of the pty, and a user can log in normally. When the user's shell exits, *xterm* exits, and *init* can then detect the user has logged out normally.

Init can now be used to guarantee that a process will be kept running despite failures as long as the system is multi-user. Another approach not seriously examined would have made it possible for an orphan process to have a parent other than *init*.

Resource Location and Authentication

At this time, UNIX lacks good network authentication and resource location. The only example of a real name server in widespread use is the internet name server. As UNIX moves toward a distributed systems environment, questions of distributed resource location become important. X at this time does little to solve this problem, relying on either command line arguments or an environment variable to specify the host and display you want the application to use. In reality, it should be closely tied to the user's name, since the name of a machine is basically irrelevant as users often move. X seems to highlight some issues in the future design of such servers that may not be widely appreciated.

The model used to best describe distributed computing goes under the name of the "client/server" model. That is, a client program connects to a "server" which provides a service somewhere in the network. The additional twist is that the window system is a "server" in this model, and other network services may become "clients" of the X server. For example, one can envision using services that want to interact with the user's display. The result is that the "name" of the X server must somehow propagate through such service requests, along with whatever authentication information may be required to connect the X server in the future. This "cascaded" services problem has not been well explored.

The access control currently in X requires no authentication, but is only adequate for workstations, and fails badly in an environment which includes timesharing systems. X can be told to only accept connections from a list of machines. Unfortunately, if any of them are timesharing machines, and you allow access from that machine, then anyone on that machine may manipulate your display arbitrarily. This has the unfortunate side effect of making it trivial to write password grabbers (across the net!) or otherwise disturb the display if access is left open.

The "name" of the user's display server also comes and goes with some frequency, as each time you log out, any previously authenticated connection information needs to be invalidated, so no background process from a previous user will disturb the user's display. It is also not uncommon that a single user may use multiple displays, possibly on multiple machines simultaneously. This might be common, for example, in a laboratory environment. Interesting questions arise as to which display to use on what machine. (For example, the user may initiate a request on a black and white

display that really works better on a color display; which display on what machine should be used?) We do not believe these issues, in particular the transient and cascading nature of such display services and authentication information, have been properly taken into account in the design of resource location and authentication servers.

Stub Generators and the X Protocol

The X protocol is not a remote procedure call protocol as defined in the literature [4,5], as client calls are not given the same guarantee of completion and error handling that an RPC protocol provides. The X protocol transports fairly large amounts of data and executes many more requests than typically seen in true RPC systems. Given this generation of display hardware and processors, X may handle greater than 1000 requests/second from client applications to a fast display.

X clients only block when they need information from the server. Performance would be unacceptable if X were a synchronous RPC protocol, both because of round trip times and because of system call overhead. This is the most significant difference between X and its predecessor W, written by Paul Asente of Stanford University. On the other hand, a procedural interface to the window system is essential for easy use. We spent much time crafting the procedure stubs for the several library interfaces built during X development.

The original implementation of the client library would always write each request at the time the request was made. This implies a write system call per X request. There was implicit buffering from the start in the connection to the server due to the stream connection. Over a year ago, we received new firmware for the Vs100, and were no longer able to keep up with the display. We changed the client library to buffer the requests in a manner similar to the standard I/O library; this improved performance dramatically, as the client library performs many fewer write system calls.

Many current RPC [6] argument martialing mechanisms perform at least one procedure call per procedure argument to martial that argument. This is almost certainly too expensive to use for this application. Even if martialing the argument took no time in the procedure, the call overhead would account for ~10% of the CPU. Stub generators need to be able to emit direct assignment code for simple argument types. Complex argument types can probably afford a procedure call, but these are not common in the current X design.

Proper stub generation tools would have saved several months over the course of the project, had they been available at the proper time. Arguments could be made that the hand-crafted stubs in the X client library are more efficient than machine generated stubs would have been. On the other hand, to keep the protocol simple, X often sends requests with unused data, for which it pays with higher communications cost. It would be instructive to reimplement X using such a stub generator and see the relative performance between it and the current mechanism.

Machine dependencies in such transport mechanisms need further work. The protocol design deserves careful study. Issues such as byte swapping cannot be ignored. With strictly blocking RPC, the overhead per request is already so high that network byte order is probably not too expensive, given the current implementation of RPC systems on UNIX. With the higher performance of the X protocol, this issue becomes significant. It is desirable that two machines of the same architecture pay no penalty in performance in the transport protocols. Our solution was to define two ports that the X server listens at, one for VAX byte order connections, and one for 68000 byte order connections. At a late stage of X development, after X client code had already been ported to a Sun workstation and would interoperate with a VAX display, another different machine architecture showed that the protocol was not as conservatively designed as we would like. Care should be taken in protocol design that all data be aligned naturally (words on word boundaries, longwords on longword boundaries, and so on) to ensure portability of code implementing them.

X would not be feasible if round trip process to process times over TCP were too long. On a MicroVAX† II running Ultrix‡, or on a VAX 11/780 running 4.2, these times have been measured between 20 and 25 milliseconds using TCP. As this time degrades, interactive "feel" becomes worse,

† VAX is a trademark of Digital Equipment Corporation.

as we have chosen to put as much as possible in client code. Birrell and Nelson report much lower times using carefully crafted and tuned RPC protocols on faster hardware; even extrapolating for differences in hardware, UNIX may be several times slower than it could be. Given a much faster kernel message interface, one should be able to improve on the current times substantially. The X protocol requires reliable in order delivery of messages.

The argument against using such specific message mechanisms are: 1) the buffering provided by the stream layer is used to good advantage at the server and client ends of the transmissions. 2) Less interoperability. X has been run over both TCP and DECNET, and would be simple to build a forwarder between the domains if needed. This reduces the number of system calls required to get the data from the kernel at either end, particularly when loaded.

These times have been improved somewhat by optimizing the local TCP connection, and could be further improved by using UNIX domain connections in the local case.

In general UNIX needs a much cheaper message passing transport mechanism than can currently be built on top of existing 4.2BSD facilities. Stub generators need serious work both for RPC systems and other message systems particularly in light of some of the issues discussed above. We would make a plea that there be further serious study of non-blocking protocols[7]. There should be some way to read multiple packets from the kernel in a single system call for efficient implementation of RPC and other protocols.

Select and Non-blocking I/O

Without select(2), building X would have been very difficult. It provides the only mechanism in UNIX for multiplexing many requests in a single process. It is essential for the X server to be able to block while testing for work to do on any client connection and on the keyboard device. X will then wake up with the information required to determine which device or connection needs service.

In actual interactive use of X on a very fast display, select accounts for both the most CPU time and the most subroutine calls. Over an afternoon's use on this display, it accounted for more than 20% of the CPU time used. This is not surprising, since most use of the window system is generated by input events going to editors (in our environment), and output character echoing as well as clock and load monitor graphics calls. When not loaded, one would expect on the order of one select call per X request performed. In fact, there are approximately two X requests performed per select call.

One should remember that select's overhead diminishes as the load on the window system increases, both because you are likely to have many requests on a single connection, and because multiple connections may be processed on a single call. Profiling of the server when the display is loaded shows select using a much smaller percentage of the total CPU time.

Note that for the typical case under normal use, TWO system calls will be occurring where one might potentially do. In the output case (from a client), X will be blocked in select awaiting input (one call). It must then read the data from the client and process it (second call). Due to the shared memory described above, we are avoiding a write system call to the display. On input (keyboard or mouse), X will be blocked in select (one call). It then gets the input event out of the queue, determines which client should get the event, and writes it (second call). Again, we have saved a system call to read the data. Note that since buffering may occur on both input and output, the overhead per graphics operation performed will diminish as the load on the server goes up, since the server will perform more work for the same amount of overhead.

Optimally, select should be very cheap. On fairness grounds, one would like to see if more input from a different client is available after each X request. The original X request handler would check after every request for more requests. The current scheduler only checks for more input when all previously read data has been processed, and provides an approximately 30% reduction in X server overhead (all in the select and read system calls).

‡ Ultrix is a trademark of Digital Equipment Corporation.

Summary

The current UNIX kernel implementation is quite inflexible, closing off what might be interesting design choices. Lightweight processes both in the kernel and in user processes could be used to good advantage. The kernel is not properly structured to allow easy use of different facilities together. Streams may be a decent first step in this direction.

Stub generators, message passing and RPC transport protocols all need substantial work as UNIX moves into the distributed world. Using these protocols without stub generators is like a day without sunshine.

Resource location, authentication and naming are issues UNIX has not faced in the distributed environment. Cascaded services present another level of issues which need to be faced in their design.

UNIX has ASCII terminals ingrained into its very nature. It will take much more work to smooth the rough edges emerging from the forced marriage of workstation displays with UNIX.

If a system resource is in short supply (as file descriptors are), the correct solution is to lift the limit entirely. Doubling or tripling a limit on a resource only delays the day of reckoning, while still preventing those design strategies that found them in short supply originally.

Shared memory should allow sharing of memory between processes, between the kernel and a process, and between a process and hardware. Shared libraries would open up design opportunities.

More work needs to be done on performance of some of the new kernel facilities. The X server uses select more heavily than any other system call, accounting for the largest single component of CPU time used, though select is not the limit in absolute performance.

Acknowledgements

Without Bob Scheifler of MIT's Laboratory for Computer Science, there would be no X window system.

The list of contributors is now too long for an exhaustive list, and includes Paul Asente, of Stanford University, Mark Vandevoorde, Tony Della Fera, working for Digital at Project Athena, Ron Newman of Project Athena, the UNIX Engineering Group and the Workstations group of Digital. My thanks also go to Sam Leffler, Steve Miller and Noah Mendelsohn for helpful comments during the writing of this paper. My thanks also go to Branco Gerovac for Figure 1.

References

- [1] Gosling, J. and Rosenthal, D. "A Window-Manager for Bitmapped Displays and UNIX," to be published in *Methodology of Window-Managers*, F. R. A. Hopgood et al (eds) North-Holland.
- [2] Newman, R., Rosenthal, D., Gettys, J. "User Extensible Streams," In preparation.
- [3] Ritchie, D. M., "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, Part 2, pp. 1897, October 1984.
- [4] Birrell, A. D. and Nelson, B. J., "Implementing Remote Procedure Calls," *Transactions on Computer Systems*, vol. 2, no. 1, February 1984.
- [5] Nelson, B. J., "Remote Procedure Call," *Technical Report CSL-81-9*, Xerox Palo Alto Research Center, 1981.
- [6] "Sun Remote Procedure Call Specification," Sun Microsystems, Inc. Technical Report 1984.
- [7] Souza, R. J. and Miller, S. P., "UNIX and Remote Procedure Calls: A Peaceful Coexistence?," Project Athena Internal Paper, 1985.

SUNDEW: A Distributed and Extensible Window System

James Gosling

Sun Microsystems

ABSTRACT

SUNDEW is a distributed, extensible window system that has arisen out of an effort to step back and examine various window system issues without the usual product development constraints. It should really be viewed as research into the *right* way to build a window system. The key unique feature of SUNDEW is the ubiquitous use of an extension mechanism. The extensibility of the system has proven to be crucial to its functioning well in a distributed environment. Performance is enhanced through closer interaction between client and server; data compression on the communication channel can be done in an application-specific way; semantic issues are cleared up by having a centralized authority; and user interface changes are easier to make.

The paper is organized as two parts. The first provides some background to motivate the design of SUNDEW, and the second presents the design. Several aspects of the design are rather unusual, and hence need a lot of motivation.

1. Background

There is a wide range of flexibility in window systems. On the one extreme are systems like Andrew¹ and the Macintosh² where essentially nothing can be changed in either the user or programmer interface. In the middle are systems like X³ which have provisions for new menu packages or new layout managers, but where the difficulty of exploiting the flexibility is fairly high. At the other end are totally open systems like Smalltalk⁴ where it is fairly trivial for a skilled user to modify parts of the system's behavior.

Take as an example what you must do to change the background grey pattern on the window system's desktop. On the Mac, this is easy because someone thought to include it as a configuration option. On the other hand, if the scrollbar grey needs to be changed, you're stuck. With Andrew, since changing the background grey isn't a configuration option you're stuck unless you can get at source. X is somewhat better since it is more broken up, but you're still faced with rebuilding a large part of the system. Smalltalk makes it fairly easy since the component of the window system that deals with the background grey is small and well-contained and can be replaced incrementally without disturbing the things around it. The hard part is finding out which piece to replace and what its specification is. Smalltalk systems generally have the full source available along with a powerful browsing facility: this makes the task possible and easy, but only for a skilled person.

Window systems have a wide range of complexity in their user interfaces. Some, like the Macintosh, have very simple and clean interfaces that are easy for novices to learn. The Andrew window system has a very simple style that is easy to teach, easy to use, and easy to document; but this simplicity comes at the cost of a more rigid system. In some window systems experienced users find that all the help and menu interaction can get in the way, so at the other end of the scale are systems that are tuned to expert use but which novices find hard to learn. Systems are rarely at one of these extremes: they usually have accelerators for expert users or simple menu interfaces for novices. The one thing that is clear is that no interface style is satisfactory or even adequate in all situations.

Similar comments can be made about the programming interfaces to window systems. Simple interfaces often make unusual operations difficult; it can become necessary to take pliers to the beast and bend it in unintended directions. For instance, in the Andrew system, direct program manipulation of bitmaps is almost impossible. In the SunWindows⁵ system it is impossible to avoid. Powerful interfaces tend to be baroque. This is partly an inherent problem, and partly due to the tendency of systems to accrete features as they mature. The best compromise seems to be an interface that can be viewed in parts, starting out at a simple but complete base, and having complexity that can be incrementally learned.

Another sort of flexibility that varies widely between window systems is their device independence. Many window systems start out being intended for a particular technological base, and the assumptions built into that base often creep into the higher levels of the design. A common problem is the use of the 'bitblt' graphics model. While this deals fairly well with monochrome displays, it doesn't extend in a clean and useful way to color. Boolean combination functions between color pixel values don't make much sense. For instance, one often draws transient rubber band lines by XORing them with the image. XORing color map indices can lead to some pyrotechnic effects.

Most window systems are initially built for a particular piece of hardware. Decisions tend to be made less in favor of what is 'right' and more in favor of what fits in with the hardware at hand. A good example of this is the X window system. It has been going through substantial growing pains as it has developed. X started life as a window system for VAXes with VS100 displays. The communication protocol between the X server and client programs was based on C structures, whose internal representation was very VAX-specific. It also started out using the VS100 font format. Unfortunately, the VS100 font format has some major technical problems, and the VAX C structures don't map well to other machines. The process of cleaning out these VAX-specific aspects has taken quite a while.

Andrew is a good example of a window system that was designed without a specific piece of hardware in mind. This was an accident of the political situation at the time it was being written: the hardware that it was being designed for didn't exist, hadn't really been designed, was being designed in relative isolation from the design of Andrew, and, in fact, there were several display designs going on simultaneously. Andrew was designed for a black box; all that was known about the eventual system was that it would run some kind of Unix and that it would have some kind of bitmap display. At the time, this was a very painful situation, but in retrospect, it was a great blessing.

The correct choice of a graphics model is crucial to achieving device independence. The more abstract the model, the more room there is for the underlying implementation to accommodate different technologies. For example, consider the representation of color. There are three common ways that color is represented in display devices: 1-bit black and white (constant small set of colors); 8-bit color with a colormap (variable small set of colors); and 24-bit color (all possible colors available everywhere). Integrating the views of color that these three implementations present is a very hard but important problem.

The choice of a graphics model has a strong impact on the usefulness of the window system for doing graphics. Many systems provide only rasterop, vector drawing, and simple text. On the other hand, systems like the Macintosh which have a much richer graphics model, have a flair for much more graphically interesting applications. This is a balancing act: richer models are more difficult to implement and more difficult to understand.

In the kind of distributed networked environment that Sun promotes, it is natural to want to be able to access windows on another machine as naturally as the Network File System supports accessing remote files. The experiences with Andrew and X have shown that the flexibility that this allows in the choice of where a client program runs is very valuable. Non-networked systems like Smalltalk or SunWindows have, by contrast, a very closeted feeling.

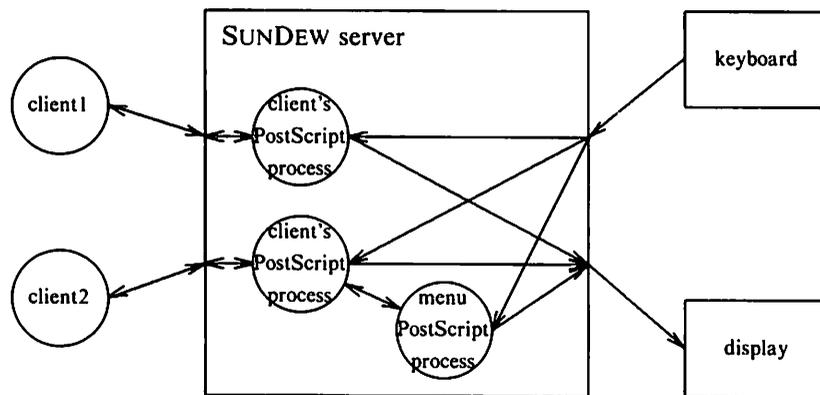
2. The Design

SUNDEW is based on a novel sort of interprocess communication. Interprocess communication is usually accomplished by sending messages from one process to another via some communication medium. Messages are usually a stream of commands and parameters. One can view these streams of commands as a program in a very simple language. What happens if this language is extended to being Turing equivalent? Programs don't communicate by sending messages, they communicate by sending programs that are elaborated by the receiver. This has interesting effects on data compression, performance and flexibility.

The POSTSCRIPT programming language defined by John Warnock and Charles Geschke at Adobe Systems is used in just this way.⁶ What Warnock and Geschke were trying to do was communicate with a printer. They transmit programs in the POSTSCRIPT language to the printer that are elaborated by a processor in the printer, and this elaboration causes an image to appear on the page. The ability to define a function allows the extension and alteration of the capabilities of the printer.

This idea has powerful implications within the context of window systems: it provides a graceful way to make the system much more flexible, and it provides some interesting solutions to performance and synchronization problems. For example, if you want to draw a grid, you don't have to transmit a large set of lines to the window system, you just send down a loop. Downloading programs to the server is not just a nice feature that has been tacked on: it's an integral part of the window system.

POSTSCRIPT is the extension language used by SUNDEW. It is a clean and simple language, it has a well-designed graphics model, and it is compatible with most of the printers that Sun supports. SUNDEW is structured as a server which contains a POSTSCRIPT interpreter. Within this server process is a collection of lightweight processes that execute POSTSCRIPT and C programs. Client programs talk to SUNDEW through byte streams (4.2 BSD sockets). Each of these streams generally has a lightweight POSTSCRIPT process within the SUNDEW process that executes the stream.



Messages pass between client processes, that exist somewhere out on the network, and POSTSCRIPT processes that exist within the SUNDEW server. These processes can perform operations on the display and receive events from the keyboard. They can talk to other POSTSCRIPT processes that may, for example, implement menu packages.

Everything in SUNDEW is centered around POSTSCRIPT as an extension language. All that is provided by SUNDEW is a set of mechanisms; policies are implemented as POSTSCRIPT procedures. For example, SUNDEW has no window placement policy. It has mechanisms for creating windows and placing them on the screen given coordinates for the window. The choice of those coordinates is up to some POSTSCRIPT procedure.

What is usually thought of as the *user interface* of a window system is explicitly outside the design of this window system. *User interface* includes such things as how menu title bars are drawn and whether or not the user can stretch a window by clicking the left button in the upper right hand

corner of the window outline. All these issues are addressed by implementing appropriate procedures in POSTSCRIPT.

The rest of this paper presents SUNDEW in four parts: the imaging model, window management, user interaction, and the client interface. The imaging model refers to the capabilities of the graphics system — the manipulation of the contents of a window. Window management refers to the manipulation of windows as objects themselves. User interaction refers to the way a user at a workstation will interact with the window system: how keystrokes and mouse actions will be handled. The client interface defines the way in which clients (programs) will interact with the window system: how programs make requests to the window system.

2.1. Imaging

Imaging in SUNDEW is based on the stencil/paint model, essentially as it appears in Cedar/Graphics⁷ and POSTSCRIPT. A stencil is an outline specified by an infinitely thin boundary that is piecewise composed of spline curves in a non-integer coordinate space. Paint is some pure color or texture — even another image — that may be applied to the drawing surface. Paint is always passed through a stencil before being applied to the drawing surface, just like silkscreening. This is the *total* model: lines and characters can be defined using stencils. Lines are done as narrow stencils. Underneath it all, it isn't really done this way: special cases are exploited wherever possible. One can think of a stencil as a clipping region. Stencils may be composed by union, intersection and difference to create new stencils.

One of the attractive characteristics of this imaging model is its very abstract nature. For example, the definition of a font allows many implementations: as bitmaps, as pen strokes, or as spline outlines. No commitment is made about exactly which pixels are affected, or even that there are pixels at all. The extension of the system to deal with anti-aliasing will not affect the interface.

The specification of this model is simple and elegant, but the way in which its various features can be combined leads to a very tricky implementation. For example, the mechanism for specifying a stencil allows straight lines, arcs and higher order curves to be a part of its boundary. Stencils can be used both for clipping and for filling. This implies that it must be possible to compute the intersection of curved boundaries. This is difficult, but possible, to do fast.

The work done by Vaughan Pratt on Conic Splines provides a fast way to deal with the generation of curves.⁸ A further set of algorithms for putting these curves together and dealing with the various operations on shapes that results has been developed.⁹

2.2. Window management

The basic windowing object is something called a *canvas*. This nonstandard term was picked to avoid the semantic confusion that surrounds the word 'window'. A canvas is just a surface on which an image may be drawn. A set of canvases, called a scene, can be laid out in three dimensions on a display surface. The actual implementation of canvases depends heavily on the graphics package described in the previous section. Each canvas is made up of two parts: one on the screen, and one not. By playing with these two parts one can get double-buffered, retained and non-retained behavior.

Canvases are cheap and easy to create. Menus, windows and pop-up messages are all based on canvases. POSTSCRIPT has been extended with primitives to create and manipulate canvases. All POSTSCRIPT graphics operations are performed on some canvas.

2.3. User interaction – Input

Each possible input action is an *event*. Events are a general notion that includes buttons going up and down (buttons may be on keyboards, mice, tablets, or whatever else) and locator motion.

Events are distinguished by where they occur, what happened, and to what. The objects spoken about here are usually physical, they are the things that a person can manipulate. A example of an event is the ‘E’ key going down while the mouse is over canvas *x*. This might trigger the transmission of the ASCII code for E to the process that created the canvas. These bindings between events and actions are very loose; they are easy to change.

The actions to be executed when an event occurs can be specified in a general way, via POSTSCRIPT. This strongly resembles the *squeak* language, with lightweight processes replacing concurrency compilation.¹⁰ The triggering of an action by the striking of the ‘E’ key in the previous example sends a message to a POSTSCRIPT process that is responsible for deciding what to do with it. It can do something as simple as sending it in a message to a Unix process, or as complicated as inserting it into a locally maintained document. POSTSCRIPT procedures control much more than just the interpretation of keystrokes: they can be involved in cursor tracking, constructing the borders around windows, doing window layout, and implementing menus.

2.4. Client interface

A client program exists in two parts: one that is written in POSTSCRIPT and lives inside SUNDEW, and one that lives outside SUNDEW and talks to it through a byte stream. This leads to a number of levels at which the client interface can be viewed. At the lowest level, the programmer is writing POSTSCRIPT programs and is dealing with an entirely POSTSCRIPT universe. Menu packages and window layout policies are examples of objects that will usually be implemented this way.

One step above that, the programmer is writing programs in C, or some other language, that write POSTSCRIPT programs – the programmer is explicitly aware of the existence of POSTSCRIPT. SUNDEW emulators for other window systems are generally implemented this way.

The highest level, and the one at which most programmers deal, the existence of POSTSCRIPT and message passing is completely hidden by an interface veneer. The flexibility of POSTSCRIPT allows this veneer to have many possible appearances: it can emulate other window systems like X or Andrew.

3. An Example

This example defines a POSTSCRIPT function that pops up a message on the screen under the mouse and removes the message when the user clicks a mouse button on it.

```
/popmsg { { HighlightFont setfont
          dup stringwidth pop 15 box
          createcanvas setcanvas
          0 3 moveto show
          currentcanvas currentlocator movecanvas
          /MouseButtonUp enableevent
          waitevent
        } fork pop
      } def
```

POSTSCRIPT is a completely postfix language. There is an operand stack from which arguments are taken and onto which results are pushed. Literal numbers like 0, and literal names like */popmsg* are just pushed on the stack. Names which correspond to variables, like *HighlightFont*, have their values pushed onto the stack. The construction ‘*{stuff}*’ defines a program block whose contents is *stuff*. */popmsg {...} def* defines the function *popmsg* by first pushing the name *popmsg* on the stack, then a program block, then calling *def* to define the function based on these two arguments.

SetFont takes a single argument, a font, and makes it be the current font. *Popmsg* is written to take a single string argument on the top of the stack. *Dup* duplicates this argument and *stringwidth* calculates its width. *Pop* throws away the y component of the width. *Box* uses the width and 15 to set the current path to be a box with those dimensions. *Createcanvas* then creates a canvas of that shape, which is installed as the current canvas by *setcanvas*. *0 3 moveto show* moves to the starting position of the string and draws it. *currentcanvas currentlocator movecanvas* draws the canvas on the display positioned by the mouse. The *MouseButtonUp* event is then enabled and waited for. This whole block of code is run as a separate process so that it doesn't block the rest of the system and so that its canvas and font get cleaned up automatically when the mouse is clicked. The *pop* gets rid of the unwanted process handle.

4. Conclusion

Extensibility has proven to be very beneficial in the construction of a distributed window system. It can be used to improve both the effective bandwidth and latency of the communications network. The bandwidth improves by tighter encoding of the operations and the latency improves by reducing the number of situations where messages need to be sent. The flexibility it provides, even in a non-distributed environment, allows a clean separation of policy and mechanism, which aids in user interface design.

1. David Rosenthal and James Gosling, "A Window Manager for Bitmapped Displays and Unix," in *Methodology of Window Managers*, ed. F. R. A. Hopgood et al., North Holland (To be published).
2. C. Espinosa and C. Rose, *QuickDraw: A Programmer's Guide*, Apple Computer (March, 1983).
3. James Gettys, "Problems Implementing Window Systems in Unix," *Usenix Proceedings* (January, 1986).
4. Adele Goldberg and David Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley (May, 1983).
5. *Programmer's Reference Manual for SunWindows*, Sun Microsystems (April, 1985).
6. *POSTSCRIPT Language Reference Manual*, Addison-Wesley (July, 1985).
7. John Warnock and Douglas Wyatt, "A Device Independent Graphics Imaging Model for Use with Raster Devices," *Computer Graphics* 16(3) (July, 1982).
8. Vaughan Pratt, "Techniques for Conic Splines," *Siggraph Proceedings* (July, 1985).
9. James Gosling, "If the Earth is Round, Why is the Sun Square," *Usenix Workshop on Computer Graphics* (December, 1985).
10. Luca Cardelli and Rob Pike, "Squeak: A Language for Communicating with Mice," *Siggraph Proceedings* (July, 1985).

UNIX on Big Iron

January 16, 1986

Denver, Colorado

User Requirements for UNIX† on “Big Iron”

E. N. Miya

Computational Research Branch
NASA Ames Research Center
Moffett Field, CA 94035
eugene@ames-nas.ARPA
UUCP: {ihnp4,hplabs,hao,decwrl}!ames!amelia!eugene

ABSTRACT

The UNIX operating system was developed during an unusual period when mini-computers were prominent. This is the source of many problems hampering the scale-up of the system. Some historical perspective is useful for determining near-term and far-term problem areas of UNIX running on large-scale systems including multiprocessors, increased file system size, new performance criteria, and the like.

Recently, a user was asked ‘what will the operating system of the 22nd Century would look like’ He said, “I don’t know what it will look like, but it will be called UNIX.”

>From an anonymous computer rag in the year 2085.

Introductory Comments

The development of the UNIX operating system [1] was largely “user”-driven. Users, as history has frequently shown, have tended to demand ever-increasing quantities of computer speed and storage (typically at rates faster than $O(n)$). This placed a greater burden on UNIX to operate in new and unintended environments. These environments were identified by Thompson and Ritchie years ago in a now ancient (classical) issue of the *Bell System Technical Journal*: e.g., multiprocessors, [2] high-speed input/output (I/O) devices, [3] and other realms. This demand strains traditional resources like the file system.

It is ridiculous to think that file system size, for instance, should be bounded when there exist requirements for greater speed and storage. Dennis Ritchie in his Turing award lecture [4] even mentions the possibly of “a future Ken Thompson finding a little used CRAY-1‡ ...” Ironically, Ken and Dennis have just received a CRAY-X/MP.

The bottom line for any discussion of Big Iron? must be:

We must not sacrifice performance (either human or machine)!

The intention of this paper is to contrast the large-scale environment to the UNIX environment. Specific features mentioned in the original Thompson and Ritchie paper [1] are briefly analyzed from the large-scale perspective.

Big Iron

What constitutes “Big Iron?” The dividing line is difficult to draw: it is beyond a super-minicomputer, and less than a “mainframe.” Both of these are vaguely defined terms. Many

†UNIX is a trademark of AT&T Bell Laboratories.

‡CRAY is a trademark of Cray Research, Inc.

new multiprocessors are composed of collections of microprocessors: they are certainly interesting. To avoid ambiguity, it is best to discuss user requirements at the high-end to understand big iron requirements.

Supercomputers are defined as the fastest machines existing at a given point in (relative) time. Supercomputer applications tend to have extreme requirements for both memory and speed. These programs cannot trade time for space or vice versa. These applications are not restricted to "scientific programming." Big-iron skeptics might question the above statement. Fortunately, a popular, easy to understand example exists in computer graphics: making high-quality, "Lucasfilm" or "Star-Wars*" movies.

Consider the development of spatial resolution in computer graphics. Frame buffers started with 256 by 256 pixel resolution. Color was added, so more bits were added. Next, spatial resolution increased from 256 pixels to 512 pixels per side, then to 1024 pixels, and now we have 2048 pixels. Resolution increases (at a rate of $O(n^2)$), but aliasing problems still exist for big screen cinema. Aliasing can only disappear with a combination of adequate resolution (greater than 2048 pixels) and anti-aliasing techniques.

Note that the design of resolution went up linearly, but the storage requirements squared with each linear increase. The execution time may similarly square. Suppose the film-maker decides that 16K by 16K resolution is needed. Improving the resolution a factor of 64 (over 256 pixels) yields a 4096 fold increase in processing time and storage. The solution is to make the execution time constant (ideal), linearly increasing, or at least, increasing at a manageable non-linear rate.

The motion picture example could easily generalize to an image or signal processing problem, or even modeling the motion of air/space craft. If realism is a goal, why not model more than the surface characteristics using the real equations of motion? The processing here may easily exceed $O(n^3)$.

One computing faction says that large systems should be used in a "batch" mode. Another faction says that interactive computing is the wave of the future. The batch-proponents argue the problem with interaction is the cost of interrupt handling is too expensive. The interaction-proponents counter that the bulk of data is larger than the ability for humans to comprehend. Interaction and graphics are now required to do large-scale computation.

The reality will probably come with a middle ground, perhaps, a transaction type of distributed system with some local interactive processing and more powerful remote facilities.

UNIX?

Our perspective might be a little different if Bell Labs had approved Thompson and Ritchie's request for a DECSYSTEM-10 or if they had used an Interdata 8/32 for their first CPU. The UNIX operating system was developed on minicomputers: now an uncommon backwater of computing. The design span of minicomputers was short, and it cast an uncommon perspective. The microcomputer community sees UNIX as an operating system for "big" machines, and the mainframe environment sees UNIX as a system for "small" machines. Nobody is designing 16-bit minicomputers any more. Our problem comes from the terminology (prefixes really) of computing. Technology overpowered terminology.

Another problem is that microcomputers are gaining more computing power and catching up to if not exceeding the power of many past mainframes. Cycle time improves first. The next area is memory hierarchy. Storage requirements are beginning to drive computer requirements as much as speed which in turn requires more speed to process the large memories. Big disks in 1970 were measured in the tens of Megabytes. Big mass-storage is now measured in the

*Lucasfilm and Star-Wars are probably trademarks of Lucasfilm, Ltd.

terabytes.

The fundamental debate regarding the use of large expensive computers raises a question from non-UNIX people:

Is an operating system designed for yesterday's 16-bit mini-computer appropriate for the 64-bit supercomputers of today and/or the 64-bit ultra(?) micros of tomorrow (having gigaword addressing and teraword secondary storage)?

This question can be broken into several subproblems: reliability, economy of scale, and existing problems.

Computational reliability is one concern. Today's fault-tolerant computers are adequate for the jobs for which they are designed: transaction and switching systems. Computation designed to run a lifetime (i.e., the set of all computations for which the answer is 42) need some restart facility without extensive user coding. Computation of this type is becoming more frequent in the engineering and science domains.

The *scale-up* in both hardware and software is probably the most difficult issue to address. Is an operating system for 1 to 4 CPUs adequate for 16? for 100? for 1,000? or for 1,000,000 CPUs?

On a more technical level, how closely will the operating system kernels of today match those of 20 years from now? [i.e., UNIX the FORTRAN of operating systems] Will the functionality be identical? Will UNIX run on radical data flow architectures? Will UNIX run on the CRAY (or 370) on a desk? On the wrist?

Every UNIX programmer has his or her own set of gripes: better I/O, problems with small critical sections, and so forth. The remainder of this paper briefly surveys these problems from the perspectives of the original six features that made UNIX popular.

A Review of Six Features

It is said that no single feature accounts for the popularity of UNIX, but rather, in the combination of these six features mentioned by Thompson and Ritchie in their seminal paper: [1]

- A hierarchical file system incorporating demountable volumes
- Compatible file, device, and inter-process I/O
- The ability to initiate asynchronous processes
- System command language selectable on a per-user basis
- Over 100 subsystems including a dozen languages
- High degree of portability

The issues facing the UNIX operating system are prefaced in the light of these six features. These features have sometimes succeeded, while in other cases, they have reached limitations far beyond the minicomputer environment for which the system was developed. The following sections summarize the apparent success and future problems faced by each UNIX feature.

Portability

High degree of portability --
Clearly successful, but . . .
System portability/high-level language -- Yes
Applications portability -- less so

Portability is mentioned first because it is the first issue any new UNIX implementation faces. There is beauty in system portability, and the degree of UNIX portability cannot be questioned as this is the first operating system to be suggested as a standard. No other operating system can yet make the claim that it spans the smallest to the largest machines.

Our first concern involves getting the operating system running on a new piece of hardware. The second concern involves moving software tools and running applications. We

have amassed significant quantities of experience with the former. A programmer can open(2) a file (i.e., *dataset* if you come from the IBM-world) regardless of the underlying hardware. The latter is still a major problem.

Porting an application involves analysis of the intended operating environment. Programming environments and languages are made much less portable if the operating environment differs significantly between machines. The different non-UNIX ways of opening files require a slew of different source code and job control language changes. UNIX should standardize this.

So then, why are C portability classes and discussions suddenly cropping up? Is C less portable than touted? Probably. It is now used in a wider variety of environments than probably intended. Consider these novel environments:

- 16-bit, NUXI (reversed word) systems
- 32-bit, XINU systems (reversed byte-patterns)
- 36-bit, word-oriented Univacs
- 64-bit, byte oriented CDC systems
- 64-bit, word-oriented CRAY
- and other systems

Symptoms (features?) of non-portable code are `includes` and conditional compilation. The reader is given an exercise of considering some of the features which might appear in the following header files or machine specific code blocks:

```
#ifdef CRAY
#include <cray.h>
#ifdef CRAY2
#include <cray2.h>
#ifdef IBM370
#include <ibm370.h>
#ifdef IBM3090
#include <ibm3090.h>
#ifdef u1100
#include <u1100.h>
#ifdef iPSC5d
#include <iPSC5d.h>
#ifdef iPSC7d
#include <iPSC7d.h>

#include <4.2.h>
#ifdef 4.2BSD
#include <SV.2.h>
#ifdef SV.2
```

Ideally, code should not need any of the above conditions, but the reality is that we can expect more statements of this type to appear as we explore architectural diversity in the future. Currently, we can only try to isolate machine dependencies.

File System

A hierarchical file system -- Largely successful, but . . .

Success: the simplicity of the UNIX file system is helpful although many future UNIX-users do not fully appreciate this fact. There are four main issues: capacity, performance, structure, and reliability. The UNIX community must quickly address the issue of file system capacity. File system organization is not an immediate issue, but capacity limitations, large-scale I/O performance, and reliability are issues. Dennis Ritchie's advice [5] in the use hierarchical file systems was heeded by developers of new operating systems. Performance is covered in the next section.

Reliability is given considerable attention by the business community. This leaves capacity.

There are applications where a 16-gigabyte file space is not adequate. It is not simply a matter of saying that hydrodynamics is a special application or that users should use many small files. On a gut-level, the high-resolution Star Wars images are an example, again. 16K pixels per side represents 256 M pixels (words) total image for a composition. 'Cat'-ing pieces of an image together is a joke. The original developers of computers never thought there was much use for computers with greater than 8 Kwords of memory. We must not make their mistake.

One last thought regarding structure. Simple hierarchies may not have enough structure for organizing vast quantities of data. Scientific database systems are lacking in particular. Our imperfect movie analogy breaks down with its mostly pure sequential structure. This author is unable to comment on business database systems, but their requirements are probably also severe. Some compatibility between these supplementary organization schemes requires consideration.

Compatible I/O

Compatible file, device, and inter-process I/O -- Successful in principle, but

Ken Thompson [3] noted the potential inadequacies of the UNIX file system and I/O in his *Bell System Technical Journal* paper on implementation. Low latency devices, Thompson noted, would be problems. Devices with low latency like RAM-disks are now on the market. While the overall issue of compatible I/O was important, the issue of some degree of performance was also important.

Several non-UNIX operating systems, notably Cray's and CDC's, have used *disk striping* across spindles as a means of increasing disk I/O, but this technique has reached its limit. It is understood that some microprocessor users are considering this technique for higher disk throughput. So high-end machines pave the way for their less expensive peers.

The next problem with I/O and the file system occurs with extremely large processes (e.g., our Star Wars movie again). Accessing files from disk is painfully slow. As the volume of data increases, our capability to swap data out of secondary storage has not kept pace with main memory movement. This in turn has an effect on the UNIX scheduler which requires change (swap time taking longer than a time slice). The reader should visualize extremely large processes whose swap times might be measured in whole seconds. Partial swapping offers one answer to this problem. There are likely to many side-effect problems of this type which will affect UNIX.

Another major problem in the area of I/O is the increased use of high-speed communication networks. Large-scale systems are rarely, if ever, used in a stand-alone mode. Large quantities of data need to be moved from remote storage units or data collection devices and buffered (staged) for execution.

Software Tools

Over 100 subsystems including a dozen languages (software tools) -- Largely successful

There are two issues here: the tools that exist and those that need to be developed. The business community thinks it needs a COBOL compiler. The scientific community does need a better FORTRAN compiler with vectorization, multitasking, and other features.

In the first case, most end-users have amassed their own software tools. These require porting a better FORTRAN or COBOL compiler or whatever language was used before. A 'better' compiler is an example of the second type of tool. Once this happens, the end-user applications can follow. So there are a variety of obvious dusty deck related problems.

Remembering that user requirements increase with time, we can be certain that newer software tools must incorporate things which were not a concern in the easiest days of UNIX: better schemes for organizing data on top of the file system, more graphics tools, better network/distributed systems communication. With the hundreds of tools floating within UNIX, a navigation system would help the user to put tools together: a tool to help use tools.

Software tools on portable operating systems bring up some interesting issues. If company X (using UNIX) develops in-house package Y, should this package run on company Z's UNIX machine? There is some tendency for a manufacturer to customize UNIX, its surrounding packages, and the associated hardware as new features. This is the portability issue again. This is a touchy area, since hardware companies make their money selling boxes, not software.

Shells

System command language selectable on a per-user basis -- Largely successful

The problems of obscure command naming are well documented. [6] Obscure command names are typically not shell problems but utility-name problems that are distinct from specific shells or system calls. The *functional* capabilities of most UNIX shells are unmatched by most vendor operating systems. Thoughtful exposition on improving shell functionality is scarce. [7]

Users need a variety of more graphical and more intelligent (perhaps voice?) interfaces. UNIX had its earliest origins on a graphics system and graphical interaction has a degree of human-interaction that is unmatched.

Parallelism

The ability to initiate asynchronous processes (parallelism) -- Clearly successful

User-controllable parallelism became a issue when the Fifth-Generation Project announced it would solve its problems using parallelism. Highly parallel systems appear to be the only architectural way of increasing performance as designers reach the physical limits of semi-conductors: e.g., Silicon and Gallium. No further comment about this is necessary.

Conclusion

The UNIX operating system is unquestionably a successful system. To remain successful, historical issues must now be addressed to increase performance and adapt to new environments. This paper has outlined some of the areas where we may expect to see change to the operating system.

References

- [1] Dennis M. Ritchie and Ken Thompson, "The UNIX Time-Sharing System," *Communications of the ACM* **17**(7), pp. 365-375 (August 1974).
- [2] J. A. Hawley and W. B. Meyer, "MUNIX, A Multiprocessing Version of UNIX," M.S. Thesis, Naval Postgraduate School, Monterey, Cal. (1975).
- [3] K. Thompson, "UNIX Time-Sharing System: UNIX Implementation," *Bell Sys. Tech. J.* **57**(6), pp. 1931-1946 (1978).
- [4] Dennis M. Ritchie, "Reflections on Software Research," *Communications of the ACM* **27**(8), pp. 758-760 (August 1984).
- [5] D. M. Ritchie, "UNIX Time-Sharing System: A Retrospective," *Bell Sys. Tech. J.* **57**(6), pp. 1947-1969 (1978).
- [6] Don A. Norman, "The Trouble with UNIX," *Datamation* **27**(12) p. 139 (1981).
- [7] Rob Pike, "Shells, features and interaction," *net.unix* **4575@alice.UUCP**(17 Nov 1985).

Experience with Large Applications on Unix

**Bob Bilyeu
McNeill-Schwendler, Inc.**

Didn't make deadline. Copies available at the Conference.

UNIX Scheduling for Large Systems

Jeffrey H. Straathof, Ashok K. Thareja, Ashok K. Agrawala

Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

UNIX^{*} derives much of its power and versatility from its simple and elegant design. While simplicity and elegance have been hallmarks of UNIX, there is an important aspect of UNIX that does not exhibit these characteristics, i.e., the UNIX scheduler. A close examination of most implementations of UNIX reveals that the scheduling concepts are *ad hoc*, cumbersome, and full of hacks.

Most large computer systems require the management of a vast number of processes arising from a diverse population of users, applications, and requirements. This makes the role of a scheduler very important in a large system. This paper identifies the requirements imposed by a large system on a scheduler. The fundamentals of the UNIX 4.2BSD and UNIX 4.3BSD schedulers are described in detail along with their design and implementation drawbacks. These drawbacks are discussed in light of the scheduling requirements imposed by a large system. This discussion is followed by the description of a new scheduler for UNIX that has been designed and implemented at the University of Maryland. The new scheduler is aimed at eliminating the shortcomings of UNIX 4.2BSD and UNIX 4.3BSD schedulers.

* UNIX is a trademark of AT&T Bell Laboratories.

1. Introduction

Each version of UNIX succeeding the original has included implementation changes in the cpu scheduler. The collection of scheduler changes over the years has produced a UNIX 4.2BSD implementation that can consume as much as ten percent of total cpu time just recomputing priorities every second, as estimated in [2]. The current scheduler implementations show the signs of hacking, complete with outdated comments and messy code.

This paper explains the critical issues involved in large system scheduling and the fundamentals of the UNIX schedulers. The design and implementation shortcomings of the current UNIX schedulers are examined, and a description of the design and implementation of a new scheduler is presented.

2. Critical Scheduling Issues In Large Systems

In order to understand the impact of schedulers, consider the simplest single user environments which place very few demands on a scheduler. The scheduler in such an environment never has to decide as to which of many processes the cpu should be given. Even when the case of background processes is allowed, demands increase only by a small amount as all of the processes still belong to the same user and it is his work that is always being done. In short, in a single user environment the role of a scheduler is a) to manage multiple tasks, and b) to utilize the hardware in a manner that allows for maximum overlap of processing between different devices, thus providing the maximum throughput for a given system configuration.

As systems and environments grow to accommodate many users, and then to allow each user to have many active processes, scheduling demands increase dramatically. A large system can have a few thousand processes active simultaneously, requiring the

scheduler to make process selections from a much larger set of choices. A large machine has an instruction time of a few nanoseconds, thus a new process may have to be scheduled every few tens of microseconds. Furthermore, the scheduler must be flexible enough to operate under changing load conditions, since large systems serve both small and large user groups.

The increase in demands of a large system makes the scheduler a more significant factor in system performance. An effective scheduler must maintain a reasonable amount of responsiveness and utilize the hardware to optimize throughput so the best system performance can be achieved. Large system schedulers must also allow convenient ways to alter the usual priorities of processes so users can regulate the workload of a running system. These goals of effective schedulers must be met under the variety of system loads experienced by large systems and in the variety of user environments large systems reside. Let us examine each of these objectives of a scheduler for large systems.

Responsiveness

The interactive response time has always been an important performance measure of a large system. Large systems running UNIX place even more importance on the interactive response time, since UNIX supports primarily interactive processing.

Most interactive users prefer a small and stable response time. A small response time means that users can retrieve results quickly; a stable response provides a sense of performance predictability to the users. Before requests are made, users can accurately weigh the time needed to retrieve results against the benefits of the results. Thus, a primary goal of a large system scheduler is to provide users with a stable response time that is proportional to the size of user requests. Several studies have been conducted in which system responsiveness has been related to user goals such as productivity, creativity and satisfaction.

Throughput

The throughput of a system has also always been an important performance measure of a large system. Large systems not only support a few thousand active processes simultaneously, but also support a large number of peripheral devices. These devices, whose speeds and capabilities vary, make the task of reaching optimal throughput more complex. Reaching the optimal throughput of a system requires each device to operate at its best throughput, since the lagging device will be the bottleneck of a system. The scheduler must select processes in an order so that the devices are utilized to maximize the throughput of a system.

One common device in all systems is the cpu. The throughput of a busy cpu is increased when the processing overhead is decreased, since the cpu can then use the overhead time saved to complete user requests. Therefore, running a scheduler that consumes little overhead is very desirable.

Maintaining the high throughput of a system, accomplished through good process scheduling and reducing overhead, is another important goal of a scheduler for a large system.

External Control of Resource Usage and Response Time

With the vast number of active processes in a large system, more processes are available for user-imposed priority increases and decreases. Externally decreasing the priority of unimportant processes, done usually during peak hours, is a quick way to artificially decrease the workload and thereby shorten the response time of a system. Externally increasing the priority of important processes is a quick way to make the scheduler favor them, thereby shortening their individual response times.

Convenient mechanisms for the external adjustment of priorities have been prevalent in operating systems used on large systems. There are several desirable characteristics of

such mechanisms. It is highly desirable that a priority adjustment algorithm produce predictable results. It should also be possible to reverse the effects of manual intervention.

Adaptability

Most large systems vary considerably in the application mix, performance requirements, system configuration, and system load. A scheduler is one of the basic entities that must be tunable to adapt a system to different environments.

3. Fundamentals of UNIX Scheduling

The original implementation of UNIX described in [4] and [5] included a simple scheduler that based the priority of a process on its compute-to-real-time ratio. A process that used a lot of compute time in the last real time unit was assigned a low priority, and a process that had received little compute time received a high priority. Runnable processes were kept in a multilevel queue, higher priority processes could preempt lower priority processes, and a quantum ended every second. Use of the compute-to-real-time ratio made the scheduler favor interactive processes, albeit also nonserviced cpu bound processes. No major considerations were given to the cost of the scheduling, since the goals of UNIX were elegance and simplicity.

A look at the current implementations of the UNIX 4.2BSD and UNIX 4.3BSD schedulers reveals that they operate in primarily the same manner. Priority assignments do occur more often and consider more parameters than just the compute-to-real-time ratio, and the clock frequency has been increased permitting quanta to end more frequently. Their core design is still that of the original scheduler, though as we will see in the next section, their elegance and simplicity have faded and their cost risen.

The remainder of this section presents and explains most of the code used in the implementation of the UNIX 4.3BSD scheduler*. To understand its working, it is important to remember that the implemented scheduler is not a process; it is a collection of kernel procedures executed at fixed frequencies or upon the occurrence of certain events. This description explains those procedures and their role in a running system. It is assumed that the reader is very familiar with the C programming language.

Process Activity in the Run Queue

At the end of a system reboot when all system processes have entered blocked states, *Swch()* is called to idle the cpu and wait for the first user to login. As users log in, new processes desiring service from the cpu are created. These processes are made runnable by inserting their process table slots into the multilevel run queue. The procedure *setrq()* is called to perform an insertion. The level of the run queue in which a process is placed is determined by the value of the process' priority, *p_pri*. The lower the value of *p_pri*, the higher the level. A process is removed from the run queue with a call to *remrq()*. An invocation of *swch()* will remove a process from the run queue and give it cpu service. *Swch()*, *setrq()* and *remrq()* are written in assembly language. Their code need not be examined to understand the operation of the scheduler; only familiarity with their functions is required.

Scheduling Event Waits and Timer Interrupts

When a process has control of the cpu but desires to wait for an event before continuing, it places itself in a blocked state and releases the control via a call to *swch()*. *Swch()* idles the cpu if the run queue is empty, or gives control of the cpu to the first process in the top level of the run queue if it is not.

* The UNIX code excerpts have been taken from the Fourth Berkeley Software Distribution under license from the Regents of the University of California.

Clock interrupts in UNIX 4.3BSD occur every ten milliseconds. At every tenth clock interrupt, the process that has control of the cpu is forced to release the control. This release of control, referred to as a quantum expiration, occurs regardless of when the currently running process gained its control of the cpu. This type of quantum expiration occurs in the following manner.

When a kernel procedure needs execution at a specific future time, its starting address and wakeup time are placed in the kernel's *callout* queue using the procedure *timeout()*. The system's clock interrupt handler *hardclock()*, among other things, examines the contents of the *callout* queue for the existence of any procedures that need to be *called out* and executed. If any are found, a low priority interrupt is scheduled that will occur as soon as the interrupt priority level of the cpu drops to a low level. The low priority interrupt handler, *softclock()*, will execute each of the *called out* procedures.

Timeout() and the portions of *hardclock()* and *softclock()* responsible for examining and executing procedures in the the *callout* queue follow. The text enclosed by */*** and **/* are comments not part of the original source.

```
/** Arrange that (*fun)(arg) is called in t/hz seconds by placing it in the callout queue.
    hz represents the frequency of the clock interrupt. In 4.3, hz is 100 */

timeout(fun, arg, t)
    int (*fun)();
    caddr_t arg;
    register int t;
{
    register struct callout *p1, *p2, *pnew;
    register int s = spl7(); /** raise the cpu interrupt priority level */

    /** Procedure addresses are inserted in the callout queue such that their order from
        front to back represents the order in which they are to be executed. The
        wakeup time of a procedure is stored as a delta from the time of the
        procedure in front of it. This method makes searching for a "due"
        event fast, but requires scanning of the callout queue at every insertion.
        Arguments used when the procedure is finally executed are also stored. */

    if (t <= 0)          /** make sure wakeup time is in future */
        t = 1;
    pnew = callfree;    /** get the head of free callout queue element list */
    if (pnew == NULL)  /** test for empty free list */
```

```
        panic("timeout table overflow");
    callfree = pnew->c_next;/** remove a free element **/
    pnew->c_arg = arg; /** load the new callout queue **/
    pnew->c_func = fun; /** element **/

    /** find the right place in the callout queue for the insertion and make the wakeup
        time of the new queue element a delta from the previous element's time **/
    for (p1 = &calltodo; (p2 = p1->c_next) && p2->c_time < t; p1 = p2)
        if (p2->c_time > 0)
            t -= p2->c_time;

    p1->c_next = pnew; /** insert the new element **/
    pnew->c_next = p2;
    pnew->c_time = t;

    if (p2) /** update the time of the next element **/
        p2->c_time -= t;

    splx(s); /** return to previous interrupt level **/
}

/** Portion of the clock interrupt handler executed every 10 msec. **/

hardclock()
{
    register struct callout *p1; /** points to callout queue element **/
    register struct proc *p; /** points to a process table slot **/
    register int s;
    int needsoft = 0;

    /** look at the procedures on the callout queue and if we find any that should
        be called, remember it so we can cause a low priority interrupt later to
        execute them. We cannot call the procedures directly from here, since
        doing so might make this execution of hardclock so long that we would
        miss the next clock interrupt. **/
    p1 = calltodo.c_next;
    while (p1) {
        if (--p1->c_time > 0)
            break;
        needsoft = 1;
        if (p1->c_time == 0)
            break;
        p1 = p1->c_next;
    }
    /** . . . . **/
    if (needsoft) {
        if (BASEPRI(ps)) {

            /** Since we were operating at a low interrupt priority
                level when the clock interrupt occurred, the
                low priority interrupt will occur as soon as
                we return. We can call softclock() here to
                save the overhead of the interrupt. **/
```

```
        (void) splsoftclock();
        softclock(pc, ps);
    } else
        setsoftclock(); /** cause the low priority
                        interrupt **/
    }
}

/** Softclock() is called directly from hardclock() or as a result of an interrupt caused
    by hardclock() to execute some procedures in the callout queue. **/

softclock()
{
    for (;;) {
        register struct callout *p1;
        register caddr_t arg;
        register int (*func)();
        register int a, s;

        s = spl7(); /** raise interrupt priority level **/

        /** if there is nothing on the callout queue, or the next procedure on
            it is not to be executed yet, return **/
        if ((p1 = calltodo.c_next) == 0 || p1->c_time > 0) {
            splx(s);
            break;
        }

        /** get the address and calling arguments of the procedure on the front of
            the callout queue, advance the head of the callout queue, return
            the used element to the free list, reset the interrupt priority
            level to its previous level, and execute the procedure **/

        arg = p1->c_arg; func = p1->c_func; a = p1->c_time;
        calltodo.c_next = p1->c_next;
        p1->c_next = callfree;
        callfree = p1;
        splx(s);
        (*func)(arg, a);
    }
    /** ..... **/
}
}
```

Round-robin Scheduling

One procedure that is usually found in the *callout* queue, absent only when it is being executed, is *roundrobin()*. *Roundrobin()* is placed in the queue during system initialization by invoking it. Its execution causes an asynchronous system trap, AST, to occur as soon as

the interrupt priority level of the cpu drops to a low level. The AST handler removes control of the cpu from the running process by placing it back in the cpu run queue. The handler then gives control of the cpu to the next process via a call to *swtch()*. The periodic execution of *roundrobin()* forces a rescheduling of processes every tenth of a second, never permitting a single process to keep exclusive control of the cpu.

The code for *roundrobin()* and a portion of the AST handler follows. It is important to see that *roundrobin()* always places itself back in the *callout* queue.

```
/* Force switch among equal priority processes every 100ms. */

roundrobin()
{
    runrun ++;    /** informs code to be executed before the next AST
                  that a rescheduling is about to occur. **/
    aston();     /** #defined as mtp(ASTLVL, 3) which will cause the AST. **/
                /** call to timeout() places the address of roundrobin() back on
                  the callout queue to be executed 100ms from now, thereby
                  setting up the next quantum expiration. **/
    timeout(roundrobin, (caddr_t)0, hz / 10);
}

/** Portion of trap code executed when an AST occurs. **/

trap()
{
    /**      . . . . .      **/
    (void) spl6();           /** increase cpu interrupt priority level. **/
    setrq(p);               /** put slot of current process back in the run queue. **/
    swtch();                /** give cpu control to the next process. **/
    /**      . . . . .      **/
}
```

Priority Recomputation

Another procedure that is usually found in the *callout* queue, absent only when it is being executed, is *schedcpu()*. *Schedcpu()*, like *roundrobin()*, is placed in the queue during system initialization by invoking it. Its execution causes process priorities to be recomputed every second. The method used to determine the priority of a process incorporates many parameters, some of which are updated by the clock interrupt handler, *hardclock()*. The

portion of *hardclock()* relevant to *schedcpu()*, and all of *schedcpu()* follows. It is important to see that *schedcpu()* always places itself back in the *callout* queue, just as *roundrobin()* does.

```
/** Called out to recompute process priorities every second. */

#define      filter(loadav) ((2 * (loadav)) / (2 * (loadav) + 1))
#define      NQS      32
#define      PPQ      (128 / NQS)

/* fraction for digital decay to forget 90% of usage in 5*loadav sec */
double ccpu = 0.95122942450071400909;          /* exp(-1/20) */

schedcpu()
{
    register double ccpu1 = (1.0 - ccpu) / (double)hz;
    register struct proc *p; /** points to a process table slot **/
    register int s, a;

    /** avenrun[0] contains the average number of runnable processes over
        the last minute. Its value is recomputed every 5 seconds. **/
    float scale = filter(avenrun[0]);

    /**      . . . .      **/

    /** for every process table slot ... **/
    for (p = allproc; p != NULL; p = p->p_nxt) {

        /** increment the time the process has been in core **/
        if (p->p_time != 127)
            p->p_time ++;

        /** increment the time the process has been blocked **/
        if (p->p_stat == SSLEEP || p->p_stat == SSTOP)
            if (p->p_slptime != 127)
                p->p_slptime ++;

        /** if the process has been blocked for more than a second,
            stop recalculating its priority. **/
        if (p->p_slptime > 1) {
            p->p_pctcpu *= ccpu; /** updated for ^T and ps command. **/
            continue;
        }

        /** update the percent cpu usage of the process for ^T and the ps
            command. p_cpticks is the number of times a clock interrupt
            occurred while this process had control of the cpu over the last
            second, estimating its cpu usage **/
        p->p_pctcpu = ccpu * p->p_pctcpu + ccpu1 * p->p_cpticks;
        p->p_cpticks = 0;          /** reinitialize for next interval **/
    }
}
```

```
/** begin actual priority computation. p_cpu is the value of 'a' from the
    previous call to schedcpu(). p_nice is a user imposed priority
    value, ranging from -20 to 20. 'scale' was #defined above. */
a = (int) (scale * (p->p_cpu & 0377)) + p->p_nice;
if (a < 0)
    a = 0;
if (a > 255)
    a = 255;
p->p_cpu = a;
(void) setpri(p); /** finish priority computation in setpri() */

/** increase the interrupt priority level of the cpu to ensure atomicity
    while examining and manipulating the run queue */
s = splhigh();

/** if the signal reception level of the process is low, then
    we can reset the priority */
if (p->p_pri >= PUSER) {

    /** if the process is in the run queue and its new priority would
        require it to move levels, remove it from the queue,
        change its priority, and insert it into the new level. Note
        the range of p_pri is 0..127 and must be divided by 4 to
        make is correspond to a queue level.
        if it is not in the run queue, simply change its priority. */
    if ((p != u.u_procp || noproc) &&
        p->p_stat == SRUN &&
        (p->p_flag & SLOAD) &&
        (p->p_pri / PPQ) != (p->p_usrpri / PPQ)) {
        remrq(p);
        p->p_pri = p->p_usrpri;
        setrq(p);
    } else
        p->p_pri = p->p_usrpri;
}
splx(s); /** restore old cpu interrupt priority level */
}
/** . . . . */

/** call to timeout() places the address of schedcpu() back on the callout queue
    to be executed a second from now, thereby setting up the next priority
    recomputation. */
timeout(schedcpu, (caddr_t)0, hz);
}

/** Setpri() is called from schedcpu() to complete the remainder of a single priority
    recomputation. If the new priority computed is better than the priority of the
    process running when schedcpu() was called out of the callout queue, a process
    rescheduling will be forced to occur via an AST. This occurs even if new priority
    belongs to a blocked process. */
```

```
setpri(pp)
register struct proc *pp;      /** points to a process table slot **/
{
register int p;                /** the new priority **/

p = (pp->p_cpu & 0377)/4;      /** p_cpu was computed above **/

/** PUSER is #defined as 50; p_nice is the process' user imposed
    niceness parameter, ranging from -20 to 20. **/
p += PUSER + 2 * pp->p_nice;

/** p_rssize is process' resident set size; p_maxrss is the process' maximum
    resident set size; freemem is the amount of free real memory in the system;
    desfree is the desired amount of free real memory in the system. **/
if (pp->p_rssize > pp->p_maxrss && freemem < desfree)
    p += 2*4;      /* effectively, nice(4) */

if (p > 127)
    p = 127;

if (p < curpri) {            /** if the priority is better than the current process', **/
    runrun ++;              /** force a process rescheduling **/
    aston();
}
pp->p_usrpri = p;
return (p);
}

/** Portion of the clock interrupt handler executed every 10 msec. **/

hardclock()
{
register struct proc *p;
register int s;

/** ..... **/
/** if the cpu was not idle when the clock interrupt occurred, charge the running
    process for using the cpu the entire last interval. if this is a fourth clock
    interrupt the process has fielded, then recompute its priority to allow
    others to preempt it more easily. **/
if (!noprocs) {
    p = u.u_procp;          /** get process slot of current process **/
    p->p_cpticks ++;        /** charge process with this interval **/
    if (++p->p_cpu == 0)
        p->p_cpu--;
    if ((p->p_cpu&3) == 0) { /** fourth clock interrupt fielded? **/
        (void) setpri(p); /** recompute priority **/
        if (p->p_pri >= PUSER)
            p->p_pri = p->p_usrpri;
    }
}
/** ..... **/
}
```

Unblocking Processes

Preemption of the cpu occurs not only when *setpri()* is called from *schedcpu()* or *hardclock()* as shown above, but when a process exits a blocked state and enters the cpu run queue. The kernel procedure *wakeup()* is called by interrupt handlers when events complete and processes should be unblocked. The following code is the portion of *wakeup()* that completes the priority recomputation, run queue insertion, and preemption.

```
/** Unblock all processes that were waiting for the completion of
    the event identifiable by the value of chan. */

wakeup(chan)
register caddr_t chan;
{
    register struct proc *p; /** points to a process table slot **/
    int s;

    s = splhigh();          /** raise interrupt priority level **/
    /**      . . . .      **/

    /** if the process being unblocked was in the blocked state for
        more than a second, recompute its priority. **/
    if (p->p_slptime > 1)
        updatepri(p);

    /** reinitialize the blocking time; set the process runnable, and
        put it in the cpu run queue if it is not swapped. **/
    p->p_slptime = 0;
    p->p_stat = SRUN;
    if (p->p_flag & SLOAD)
        setrq(p);

    /** Always cause a process rescheduling to occur. **/
    runrun ++;
    aston();

    /**      . . . .      **/
    splx(s); /** restore interrupt priority level **/
}

```

```
/** Updatepri() is called by wakeup() to recompute the priority of a process that was
    blocked for more than a second. Setpri() and its variables were explained earlier. **/
```

```
updatepri(p)
    register struct proc *p; /** points to a process table slot **/
{
    register int a = p->p_cpu & 0377;
    float scale = filter(avenrun[0]);

    p->p_slptime--;          /* the first time was done in schedcpu() */
    while (a && --p->p_slptime)
        a = (int) (scale * a) /* + p->p_nice */;
    if (a < 0)
        a = 0;
    if (a > 255)
        a = 255;
    p->p_cpu = a;
    (void) setpri(p);
}
```

The combination of the described procedures yields the standard UNIX 4.3BSD scheduler. The implementation involves procedures executed at fixed frequencies to recompute priorities and force process reschedulings, and involved procedures executed as events completed to unblock and recompute the priority of blocked processes. The next section examines the design and implementation problems associated with the standard UNIX schedulers.

4. Shortcomings In UNIX 4.2BSD AND UNIX 4.3BSD Schedulers

4.1. Design Specific Problems

Design specific problems are those that can be attributed to poor planning in the earliest stages of problem solving. These problems are engrained in the respective solutions, correctable only through completely new designs. The UNIX schedulers incorporate several design specific problems.

Costly Priority Assignments

As observed in the last section, the process priorities in UNIX are based on: 1) the amount of time the process has existed, 2) the amount and percent of cpu time it has consumed, 3) the amount of time it has spent in a blocked state, 4) the amount of real memory it has accrued, 5) the amount of real memory currently free, 6) the current load of the system, and 7) a user imposed *niceness* parameter. The many values are passed through an algorithm which produces a value deemed the process' scheduling priority. The algorithm, coded in *schedcpu()* and *setpri()*, is costly to execute, difficult to understand, and almost impossible to modify with predictable results. A separate, but just as complex, priority computation algorithm exists in *updatepri()*.

Not only are many parameters used in the algorithms, the ones measuring time are merely estimates. *Hardclock()*, the clock interrupt handler, and *schedcpu()*, the priority recomputing procedure, increment several time values when they are executed, including values not presented in the code excerpts of this paper. Each procedure examines the current state of the system when its execution occurs and assumes that the system was in that state since the last execution. For example, *hardclock()* charges the process that had control of the cpu when the interrupt occurred with use of the cpu for the entire ten millisecond interval since the last clock interrupt. This method of measuring time based on snapshots of the system produces values that are inaccurate and nonrepeatable. Using these values contributes to the complexity of the algorithms that recompute priorities.

In addition to their complexity, priority recomputations occur at somewhat arbitrary times. The priorities of processes waiting for cpu service and of those in the first second of a blocked state are recomputed every second by *schedcpu()*, requiring a complete pass of a process table that can contain a few thousand slots in a large environment. The running process has its priority recomputed every 40 milliseconds in *hardclock()*, and the priority of a process gets recomputed when it exits a blocked state by *updatepri()*. The tremendous

amount of overhead associated with priorities creates a negative impact on system throughput, since the cpu cannot service user requests while computing priorities.

Few Tuning Opportunities

Another significant shortcoming engrained in the design is that no simple methods for tuning were considered in such a complex priority computation algorithm. The arbitrariness in the frequency and points of priority computations also yields little room for tuning. All environments running UNIX 4.xBSD, large and small, rely on primarily the same untunable schedulers. Large systems are particularly affected since a very significant amount of the cpu is wasted irrespective of whether an environment needs these computations.

Unpredictable External Control

The only way a user can affect the normal priority of a process in UNIX 4.xBSD is to alter the process' *nice* parameter. The *nice* parameters, used in the priority computation algorithms, make priorities rise or fall relative to those of other processes. An examination of the code does not reveal what the exact effect of a change in a *nice* parameter is, only that a general scheduler favoring or shunning will take place.

4.2. Implementation Specific Problems

We can consider implementation specific problems as those problems that can be attributed to the coding of a solution. Such problems can be corrected without a corresponding redesign. The UNIX 4.xBSD schedulers also suffer from several implementation specific problems.

Fluctuating Quantum Sizes

In the currently implemented UNIX, a quantum expires every tenth of a second, regardless of when the running process obtained control of the cpu. The fixed frequency of quantum expirations hampers higher throughput when a system is lightly loaded since the cpu must service an unnecessary trap and perform an unnecessary context switch each tenth of a second.

When a large system has many active processes and voluntary context switching occurs frequently, it is often the case that a process must relinquish control of the cpu immediately after obtaining it because the next tenth of a second had arrived. These context switches also contribute to the UNIX scheduling overhead and the resulting impact on throughput.

Priority Misuse

A process' priority is rewritten with a value designating its signal reception level every time the process enters a blocked state. When the process exits a blocked state that it had been in for less than a second, *updatepri()* is not called from *wakeup()* for priority recomputation; the signal reception level is used as the new priority of the process. If the process had been in the blocked state for more than a second, a new priority must be computed since the one available before the block was overwritten. This 20 line priority computation algorithm, *updatepri()*, uses most of the parameters stated earlier and includes a loop whose execution time is proportional to the time spent in the blocked state, limited to a time of 127 seconds. Interactive processes, e.g. editors, must execute the costly priority assignment algorithm frequently since they often block more than a second for terminal input. This could have been avoided by simply providing an additional field in each process table slot for the signal reception level.

4.3. Scheduler Implementation Differences of UNIX 4.2BSD and 4.3BSD

All of the problems described above still exist in the field test versions of UNIX 4.3BSD. The previous version, UNIX 4.2BSD, had more severe scheduler problems. Two UNIX 4.3BSD scheduling changes described in [3] were implemented to relieve some of the vast scheduling overhead of UNIX 4.2BSD.

In UNIX 4.2BSD, when the process table was scanned every second by *schedcpu()* to recompute the priorities of processes, all blocked processes also had their priorities recomputed. Furthermore, a process never had its priority recomputed when it exited a blocked state. Also in UNIX 4.2BSD, a frequent type of unnecessary process rescheduling described in [3] occurred.

4.4. Summary of UNIX 4.2BSD and UNIX 4.3BSD Shortcomings

The concepts and methods of priority assignments used in the UNIX 4.xBSD schedulers are too complex and involve far too much overhead. The schedulers are difficult to modify and provide no facilities for tuning. The mechanism provided for user adjustments of priorities is simple but not predictable.

Problems in the current implementation of the UNIX 4.xBSD schedulers justify the need for their reworking. The fluctuating quantum size used in the run queue and the misuse of a process' priority field are two of the problems.

5. An Alternate Approach to UNIX Scheduling

This section describes the design and implementation of a new UNIX 4.3BSD scheduler developed at the University of Maryland. Our goal was to undertake a surgical operation of the UNIX 4.xBSD scheduler, leaving all of the fundamental design and implementation concepts of the remainder of UNIX intact. The explicit goals of the scheduler were maximizing responsiveness, maximizing throughput and available

adaptability, where maximizing responsiveness took priority over the latter two. We recognized that in order to fit these in with UNIX, the design had to be very simple. The simplicity was also an approach to keep the scheduler overhead to a minimum.

5.1. Design Specifications

5.1.1. Run Queue Structure and Operation

The cpu run queue is of the multilevel feedback type [1]. When a process is created, it is first placed in the run queue. A process in an upper level of the queue receives cpu service before a process in a lower level. Control of the cpu is taken away from a process when a) it enters a blocked state and releases control voluntarily, b) a different process with a higher priority exits a blocked state and preempts the running process, or c) the running process' quantum expires. Quantum sizes vary with each level of the queue, being smaller at the upper levels and larger at the lower levels. The quantum of a process begins when the process is removed from the run queue and given control of the cpu.

5.1.2. Process Priorities

The priority of a process waiting for the cpu directly determines the level of the run queue in which it resides. The higher priority processes are in the upper levels; the lower priority processes are in the lower levels. Changes to the priority of a process occur strictly on an event basis. When a process performs some type of interactive activity its priority is boosted, and when it completes an entire quantum of cpu service its priority is bumped. The small quantum sizes at the upper levels, and priority boosting for interactive events are designed to make interactive processes receive quicker service than their cpu bound counterparts.

5.1.3. Available External User Control

Each process has an associated *priority-min* and *priority-max*. The priority of a process never decreases below its *priority-min*, and never increases above its *priority-max*. The *priority-max* is never set above the highest level of the run queue, and likewise the *priority-min* is never set below the lowest level. Users are able to modify the *priority-max* and *priority-min* values of a process, thereby limiting the range of run queue levels in which it can reside.

5.2. Implementation Specifications

5.2.1. Run Queue Structure

The multilevel feedback queue is implemented as an array of 32 linked lists, similar to the one in UNIX 4.3BSD. Each linked list contains process table slots and corresponds to one level of the run queue. When a process desires service from the cpu, its process table slot is appended to the linked list associated with its priority value by the procedure *setrq()*. When the process is selected to receive cpu service, its process table slot is removed from the linked list by *swtch()* and the process is then referred to as the *running process*.

The quantum sizes for the levels are kept in a 32 element array of integers. When a process switch occurs, the quantum size associated with the level from which the next *running process* was selected is copied to a place referred to as the *current quantum*. The quantum size unit is dependent upon the frequency of the system's clock interrupt. At a clock interrupt frequency of 50 hz, the quantum unit is 20 msec; at a clock interrupt frequency of 100 hz, the quantum unit is 10 msec. The value of the *current quantum* is decremented at each clock interrupt in *hardclock()*. When the value reaches zero, the *current quantum* has expired and another process switch occurs. This implementation does allow for some variance in the actual *current quantum* of a process, since processes do not

necessary gain control of the cpu immediately after clock interrupts. The quantum decrementing occurs during clock interrupts to avoid the overhead of having a separate interrupt timer run for scheduling purposes only, and because hardware does not usually provide an additional interrupt timer.

5.2.2. Priority Boosting and Bumping

A new process' priority, *priority-min*, and *priority-max* are copied from its parent during process creation. Priorities, bumped only at quantum expirations, are boosted at the execution of several interactive events.

Creation Boosts

The first place a boost occurs is during the process' creation. This permits a new process born from a cpu bound parent to demonstrate its interactive nature quickly. If the new process is not interactive, its rapid set of quantum expirations at the upper levels of the run queue will force it to a lower level along with the other noninteractive processes.

Disk Operation Boosts

A priority boost also occurs when a process performs a block disk read. Giving boosts for disk operations keeps the disks active and the system throughput high. If the block already exists in the operating system's block cache though, the boost is not given. This is done to prevent a process from constantly reading the same disk block, then always found in the cache, just to keep its scheduling priority high. Boosts are also not given for block disk writes, since a process usually does not wait for them to complete before continuing. A properly contrived process could then constantly write the same disk block, which would only periodically get written to the disk from the block cache, just to keep its scheduling priority high.

Terminal Operation Boosts

A large number of priority boosts are given to a process that demands terminal interaction. A boost is given each time a process executes a terminal input command. Giving boosts for terminal input will keep a highly interactive process in the top levels of the run queue and interactive response time low. A boost is not given for terminal output because a process usually never waits for the write to complete before continuing. A process could then simply perform terminal output just to keep its scheduling priority high.

Priority boosting for terminal input is distinguished by the type of input being done. Most window editors and other extremely interactive processes operate in *raw* input mode, for which a small boost is given each time a character is read. Other very interactive processes, usually ones producing a lot of information for very little input, operate in *cbreak* mode and are given a moderate boost for each character read. The majority of interactive processes operate in *cooked* mode, for which a significant boost is given each time a line of input is read. The implementation recognized the distinction so that scheduling can be accomplished properly between the interactive processes themselves.

Message Passing Boosts

Sending or receiving a message through an interprocess communication channel also generates a priority boost for a process. Giving a boost for message passing is done to prevent filters from sharply decreasing the response time of a set of concurrent processes whose outputs are piped to other inputs.

Termination Boost

Receiving a signal requiring immediate termination generates a large boost to the priority of a process. This permits a process executing at a very low scheduling priority to terminate quickly and free its system resources, providing the user with fast external

control.

5.2.3. User Control

The system call *setpriority()* permitting user programs to change the priority of a process has been modified. For example, the user program *renice* created before the implementation of the new cpu scheduler, assumes its call to *setpriority()* modifies the *nice* parameter of a process. The modified *setpriority()* instead maps the argument to an appropriate *priority-min* and *priority-max* for the process. A new user program has been created to allow explicit setting of a process' *priority-min* and *priority-max*.

The *swapper*, *page daemon* and *init* are the only processes created during the initialization of UNIX. All other processes are created from *init* or its descendents, and have copied the *priority-min* and *priority-max* from those specified for *init* during initialization. Setting the initial priority values for *init* appropriately can lead to any of the following control scenarios.

If the *priority-min* and *priority-max* for *init* are zero and 31 respectively, then all processes created from *init* or its descendents will use the full range of priority run queue levels for scheduling purposes. If the *priority-min* for *init* is set to a value greater than zero, then all of the processes will not use the lowest run queue levels, referred to as the *basement*, for normal scheduling purposes. Processes that should receive service only when no normal process needs service should have its *priority-min* and *priority-max* set to values less than the *priority-min* of *init*.

If the *priority-max* for *init* is set to a value less than 31, then all of the processes will not use the highest run queue levels, referred to as the *attic*, for normal scheduling purposes. Processes that should receive service before any normal process needs service should have its *priority-min* and *priority-max* set to values greater than the *priority-max* of *init*.

In all cases, the *priority-min* and *priority-max* of any or all processes can be set to cover any range of run queue levels.

5.2.4. Scheduler Tuning

The ability to tune the new cpu scheduler is inherent in its design and implementation. All environments have the ability to select the parameters best suited for their needs. Quantum sizes, quantum size resolution, priority boost values, and initial values for the priority minimum and maximum of *init* are available for individual setting.

In environments where copious amounts of cpu intensive work is undertaken, tuning would be geared to emphasize the maximum throughput of a system. The quantum sizes would be considerably large, the quantum resolution coarse, and the priority boost values high. In environments where a significant percentage of the work is highly interactive, tuning would be geared to emphasize a stable and minimum response time. The quantum sizes of the upper levels would be small, the quantum resolution fine, and the priority boost values low. In either case, the initial values for the priority minimum and maximum of *init* would depend upon the applicable control requirements.

5.3. Implementation Details

This section presents and explains most of the code used in the implementation of the new scheduler. The new scheduler, like the UNIX 4.3BSD scheduler, is not a process; it is a collection of kernel procedures. It does not use most of the procedures of the old one, including *roundrobin()*, *schedcpu()*, *setpri()*, and *updatepri()*. It is much smaller, in terms of the number of lines of code, and is primarily event driven as opposed to clock driven. The following description assumes that the reader is very familiar with the C programming language.

The assembly routines *swtch()*, *setrq()*, and *remrq()* are basically the same. The procedure *setrq()* has been modified to reflect the fact that the priority of a process now directly determines the level of the run queue into which it is to be placed. *Swtch()* has been expanded from its original 40 lines to include one line which reinitializes the *current quantum*. The value is taken from the array of quantum sizes and corresponds to the level from which the next running process came.

The implementation of quantum expirations is unique in the new scheduler. The overhead involved in every insertion and removal from the *callout* queue of the procedure *roundrobin()* has been completely eliminated. Quantum expirations occur directly in *hardclock()* as follows.

```
/** Portion of the clock interrupt handler executed every 10 msec. **/

hardclock()
{
    /** if there was a process running, decrement the current quantum. if the current
        quantum becomes zero, decrement the process' priority so long as it doesn't
        go below the process' priority-min, and then cause a rescheduling via
        the AST. **/
    if (!noprocs) {
        if (!--quantum) {
            if (--p->p_prior < p->p_min) {
                p->p_prior ++;
            }
            runrun ++;
            aston();
        }
    }
    /** . . . . **/
}
```

Because the design associated with priority assignments in the new scheduler is so very different from the old one, the corresponding implementation is also very different. The old procedure to recompute priorities every second is never called, the priority of the running process is not recomputed every fourth clock interrupt it fields, and the priority of a process is not recomputed when the process exits a blocked state. Priorities, now bumped in *hardclock()*, are boosted as described in the implementation specifications. The boosting for

a particular event is done in the upper layers of the respective device driver code. The boost given for a block disk read, as an example, follows.

```
/* Read in (if necessary) the block and return a buffer pointer. */

struct buf *
bread(dev, blkno, size)
    dev_t dev;
    daddr_t blkno;
    int size;
{
    register struct buf *bp; /** points to a block cache buffer **/
    register struct proc *p; /** points to a process table slot **/

    /**      . . . .      **/

    /** try to get the block from the block cache. return if obtained. **/
    bp = getblk(dev, blkno, size);
    if (bp->b_flags & B_DONE) {
        return(bp);
    }

    /** active the low level device driver to perform the read. **/
    bp->b_flags |= B_READ;
    (*bdevsw[major(dev)].d_strategy)(bp);

    /** boost the priority of the process making the request by the
        value of the boost associated with block io. don't let
        the priority exceed the process' priority-max. **/
    newprior = p->p_prior + bioboost;
    if (newprior > p->p_max)
        p->p_prior = p->p_max;
    else
        p->p_prior = newprior;

    biowait(bp);          /** wait for the read to complete. **/
    return(bp);
}
```

Preemption in the new scheduler occurs when quantum expires in *hardclock()* and when processes exit a blocked state. If a process is removed from a blocked state by an interrupt handler's call to *wakeup()*, and has priority higher than that of the running process, an AST is forced to occur. The difference between the old *wakeup()* procedure and the new one is that the new one does not contain the call to *updatepri()*. A process exits a blocked state with the priority it had when it entered the blocked state; *p_pri* is used only

to specify the signal reception level.

6. Conclusion

In this paper, we have shown that a large system places heavy demands on the scheduler, and that the scheduler plays a key role in system performance. We have described in detail the standard UNIX schedulers and have shown that they have not evolved as systems have. Because of the design and implementation shortcomings of the UNIX 4.2BSD and UNIX 4.3BSD schedulers, a thorough design review and reimplementaion of the scheduler is an essential step in the advancement of large system UNIX operating systems. The concise design and the corresponding implementation of the new scheduler described should make the current goals of a large system UNIX scheduler attainable.

The University of Maryland's new scheduler is still being refined. Discovering characteristics about interactive processes that receive little or late cpu service gives way to the consideration of new events for which boosting might occur. The standard set of boost amounts and quantum sizes is still being developed. Further research and an extensive analysis should yield the best parameters as well as valid performance data. We are looking forward to testing the effectiveness of this scheduler on very large systems and in varieties of user environments.

References

- [1] E. G. Coffman and P. J. Denning, *Operating Systems Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1973.
- [2] S. Leffler, M. Karels and M. K. McKusick, "Measuring and Improving the Performance of 4.2BSD," in *Proc. Salt Lake City Usenix Conf.*, pp. 228-236, June 1984.
- [3] M. K. McKusick, M. Karels and S. Leffler, "Performance Improvements and Functional Enhancements in 4.3BSD," *Comp. Syst. Research Group, Dep. Comput. Sci. and Elec. Eng., Univ. of California at Berkeley, Berkeley, CA*, 1985.
- [4] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *CACM*, vol. 17, no. 7, pp. 365-375, July 1974.
- [5] K. Thompson, "UNIX Implementation," *Bell Syst. Tech. J.*, vol. 57, no. 6, part 2, pp. 1931-1946, July/Aug. 1978.

A Straightforward Implementation of 4.2BSD on a High-performance Multiprocessor

Dave Probert, Jeff Berkowitz, Mark Lucovsky

Culler Scientific Systems Corporation

ABSTRACT

The CULLER 7 scientific computer system is a high-performance multiprocessor consisting of a single kernel processor and one to four application processors. 4.2BSD was implemented on this machine by executing the majority of the operating system on the kernel processor, and executing user code on the application processors. This paper describes the implementation of the operating system kernel, with emphasis on the effects of asynchronous multiprocessing, process pipelining, the virtual memory implementation, and the utilization of special hardware for forks, context switches, and I/O.

UNIX is a trademark of AT&T

Multibus is a trademark of Intel

4.2BSD is a trademark of the UC Regents

Culler 7 is a trademark of Culler Scientific Systems

VAX and UNIBUS are trademarks of Digital Equipment Corporation

1. Introduction

The Culler 7 is a high performance computer system targeted at scientific and engineering applications, particularly those addressing the modeling and simulation area where a balance of scalar and vector capabilities are required. The Culler 7 architecture does not depend on pipelines for performance but incorporates the capability of executing multiple operations simultaneously within one clock cycle. This parallel execution of non-repetitive code as well as repetitive functions provides a high level of delivered performance for both scalar and vector code.

In the sixteen years preceding the announcement of the Culler 7, Culler Scientific developed a series of special purpose scientific computers for university research groups, the National Science Foundation, and the U.S. Department of Defense. These processors addressed a wide range of applications including signal and image processing, plasma physics simulation and linear algebra. The design for one of these products, the AP-120, was licensed to Floating Point Systems in 1975 and became an integral part of their commercial product line.

The design goal of the Culler 7 was to implement an extendible multiprocessor architecture that could support general purpose scientific programming by taking advantage of the array processor architectures Culler had pioneered. The design team focused on three primary areas: improving the fundamental array processor design to eliminate performance bottlenecks, efficient execution of programs compiled from standard FORTRAN and C, and designing hardware capabilities to allow the effective implementation of modern operating systems.

2. The CULLER 7 System Architecture

The system architecture consists of one to four *User Processors* connected to two high speed bus systems and coordinated by a separate *Kernel Processor*. I/O is provided by a Multibus I/O subsystem and through high-speed attachments directly to the system data bus.

Figure 1 contains a system level block diagram of a Culler 7 with two user processors. The kernel processor controls the user processors and I/O activities via the System Control Interface (SCI). The kernel processor offered with the Culler 7 is a 68010 attached to a Multibus. By replacing the SCI it is possible to attach alternative kernel processors and busses.

The SCI provides hardware support for I/O, virtual memory, and context switching. This enables the kernel processor, which is much slower than the user processors, to adequately service the I/O and virtual memory requirements for CPU intensive application programs. Interactive response under heavy CPU load is comparable to that of a VAX 11/750.

The existence of the separate kernel and user processors leads to the identification of three process types in the system. *Kernel processes* are processes that have no user state and execute solely within the kernel, such as the pageout and swapper daemons. *User processes* are ordinary processes that execute on a user processor when in user mode. We refer to the processes that execute in user mode on the kernel processor as *system processes*.

2.1. The User Processor

Figure 2 is a block diagram representation of a single user processor. Each user processor consists of two parts, the XY-machine and the A-machine. The A-machine is designed to perform addressing and program sequencing activities. The XY-machine contains the XY register memory, an IEEE floating point adder and multiplier, and a datapath architecture that has evolved from Culler's earlier array processor designs. The division of the user processor into two machines has some similarities to the PIPE architecture [Goodman85], but was developed

CULLER 7

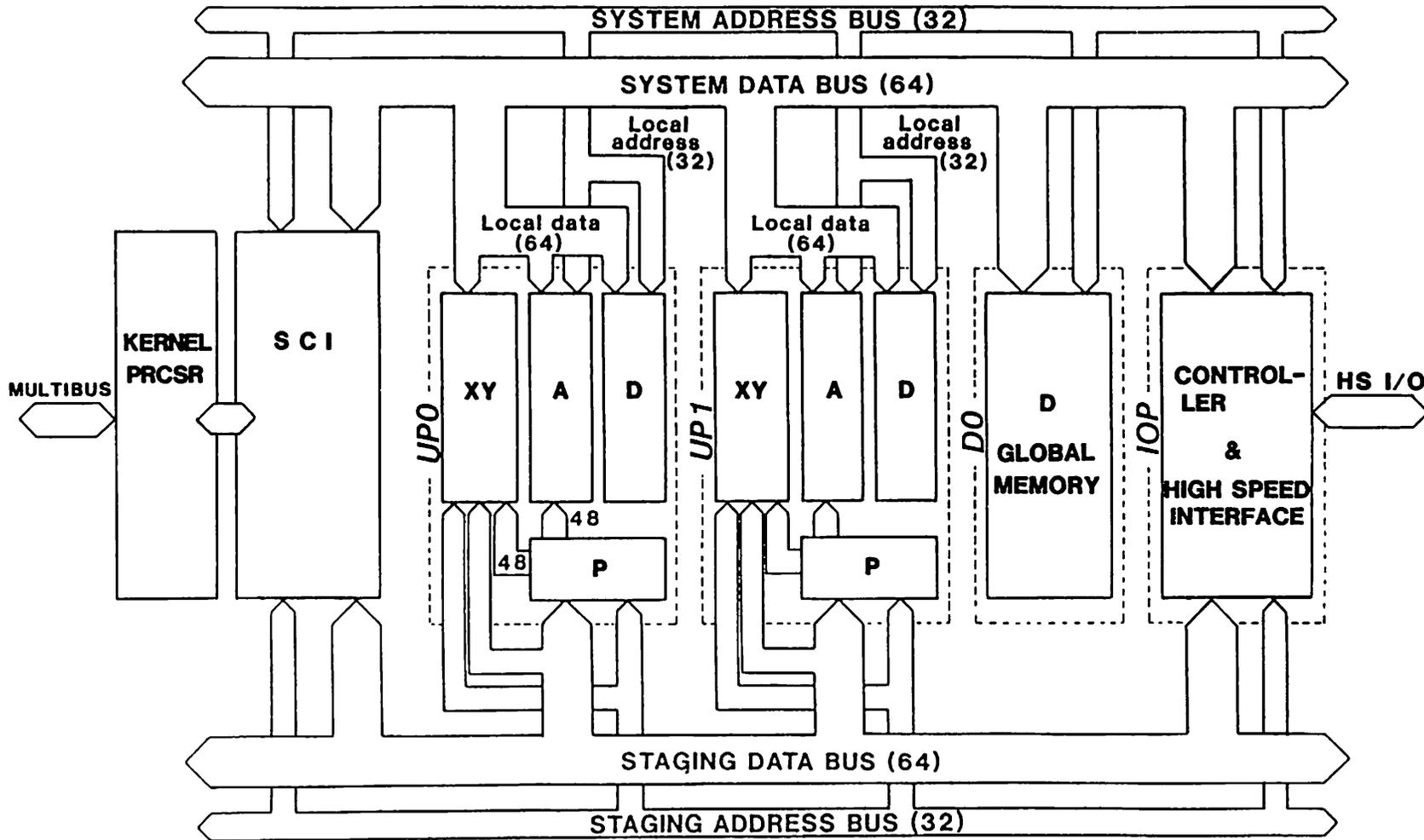


Figure 1: System block diagram of a Culler 7 with two user processors.

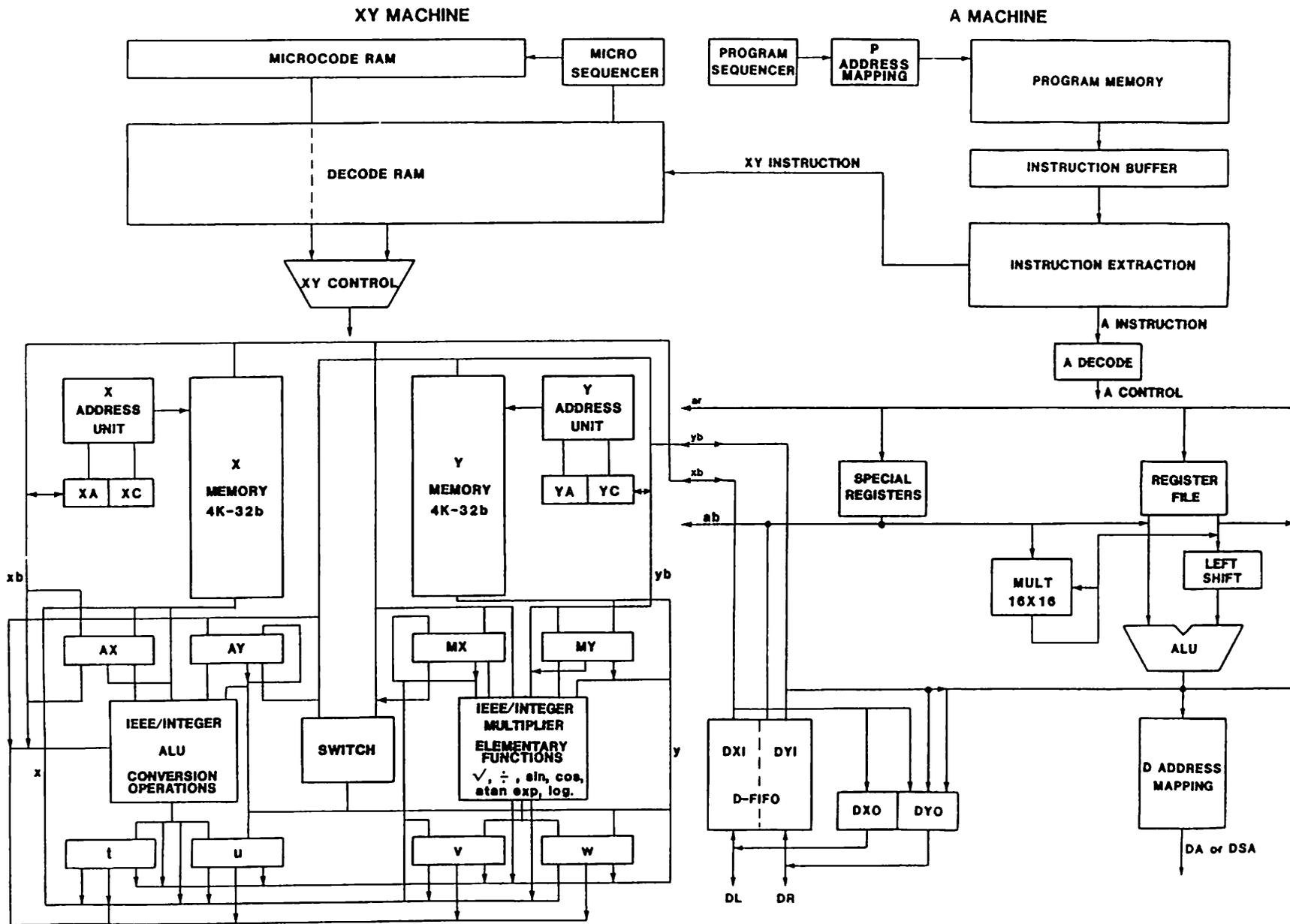


Figure 2: Block diagram of the architecture of a user processor.

independently.

The user processor has two sets of XY-memories and two D and P hardware page tables (D-map and P-map). One set is associated with a *half* of the user processor. Only one half of the user processor can be active at a time. The active half of the user processor is said to be the foreground half. The XY-memories and page tables associated with the background (inactive) half are staged using the block transfer hardware in the SCI, which transfers between D-memory and the staging bus. P-memory is shared by both halves of the user processor, and is written via the staging bus by stealing cycles from the user processor.

2.1.1. XY-memory

The XY-memory is 4K 64bit words of fast access memory allocated as a stack. It is used for the allocation of procedure call stack frames, which include subroutine linkages, procedure call arguments, and storage of register variables, automatic arrays, and vector registers. XY-memory is liberally allocated for these purposes by the compiler, so 32KBytes may not be adequate for some programs. Privileged bounds registers are used to trigger *roll-outs* of the bottom of the XY-memory stack into data memory. The top of the stack is always kept inside the user processor. XY-memory must be staged (or destaged) every time a process is put on (or taken off) a user processor. Staging normally overlaps with the execution of another process on the foreground half of the user processor.

2.1.2. Instruction Sequencing

The A and XY-machines have separate instruction sets which are fetched from the common program (P) memory by the A-machine. Each A-instruction is executed in a single clock, as are most of the X-instructions. However, an XY-machine instruction may invoke microcode that will cause the XY-machine to sequence independently from the A-machine. A combination of microcode and special hardware is used to implement the elementary functions and vector instructions. Programs can also include microcode segments, generated by the compiler, which extend the XY-machine instruction set for the execution of that program. The microcode segments are loaded into the user portion of microcode memory when the process is run on a user processor. The operating system manages the microcode segments as an extension of program text.

P-memory is operated as a cache for the text pages that are the working set of currently executing processes. Copies of all pages cached in P-memory are always maintained in D-memory. Since P-memory is read only, this implies that it never requires destaging. Programs access P-memory through the P-map. This allows P-memory to contain pages from multiple processes concurrently.

2.1.3. D-memory

All user processors share a large data (D) memory accessed over the D-bus. There are several facets to the architecture that eliminate contention for D-memory: program text is fetched from the separate P-memory, the compilers make heavy use of the XY-memories, and the D-bus is split into multiple busses, allowing for up to five separate D-transfers in the same cycle.

Data requested from D-memory is received by the user processor in a three-deep FIFO. Addressing is performed by the A-machine, but the data can be transferred from the FIFO into either the A-machine (up to 32 bits) or the XY-machine (up to 64 bits). The hardware supports references of 8, 16, 32, and 64 bits, and the user processor will recover from unaligned references (with some

performance impact).

2.1.4. Virtual Addressing and Process Structure

Program text and data live in separate virtual address spaces. Since text is always transferred into P-memory from D-memory, it is possible to make text writable by mapping it into the data virtual address space of a process.

Virtual addresses for text and data are mapped into physical addresses for P and D-memory by the P and D hardware page tables (P-map and D-map). The tables are organized as two-way set-associative maps. The P-map translates program page addresses into P-memory physical addresses, while the D-map translates data addresses into 32 bit D-addresses. Both the foreground and background halves of each user processor have a set of maps, so the maps can be staged by the SCI along with XY-memory. The SCI copies addressing information into the hardware maps from map images maintained in D-memory by the kernel processor. Since the maps are always copies of the D-memory image, they never need to be destaged.

Figure 3 shows the structure of a loaded Culler 7 process. Kernel memory contains the same structures as found in VAX 4.2BSD, but with extensions to associate the user microcode segment and a separate page table with the text structure. D-memory contains the backing images for the D-map, P-map, user microcode, and the text segment. The data/stack segment is also in D-memory. The process control block (PCB) is the first page of the data segment, and the XY-memory rollout area exists above the stack segment in process virtual space. The user processor is loaded from the appropriate areas of D-memory whenever a process is staged.

Each process has a single D-map image that is used for all user processors that it may be staged onto. A separate P-map image is maintained for every processor that a process is staged on. This is required because each user processor has a separate P-memory.

2.1.5. Synchronization and Interrupts

Synchronization between the A and XY-machines occurs through the interleaving of X and A-instructions in program memory. Additional synchronization is provided by the D-FIFO and synchronization bits on the inter-machine busses.

The user processor has a single level interrupt/trap structure that will put the user processor in privileged mode and begin execution of a local interrupt handler (KUP0). Interrupts can occur due to exceptions, arithmetic traps, system service requests within the user processor, or a preemption interrupt from the kernel processor. KUP0 is responsible for saving miscellaneous register and hardware state into the PCB of the process.

2.2. System Control Interface

The System Control Interface (SCI) is the hardware that provides the connection between the Multibus and kernel processor and the user processors and D-memory. It contains a collection of functional units that operate in parallel. These functional units make a significant contribution to operating system performance by offloading time critical functions from kernel processor software.

2.2.1. Staging

The local memories and page maps in the user processor are loaded from D-memory through the SCI. When a process is selected to run on a user processor, the kernel queues a series of staging transfers to the SCI. For each transfer, the SCI copies from a specified block of D-memory onto

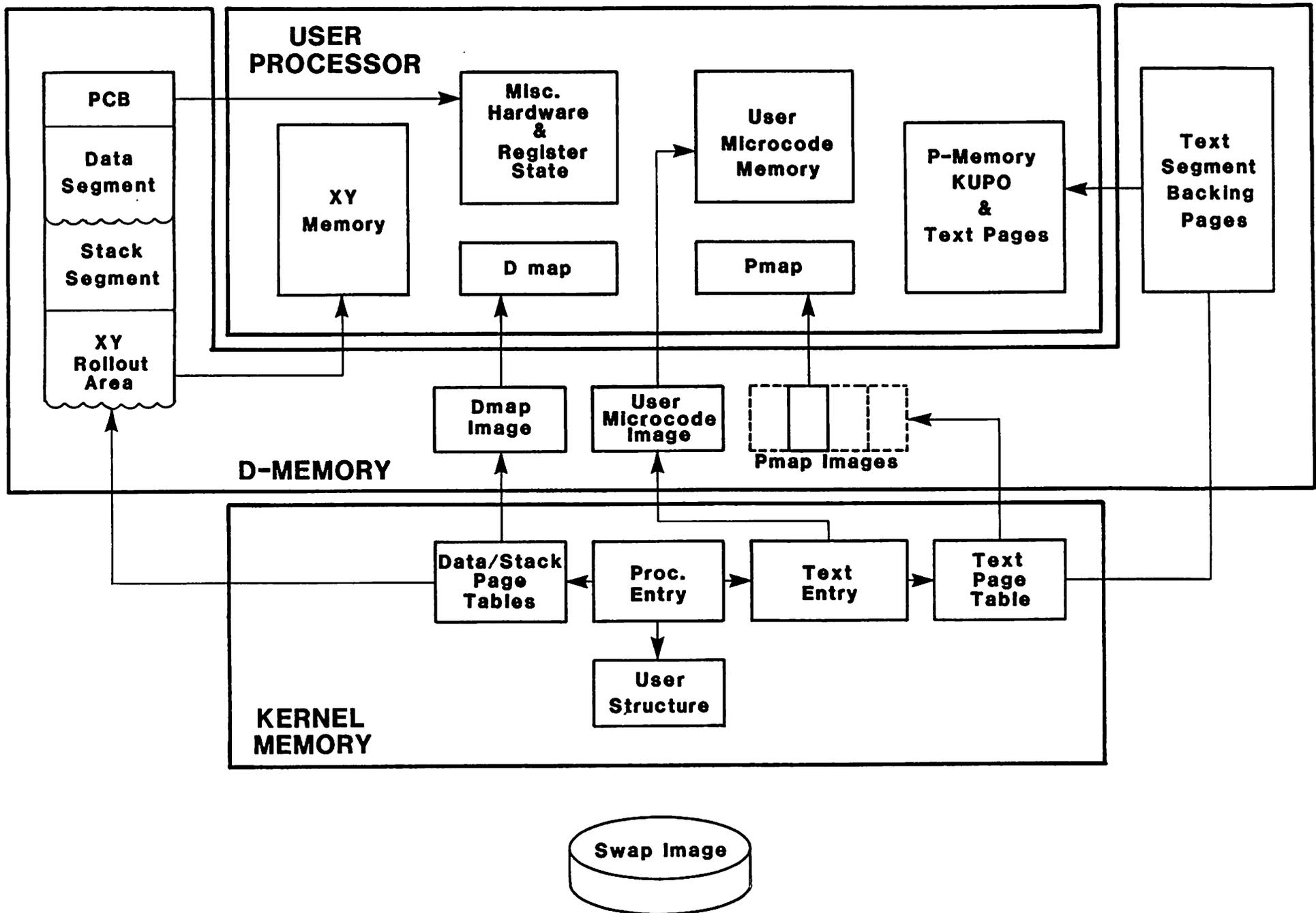


Figure 3: Structure of a loaded Culler 7 process.

the staging bus. The memories in the background half of each user processor are addressable over the staging bus. When a process is removed from a user processor, its XY-memory is destaged back into D-memory. Because P-memory and the page maps are not directly modified by the user processor, they do not need to be destaged.

2.2.2. D-memory Access

The Multibus provides 20 bits of address and 16 bits of data. The D-bus has 32 bits of address and 64 bits of data, but also allows aligned 8-, 16- and 32-bit transfers. The SCI provides two mechanisms for mapping the Multibus to the D-bus. These mechanisms are similar to the functions of the UNIBUS adapter on a VAX. The first facility is a set of registers that allow the specification of a full 32 bit address and transfer size (byte, shortword, longword, or double). This facility is currently used by the kernel, and is convenient for transferring small amounts of data with minimal overhead.

The second facility consists of windows onto the Multibus, each of which contain a set of address mapping registers and a data assembly register. This facility is analogous to the buffered data path mechanism on the VAX UNIBUS adapter. The kernel allocates the windows using a mechanism similar to the *ubasetup()* mechanism in 4.2BSD. The windows are used primarily by the Multibus disk and tape controllers, but are occasionally used by the kernel to transfer data blocks into the D-memory portion of the buffer cache. The mapping table supports scatter/gather I/O and allows the disk controller to execute chains of I/O requests without kernel intervention.

The SCI contains a hardware transfer unit for copying blocks of D-memory. This facility is used to transfer data from the D-memory buffer cache into process data space. Hardware D-to-D transfers are also used for zero-filling pages and copying memory during forks.

2.2.3. Miscellaneous Functions

Several control and diagnostic functions are accessed through SCI registers in Multibus I/O space. Each user processor has a control/status register. Control functions to *reset*, *halt*, *run*, *interrupt*, and set *readyexchange* are provided via these registers. Serial state chains are threaded through each user processor and accessed via other registers in the SCI. They provide access to internal processor state for diagnostics, and are also used to determine processor and memory configurations at system boot time.

The SCI is responsible for arbitration of the system D-bus. In this role it sees every global D-bus transfer and records usage/dirty bits for physical addresses assigned to D-memory. Usage/dirty bits for D-memory transfers accomplished wholly within a user processor (i.e., across the local D-bus) are recorded locally in a table that is accessible on the staging bus for each user processor.

3. Kernel Implementation

The kernel implementation began with an existing port of 4.2BSD running on the 68010-based kernel processor. The CSD (Culler Software Distribution) kernel consists of a straightforward set of additions to this base. The capability of executing user programs on the kernel processor was retained throughout the kernel development. This eliminated many of the problems usually associated with bootstrapping a new UNIX port. During normal system operation, the only program to run on the kernel processor is */etc/init*, but a full single user environment is supported on the kernel processor for system initialization and diagnostics.

The implementation of the kernel was accomplished in the following phases:

- 0: Started with a fully functional 4.2BSD kernel on the kernel processor
- 1: Wrote a program on the kernel processor, the Single Process Executive (SPE), which enabled us to run one process at a time on a user processor. SPE acted as the surrogate for system calls. This allowed much of the KUP0 code to be debugged and compiler testing to start at a very early stage.
- 2: The rudiments of asynchronous multiprocessing were implemented in the kernel and the the SPE code was migrated inside to form a version of the kernel that was capable of running a single job at a time on a user processor.
- 3: Multiprocessing of user processor jobs was enabled by adding the staging code.
- 4: Finally, paging and swapping were added.

The additions made to the original 4.2BSD kernel can be divided into two categories: those required for asynchronous multiprocessing and hardware support, and those designed to support system performance. Examples of the first class are the KUP0 (local trap handler) code, the user processor system call mechanism, and the changes to the virtual memory code. The second class includes the XY-memory rollout mechanism, support of user microcode, and the distribution of the kernel buffer cache across both kernel and D-memory.

Code was added to the kernel in the following areas:

(1) KUP0 (user processor resident trap handler)

This code is responsible for first level response to all user processor interrupts, traps, and exceptions. It categorizes the interrupt cause and communicates with the kernel processor by leaving a message in the PCB and causing a kernel interrupt for cases that require kernel processor intervention.

(2) User Processor Interface

(a) System Call Support

A new mechanism was implemented to accept system call interrupts at arbitrary times and synchronize the resulting system call execution through a normal kernel context switch. This mechanism is used to support all operations that may cause the requesting process to block. Some non-blocking system calls are handled at interrupt time, avoiding the kernel context switch.

(b) Stager

The user processor is supported by an interrupt driven staging driver that implements pipelining at the process level. This pipelining is made possible by the user processor's foreground/background memory implementation.

(c) Configuration

Code was added to support the autoconfiguration, initialization, and dynamic deconfiguration or offlining of user processors and D-memory. The ability to dynamically remove a processor from the available set is important for diagnostics in a multiple processor environment.

(3) Hardware support

Special purpose hardware is provided by the SCI to support critical operations such as process staging, block I/O, and process forks. Routines were added to the kernel for

allocating, managing, and accessing these resources.

(4) Virtual memory support

The base kernel contained a virtual memory implementation for kernel memory. A separate pageout daemon, and parallel virtual memory code were added to support the Culler 7 D-memory. Additional code was added to manage the P-memory cache within each user processor and to support the separate text and data address spaces.

(5) Additional program segments

In addition to the text, data, and stack segments of standard UNIX, CSD supports two additional program segments. A user microcode segment is implemented as a part of the shared text mechanism. Each process also owns a second stack segment, called the XY rollout segment, the top of which is kept in the local XY-memory of the user processor. As the XY stack grows, more of the bottom rolls out to D-memory. When the stack contracts, some of the bottom rolls back in. The CSD compilers use the XY stack for local procedure frames and vector registers. A separate D-stack is maintained for dynamic allocation of dereferenced data.

(6) Buffer cache

The system buffer cache is split between D-memory and kernel memory. Superblocks, inodes, bitmaps, directory blocks, and indirect blocks are transferred into kernel memory, while data blocks from ordinary files usually go directly into D-memory. The operational division of labor between the buffer caches is a consequence of the UNIX filesystem design, and any block could end up in either cache. Code was added to the kernel to detect blocks that end up in both caches; and one of the copies is always invalidated before the other can be written.

(7) Miscellaneous kernel additions

The 4.2BSD implementation of signals was extensively modified to move functionality into the user processors. Code relating to timing and timeslicing also required modification in the multiple processor environment. The *exec* system call was modified to allow the coexistence of kernel processor and user processor code files. Programs are *exec'd* on the machine appropriate to their type, as indicated by the magic number. User physical I/O and swap I/O were modified to support D-memory access. Changes were also made to the kernel to take advantage of some of the performance enhancements suggested in [Leffler84].

Many of the areas traditionally modified during UNIX kernel ports are absent from our list. This is due to our decision to start with an existing 4.2BSD port and add the functions required to support the Culler 7 multiprocessor. On the other hand, the implementation of the XY-memory and user microcode segments required us to make substantial modifications to parts of the kernel that are normally machine independent.

3.1. Job Stream Pipelining

XY-memory, P-map, and D-map contain a significant amount of state. Context switches would be very expensive if the user processor had to remain idle while the new process state was loaded. Providing two sets of these memories within each user processor, and allowing a process to execute using the foreground set while the background set is being staged, retains the bandwidth and performance advantages of local memories without incurring the additional context switch

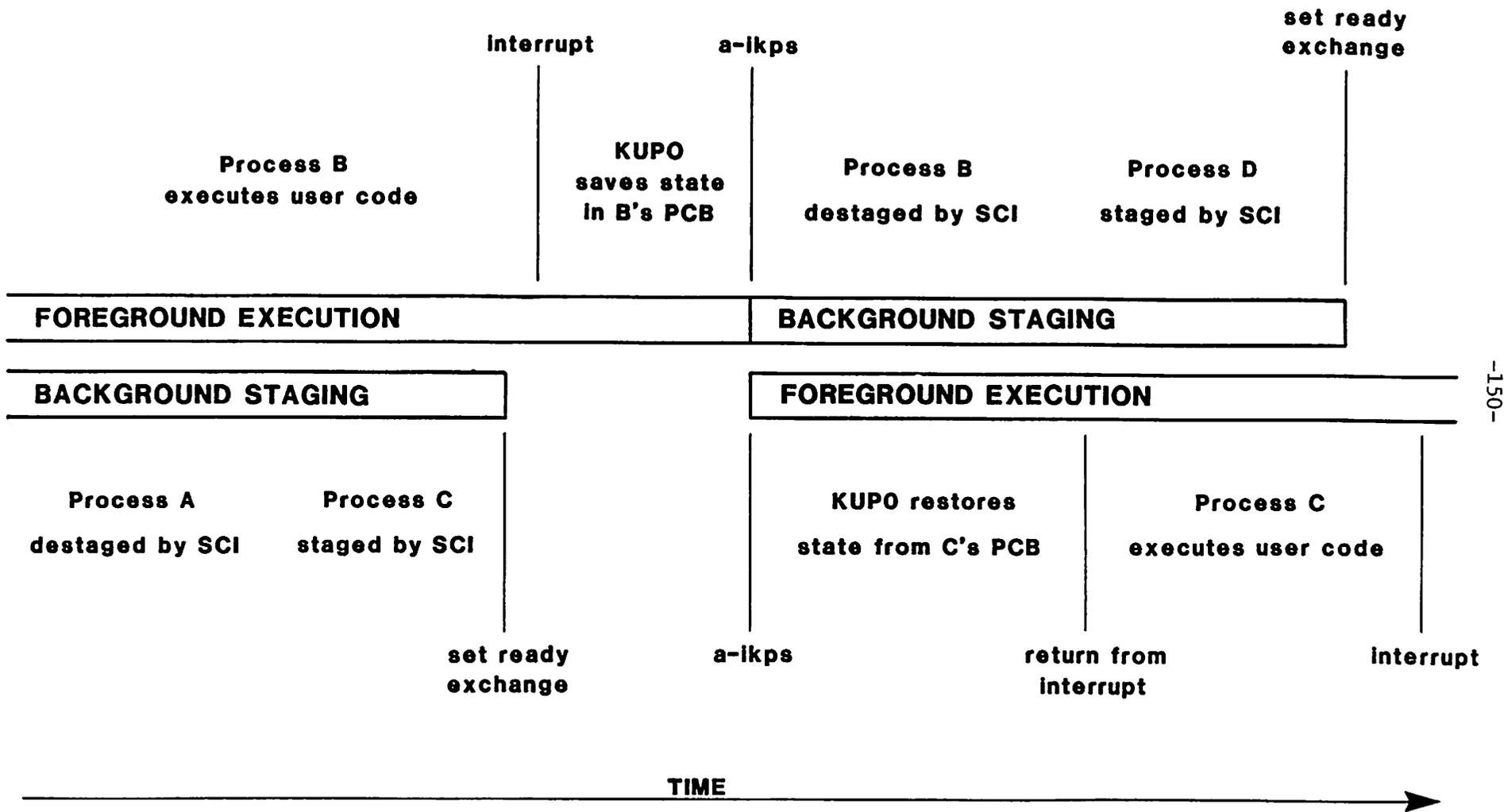


Figure 4: Job stream pipelining. Time from interrupt of process B until return from interrupt in process C is a few dozen microseconds.

overhead.

The result of this design is that the user processor effectively becomes a two-stage pipeline for processes. Figure 4 shows the operation of the two stages of the pipeline.

In the worst case, staging of a user processor can take a couple of milliseconds. Frequently, only small fractions of the memories and maps are used and the rest are not staged.

The pipelining of processes exacerbates the problems of asynchronous multiprocessing. Once a process is staged and the *readyexchange* bit is set the process might start running at any time. By the time the kernel processor responds to the *a_ikps* from the first process on the processor, the second process may have already trapped and also be requesting kernel intervention. Careful coding was required to avoid the handful of race conditions that pipelining created.

3.1.1. Context Switching

A user context switch may occur whenever a running user process requires service from the kernel processor or the kernel processor preempts a user processor. In either case, the user processor begins executing KUP0, which saves miscellaneous hardware and register state in the PCB, and then executes an *a_ikps* instruction. The *a_ikps* instruction accomplishes the exchanging of the foreground and background halves of the user processor. *a_ikps* sends an interrupt to the kernel processor and then the processor hangs until the kernel processor sets the *readyexchange* bit in the user processor control register. The background halves of the memories and maps are then exchanged with the foreground halves and the processor continues execution, but in the context of the process that had been staged onto the background half. In general, processes take longer to execute than to stage, either because they are CPU intensive, or have relatively little active context to stage. Thus the *readyexchange* is usually set long before the *a_ikps* instruction is executed, and the exchange of processor halves takes place immediately.

When the exchange occurs, execution resumes after the *a_ikps* instruction. KUP0 restores the state of the hardware registers and any parts of the user microcode memory that are in use, and then executes a *return-from-interrupt* instruction which, re-enables interrupts and turns off privilege until the next time an interrupt occurs.

When a process is ready to run, the kernel selects a user processor, and then queues a series of staging transfers to the SCI driver. The last staging transfer causes the *readyexchange* bit to be set for the user processor. At the next system call, page fault, or kernel preemption interrupt, the halves are exchanged, and execution of the new process begins.

A few special cases complicate this straightforward model of process pipelining. For example, there may be only a single runnable process in the system. There is then no process to stage onto the background half of the processor, and when the process exits the kernel, it is desirable to resume the process without first destaging its XY-memory and then restaging the process onto the other half of the processor. The kernel makes this optimization by artificially running a dummy process on the other half of the processor which, immediately switches back to the real process.

3.1.2. Initialization

When the system first boots, */etc/init* is executed on the kernel processor. The */etc/rc* script runs a configuration program that initializes the user processors and enables the execution of user processor programs. Until the user processor is configured, processes can be run on the kernel processor only. Initialization consists of loading KUP0 into the P-memory of each user processor, setting the P-map and D-map appropriately, and then sending an interrupt from the kernel

processor. KUP0 will begin execution and end up hanging at the *a_ikps* instruction, waiting for the kernel to stage a process onto the background half of the user processor.

3.2. Kernel-mode User Page 0 (KUP0)

The local trap handler (KUP0) is the part of the operating system that runs on the user processor. KUP0 is privileged code that responds to all traps and interrupts within the user processor. Interactions with the kernel processor are accomplished through a message buffer in the process control block (PCB) of the currently running process. The PCB for each loaded process is locked in D-memory.

The following example of how a system call is performed illustrates the activities of KUP0:

The user code pushes the system call arguments on the stack and executes an *a_trap* instruction.

KUP0 begins executing in privileged mode on the user processor.

KUP0 saves the non-staged hardware context into the PCB.

KUP0 copies the system call arguments into the PCB.

KUP0 executes an *a_ikps* instruction, which interrupts the kernel processor and may cause an immediate context switch to the process staged in the background.

The kernel picks up the system call from the PCB and performs the system call. If the process blocks inside the kernel, its processor half may be destaged and used for another process. In this case the process will have to be staged back onto a user processor at the end of the system call.

The process is ready to be resumed. The other half of the processor will ultimately execute an *a_ikps* instruction, which will cause a context switch back to this process. KUP0 will continue execution after the *a_ikps* instruction.

KUP0 handles any messages it finds inside the PCB.

KUP0 restores the non-staged hardware context from the PCB.

KUP0 returns from the interrupt and begins executing the user code after the *a_trap* instruction.

Page faults are handled similarly to system calls, except that the arguments are conjured by KUP0 from the hardware state, and the faulting reference is restarted.

Signal handlers are built and dismantled by KUP0 based on the signal vector information which is cached in the PCB. Signals can originate due to kernel processor messages (e.g., the *kill()* system call) or within the user processor (e.g., floating point exceptions).

KUP0 also contains code for fixing up unusual cases of IEEE arithmetic, repairing unaligned D-memory references, rolling in/out the XY-memory stack, managing user microcode memory, and single-stepping through user instructions or microcode for the debugger.

3.3. Asynchronous Multiprocessing

In standard UNIX, the system call interface is implemented through the execution of some sort of trap instruction. The processor begins execution of the kernel trap handler in privileged state, but still in the context of the user process, i.e. the processes MMU/VM state and *u* structure. The Culler 7 can have multiple user processes active at any one time. For each user processor in the system there can be active processes on the both the foreground and background halves, and

the kernel processor itself may have a process active either in user mode, or running inside the kernel. To simplify discussion, we coined the term *asynchronous multiprocessing* to describe this characteristic of the Culler 7 system.

3.3.1. Asynchronous System Calls

Asynchronous multiprocessing requires a means for creating arbitrary kernel mode contexts at interrupt level. This is implemented by an interrupt handler that associates an interrupt from the SCI with the kernel state of the user process that executed *a_ikps*, and a routine that builds a kernel-stack for the process and places it at a high priority on the kernel run queue. When the kernel processor is about to return to user mode, e.g., when returning from the SCI interrupt, a kernel context switch will occur. This mechanism is similar to the use of *aston()* in 4.2BSD. User processes don't have any 68010 user MMU state to be faulted in, which reduces the cost of 68010 context switches.

The kernel-stack is constructed so that when the kernel mode context switch occurs, the process will return from *switch()* into the equivalent of *trap()*. This mechanism assumes that the kernel-stack for a user process is valid only while the process is executing within the kernel. As described below, this assumption is not always correct; there is a race condition which must be protected against. To build the kernel stack required that every loaded user process have its user structure mapped into the virtual address space of the kernel. This was not difficult to implement, but does consume slots in the kernel page table.

A user process can exit the kernel only by terminating or being staged onto a user processor and calling *switch()*. The calls to the staging driver to stage the process can complete before *switch()* is called. Thus for processes that have only small amounts of state to stage, there is a race between the process on the user processor and the kernel. If the process tries to reenter the kernel before the kernel has switched away from the process' kernel-stack, the SCI interrupt routine may try to build a new kernel context over the top of the stack during the call to *switch()*. This hazard is avoided by deferring alterations to the kernel stack in cases where the process reenters the kernel immediately.

3.3.2. Stager

The stager is responsible for processor allocation, staging and destaging of process state from the user processors, and the manipulation of the hardware context switching support on the SCI.

When a user process becomes ready to exit the kernel, an attempt is made to allocate to it the background half of an available processor. If all processors are busy, the process is placed on the user processor run queue until a processor becomes available. Once a processor is allocated to the process, the staging transfers are queued up for the SCI.

One of the optimizations within the stager is that a process may *reclaim* the processor that it was last on without having to be staged. This circumstance occurs when a process enters and exits the kernel without blocking.

A user process explicitly releases its processor inside *sleep()* and *exit()*. The routine called to release the processor triggers the allocation of the processor to an awaiting process for staging. The kernel does not have to be executing in the context of a user process in order to stage it. Staging is implemented using interrupt driven queues, similar to the *buf* queues used by the *strategy/start/interrupt* routines in block I/O drivers.

When a process becomes runnable, it may be placed on the user processor run queue because there is no available processor. The user processor run queue is maintained separate from the normal kernel run queue, which is used to schedule the 68010. Both run queues are implemented with 32 linked lists of processes. Processes are moved from one linked list to another as their priority changes. When a processor becomes free, the highest priority process on the user processor run queue is allocated to the free processor and staged.

3.4. Clock Considerations

The CSD kernel required changes in the areas of priority adjustment, interval timer support, time accounting, profiling, and time slicing.

3.4.1. Priority Adjustment

For the purposes of priority adjustments, the 4.2BSD clock interrupt code looks at the currently mapped *u* area to determine what process was active at the time of the interrupt, and looks at the stacked *psw* to see if the process was executing in kernel mode or user mode. On the Culler 7 there may be up to nine processes (foreground and background of each of four user processors, and a system process) that appear to be executing simultaneously. The kernel is careful to not tick processes that have been staged, but are still on the background half of the processor.

3.4.2. Process Timing

The user processors are too fast for the resolution of the system clock interrupt to be useful for measuring a process' user time. The CSD kernel takes advantage of a 3-microsecond resolution clock on the kernel processor to provide more accurate user time accounting for user processes. System time accounting is still done using the statistical sampling in the clock interrupt routine. The 4.2BSD interval timers run off of the data collected from the high resolution clock.

3.4.3. Time Slicing

4.2BSD periodically switches between processes of the same priority by calling *roundrobin()* from a *softclock* interrupt. *Roundrobin()* preempts the currently running process, which will call *switch()* when it enters the kernel and start another process running. The same mechanism is used for user processes on the Culler 7, but care is taken to stagger the preemptions so that the user processes don't all try to enter the kernel at once. The staging driver always keeps the user processors full of available processes, so sending a kernel interrupt to a user processor is sufficient to effect a context switch.

3.5. Virtual Memory Implementation

4.2BSD was originally implemented on the VAX architecture. At the conceptual level the VAX looks up every page table entry in main memory, while at the practical level most address translations are cached in the instruction lookaside buffer. The Culler 7 has two-way set-associative page tables implemented in the hardware. For the CPU intensive programs found in scientific and engineering applications, there is minimal cost associated with loading the page maps because the transfers are overlapped with process execution, using the alternate set of maps.

It is inconvenient to manipulate the images of the map directly, and so VAX style page tables are maintained in kernel memory. PTEs from the kernel page tables are written to the D-memory images of D-map and P-map by a routine that handles the set-associative details of the map

entries.

3.5.1. Separate I and D Considerations

A significant change made to the virtual memory code was the creation of a separate page table for text. The VAX architecture uses a single page table (P0) to map both text and data. The 4.2BSD implementation of shared text required that there always be at least one process loaded for each in-core text so that the front of the process' P0 page table could be used to maintain the page table for the text. Every operation made to a page table entry for text had to be distributed to the P0 page tables for all other loaded processes that were attached to the text.

The 4.2BSD page table mechanism for text would have been difficult to implement on the Culler 7 because text and data reside in overlapping virtual address spaces. Instead we allocate a separate page table for text and associate it with the text structure. This simplified the implementation of shared text at the cost of increased usage of kernel memory.

3.5.2. Program Text

The instructions executed by a program reside in P-memory. There is a separate P-memory for each user processor, but any P-memory may contain pages for more than one process concurrently. Support for P-memory was provided by adding a layer on top of the virtual memory code. The text segment is maintained in 8KB D-memory pages. When a P-page fault is encountered, the kernel transfers a 2KB subpage from D into P using the SCI staging facility. If the page was not resident in D, the page must first be paged in from disk, using the same mechanism that handles data faults in D-memory.

Reclaiming pages of P does not require a separate pageout daemon. P-memory is relatively small, its pages never get dirty and the penalty for having to transfer a page back in from D-memory is fairly small. The P-memory allocator simply takes what it needs by reclaiming the least-recently used P-page that it can find. The allocator never needs to block, and can even be invoked at interrupt level.

3.5.3. Pageout Daemon

Supporting asynchronous multiprocessing posed some interesting problems for the D-memory pageout daemon. In a uniprocessor, pages may be taken at will by simply unmapping them. Because there is only one processor, there cannot be a user mode process racing to modify the reference/dirty bits while the kernel is examining them.

On the Culler 7, the pageout daemon may be running in parallel with several executing user processes. The pageout daemon is biased against stealing pages from running processes, but when it becomes necessary, special care is taken to check the hardware reference/dirty usage bits again after the page has been unmapped. Unmapping the page is itself complicated because the process must be temporarily put on the background of the processor in order to modify its maps via the staging bus.

4. Conclusions

The architecture of the Culler 7 multiprocessor has enabled us to make a straightforward implementation of 4.2BSD. By running the operating system on a separate kernel processor, and starting with an existing kernel port, many aspects of the implementation were simplified. We were able to spend more of our time implementing *visible* functionality, such as multiprocessing and

staging, and much less time making *invisible* changes to boilerplate code like `locore.s` and the device drivers.

Because the kernel didn't have to be bootstrapped, it supported a fully functional development environment from the beginning. We were able to use the capability of running processes on the kernel processor to incrementally integrate the user processors into the system. In the early stages when the hardware and compilers were still shaky, occasional compiler bugs and hardware failures in the user processor or SCI resulted in fewer mysterious kernel crashes than would otherwise be expected.

The most significant advantage of our approach is that while for the short-term we have avoided the hard problems of distributing the operating system onto multiple processors, we have created a platform from which functionality can be migrated onto the user processors based on analysis of performance bottlenecks for real applications. The first code to be migrated was the signal handling mechanism. Future operating system releases are expected to transfer more of the virtual memory and interprocess communication functionality onto the user processor.

The involvement of the operating system implementors early in the design of the Culler 7 architecture allowed design trade-offs to be made jointly from the viewpoints of both the operating system and the hardware implementation. The result was a system that fits naturally into the framework of 4.2BSD UNIX. It was designed with 4.2BSD in mind from the beginning.

Contributors

The initial design work was done by Dave Probert. Jeff Berkowitz and Mark Lucovsky completed the design and did some of the trickier parts of the implementation. Steve Byrne modified the code for the pageout and swapper daemons, the buffer cache, and the disk and tape drivers. Dave McMillen wrote KUP0. John Gerngross uncovered numerous bugs in our system and 4.2BSD in general by applying systematic testing to the kernel.

References

- [Leffler84] Leffler, S., Karels, M., and McKusick, M. K., *Measuring and Improving the Performance of 4.2BSD*. Proceedings of the 1984 Summer USENIX Conference at Salt Lake City
- [Goodman85] Goodman, J., et al, *PIPE: A VLSI Decoupled Architecture*, Proceedings of the 12th Annual International Symposium on Computer Architecture

Porting UNIX to the System/370 Extended Architecture

Joseph R. Eykholt

Amdahl Corporation
Sunnyvale, California

ABSTRACT

The UNIX¹ operating system has been ported by various groups to the System/370 mainframe architecture. The new System/370 Extended Architecture offers capabilities that improve the power and flexibility of UNIX systems. These capabilities include a significantly larger address space, an enhanced I/O subsystem, and advanced multiprocessor features.

This paper describes an implementation of Amdahl Corporation's UTS², a System/370 UNIX product that takes advantage of the Extended Architecture. Emphasis is placed on the design considerations required for the new features of XA that are unique to very large mainframe UNIX implementations.

1. A comparison of System/370 and System/370 Extended Architecture

To understand the motivations and effort involved in porting UNIX from System/370^[1] to the System/370 Extended Architecture^[2] (XA), it is necessary to understand the differences between these two architectures. Readers who are more familiar with minicomputer and microcomputer architectures may find the differences interesting, especially the difference in scale. This description is a bit simplified, leaving out details not essential to the discussion.

The Extended Architecture has provided new features and removed some of the restrictions imposed by the earlier System/370 architecture. This extension has been provided in such a way that user programs written for System/370 should require no change to execute correctly. However, the changes to the operating system were significant. Therefore, this discussion will concentrate on the areas which are different between the two architectures.

1.1 System Architecture

Figure 1 shows the primary connections between the major components of a System/370 mainframe. Several central processors may be present, sharing main memory. Currently, the largest non-XA System/370 implementations can support two CPUs, 64MB of main memory, and 32 I/O channels. With XA, the Amdahl 5890-600 will support four CPUs with 512MB of main memory and 128 channels.

1.2 I/O Architecture

The I/O processors operate somewhat autonomously, transferring data between main memory and devices via the channels. Each channel is in many ways similar to an intelligent direct memory access (DMA) controller, especially with respect to the way it is viewed by the CPU. Internally the channel has a processor that can be as powerful as some minicomputers. Channels multiplex their activity to permit several transfers to be active simultaneously on the same channel. Each channel can have a transfer rate of up to 3 megabytes per second. The operating system performs I/O operations by building a channel program, and signaling the channel to start executing it. The channel program is simply a sequence of one or more channel command words (CCWs), which each

1. UNIX is a trademark of AT&T Bell Laboratories.

2. UTS is a trademark of Amdahl Corporation.

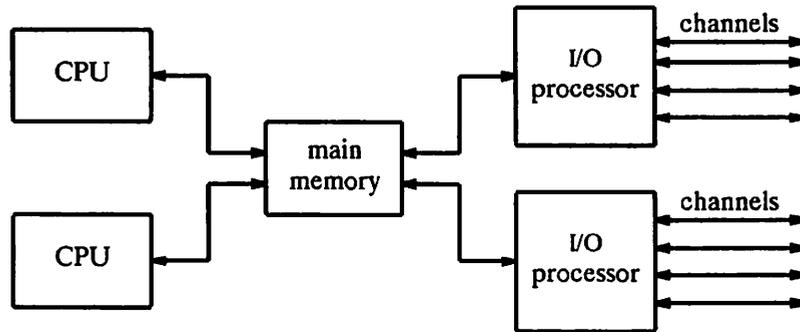


Figure 1. System/370 mainframe organization

describe an operation (read, write, seek, etc.), and a data buffer to be used. The channel communicates status back to the CPU with an I/O interrupt, during which a channel status word (CSW) is stored.

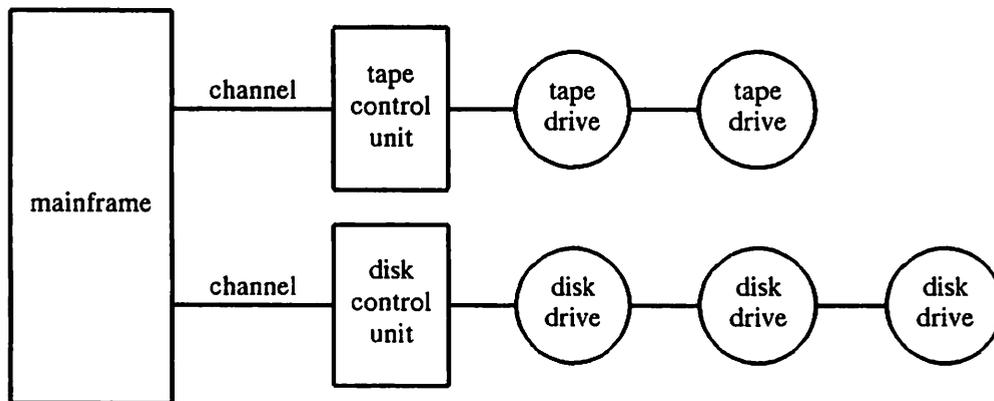


Figure 2. Example channel, control unit, and device connections

1.2.1 Channels, Control Units, and Devices Each I/O device has a control unit that is connected to the mainframe channel. See figure 2. The control unit is sometimes integrated into the device or the channel, but there is always the logical concept of a control unit, whether it exists as a separate physical entity or not.

Most control units serve several devices. For example, a disk control unit will typically control as many as 32 disk drives.

1.2.2 Alternate Paths and Sharing Devices can be attached to one or two control units, and control units can also be attached between up to four channels, as shown in figure 3. Attaching a device to two control units is useful because it can provide an alternate path to a device when a control unit is busy. Attaching control units to different channels is useful for providing alternate paths when a channel is busy and for sharing devices among several mainframes.

Another reason for alternate paths is to enhance reliability. When a path fails because of a hardware problem, the error can usually be detected and the path disabled. If there is an alternate path that remains operational, access to the device is not lost.

1.2.3 Busy Conditions All this sharing makes it inevitable that busy conditions occur. Since the device, control units, and channels cannot queue all requests, the I/O supervisor in the operating system has to be aware of the configuration and manage the appropriate queues. With alternate paths to devices, the I/O supervisor can re-route requests to another channel.

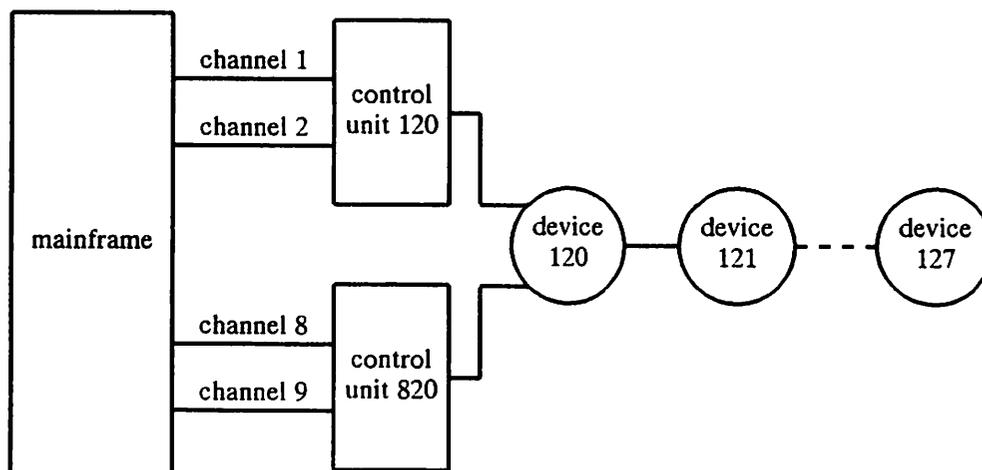


Figure 3. Example alternate paths connections

1.2.4 The XA I/O Subsystem One major area of difference between System/370 and the Extended Architecture is the I/O subsystem, which consists of the I/O processors, channels, and the associated microcode and tables. In XA, most of the queuing and path selection functions of the I/O supervisor were moved into the I/O subsystem, freeing the operating system's I/O supervisor from the overhead of those tasks.

To provide these functions, the XA I/O subsystem requires that it be told about the configuration of the peripheral farm. This is done with an I/O configuration file that is used to build tables in the I/O subsystem's memory during system initialization. I/O requests and interrupts no longer refer to the channel and unit address, but use an arbitrarily assigned subchannel number that corresponds to the device involved. For operator convenience, there is a user-assigned device number that is used in all console messages and administrative commands. This device number is mapped into the subchannel number by the operating system.

1.2.5 Multiprocessor Considerations In a System/370 multiprocessor configuration, the channels are divided into one or more channel sets. Each CPU may connect dynamically to at most one channel set, and at most one CPU can be connected to each channel set. For more than one CPU to be capable of I/O, there can be alternate paths to the device, one from each channel set, and each CPU can use the channel set connected to it. This reduces the options open to each CPU in initiating the I/O when its path is busy. I/O interrupts providing status on a request always occur on the channel that issued the request, meaning that the same CPU that initiated the request must take the interrupt.

In the System/370 Extended Architecture, there is only one channel subsystem, regardless of the number of processors, and any CPU can start I/O requests to any device. I/O interrupts can be presented to any CPU that is enabled for them. Therefore, XA is a much more suitable environment for multiprocessor operating systems. Because the CPUs no longer have all of the burden of managing I/O, higher levels of multiprocessing can be supported by XA.

Although the current dual processor implementation of UTS does I/O from only one CPU, XA allows us to begin development toward a completely symmetrical multiprocessing implementation.

1.3 Addressing

In System/370, all effective addresses are 24 bits wide, limiting the address space to just 16 megabytes. When a 32-bit register is used to generate an address, the high order byte (ordinarily the most significant) is ignored. Unfortunately, this feature has been exploited by assembler programmers in many ingenious ways. It has also been exploited by the architecture, by using the high order byte of words containing addresses for other purposes. For example, the channel

command word (CCW) contains the buffer address in bytes 1, 2 and 3, but uses byte 0 for the command code.

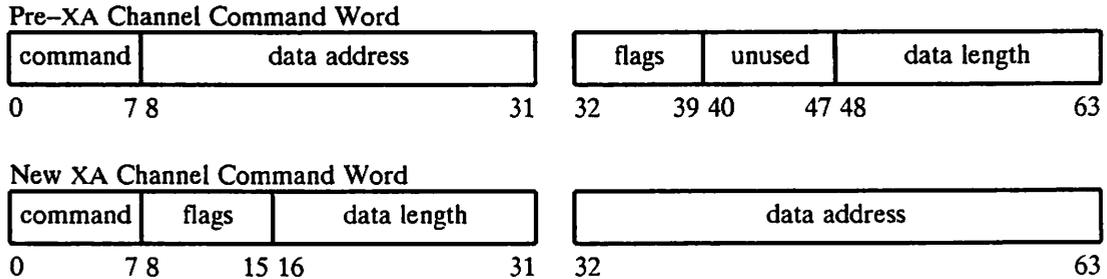


Figure 4. Old and new channel command word formats

XA provides compatibility with 24-bit mode, but also provides a 31-bit addressing mode. This allows an address space of up to two gigabytes. To provide larger address fields, most hardware-associated structures, including the PSW, CSW, CCW, and page table entries have changed. Figure 4 shows the CCW formats, both of which are available in XA.

1.4 Virtual Memory

Non-XA systems can have 64K or 1M segments, and 2K or 4K pages, although only 64K segments and 4K pages are standard features of the architecture. Although programs can directly access only 16 megabytes, the non-XA page table entry, shown in figure 5, allows support of up to 64 megabytes of real memory, by squeezing two extra bits of real address into bits 13 and 14.

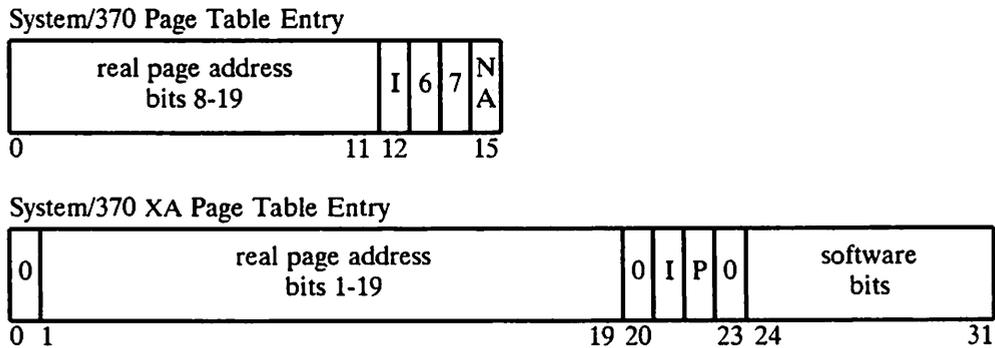


Figure 5. Old and new page table entries

XA systems only provide 1M segments and 4K pages. To support larger virtual and real address spaces, XA uses a new segment table and page table format. To accommodate the larger page frame address, the XA page table entries are 4 bytes wide. The maximum size of the hardware page tables has grown from 32 bytes to 1K bytes, and the segment table from 1K to 8K bytes.

2. Motivations for Porting UTS to XA

The prime motivation for using XA mode machines has been from applications that require address spaces greater than 16M. So far, these applications have been mostly digital logic simulations, and other LSI and VLSI chip design applications. One user, doing LSI design work under IBM's non-XA VM operating system, must run all applications larger than 16M bytes on minicomputers. Some estimate that even the two gigabyte address space provided by XA will soon be insufficient for certain applications.

Since processor speeds are constantly improving, the 64M byte limit of mainstore was becoming a system bottleneck for some operating systems. Larger real memory sizes allow more users to be supported, and allow better swapping policies. Already XA systems have been announced which will provide 512 megabytes of real memory. If the trend in memory size growth continues, 31 bits

will not be enough for real memory addressing.

Currently all machines that support the XA mode can also be used in non-XA mode, by selecting the mode during system initialization. However, it is clear that XA is the preferred mode of operation on these machines.

3. UTS Changes Required for XA

3.1 I/O Supervisor Changes

Since many of the functions of the 370 mode I/O supervisor have been taken over by the hardware, the XA version software is much simpler than its 370 mode counterpart. All queuing for channels and control units was removed, since the operating system no longer requires any knowledge of them.

3.1.1 Error recovery The XA I/O subsystem reports channel-detected errors in a new way. All I/O interrupts must come from a subchannel. If the I/O subsystem detects an error on a channel interface that cannot be associated with a subchannel, it presents a new class of machine check, called "channel report pending." When this machine check occurs, channel report words are obtained with a new instruction, and the contents are used to guide the recovery.

The method by which errors that can be attributed to a device are reported hasn't changed significantly between the two modes.

3.1.2 Path Selection Control Path selection in XA is performed by the I/O subsystem, so that function was easily removed from the UTS I/O supervisor. It is sometimes necessary for the system administrator to have control over which paths to a device are enabled (varied on). If, for example, a control unit failure caused the recovery system to disable paths using that control unit, those paths would need to be manually enabled after repairs are completed. Control over which paths can be selected is still available to the system administrator through the *vary* command. This command and its associated system call were modified so that they would function in both XA and non-XA modes.

3.2 Device Driver Changes

To perform I/O operations, the device driver calls the start I/O routine with the device address, a CCW pointer, a pointer to a function to handle the interrupt, and an argument to be passed to that function. When an interrupt is accepted for that request, the interrupt handler function is called with the given argument and the channel status word.

It was possible to maintain the same interface between the device drivers and the I/O supervisor for both architectural modes. The device drivers call the I/O supervisor routines with device numbers, and the I/O supervisor translates these to subchannel numbers before performing the requested operation. The channel command and channel status words are described in C language structures that use bit fields to declare the status and flag bits. To reflect the new CCW and CSW formats, it was necessary to redefine these structures. This only required changing the sequence of the existing fields and adding new fields. Some drivers, which did not use the defined structures to access CCW and CSW fields had to be modified so that they now do, and so now all drivers have the same source code for both architectural modes, with *ifdefs* only in the header files.

3.3 Memory Management Modifications

The page table and segment table format change was handled by simply changing the structure definitions in the header file. Most of the memory manager source is common to both modes.

3.3.1 Copy on access bit The UTS memory manager has a feature that delays copying pages after a *fork* until the pages are referenced. During a *fork* the page tables for the data and stack segments are copied with all pages marked invalid and a "copy-on-reference" bit set in each page table. If the pages are referenced, a page fault occurs and each process is given its own valid copy of the page.³ In non-XA mode, there were no free bits in the page table entry, so a word was used to hold

the copy-on-reference bits for each of the 16 entries. In XA mode there can be up to 256 entries in a page table, so a bit map could be used, but since there are eight bits in each XA page table entry that are reserved for use by the operating system, one was used as the copy-on-reference bit. Code which inspects or alters these bits was changed to use a preprocessor macro, which is redefined for XA mode.

As previously mentioned, the maximum size of page tables increased from 32 to 1K bytes, and the segment tables from 1K to 8K bytes. Since UTS currently manages free space in 4K pages, it was decided to use only 4K byte segment tables temporarily. This still provides users with 1 gigabyte address spaces, which should be enough until the memory allocation routines are rewritten.

Since most text and data segments are nowhere near 256 pages long, the large page tables are wasteful users of real memory. Consider that most UTS utilities use around 20K bytes of memory in three segments (text, data, and stack). This requires three page tables and a segment table, for a total of 7K in translation tables. Eventually, the memory manager will be modified so that it adjusts the page and segment table sizes for the process.

3.4 Changes to the exec System Call

24-bit addressing is usually sufficient, and a few applications require it, so it was decided to make that the default mode for user processes. The architecture provides the ability to use shorter segment tables for 24-bit address spaces, so UTS may eventually take advantage of this and handle small address spaces more efficiently. It seems desirable to allow the kernel to change the default in the future, so two flags were added to the Common Object Format File (COFF) header to identify programs with special requirements. One flag is for programs that need their address space to be as large as possible. The other is to specify that the executable must be run in the 24-bit addressing mode.

3.4.1 Requirements for shared text segments UTS requires that the text and data sections of shared text programs not share the same page table. This provides for a more efficient method of mapping the text sections into multiple address spaces. Since XA increases the amount of memory represented by each page table from 64K to 1M bytes, shared text programs for XA machines must have their data segments in a different 1M section than their text. The changes to the kernel allow shared text programs that do not obey this constraint to be treated as non-shared programs. The loader was changed to start data sections on a 1M boundary by default, even for non-XA machines. Although some of the address space is wasted by this restriction, it does not change the number of real memory pages used.

3.4.2 The Gigabyte core file When a program aborts, its address space may be larger than the available disk space for a "core" file. If this occurs, it should be still possible to write the stack segment or at least the user page to the file.

3.5 Utility Changes

Several standard utilities required changes. Where possible, changes were made in such a way that the program would work in both modes. User applications should require no change at all. However, re-linking of user programs may be desirable to take advantage of shared text.

3.5.1 CCS Changes The C compiler was changed to allow the new 31-bit addressing subroutine linkage instructions to be used. This is necessary only on those programs that have code sections larger than 16M bytes.

3. It would have been better to do the copy only on references that modify the page, but the architecture doesn't provide enough support to do that cleanly, and it seems unlikely that performance would be improved greatly.

The assembler was changed to add several new instructions, including new instruction formats, primarily for use by the few small assembler routines in the kernel. The linker was modified to allow the large address space flag to be set, to handle larger sections, and to change the default alignment of data sections, as discussed earlier.

3.5.2 ps The portion of the *ps* command that prints the command line arguments of processes currently operates by reading through the translation tables to find the stack page of the process, either in memory or on a paging device. Since the header files containing page table access macros were modified for the kernel so that they work correctly if the preprocessor symbol *XA* is defined, *ps* worked correctly in *XA* mode after it was recompiled. However, it was irritating for us to have to recompile *ps* every time we switched our test system between *XA* mode and non-*XA* mode. In order to save ourselves that bother, the code that prints the command line arguments was placed in a separate module which was compiled twice, once with the preprocessor symbol *XA* defined, and once without it. The routine name was also changed for the *XA* case. It was modified so that it could determine at execution time whether it is being used on an *XA* machine or not, and use the appropriate routine.

Eventually, it may be possible to eliminate this code in *ps* entirely. Some systems store the command line arguments in the process's user structure, freeing *ps* from having to know the layout of the page tables. This has the disadvantage that programs cannot hide their arguments, as the *crypt* command does. When the */proc* file system is available, providing a file for each process's memory image, *ps* could use these files to read the arguments. ^[3]

3.5.3 sysdump The dump analyzer program, *sysdump*, which is similar in function to *crash*, formats several system structures, and some of these have changed format for *XA* mode. The technique used for *ps* couldn't cleanly be used for *sysdump*, because of the number of special cases. Instead the *makefile* for *sysdump* was modified to make both a non-*XA* version and an *XA* version. These both come from the same source, except that the latter one has *XA* defined. A shell script decides which version to use based on the kernel type in the memory image.

3.5.4 booting The *ipl* (initial program load) program has been changed to adapt to the architecture of the machine, and to check the kernel type after it is loaded to be sure the correct kernel has been specified.

4. Summary

UTS has been adapted from the System/370 environment to the System/370 Extended Architecture. This has been done with little or no impact on applications programs. The resulting system provides a much larger address space, to support larger applications, as well as larger real memory configurations.

The changes to the system allow kernels for the two modes to share most of the source code. A UTS system with these modifications that is running in 370 mode can be converted to *XA* mode or vice-versa without changing any programs or files, except for those that require the special features provided by *XA*.

After reading this paper, one might conclude that the UTS kernel is very different from other UNIX ports. While there are significant differences, that perception may be exaggerated because we have been describing only those low-level portions that are different. The rest of UTS is very recognizable as the UNIX we all know and love.

5. Acknowledgements

A special thanks to George Cameron who shared in the conversion effort. Thanks also to John Marshall, who quickly modified the loader and assembler, and fixed several problems with the *exec* system call. Thanks also for the generous support by the entire UTS group at Amdahl.

REFERENCES

1. *IBM System/370 Principles of Operation*, Publication number GA22-7000-9.
2. *IBM System/370 Extended Architecture Principles of Operation*, Publication number SA22-7085-0.
3. *Processes as Files*, T. J. Killian, USENIX 1984 Summer Conference Proceedings.

Full Duplex Support on Mainframes

Don Sterk

Amdahl Corporation
Sunnyvale, California

ABSTRACT

The usual method of providing character by character I/O with echoing by the host requires two interrupts per character. Minicomputers frequently become bogged down with terminal I/O with tens of users. This paper examines how to provide full duplex ASCII support on a mainframe for hundreds of users without noticeable delay. A case study is made of UTS¹ full duplex using an Amdahl 4705 as a front end and a packetizing scheme preventing the necessity of an interrupt for each character read.

INTRODUCTION

Although large mainframes provide the power for rapid computation and processing large blocks of I/O, they may incur high overhead per I/O transaction. The character-by-character nature of full duplex I/O leaves them prey to the same problem minicomputers have: becoming sluggish as the number of full duplex users increases. If an interrupt is received every time a character is read and another when the echo completes, a mainframe may not be able to support as many users per MIP as a minicomputer. The greater expense of a mainframe justifies the cost of a front-end processor to relieve it of some of its I/O overhead and thus use its greater computational power more effectively. This paper will discuss the implementation of full duplex support in UTS, which uses an Amdahl 4705 as a packetizing front end processor for an Amdahl 470 or 580 mainframe.

1. UTS FULL DUPLEX ENVIRONMENT

Before the architecture of UTS full duplex can be explained, an introduction to I/O on System/370 compatible computers will be presented.

1.1 System/370 I/O Overview

I/O on a System/370 computer is performed by a *channel*, an auxiliary processor that shares main memory. A channel executes *channel programs* to transmit and receive data through main memory. It is connected via cables to a *channel adaptor* in a peripheral controller. A channel program consists of one or more *channel command words*, or CCW's. A CCW is 64 bits long and has fields designating a command, operand address, operand length, and numerous options. Execution normally proceeds through a sequence of CCW's until one is found with the *command chain* bit set to 0. A special command, *Transfer In Channel*, or TIC specifies that the next CCW to be executed is to be found at the address contained in the operand field of that CCW. A typical CCW would be to read *n* bytes of data into a buffer.

The device address on which the I/O is to be performed is designated by an argument to the *Start I/O* or *sio* instruction, by which the mainframe submits a channel program for execution by a channel. The device address consists of a channel address, designating a particular channel, and a subchannel address, designating a particular device attached to that channel.

The completion status of a channel program is presented as an interrupt in a *channel status word*, or CSW. Besides the completion status, the CSW contains the address of the CCW that generated it. One of the options available in a CCW, *Program Controlled Interrupt*, or PCI, specifies that an interrupt is to be received when that CCW starts to execute. In this way the progress of a long channel program can be

1. UTS is a trademark of Amdahl Corporation.

monitored.

1.1.1 UTS I/O Scheduler UTS has an I/O scheduler that permits each driver to submit a channel program and designate an interrupt routine to handle the interrupt. The scheduler queues such I/O requests to minimize busy conditions on channels, controllers, and devices, and handle them when they do occur. If the interrupt indicates an unusual termination, it requests sense data from the device and provides it to the interrupt handler.

1.2 The 4705 Communications Controller

A 4705 is Amdahl's version of the standard communications controller for System/370. It can have up to 350 data lines attached to it, which are accessed by the host through up to 255 subchannel addresses. The mapping from external lines to internal subchannel addresses is provided by the software in the 4705. A 4705 can support both asynchronous and synchronous data communications. It can be used for communicating with remote job entry systems, line printers, and both ASCII asynchronous and EBCDIC BSC terminals. It is often used to provide virtual terminal support from one computer to another. Since these other communications media are often needed by UTS customers, the 4705 was chosen as the front end processor for UTS full duplex communications. Figure 1 shows the physical configuration of the UTS full duplex system.

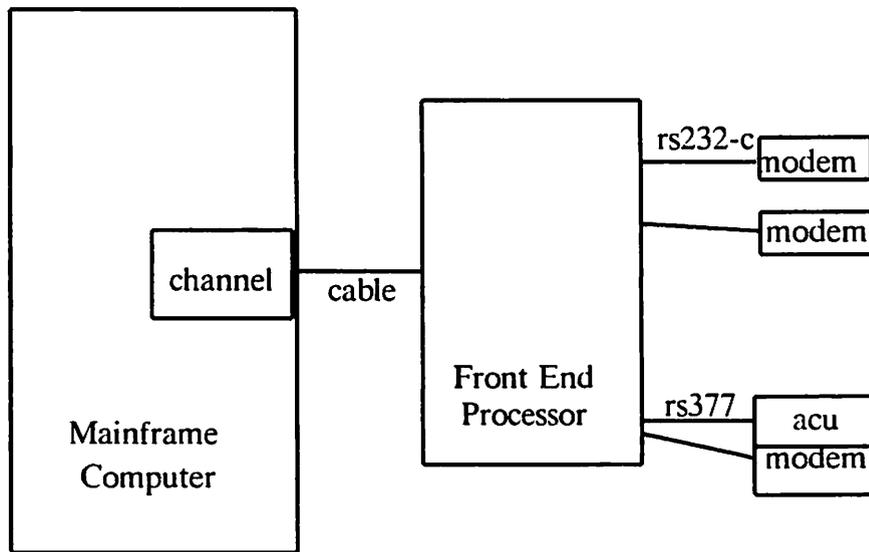


Figure 1. UTS Full Duplex Physical Configuration

1.2.1 4705 Software Various control programs with different capabilities can be run in the 4705. The one selected for UTS full duplex is the Emulator Program, EP, which emulates an older controller. EP is available in public domain versions and can support half duplex ASCII terminals. Other vendors have modified EP to provide X.25 support, so UTS customers with a 4705 can run full duplex, half duplex, X.25 and remote 3270 terminals through it. The modification made to provide UTS full duplex support in EP is called UTS/F.

2. UTS Full Duplex Considerations

The two main constraints in designing UTS full duplex support was to provide a full duplex interface as described in the System V *termio* manual page while reducing the amount of overhead normally incurred.

2.1 Full duplex Requirements

The term *full duplex* means many different things to different people depending on their background. The literal definition is that data can flow in two directions at the same time. Since the hardware in the 4705 that interfaces to asynchronous lines can only provide I/O in one direction at a time, we were required to use two asynchronous lines for one full duplex line: one for reading, the other for writing.

Most UNIX² users do not think of full duplex as simply handling I/O in both directions at once. Instead, they associate it more closely with the following features.

2.1.1 Character by Character I/O Some data links, such as X.25, permit data to flow in both directions at once, but packetize blocks of characters together for transmission. Data is only sent to the host when a line end character is seen or a minimum number of characters is read. Although UNIX canonical input only presents data to user programs when a line of data has been read, UNIX raw I/O presents individual characters as they are typed by the user. Therefore, although a substantial overhead reduction can be achieved by packeting the characters typed a line at a time, we could not rely on this for UTS full duplex support.

2.1.2 Echoing The *termio* manual page specifies that a user program can enable and disable echoing via an *ioctl*. The echoing can be performed by the front end processor if the *ioctl* to turn echoing on and off is transmitted to it. But since most users like knowing that what appears on their screen has been received by the host computer, we decided to have the host echo.

2.1.3 uucp A major incentive for providing full duplex support is to run uucp to communicate between systems. This requires dial out capability to initiate communications. Therefore, UTS full duplex support was designed to permit the use of the standard library routine *dial*.

2.1.4 vi Editor The other popular program requiring full duplex support is the vi editor. As a measure of compatibility, it was decided that UTS full duplex had to support the vi editor without modification. To do this, the standard line discipline is used with the same *termio* structure, so that the TCSETA and TCGETA *ioctl* calls have the standard UNIX behavior.

2.2 Overhead Reduction For Large Number of Users

Despite the above requirements, UTS full duplex must support many users without degrading performance. To provide character by character I/O without receiving an interrupt for every character typed, UTS multiplexes the data read from all the terminals on a 4705 onto a single packet subchannel.

3. UTS Full Duplex Solution

Figure 2 shows the logical configuration of UTS full duplex. Each terminal is accessed by UTS through its own read and write subchannels. All data is written to the write subchannel. The read subchannel is only used to open the terminal and to receive line conditions, such as break and hangup. A single packet subchannel per 4705 is used to receive the read data from all the terminals.

3.1 Full Duplex Packets

A packet of data is sent from the 4705 to the host every 100 milliseconds when there is data present, or whenever its buffer fills up. This is sufficiently frequent to appear no slower than direct full duplex to most users. Each packet contains a header and data sections. (See figure 3). The header contains status information and the size of the data section. The header status, sense, and address are used to indicate

2. UNIX is a trademark of AT&T Bell Laboratories.

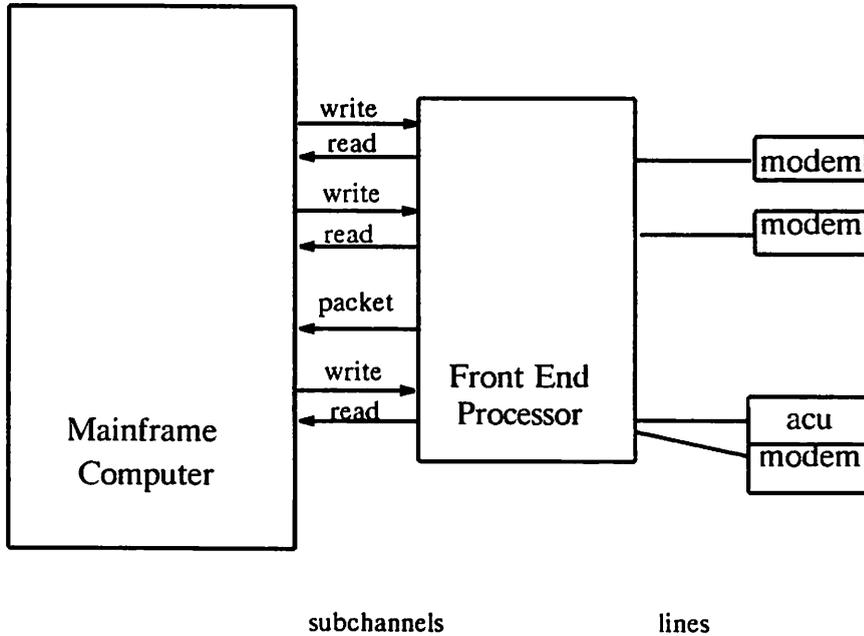


Figure 2. UTS Full Duplex Logical Configuration

Packet	
unused	address
status	sense
count	
Packet Body	
address	data
address	data

Figure 3. UTS Full Duplex Packet Format

overrun conditions on the packet subchannel or individual lines. The data section contains address/data pairs. Each pair contains a byte of data and the subchannel address of the terminal that presented it.

3.2 4705 Software

The EP control program was modified to collect the data from full duplex lines in a common buffer instead individually. The timer was used to generate an interrupt every 100 milliseconds (its finest granularity) and the interrupt handler changed to check for presence of data to send to the host. Additional code was written to handle the interface to the host on the packet subchannel. Autocall service was already provided by half duplex code in EP. The autocall data is written via a separate line to the autocall unit attached to the modem.

3.3 UTS Software

Two drivers are used for full duplex, the autocal (ACU) driver for out-dialing and the full duplex driver for ordinary I/O.

3.3.1 ACU Driver The acu driver is only used to write the number that is to be dialed. This is done by issuing a CCW with a special DIAL command on the write subchannel of a full duplex line. This is mapped by the 4705 software to the autocal line.

3.3.2 Full Duplex Driver The full duplex driver must handle the read and write subchannels as well as demultiplex the data from the packet subchannel to the respective tty structure within the kernel. Accessing the driver only occurs through the line discipline, which handles buffering and copying data between the user and the tty structure.

3.3.2.1 Opening a terminal If a terminal is being opened for the first time, an *enable* command is issued to the read and write subchannels. These commands are processed by the full duplex software in the 4705. The write subchannel will enable immediately and return an interrupt. The read subchannel will not enable until the modem presents DSR, indicating a connection is made. This happens immediately for dedicated lines and when a carrier is detected on dial up lines. Once the read subchannel is enabled, the count of open terminals on this 4705 is incremented. If this is the first terminal open on this 4705, the packet line is also enabled. When it enables, a channel program to read the packet subchannel is issued. After the packet subchannel is initialized a *read* command is issued on the read subchannel and the *open* system call returns. The read on the read subchannel is not issued to receive ASCII data, only to arrange for notification when the line condition *hangup* or *break* occurs.

3.3.2.2 Writing to a terminal Writing to a terminal occurs through the line discipline. The driver itself simply transfers data from the tty outq to the transmit buffer and uses a *write* CCW on the write subchannel to write it.

3.3.2.3 Depacketing packets The channel program on the packet subchannel consists of a number of *read* CCW's with the PCI flag turned on, followed by a TIC back to the first *read* CCW. Each CCW will read in one packet. After the packet is read in the PCI in the next CCW causes an interrupt to be generated. The CSW presented by the interrupt points back to the CCW that generated it; the previous CCW is the one that just completed. Since the CCW contains the address of the buffer read in, the packet interrupt handler can find the buffer just read. Its header specifies the number of data/address pairs to process. Using the address of the packet subchannel and the subchannel addresses in the packet, the data is put in the *rbuf* of the respective tty structure. The standard line discipline transfers it to the rawq and performs canonical processing.

The *tic* eliminates the need to reissue the command when it completes.

The purpose of chaining several reads together is that under certain conditions PCI interrupts can be missed. By comparing the buffer address in the current CCW with the last one received, missing interrupts can be detected and their packets processed.

3.3.2.4 Reading from a terminal Since data is received on the packet subchannel and put into the rawq and canq, reading a terminal amounts to copying available data out to user space, and if none is present, sleeping until it is available. This is handled by the line discipline.

3.3.2.4.1 Closing a terminal When the last file descriptor to a terminal is closed, the driver terminates the *read* CCW on the read subchannel with a *halt device* command and issues *disable* CCWs on the read and write subchannels. It decrements the count of open terminals on this 4705 and if this was the last one, terminates the channel program on it.

3.3.2.4.2 Hangups A hangup is normally detected by the sense data returned on the read subchannel. However, if the hangup occurs while data is being written to the terminal, the *write* channel command will terminate with sense data indicating a hangup. In either case, the tty structure is updated to reflect its status and a signal is sent to the process group sharing the terminal. Any process reading the terminal will be returned a character count of zero, indicating the loss of carrier. The channel commands, if any, on the read and write subchannels are terminated. It is expected that the processes with the tty open will soon

close it, either explicitly or by exiting.

4. Features

The UTS full duplex solution provides the following features.

4.1 Reduction of I/O Overhead

By eliminating the interrupt for every character received, nearly half the overhead in full duplex support is eliminated. The largest single source of remaining overhead is in the single character writes incurred by host echoing.

4.1.1 Performance Our largest customer installation has 355 full duplex lines on 5 4705's, plus other synchronous lines including computer-to-computer links. Line speeds vary from 1200 to 9600 baud. There is no noticeable delay when typing.

4.2 IXON/IXOFF

IXON and IXOFF support is provided in the driver and line discipline. When a *cntrl s* is received for a terminal, the *ustop* bit is turned on in its tty structure. While this bit is on the line discipline will not present any data for writing and the driver will not transmit any data. When a *cntrl q* (or any character if IXANY is set) is received, the *ustop* bit is cleared and any pending data written.

A problem exists because there is no way for the driver to terminate a write in progress and determine the amount of data actually written. To avoid losing or repeating write data, the current write must complete. A special *wmax ioctl* and *stty* option are provided to permit the user to specify how many characters to write at a time, which controls how many characters may be transmitted after a *cntrl s* is received. Values between 1 and 255 can be specified; larger values provide greater throughput, smaller values quicken response to *cntrl s*.

4.3 Auto Call Support

As mentioned above, the standard *dial* library routine is provided. Thus, full *cu* and *uucp* service is available.

4.4 Use of Standard Hardware and Software

This solution required no additional hardware. Since the UTS group is mainly a group of programmers, this was a great advantage.

The 4705 software relies on the asynchronous service present in EP software. Although it is always preferable to not duplicate existing code, since there are no compilers for the 4705 and its instruction set is obscure, it was desirable to minimize the amount of 4705 code written.

Similarly, the full duplex driver relies on the standard line discipline for buffering, canonical processing, and post processing. This assures a uniform user interface and again avoids duplicating existing code.

4.4.1 Additional Services of 4705 Since the 4705 is the standard front end processor for Amdahl main frames, using it for full duplex also makes it available for half duplex and synchronous communications, often used for communicating with other mainframes and remote devices.

5. Complications

This section describes some of the restrictions and problems encountered.

5.1 Number of lines per 4705

Since there are only 255 subchannel addresses per channel, and each full duplex line requires two subchannel addresses, at most 127 full duplex lines can be attached through one 4705. The 4705 is not able to support even that many lines at high baud rates.

Each full duplex line interacting with three different subchannels introduces many complications handling error conditions.

Most of the problems installing UTS full duplex arise from the fact that there are several separate components that must be configured consistently. The packet, read and write subchannels are configured into UTS but must also be reflected in the 4705 EP generation and hardware. If UTS is running as a guest operating system under VM, the VM configuration must also agree with the UTS and 4705 configurations.

6. Other Possibilities

Several enhancements could be made to full duplex support that would reduce overhead or provide better support.

Most of the interrupts serviced are due to echoing single characters. If a multiplexed channel for writing was used, UTS could read the packets from the 4705, modify them to reflect the data not being echoed, and write the packet back to the 4705 for echoing.

IXON/IXOFF support could be handled more effectively in the 4705, but this would require its knowing some of the stty settings. No mechanism currently exists to convey this information to it.

7. Conclusions

A front end processor for UNIX full duplex can effectively reduce I/O overhead in a mainframe while remaining transparent to users. Providing a mechanism to convey stty information to the front end would permit offloading more full duplex features, such as IXON/IXOFF, from the host.

8. Acknowledgments

I would like to thank the entire UTS group for their patience, insight, and support in developing the full duplex driver. Special thanks to Y.C. Wang and Ludo Vennekens, who developed and support the full duplex software in the 4705. I also thank my manager, Hal Jespersen, for encouraging me to write this paper and his editorial suggestions.

Multi-Processor Management In The Concentrix Operating System

Jack A. Test

Alliant Computer Systems Corporation
42 Nagog Park
Acton, MA 01720

Abstract

Alliant Concentrix™ is the native operating system of the Alliant multi-processor machine family and is an enhanced and extended version of 4.2BSD UNIX™. Principal features include: symmetric implementation of UNIX on a multi-processor architecture, a two-Gigabyte demand-paged copy-on-write virtual memory system, shared-library image support for user programs, and management of multiple processors working concurrently on a single UNIX process. This paper describes the Alliant machine architecture and discusses the multi-processor management aspects of the Concentrix operating system.

Multi-Processor Management In The Concentrix Operating System

Jack A. Test

Alliant Computer Systems Corporation
42 Nagog Park
Acton, MA 01720

Abstract

Alliant Concentrix™ is the native operating system of the Alliant multi-processor machine family and is an enhanced and extended version of 4.2BSD UNIX™. Principal features include: symmetric implementation of UNIX on a multi-processor architecture, a two-Gigabyte demand-paged copy-on-write virtual memory system, shared-library image support for user programs, and management of multiple processors working concurrently on a single UNIX process. This paper describes the Alliant machine architecture and discusses the multi-processor management aspects of the Concentrix operating system.

1.0 Introduction

Alliant Computer Systems Corporation designs and manufactures high-performance, multi-processor computer systems designed primarily for use in scientific and engineering applications. The Alliant machine architecture provides a tightly-coupled environment consisting of interactive processors (IPs) and computational-elements (CEs) with a coherent global memory system. While every IP and CE is a fully functional, independent processor, CEs support integrated vector and floating point operations and can, through integrated concurrency operations, participate together as a “computational-complex” in the execution of a single application.

Alliant Concentrix, the native operating system for the Alliant multi-processor machine family, is an enhanced and extended version of 4.2BSD UNIX. A specialized uni-processor version of Concentrix called Diagnostix™ diagnoses hardware problems and configures the system for Concentrix operation. Tasks such as microcode loading and system sizing, for example, are handled by Diagnostix prior to Concentrix activation.

Concentrix	Trademark of Alliant Computer Systems Corp.
UNIX	Trademark of Bell Laboratories
Diagnostix	Trademark of Alliant Computer Systems Corp.

This paper is concerned with the multi-processor aspects of the Concentrix operating system and is divided into three sections. Section 2.0 describes the Alliant system architecture, Section 3.0 discusses the major attributes of IP and CE multi-processor support, and Section 4.0 discusses the management of CEs operating together on a single UNIX process.

2.0 Alliant System Architecture

The Alliant computer architecture supports up to twenty processors working in parallel to a coherent global physical memory. The processors fall into two classes: interactive processors (IPs) and computational elements (CEs). A diagram of the full Alliant architecture is shown below:

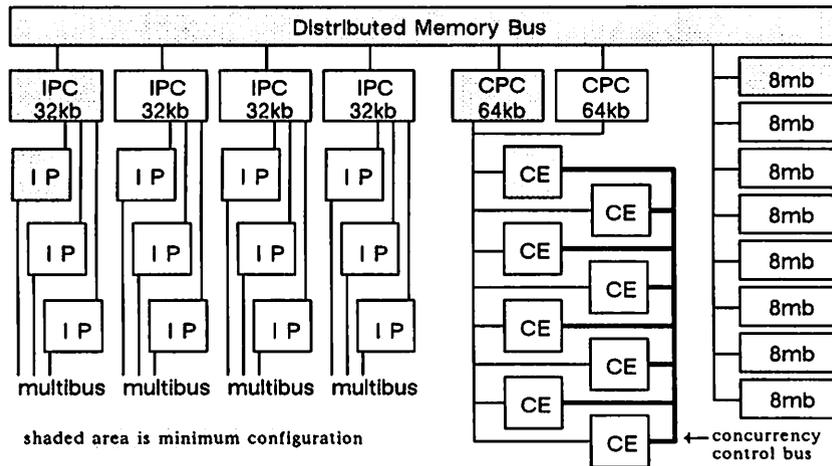


Figure 1: Alliant FX/8 System Architecture

As Figure 1 illustrates, the Alliant architecture is structured along interactive and computational lines. The interactive processors provide device support and non-compute-intensive user application support while the computational elements provide high performance computational power[1].

2.1 Global Memory

At the center of the Alliant architecture is the global physical memory system. The Alliant distributed memory bus (DMB) is a high speed, synchronous access bus that consists of two 72-bit-wide data paths (64 bits of data plus eight bits for single-bit error detection and correction and double-bit error detection), a 28-bit address bus, and a control bus. The data buses are bidirectional and driven by the memory and cache modules.

Memory modules are built with 256K dynamic RAMs and are field expandable in 8-Mb increments up to 64-Mb. Each memory module is four-way interleaved and can supply the full DMB bandwidth of 188-Mb per second for sequential read accesses and 80 per cent of the bandwidth, or 150-Mb per second, for sequential write accesses. In order to bypass hard component failures, memory modules are reconfigurable to 6-Mb or 4-Mb.

2.2 Coherent Cache System

The Alliant memory cache system is responsible for maintaining a coherent view of global physical memory to both IPs and CEs. There are two cache module types: the computational processor cache (CPC) and the interactive processor cache (IPC).

Each CPC is a two-way interleaved 64-Kb module that can interface up to four CEs to the DMB. When combined to support a full eight-CE complex, two CPCs provide a four-way interleaved 128-Kb cache with a maximum bandwidth of 376-Mb per second.

Each IPC is a 32-Kb module that can interface up to three IPs to the DMB. When combined to support a full 12-IP configuration, four IPCs provide a 128-Kb cache with a maximum bandwidth of 188-Mb per second. In the smaller Alliant FX/1 architecture, the IPC can be used to interface one CE and two IPs to the DMB.

2.3 Interactive Processors

The Alliant interactive processor (IP) module is a Multibus card containing a Motorola 68012 microprocessor operating at 11.76 MHz. The IP module contains 512-Kb of local memory, a virtual memory address translation unit, an I/O map, power-up EPROMs, and two serial ports.

The IP interfaces to the global memory system via the IPC and to peripheral devices via the Multibus (IEEE 796 compatible). Direct memory access anywhere within the physical address space, including cross page transfers, is available to peripheral devices via the IP's I/O map.

2.4 Computational Elements

The Alliant computational element (CE) is a Motorola 68020 instruction set compatible, microprogrammed, pipelined processor with integrated floating point, vector, and concurrency instruction sets. The Alliant concurrency instruction set allows CEs to work together as a computational-complex (CE-Complex) on a single application.

Individually, each CE can deliver 4450 KWhetstones single precision (32-bit) and 3630 KWhetstones double precision (64-bit). In vector mode, each CE can execute at a peak rate of 11.8 million floating point operations per second (MFLOPs) single precision

and 5.9 MFLOPs double precision. When configured as a complex, the speedup delivered to a single application approaches the number of CEs installed.

3.0 Multi-Processor Symmetric Implementation

One of the major attributes of Concentrix is that it runs symmetrically on all processors in the system. Both IPs and CEs execute a common image of the operating system and coordinate over critical code regions and data structures via a global locking scheme. The principal difference between IPs and CEs insofar as kernel-mode execution is concerned, is that only IPs execute device interrupt code (see Section 3.4).

3.1 Computing Resources

There are two classes of computing resource in the Alliant system: individual IPs and the CE-Complex as a whole. Scheduling in Concentrix is centered around the computing resource classification, each IP is scheduled independently while the CEs in the complex are scheduled as a unit. As each process is created, the image executed determines the type of computing resource on which the process can be scheduled. In particular, images that use vectorization and concurrency are schedulable only on the CE-Complex, other images are schedulable on IPs or the CE-Complex. At any given moment, every computing resource in the Alliant system is executing a different process. When a processor has no real work to do, it runs an "idle" system process.

3.2 Global Locking

In a single-instruction-stream/single-data-stream (SISD) uni-processor architecture (the traditional UNIX host) there is, as the name implies, only one active *stream* of execution at a time. The active stream can be either a *system-stream* or an *interrupt-stream*. A system-stream is a code sequence that can switch between user-mode activity and kernel-mode activity via change-of-mode traps such as system calls or memory management exceptions. An interrupt-stream is a code sequence that executes entirely in kernel-mode and is initiated by an external hardware event at a specific priority level.

In standard uni-processor UNIX, there are both implicit and explicit forms of synchronization. UNIX enforces implicit synchronization between system-streams by not allowing one system-stream to be preempted by another (i.e., a system-stream in UNIX must explicitly give up the processor before another system-stream can be scheduled to run). System-streams in UNIX explicitly protect themselves from conflicts with interrupt-streams over critical code sections by raising processor priority level.

In the Alliant multi-processor architecture the synchronization problem is more complicated. In particular, the following types of stream interaction can occur:

- A system-stream conflicting with an interrupt-stream on the same processor (this is the traditional uni-processor interaction mentioned above).
- A system-stream on one processor conflicting with a system-stream on another processor (implicit synchronization no longer holds).
- A system-stream on one processor conflicting with an interrupt-stream on another processor (raising priority level is not sufficient here).
- An interrupt-stream on one processor conflicting with an interrupt-stream on another processor (raising priority level is not sufficient here).

In order to resolve stream interaction conflicts, Concentrix utilizes both priority level locking and a hierarchy of global test-and-set based locks for synchronization purposes. Priority level locking is used to handle system/interrupt-stream interactions within a single processor. Global test-and-set locking is used to handle multi-processor interactions. Functionally, each global lock consists of a access-location, a processor-tag, a priority-identifier, and a recursion-counter.

- The access-location is used for atomic test-and-set operations by processors contending for the lock.
- The processor-tag records the identification number of the processor that currently has access to (owns) the lock.
- The priority-identifier is used to administrate a locking hierarchy for dead-lock avoidance.
- The recursion-count records how many times the lock has been reacquired on top of itself.

Locks in Concentrix are “spin-wait.” In other words, a processor contending for access to a lock “spins” (tries continuously) until access to the lock is gained. In practice, the spin-wait scheme works well because care has been taken to minimize lock holding time in critical code sections.

Concentrix tracks the lock states of processes by maintaining a *lock-stack* for each process. As locks are acquired by a process, they are pushed onto its lock-stack; as they are released they are popped from its lock-stack. The lock-stack is recorded and maintained for each process in its kernel user-area and is preserved across process sleeps. All of the locks held by a process are released in reverse lock-stack order upon going to sleep and are reacquired in lock-stack order when the process wakes up.

The UNIX teletype system provides a good example of how the Concentrix locking scheme is used. In 4.2BSD UNIX, access to the teletype subsystem is synchronized using specific priority level locking. In Concentrix, each individual teletype has a global lock used to synchronize access to it. In addition, each teletype specifies the set-

priority-level routine to be invoked when acquiring the teletype lock. Thus, pseudo-teletypes can run at zero-priority instead of raised priority as in 4.2BSD UNIX. When teletypes run out of character storage the UNIX *cfreelist* code is activated. In Concentrix, access to the common *cfreelist* code is synchronized with a *cfreelist-lock*. The *cfreelist-lock* has a higher priority-identifier than the *teletype-lock*, enforcing a locking hierarchy that prevents acquisition of the *teletype-lock* by a process already holding the *cfreelist-lock*. The *teletype-lock* recursion-count tracks multiple accesses to the teletype within the system and provides for correct unwinding out of the code. To summarize, the Concentrix teletype system, by locking at the individual teletype level, achieves a high degree of concurrency. Multiple teletypes can be serviced simultaneously and interaction between teletypes (an infrequent occurrence) is confined to the *cfreelist* code.

3.3 Processor Communication

The underlying mechanism provided by the Alliant architecture for inter-processor communication is a cross-processor-interrupt (CPI) facility. The CPI facility allows any processor in the system to interrupt any other processor in the system. CPIs can be directed to a specific processor or to a set of processors via a selective broadcast. Concentrix uses the CPI mechanism primarily for activating remote procedure calls (RPCs) on other processors, initiating remote asynchronous system traps (RASTs) on other processors, and for synchronizing the CE-Complex. CPIs occur at a level sufficient to preempt all device interrupts except the system clock and a non-maskable condition.

The RPC mechanism is used primarily to activate routines that modify remote processor or device state. For example, code that changes kernel virtual memory mapping can use a broadcast RPC to initiate translation-buffer flushes on all processors in the system. *Asynchronous* and *synchronous* RPCs are implemented using a global mailbox facility for passing arguments; target processors use a software-initiated interrupt-stream facility to actually perform the RPC. Asynchronous RPCs suspend the calling processor until all target processors have posted a software interrupt to perform the RPC. Synchronous RCPs suspend the calling processor until all target processors have actually performed the RPC.

The RAST mechanism is used primarily to reschedule remote computing resources, deliver signals, and perform user profiling. ASTs are implemented in the Alliant architecture through careful use of the Motorola 68020 trace-trap mechanism[2].

3.4 Distributed Input/Output

The Alliant architecture supports up to 12 interactive processors each with a private Multibus. Each Multibus device in the system is dependent upon a particular IP for

configuration and interrupt servicing. At system boot time, each IP configures its own Multibus by probing for attached devices and initializing the devices that it finds.

Multibus device addresses are administered on a global level so that conceptually there is one combined Multibus that contains all devices in the system. Thus, no two devices can have the same Multibus address since a board's address identifies it in a system-wide manner. The advantage of this approach is that if a controller board, for example, is moved from one Multibus to another, there is no visible effect functionally to the system as a whole (i.e., logical device names still map the same physical devices). As a consequence, interrupt load balancing can be performed by rearranging controllers on Multibuses without changing the system interface to the devices supported. A special case of interrupt load balancing, for example, could be used for real-time data processing. Attaching a real-time data gathering device exclusively to an IP guarantees immediate interrupt response because the IP has no other devices to service.

Device drivers in Concentrix manage the distributed device environment by careful use of global locks and remote procedure calls. In particular, drivers handle the general case where a I/O system call is made to a device that is "remote" to the processor (CE or IP) performing the call. In general, drivers are constructed so that an I/O call is allowed to proceed up to the point where actual device contact is required (i.e., where device control registers need to be "touched"). At that point an asynchronous RPC is initiated by the driver to activate a device control procedure on the IP that services the device.

4.0 CE-Complex Support

The most powerful feature of the Alliant architecture is the ability to apply multiple CEs concurrently to the execution of a single user application in a transparent way. Concentrix is responsible for coordinating the CE-Complex during the execution of concurrent processes.

Alliant concurrency uses the program loop construct as the source of parallel instruction streams. For example, when the Alliant FX/Fortran compiler detects loops that can be executed in parallel, it automatically generates code containing concurrency control instructions. Loops with conditional code, data dependencies, subroutine calls, potential feedback, and loop exits, can be optimized for parallel processing; such loops run serially on conventional vector computers[3].

Control of the CE-Complex executing a program such as that shown in Figure 2 represents a second level of multi-processor management within Concentrix. In particular, Concentrix manages both synchronization between multiple active processes (as discussed in Section 3.0 above) and synchronization among multiple active code streams within a single process (discussed below).

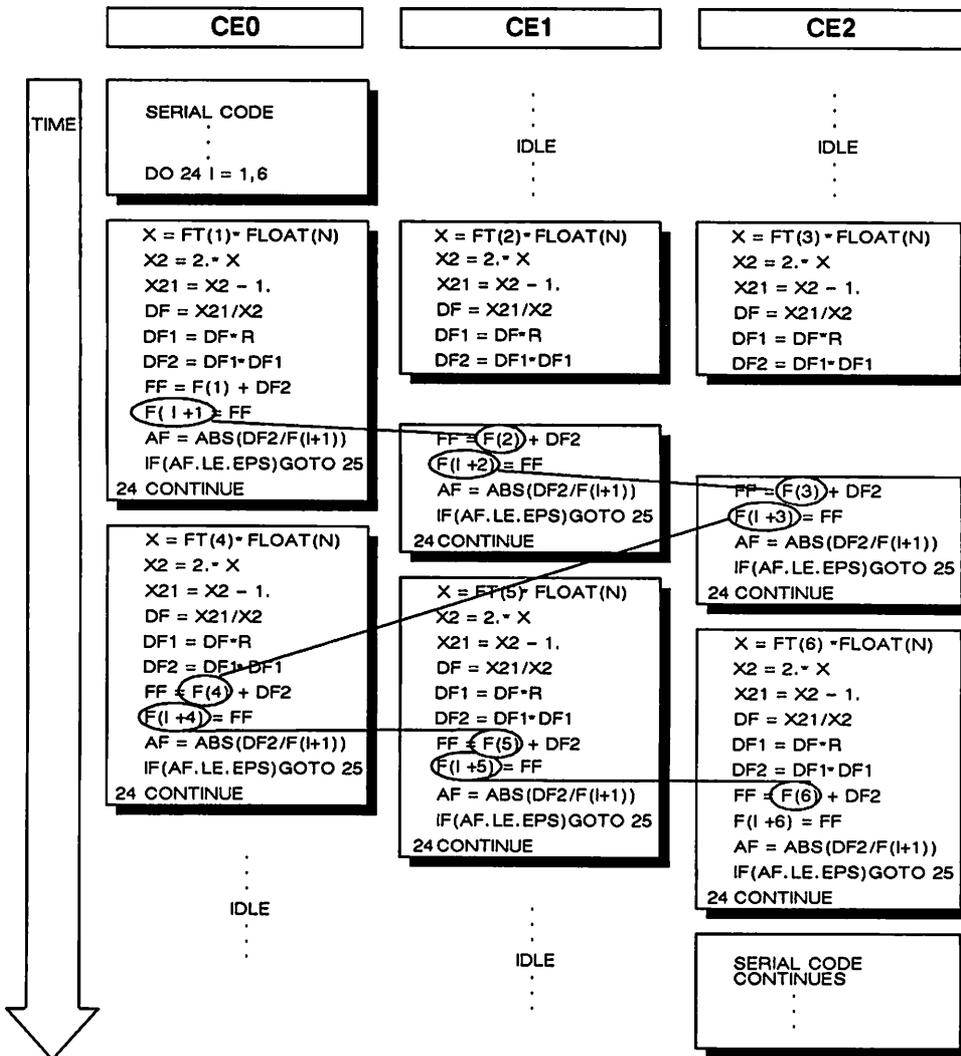


Figure 2: Concurrent Processing With Data Dependencies

4.1 Concurrency Management

In order to manage a process with multiple code streams, Concentrix replicates a number of kernel data structures. For example, up to eight kernel stacks and eight processor-control-blocks (PCBs) may be required per concurrent process, allowing each CE in the CE-Complex to have a separate kernel stack for kernel-mode execution, and a separate kernel-mode register-save-area for context switching. Concentrix allocates replicated resources on a per process basis. For example, non-concurrent processes need only one kernel stack, while concurrent processes on a three-CE complex need three kernel stacks.

As mentioned earlier, the CE-Complex is managed at a macro level as a single computing resource by Concentrix, allowing all CEs in the CE-Complex to be

scheduled as a unit to a single UNIX process. At a micro level within Concentrix, the CE-Complex alternates between *collapsed* and *expanded* states when executing a process. In the expanded state, all CEs in the complex function independently and coordination, if any, between CEs is limited to the hardware concurrency level. In the collapsed state, one CE, known as the *lead-CE* functions independently while all other CEs await *process-switch* or *expand-complex* directives from the lead-CE.

In general, collapsing of the CE-Complex is required when access to the machine state of all the CEs in the complex is needed. For example, process context switches, *fork()* and *exec()* system calls, and signal delivery collapse the CE-Complex. Most system calls do not require collapsing the complex so it is possible to have one CE in the kernel performing a system function while the other CEs are executing in user-mode. The CE-Complex is managed with minimal operating system overhead because the collapse/process-switch/expand functions are highly optimized and invoked only when absolutely necessary.

Internal management of the CE-Complex is governed by a complex-control-block (CCB). Conceptually the CCB consists of a global complex-lock, a sync-counter-lock, and various other control fields. Leadership of the CE-Complex and the right to issue collapse/process-switch/expand directives belongs to the CE that has access to the complex-lock. The sync-counter-lock is used to internally synchronize the CEs when performing collapse, process-switch, and expand functions.

4.2 Context Switching

Context switching the CE-Complex is complicated by the fact that the lead-CE going into the switch may not be the lead-CE coming out of the switch. This complication occurs because the lead-CE generally is not the same between processes. The CE process switch code, therefore, must often arrange a leadership role-swap between CEs.

4.3 Signal Delivery

Because signals are asynchronous, a process can be in concurrent execution on the complex when a signal is delivered. The policy in Concentrix is to collapse the complex and to deliver the signal on one CE while the other CEs are held in the kernel. The CE that delivers the signal is temporarily “detached” from the complex so that if a signal routine should issue concurrency directives they would have no effect on the complex as a whole.

The complete register state of the CE that delivers the signal is available to the signal handler, including all general, floating-point, vector, and concurrency registers. Signals are nestable; the CE servicing the original signal services the nested signal. The other CEs in the complex remain held in the kernel. Only when the original signal handler

returns, is the complex expanded and all concurrent CEs returned to what they were doing prior to the original signal delivery. In effect, while servicing a signal(s), the process is run on a one-CE-Complex. In order to restore the process to running on the full CE-Complex, signal processing must unwind, or the *setjmp/longjmp* mechanism invoked.

The Concentrix approach to signal delivery for concurrent programs is transparent and provides full signaling capabilities. The only limitation is that signal handlers cannot benefit from parallel execution (i.e., concurrency directives within signal handlers do not activate other CEs).

5.0 Summary

The Concentrix operating system manages two levels of multi-processor interaction within the UNIX kernel framework. Multiple active processes running on both the CE-Complex and on independent IPs share a common version of the operating system and coordinate over critical resources via a global locking scheme. Within the CE-Complex, multiple active code streams coordinate over kernel access and upon the execution of a single UNIX process.

The Alliant system is able to provide good interactive and computational performance at the same time by concentrating interactive applications in IPs and computationally intensive applications in the CE-Complex. Support for distributed I/O allows devices to be administered in a global manner among as many as 12 Multibuses.

Acknowledgments

Concentrix was developed by Larry Bakst, Herb Jacobs, Tom Jaskiewicz, Charles Monia, Roger Roles, and Jack Test. The author wants to thank Bob Perron for his help in obtaining the diagrams used in Figures 1 and 2. Special thanks go to Barry Rogoff for his expert help in editing and typesetting this paper.

References

- [1] FX/Series Architecture Manual, Alliant Computer Systems Corporation, Acton, Mass., May 1985.
- [2] MC68020 32-Bit Microprocessor User's Manual, Motorola Inc., Prentice-Hall, Englewood Cliffs, N.J., 1984.
- [3] FX/Series Product Summary, Alliant Computer Systems Corporation, Acton, Mass., June 1985.

A User-tunable Multiple Processor Scheduler

Herb Jacobs

Alliant Computer Systems
Acton, Massachusetts

Usenet Address: decvax!linus!alliant!jacobs

Abstract

The conventional Unix scheduler deals with the question of which process to schedule. In a multiple processor configuration, the question of *where* to schedule a process is an essential element in all scheduler decisions. Concentrix™ (the Alliant version of 4.2 BSD UNIX™) supports multiple processor configurations with variable numbers of processors of differing speeds. This paper discusses the implementation of the Concentrix scheduler and some of the unique ways in which it allows UNIX to be used.

The solution used in Concentrix is an intelligent, distributed class scheduler that is table driven and user-tunable: the super user can dynamically modify the tables. The overhead is proportional to the number of processors being scheduled and is quite small. Thus, it is low enough to be practical in single processor configurations.

When used with multiple processors, this style of scheduler allows UNIX to satisfy a large set of real-time applications better than many conventional real-time systems. The overheads associated with context switching and system calls for interprocess communication are totally eliminated, resulting in better response. More importantly, real-time applications can be programmed in a UNIX environment with modest effort by configuring the scheduler, rather than trying to change UNIX itself into a real-time operating system.

™ Concentrix
™ UNIX

Trademark of Alliant Computer Systems
Trademark of Bell Laboratories

Introduction

Faster computing systems remain a challenge. Supercomputers are evolving toward multiple processor configurations to achieve higher computational power. In solving a single problem, single processor systems are limited in speed by current technology and ultimately by the laws of physics. Multiple processor solutions offer a multiplicative speed improvement for problems that can be solved in parallel.

Multiple processors need not be identical. Depending on the type of problem being addressed, hardware built to deal with specific aspects of a problem can yield a better overall system solution in terms of cost and performance. It makes little sense to use a high-performance vector processor to service input/output requests.

The computer industry is just beginning to explore the use of multiple processor systems to speed up single problem solutions. Compared to more costly technologies such as submersion cooling and gallium arsenide, multiple processor configurations currently present the most promising solution to faster computational systems.

As computer systems evolve, more sophisticated scheduling techniques are needed to attain the expected performance levels. The scheduler presented in this paper solves the problem of assigning work to non-identical processors in a multiple processor hardware configuration. In the course of developing the scheduler, an interesting result presented itself: a large set of real-time problems can be addressed with absolute predictability, even within the confines of UNIX.

Overview

In an operating system, scheduling is overhead that attempts to provide some kind of deterministic behavior on behalf of consumers of the hardware resources. As currently implemented, most versions of UNIX actually have two schedulers. The job or process scheduler manages multiple tasks, owned by independent users, by allocating cpu time to processes. Multiple program execution serves two primary purposes; it attempts to use the hardware resources (peripherals, processors, and memory) as efficiently as possible, while allowing several different users access to the system at what appears to be the same time. The swap or memory scheduler deals with sharing available physical memory among active processes. Although a more optimal scheduling algorithm could be built by combining both schedulers, the problems are different enough so that two independent schedulers are a practical solution.

The first schedulers appeared with and played an integral part in the transition from standalone, single job operating systems to multiple task operating systems. Once systems began to deal with more than one job at a time, schedulers were needed to implement policy. Multiple program and multiple processor environments further complicate scheduling. The environment addressed by the Concentrix scheduler involves multiple processors of different types. Some applications can run on any processor, while others may have to run on a single unique processor or even a set of specific processors. This is analogous to using a team solution to a problem. Some members of the team can contribute to almost any part of the problem while others are experts in specific aspects. There is typically an optimal set of assignments for the team members.

Assigning work in a non-identical hardware environment is a much more complex problem than assigning work to identical processors in a multiple processor configuration. There are hardware solutions in use in industry today that deal with identical multiple processor environments. However, the problem of dealing with non-identical processors was difficult enough so that a software, rather than hardware, solution was used to implement the algorithm. The overhead in the software algorithm is small

enough to make it practical in a wide variety of hardware configurations. Actual performance by this scheduler has been measured with different combinations of hardware from a single processor system up to 14 processors with three different functional capabilities. Although it varies with workload and number of processors, the scheduler overhead in a 14-processor system in use today averages about five percent of one of the low-speed processors in the system as measured by a kernel profile. The configuration on which the measurements were taken contains six low-speed and eight high-speed processors.

The Conventional UNIX Scheduler

A brief discussion of the conventional UNIX scheduler is given here to help present the differences between it and the Concentrix scheduler. The primary characteristic of the UNIX scheduler is to give priority to the smallest CPU consumers. At a regular interval (normally one second), the scheduler reassigns priorities to all processes based on CPU usage in the last interval. UNIX priority values represent CPU consumption; the smallest CPU consumers get the highest priority (the smallest value). The more CPU time used, the higher the i-priority (inverted-priority) of the process. At a smaller interval (typically 1/10th second) the process with the lowest i-priority that is ready to run is started. At a yet smaller interval (typically the real-time clock interrupt rate) the i-priority is incremented for the active process.

As a process becomes compute-bound, its i-priority increases making it less likely to be selected. As a process remains idle, its i-priority decreases making it a likely to be selected when it becomes ready to run. As processes are running, their i-priorities creep upward, causing somewhat of a round robin effect between similar competing processes. In a purely interactive environment, with all users considered equal, this works reasonably well. However, many sites require users or applications to have higher or lower priority than normal.

Simply extending the notion of the UNIX scheduler to multiple processors leads to a ripple effect for individual processes. In other words, when a new high-priority process becomes runnable, assigning it to a specific processor displaces another process, which displaces another process, and so forth. Although the process shuffling problem can be addressed in several different ways, the object should be overall satisfactory system performance. The severity of the problem on a specific hardware configuration can be measured in terms of the context switch time between two processes. An approximation for the case of identical processors:

$$ALST = CST \times APA \times NP / 2$$

where:

ALST	Average Lost System Time
CST	Context Switch Time
APA	Average Process Activations
NP	Number of Processors

From measurements on an early five processor system, system overhead became significant enough to warrant this project.

Design Goals

This is an excerpt from the original proposal for the Concentrix scheduler. The design goals for the project, in estimated order of importance, are:

1. Provide a tunable scheduler that allows customer control over a wide variety of CPU job environments.
2. Keep it simple.
3. Scheduling must be efficient, that is, not create any kind of bottleneck.
4. For interactive systems, response must look snappy.
5. Provide a default policy that is sufficient for non-extraordinary needs. That is, the distributed system must work for a typical site.
6. Allow predictable CPU consumption for a process and for groups of processes.
7. Build the implementation in a manner such that it can be a basis for continued added value.

Feedback Schedulers

In a feedback scheduler, priority adjustments are assigned according to specific types of work. The type of work actually done by a process affects its priority. A scheduler designed to provide good interactive response time would assign high priority to processes finishing terminal input reads. Thus, keyboard-intensive programs such as text editors would respond quickly. Similarly, a scheduler designed to maximize disk bandwidth would assign high priority to completed disk reads.

The traditional UNIX scheduler is not a true feedback scheduler. The portion of the algorithm that lowers process priority as a process remains compute-bound can be considered a feedback effect.

Class Schedulers

In a class scheduler, processes can be assigned arbitrarily to one of several classes. Resource usage can then be controlled by class. The purpose is to distinguish among users having different and possibly explicit resource requirements. Different priorities and time slices can be associated among classes. For example, real-time processes that must respond within constrained timing windows can be assigned to a high-priority class and be scheduled independently from the normal workload.

The Concentrix Scheduler

The Concentrix scheduler is implemented as both a class and feedback scheduler. A feedback scheduler exists *within* the class scheduler mechanism.

At any one time, the resource pool to be scheduled consists of a collection of individual processors plus an optional group of processors, the computational complex. The individual processors themselves need not be identical; in the actual system implementation there are two different types of processors. Up to 12 low-speed processors are used for input/output and utility operations. Up to eight high-speed vector

processors perform general purpose computation. Any subset of the high-speed processors can be grouped into a computational complex that is considered a single resource.

In the Concentrix scheduler, a class is a purely logical concept. The scheduler itself has no knowledge of what it means for a process to be in a class, only how to schedule classes. By default, assignments of processes to classes take place in the *fork* and *exec* system calls. *Fork* propagates the class of the parent process to the child process. *Exec* assigns a class to a process as described by the image. *Exec* is the only place in the kernel that a relationship between processes and classes is made. This is primarily a convenience for the user. The user can construct an arbitrary program and *exec* associates the resultant process with a class that can run that specific type of program. Explicit assignment of a process to a class can be accomplished by the privileged system call:

```
setclass(pid, class)
```

For efficiency eight classes were chosen for the current implementation. Each of the eight classes contains eight priorities for the feedback scheduler mechanism. In effect there are 64 run queues in the implementation. When the scheduler is assigning processes to resources, assignment is made on a strict priority basis within class. The scheduler is preemptive within the feedback priorities in a single class. The feedback condition that *raises* priority occurs when a process voluntarily blocks itself; it is raised in priority by one notch unless blocked on keyboard input, in which case it is raised to highest priority. The feedback condition that *lowers* priority is the completion of a time slice. Lowering priority removes the process from the current run queue and places it at end of the next lowest run queue. If a process is at the lowest priority within a class, it is simply placed at the end of the current (lowest priority) run queue. When a time slice is preempted by a higher priority process, the elapsed portion of the current time slice is remembered and when the process is rescheduled, only the unused portion of the time slice is allotted.

Also, for efficiency of implementation, all processes that are possible candidates to run are kept on a run queue, even if they are currently running. Each resource runs a privately owned system null process when there is no real work for it to perform, which facilitates data gathering and system accounting.

The scheduler subroutine that maps processes to resources is a two-pass algorithm. The passes occur over the available set of resources. The first pass weakly assigns to resources processes that either must run on a specific resource or are very close to the ends of their time slices. The second pass performs assignments of processes by priorities to available resources, but attempts not to move processes around nor to break explicit assignments already performed. In actuality, the subroutine creates a list of process to resource assignments. The list is compared to the current mapping of processes to resources and those resources that need to be reassigned are notified. The resources themselves perform the process switching.

The original design assumed that it would be possible to schedule processes according to the state changes occurring in the set of runnable processes. The implementation of this approach turned out to be too complicated. When there is a poor match between processes and resources, too much special-case handling is needed.

Instead, effort was focused on an efficient algorithm to do a top down assignment of resources, using the current set of running processes as an input. To further reduce overhead, effort was given to eliminating unnecessary scheduler invocations. For example, if a low priority process reaches the end of a time slice and there is no other low priority process ready to run, it gets another time slice with no rescheduling of other resources.

The scheduler itself can be invoked from any processor in the system. Statistics were gathered to identify the conditions under which scheduling occurs. This was done with combinations of actual interactive load and artificial load via the AIM® benchmarks. Data was collected on three different combinations of processors.

System A	one low-speed processor	8 Megabytes of memory
System B	one low-speed processor, one high-speed processor	8 Megabytes of memory
System C	six low-speed processors, four high-speed processors, one high-speed four-processor computational complex	32 Megabytes of memory

Time measurements alone vary with the number of processors. Under similar workloads, system calls provide an accurate measure of work being done and are independent of a system's computational power. The numbers shown are the averages of several runs.

Reason for scheduler invocation	System Calls x 1000		
	System A	System B	System C
Process wakeup (includes child of <i>fork</i>)	143	127	159
Process sleep	121	104	133
Time/Class slice	118	108	106
Process terminating	22	22	24
Process entering new class as result of <i>exec</i>	19	19	19
Process blocking itself in <i>ptrace</i>	0	0	0
Process binding itself to explicit resource	0	0	0
Process unbinding itself from explicit resource	0	0	0
Process swapped into memory	0	0	0
Process signaled to stop	0	0	0
Runnable process being outswapped	0	0	0
Process context switches	276	287	389

The following data is configuration dependent, and no attempt has been made to normalize it.

Item	System A	System B	System C
Process context switches/second	12	37	152
System calls/second	43	125	416

As the number of processors increases, the number of context switches per second increases. Context switches are not desirable, and initially it appeared as if too many were occurring on the larger configurations. Further analysis shows that more work is being done per unit of time and, as a result, more blocking transactions occur per second, hence a higher context switch rate should be expected.

Class scheduling and multiple processors

Each resource in the system configuration is driven by a scheduling vector, a table unique to the resource that describes how that individual resource is to be scheduled. A scheduling vector determines the classes of work that can be processed by the resource. Thus, programs can be restricted to specific resources by means of class assignment. Programs that use the vector instruction set belong to a class that is restricted to either one high-speed processor or the computational complex. Other programs belong to a class that can run on any resource whose scheduling vector allows it.

The scheduling vector contains a list of time durations each of which is associated with a priority-ordered list of classes. The Concentrix scheduler is preemptive; if a process of higher class priority within the current time duration becomes available, it can preempt and displace the current process. If no work is available in the highest-priority class, the resource accepts work from the next highest-priority class, and so forth. For example, from a processor's standpoint, a scheduling vector might appear as:

300 milliseconds	1	4	5
400 milliseconds	5		

For 300 milliseconds, the resource would attempt to perform class 1 work; if none were available, it would attempt to perform class 4 work; if none were available, it would attempt class 5; and if none were available, it would go idle until the end of the interval. Then, for 400 milliseconds, it would attempt to perform class 5 work; if none were available, it would go idle until the end of the interval, then return to the 300 millisecond interval. Thus, the resource would spend $\frac{3}{7}$ of the time trying to perform class 1 work and $\frac{4}{7}$ of the time trying to perform class 5 work. In order to avoid wasting valuable resource cycles, a full list of resource classes can be placed in each time interval so that no available work is missed.

Providing each resource with its own scheduling vector creates an extremely powerful mechanism. A specific process can be guaranteed high priority service by placing it in a unique class and setting the scheduling vector for the resource on which the process is to be serviced to treat that class as highest priority. While the process is inactive, the resource would perform other work as described by the remainder of the scheduling vector. When the process activates, the resource automatically becomes available to service it.

Depending upon application requirements, the concept can be expanded to use multiple processes on multiple resources. For example, a resource can be dedicated to servicing a graphic display device, another to servicing a real-time input device, and others to performing high-speed computation. The overheads associated with context switching and system calls for interprocess communication are totally eliminated by the use of multiple processor hardware, resulting in better response than can be obtained on conventional real-time systems. Thus, on a multiple processor system, real-time applications can be programmed in a UNIX environment with modest effort by changing the scheduler, rather than trying to change UNIX itself into a real-time operating system.

This ability is used to a limited extent in Concentrix. Many of the system processes are restricted to specific processors for efficiency. The swapper process, for instance, is restricted to the resource attached to the primary swapping device, eliminating the cross-processor messages that would otherwise be needed to allow the swapper process to perform its I/O.

Some typical scheduling problems and possible solutions using a class scheduler are:

Run an interactive system with some very low priority background work.

Reserve a class x for the low priority background work. Processes enter class x only under explicit request, the system never performs automatic assignment of work to class x .

The scheduling vectors for all processors that cannot process class x do not contain class x in their class descriptions. The scheduling vectors for processors that can process class x jobs contain class x as the lowest priority class in the scheduling vector. When no other work is available, processes in class x are processed on the appropriate resources.

Guarantee 40 percent of the processing power to a special group of users who are also allowed to consume more of the resources if there is no other load. Allow users not in the special group to consume more of the resources if the special group is not using the allotted 40 percent.

When logged in, a special user is assigned to class s ; others are assigned to class n . The scheduling vectors for the resources appear, for example, as:

240 milliseconds	s	n
360 milliseconds	n	s

The choice of 600 (240+360) milliseconds as the round-robin time for the classes is a tradeoff between the number of involuntary context switches forced by class slicing and the delay before servicing processes in other classes. On an interactive system, latencies need to be quite small (a delay of .2 seconds is not unreasonable). Multiple resources improve the latency significantly. As the number of resources grows, the class slice round-robin time can be increased, reducing involuntary context switches.

In systems with non-identical processors the tables and the class assignments are more complicated. Special users receive a reserved class for each type of available resource. Other users likewise receive classes for each available resource. When a process is created, a default set of classes is propagated from parent to child. The classes can be controlled via a site-dependent database.

The scheduling vectors for each resource describe the possible classes to be scheduled in terms of the work performed by the resource. Resource A as shown chooses work only from classes $s1$ or $n1$. Resource B only performs work from classes $s1$, $s2$, $n1$, and $n2$. Since resource A is performing work in classes $s1$ and $n1$, resource B is configured to favor classes $s2$ and $n2$.

Resource A	320 milliseconds	s1	n1
	480 milliseconds	n1	s1

Resource B	280 milliseconds	s2	s1	n2	n1
	420 milliseconds	n2	n1	s2	s1

Alternatively, the scheduling vector for resource B can be configured with explicit time slot entries for each of the four classes.

Resource B

180 milliseconds	s2	s1	n2	n1
270 milliseconds	n2	n1	s2	s1
180 milliseconds	s1	s2	n1	n2
270 milliseconds	n1	n2	s1	s2

Multiple Processors and Real-time

The four main events that activate the scheduler are process sleeps, process wakeups, time slicing, and process terminations. An application being designed to run in a multiple processor environment with a class scheduler can be constructed to avoid all of these events. UNIX system overhead (process sleeps, process wakeups, and process terminations) can be eliminated from a chosen set of processes by constructing an application that separates the use of system functionality from user computation by using cooperating processes that share memory. Interprocess communication can be accomplished with memory interlocks in shared memory. The system's physical memory is the only limitation on the amount of shared memory.

In a multiple processor system, the class scheduler allows an application to use dedicated resources and at the same time schedules the non-dedicated resources. Since the scheduler runs only as needed, the dedicated resources spend no time in system state. The latencies associated with scheduling high priority processes (context switches as well as interrupts) are eliminated. Interrupts on dedicated resources can be eliminated by configuring the system peripherals onto the non-dedicated resources. The system clock, which drives time slicing, interrupts on a non-dedicated processor only.

The end result is that real-time processes, which must respond within rigorous timing constraints, are never blocked, preempted, nor interrupted.

Acknowledgements

I would like to thank Tom Jaskiewicz for the invaluable brainstorming sessions during the early phases of this project. I also want to thank Barry Rogoff for assistance in technical editing and typesetting.

High Performance Enhancements of C-1 Unix

**Rob Kolstad
Convex Computer Corporation**

Didn't make deadline. Copies available at the Conference.

Considerations for Massively Parallel UNIX Systems on the NYU Ultracomputer and IBM RP3¹

Jan Edler, Allan Gottlieb, Jim Lipkis²

Ultracomputer Research Laboratory
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012

November, 1985

ABSTRACT

Novel challenges must be met when designing UNIX implementations for highly parallel shared-memory MIMD architectures. Of primary importance is the need to avoid serial bottlenecks whenever possible, so that the potential speedup of such machines can be realized. Critical code sections far too short or infrequent to seriously impact performance on today's machines will be of concern on very large machines because the cost of each serial section rises linearly with the number of processors involved. In addition, the kernel interface must provide for a structured and natural style of general-purpose parallel programming. We present the approaches taken to satisfy these requirements for machines such as the NYU Ultracomputer and the IBM RP3. We also describe our preliminary parallel implementation of UNIX, which is currently running on an eight-processor prototype Ultracomputer.

1. Introduction

Continuing advances in microelectronics have inspired many to consider assembling large numbers of powerful processors into a single general-purpose machine capable of solving very large problems. Two such projects currently underway are the NYU Ultracomputer (Gottlieb *et al.* [83b]) and the IBM RP3 (Pfister *et al.* [85], Pfister and Norton [85], Norton and Pfister [85], Brantley *et al.* [85]), the former a shared-memory design and the latter supporting both shared and private memory. At NYU we have been investigating the adaptation of the UNIX operating system to these architectures.

However, it remains to be demonstrated that such machines can be effectively utilized, and that UNIX is well-suited to the needs of such an environment. There are two aspects to this challenge. First, several thousand processors must be coordinated in such a way that their aggregate power is applied to useful computation. Serial code sections in which one processor works while the others wait become bottlenecks that drastically reduce the power obtained, even if the serial section is so small or infrequently executed as to be entirely acceptable on a machine with only modest parallelism. Indeed, the relative cost of a serial bottleneck rises linearly with the number of processors involved. Second, the machine must be programmable by humans. Effective use of high degrees of parallelism will be facilitated by simple languages and facilities for designing, writing, and debugging parallel programs.

The present report concentrates on operating system considerations for shared memory parallel processors. The software ramifications of the RP3 private memory are only briefly discussed; a

¹UNIX is a trademark of AT&T Bell Laboratories.

²This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U. S. Department of Energy, under contract number DE-AC0276ER03077, and in part by the National Science Foundation, under grant number DCR-8413359.

more detailed presentation will appear in a future paper. We begin with an overview of the computational model and architecture of the NYU Ultracomputer. Next, we consider issues in parallel programming that impact the operating system, in particular the system services required by parallel code, and process and job scheduling issues. The following section describes in somewhat more complete fashion the programming interface presented by the kernel for parallel programs. Finally, we discuss some design issues of the ultraparallel operating system kernel itself, including data structures, resource management, and synchronization mechanisms. These mechanisms have been employed in the implementation of a parallel UNIX system running on an eight-processor prototype Ultracomputer.

2. Machine Architecture

In this section we review the architectural model on which the Ultracomputer is based, and illustrate the power of this model. Although this idealized machine is not physically realizable, a close approximation can be built. Elements of the actual machine design are briefly described in order to illustrate integrated hardware/software mechanisms for bottleneck-free coordination of a very large number of processors. The reader is referred to Gottlieb *et al.* [83b], Edler *et al.* [85], and the references therein for further details.

2.1. Paracomputers

An idealized parallel processor, dubbed a "paracomputer" by Schwartz [80] and classified as a WRAM by Borodin and Hopcroft [82], consists of a number of autonomous processing elements (PEs) sharing a central memory that they are permitted to read or write in a single cycle. In particular, simultaneous reads and writes directed at the same memory cell are effected in one cycle.

We augment the paracomputer model with the "fetch-and-add" (F&A) operation, a powerful interprocessor coordination primitive that permits highly concurrent execution of operating system algorithms and application programs (see Gottlieb and Kruskal [81]). Fetch-and-add is essentially an indivisible add to memory; its format is $F\&A(V,e)$, where V is an integer variable and e is an integer expression. The operation returns the (old) value of V and replaces V by the sum $V+e$. Moreover, concurrent fetch-and-adds are required to have the same effect as if executed in some (unspecified) serial order. The following example illustrates the semantics of fetch-and-add: Consider several PEs concurrently executing $F\&A(I,1)$, where I is a shared variable used to index into a shared array. Each PE obtains an index to a distinct array element (although one cannot predict which element will be assigned to which PE), and I receives the appropriate total increment.

Fetch-and-add is a special case of the more general fetch-and- ϕ operation (where ϕ may be an arbitrary binary associative operator) introduced by Gottlieb and Kruskal. The classic test-and-set and compare-and-swap synchronization operations are both special cases of fetch-and- ϕ as well, and the familiar load and store operations are degenerate cases.

2.2. The Power of Fetch-and-Add

Using the fetch-and-add operation we can perform many important algorithms in a completely parallel manner, i.e. without using any critical sections. For example, as indicated above, concurrent executions of $F\&A(I,1)$ yield consecutive values that may be used to index an array. If this array is interpreted as a (sequentially stored) queue, the values returned may be used to perform concurrent inserts; analogously $F\&A(D,1)$ may be used for concurrent deletes. The complete queue algorithms contain checks for overflow and underflow, collisions between insert and delete pointers, etc. (see Gottlieb *et al.* [83a]). Forthcoming sections will indicate how such techniques can be used to implement a totally decentralized operating system scheduler. We are unaware of any other completely parallel solution to this problem and note that given a queue that is neither empty nor full, the concurrent execution of thousands of inserts and thousands of deletes can all be accomplished in the time required for just one such operation.

As another example, consider the classical readers-writers problem in which two classes of processes are to share access to a central data structure. One class, the readers, are permitted to execute concurrently, whereas the writers require exclusive access. Although there are many solutions to this problem, only the fetch-and-add based solution given by Gottlieb *et al.* [83a] has the crucial property that during periods of no writer activity, no critical sections are executed³.

2.3. Hardware Realization

The paracomputer is not physically realizable, due to fan-in (and other) limitations. Furthermore, if concurrent fetch-and-add or load operations were to be serialized at the memory of a real parallel computer, then we would lose the advantage of parallel coordination algorithms, having merely moved the critical sections from the software to the hardware.

In fact, a parallel processor closely approximating our idealized paracomputer can be built as described in Gottlieb *et al.* [83b]. The resulting NYU Ultracomputer uses a message switching network with the topology of Lawrie's [75] Ω -network (equivalently, the SW Banyan of Goke and Lipovsky [73]) to connect $N = 2^D$ autonomous PEs to a central shared memory composed of N memory modules (MMs). Thus, the paracomputer's single cycle access to shared memory is approximated by a multicycle connection network.

When concurrent loads, stores, and fetch-and-adds are directed at the same memory location and meet at a switch, they can be combined without introducing any delay (see Klappholz [80], and Gottlieb *et al.* [83a]). Since combined requests can themselves be combined, any number of concurrent memory references to the same location can be satisfied in the time required for one central memory access. It is this property that permits the bottleneck-free implementation of many coordination protocols.

The impact of network latency on performance is reduced by associating a local cache memory with each PE. Frequently-used program code and data can be accessed in (approximately) a single processor cycle when resident in the cache. Thus the network latency is eliminated from many memory accesses, and all accesses benefit from the reduced network traffic.

Unfortunately, cacheing of read-write shared variables presents a coherence problem among the various caches. Different caches will in general come to contain different values for the same variable, and updates of the corresponding memory cell will occur out of sequence. Means are needed to ensure synchronized, coherent access among the multiple PEs for variables used for coordination or data transmission. An obvious mechanism, which is employed in the prototype Ultracomputer, is merely to prevent cacheing of read-write shared variables⁴. User level software is responsible for arranging the correct "cacheability" status of each memory access.

Because the cache hit ratio is a major factor in machine performance, maximizing cacheability is an important function of the compiler and operating system software. This suggests supporting a more elaborate scheme in which shared variables that are accessed read-only, or accessed only privately during a particular code segment, may be cacheable during execution of that segment (see McAuliffe [86]).

3. Parallel Programming

³Most other solutions require readers to execute small critical sections to check if a writer is active and indivisibly announce their own presence. The "eventcount" mechanism of Reed and Kanodia [79], although completely parallel detects rather than prevents the simultaneous activity of readers and writers.

⁴Because of the stochastic nature of memory access through the Ω -network, this may not be sufficient to insure that the synchronization is maintained. If the processor or cache is capable of issuing memory requests and proceeding without waiting for acknowledgment from the network, then for code sensitive to synchronization a further mechanism (a "FENCE" operation) is needed to guarantee that updates are sequenced correctly (Collier [81]).

3.1. Levels of Parallel Control

We consider applications programming of Ultracomputer-like parallel computers primarily from the standpoint of operating system design. That is, we study the requirements placed on the operating system by the desire to support effectively the number of different styles of programming permitted by the shared-memory MIMD model. While we see great potential in automatic parallelization of sequential programs, we assume here for purpose of discourse only that programs are designed originally for a shared-memory MIMD computer⁵.

Much work on concurrent programming has focused on a high-level, block-structured paradigm of parallel process control. Under this model parallel code is structured in closed-form constructs with implicit synchronization at the end of each parallel block; explicit synchronization and "fork/join" operations (Conway [63], Dennis and Van Horn [66]) are discouraged or disallowed. The argument is that the resulting parallel code is simpler, clearer, and easier to debug than code with unrestrained and unstructured process creation/destruction and synchronization. Furthermore, this programming style obviates in many cases the need for explicit shared/private declarations. That distinction is instead implicit in the usual static scope rules. Thus, for example, a variable that is visible within a block defining an "iteration" of a parallel loop, but declared in a larger enclosing scope, is taken to be shared during execution of the parallel loop; a variable declared within the block itself has scope of an individual iteration and hence is private. Finally, automatic optimization of parallel code is facilitated when the parallel structure of a program can be readily detected by the compiler. It should be possible, for example, for an optimizing compiler to implement a fine granularity of control over the cacheability of data areas, with greater reliability (and thus avoidance of cache coherence errors) than could be specified by the programmer.

However, the utility and effectiveness of this high-level parallel programming style is not yet demonstrated. Furthermore, the volatile parallelism facilitated by these closed-form parallel constructs will not be needed in all programs. When parallel processes need to synchronize very frequently, the overhead involved in the process creation and termination operations invoked by these constructs will become significant. This overhead can be alleviated to some extent by pre-spawning processes as described below. However, many parallel applications will be more suited to a lower-level programming style. One such approach involves the initial creation of a fixed number of long-lived processes, usually smaller than the number of PEs. Synchronization, scheduling, and memory management become entirely the responsibility of the application programmer. Here the syntactic structure of the program provides no information regarding the dynamic parallelism or the sharing of data.

As we will see in a subsequent section, an attractive synthesis of the two styles may be obtainable. We consider implementing these scheduling and management functions in usermode code which is part of the runtime environment provided by a language compiler. In the ideal case this would afford the advantages of high-level structured parallel programs, while the creation, destruction, synchronization, and management of parallel threads of control are accomplished without operating system overhead. Ramifications of these programming models are discussed below.

3.2. Parallel Constructs

Note that programming in the MIMD parallel environment need not be radically different from conventional sequential programming. We consider parallel languages which are variants of conventional procedural languages, augmented only with a shared/private attribute for declared variables and a small number of explicit parallel control constructs: The parallel loop, in which the iterates are executed in parallel instead of serially, is an obvious parallel extension of the loop construct found in every procedural language (see Gosden [66], Droughon et al. [67], Lundstrom and

⁵In particular, the automatic techniques of Kuck (see Kuck and Padua [79]) and Kennedy (see Kennedy[80], Allen and Kennedy [84]) will be important for running both existing and new applications.

Barnes [80], and Davies [81]). The parallel compound statement, or parallel block, contains inhomogeneous statements that are to be executed in parallel. It is also a popular structure and has appeared many times (e.g, Dijkstra [68], Brinch-Hansen [73], and the collateral expression of ALGOL 68). Neither construct contains explicit reference to processors. The PEs themselves are a resource that is only indirectly available to the program. These (and other) parallel constructs generate processes which run on PEs. Potentially, all of the processes generated by a parallel construct could execute simultaneously. Whether or not this actually occurs depends on the scheduling policy, system load, and other factors.

3.2.1. Barrier Synchronization While it is important to minimize the idle processor time caused by synchronization among parallel processes, such synchronization is occasionally necessary. A common form of synchronization occurs when each of the processes executing a parallel code section must wait at a “barrier”, or synchronization point, until they have all reached that point. Algorithms for barrier synchronization have been given by Rudolph [82] and Kruskal [81]. Other important types of synchronization are discussed subsequently in the context of the parallel kernel design, Section 6.3.

3.3. Implementation of Parallel Constructs

Having discussed the benefits of a “high-level” programming style, we now consider in detail whether that style can be supported effectively. In particular, potential parallelism must not be sacrificed due to the parallel language implementation. We shall see that the *granularity* of parallelizable processes is a crucial barometer of the effectiveness of the implementation.

The basic mechanism provided for creating parallelism is the *spawn* operation, which is used to support the parallel loop and parallel block constructs. Spawn is fundamentally an *n*-way fork of control, in which *n* identical subprocesses are created. The subprocesses are made available for scheduling on any available PEs, in a manner to be discussed. We assume here that the “parent” process waits for the termination of its spawned “children”, which occurs automatically at the end of the parallel code block. Hence the parent process—and thus the subsequent program statements—are synchronized with the completion of the spawned parallel processes.

The translation into runtime code of a parallel loop, with *n* homogeneous iterates, might make use of the following operating system primitives. First, a *spawn(n)* is executed by the original (parent) process. It stores the value *n* in a shared *children* variable, adds *n* items to the system work queue, and executes a form of *wait* so as to block until resumed subsequently by a terminating child process. The *terminate* function is executed by the spawned child processes at the end of the loop body. *Terminate* executes $F\&A(children, -1)$; if this drives *children* to zero then the current process is the last terminating child and thus resumes the parent process.

An obvious implementation for the central work pool is the fetch-and-add based parallel-access queue that was mentioned earlier.

3.4. Performance Issues in Parallel Control

There are two significant performance criteria by which any implementation of these operating system primitives must be evaluated. First, we wish to avoid algorithms that require time linear (or worse) in the number of spawned processes. Hence it is unacceptable to implement a spawn of *n* processes by a sequence of *n* insert operations on a system process queue. Instead, a spawn operation inserts a *single* item with multiplicity *n* on the process queue, and the PEs deleting such an item complete the creation of the children in parallel. Together with fully parallel memory allocation routines, this will largely prevent the occurrence of diminishing marginal benefits as the degree of parallelism is increased.

However the overhead (in absolute terms) of these operations is also crucial, because it determines the minimum granularity of parallel operations that can be efficiently spawned. Thus, although the basic unit of parallelism provided by the language constructs is the program statement,

it is clear that spawning processes that each execute a trivial statement (e.g., one assignment or arithmetic operation) would be inefficient. While careful design of portions of the operating system can reduce this overhead, the facilities described to this point must be considered useful only for relatively large-granularity parallel operations. This limitation can be alleviated in several ways.

Parallel loop iterates of smaller granularity can often be supported with a policy known as *chunking*. When the number of parallel iterates (n) is much larger than the number of available PEs (N), or the number of instructions in the body of the loop is small compared to the overhead of process creation, then the loop can be transformed into a serial loop consisting of k iterations nested inside a parallel loop of n/k iterations. Thus the scheduling overhead per iterate is reduced by a factor of k invisibly to the programmer, and, if $k < n/N$, the effective parallelism is not diminished. The value of k is most appropriately determined at runtime as a function of the number of PEs available and possibly of system load. Care must be taken to avoid setting k too high and nullifying the load-balancing properties of the "self-service" paradigm discussed below. Kruskal and Weiss [84] have examined the performance of such chunking policies and argue that even very naive schemes can perform acceptably well.

Effective granularity of programmed parallel functions can be further reduced by pre-spawning processes. Here the programmer or compiler spawns and suspends a sufficient number of processes in advance of any parallel constructs. The parallel loop or block code then activates these processes by sending a message or a signal, thus "creating" parallel threads of control without the overhead of process creation. In effect we are supporting the parallel constructs with operating system functions moved into usermode code.

Finally, one can pre-spawn non-preemptable processes that busy-wait until needed by parallel constructs. Far finer granularities of parallel functions can thus be programmed with high-level constructs, since virtually all of the overhead of process creation, scheduling, and context switching is eliminated. The cost is in tying up a larger number of PEs than may actually be used during much of the program execution, particularly if the degree of parallelism is highly volatile.

3.5. Performance Issues in Barrier Synchronization

Synchronization operations may be implemented in two ways: It is always correct for the processor to suspend the current process while waiting for the synchronization to complete, and switch to another process. However, considerable operating system overhead is incurred. The alternative is a busy-waiting synchronization routine that simply loops and tests whether the synchronization condition is satisfied. For short waits, the overhead involved is significantly less for busy-waiting than for process switching. However, busy-waiting admits the possibility of deadlock when the number of synchronizing processes is greater than the number of available processors (and no preemption is permitted). Even when deadlock cannot occur it is possible that a number of processors will loop unproductively for extended periods of time, particularly if one or more of the synchronizing processes is preempted or swapped out.

A hybrid implementation, in which each process busy-waits for a short period and then, if the condition is still not satisfied, yields the PE, would avoid rescheduling overhead in those cases where busy-waiting is suitable, and deadlock would be prevented. Synchronization issues have implications for process and job scheduling that are addressed below.

4. Process scheduling

4.1. The "Self-Service" Paradigm

Operating systems for uniprocessors and some multiprocessors usually contain a single module that schedules use of the processor(s) by assigning processes as appropriate. While this centralized approach assures favorable load balancing, it introduces a serial bottleneck that will limit overall performance on highly parallel machines. Alleviating this bottleneck by designating two or more schedulers, each managing a portion of the processors, leads to an interesting tradeoff: if the number

of schedulers is small, scheduling bottlenecks arise; if the number is large, effective load balancing suffers.

Another technique used to avoid the bottleneck is stochastic distributed scheduling. Here a work queue is maintained for each PE. Whenever a process is created on any PE, that PE assigns the process on a random basis to one of the work queues. In concert with time-slicing, or when certain constraints (on the variance of process execution times) hold, this mechanism can effectively balance the load, possibly at a cost in scheduling overhead (see Klappholz [82]). It does not permit the constant-time spawning of multiple processes discussed earlier since the assignment of each process must be randomized individually.

The scheduling paradigm adopted for process scheduling (and other resource management functions) in the Ultracomputer is that of a self-service system, in which one maintains a single central queue of ready processes. Each processor accesses this shared queue to obtain processes for execution and to insert newly-spawned processes. This self-service paradigm relies on simultaneous distributed processing of centralized data, and is highly dependent on concurrently-accessible data structures that allow concurrent operations to be performed without serialization. The fetch-and-add based mechanisms described earlier are crucial in implementing these critical-section-free operations, e.g., queue insertion and deletion.

The queue algorithms used are enhanced variants of the algorithms mentioned earlier that support three important additional features:

- (1) *Multiplicity*. A spawn of k processes is implemented by the insertion of an item of multiplicity k , which is "deleted" k times before actually being removed from the queue.
- (2) *Priorities*. Processes may be inserted onto the central ready queue at any one of a (fixed) number of priorities, with the delete operation removing the highest priority item⁶.
- (3) *Interior removal*. In order to swap out a process from memory, or to prematurely terminate a process, it is occasionally necessary to "delete" an item from the middle of a queue (see Wilson [86]).

Distributed scheduling from a central ready queue achieves optimal load balancing among the PEs and also facilitates multiprogramming. Unrelated jobs may contribute processes to the ready queue; all will be scheduled as PEs become available. In addition to its usual benefits, multiprogramming can improve throughput by allowing serial sections and highly parallel sections of different jobs to be overlapped.

4.2. Job Scheduling

As indicated above, when the number of ready processes exceeds the number of processors, the operating system uses a priority based preemptive scheduling algorithm. Although we expect this standard multiprogramming discipline to prove adequate for program development as well as for production runs of many programs, we anticipate that some sophisticated users solving large problems will require finer control over scheduling. Such users are well served by the non-preemptable allocation of a number of processors, which are then assigned to subtasks under program control. In addition to permitting a problem-specific dynamic choice of which subtask to execute next (rather than relying on the operating system's unsophisticated notion of priority), non-preemptable processes are immune from the overhead associated with involuntary context switching.

⁶In addition to other more obvious functions, process priorities are needed in management of nested spawns. The dynamic process structure of a program in which spawns are nested several levels deep may be depicted as a tree, in which spawned child processes are represented by nodes that are descendants of their parent process node. If the program is executed such that the process tree is traversed breadth-first, then there is a danger that because the processes at each level are spawning more processes before any process may terminate, the capacity of memory or system tables may be overwhelmed by an exponential explosion in instantiated processes. This is avoided by ensuring depth-first traversal of the process tree, which may be achieved by enqueueing the template for creation of spawned children on the ready queue at a priority greater than that of the parent.

The operating system supports non-preemptable processes with a variant of the *spawn* primitive, which inserts a non-preemptable item with multiplicity n onto a high priority ready queue, causing the next n available PEs to select these processes for execution. Since the operating system will not permit the total number of such processes in the system to exceed the number of PEs, the time delay between the invocation of the first and last instance of the process in question is bounded by the preemption interval.

To illustrate one use of this facility consider an application that requires tight synchronization between processes, i.e., one in which processes must synchronize after executing only a small number of instructions. Although the stochastic nature of the network prevents an exact determination of the rate of progress for an individual process, with preemption removed (and with the Ultracomputer combining network), a group of processes will execute at roughly equal rates with high probability. Thus, providing that the program segments executed by each process between corresponding synchronization points are of comparable length, a programmer using the non-preemptable *spawn* can profitably synchronize the resulting processes by means of busy waiting loops free of system calls.

In summary, we have considered a spectrum of "organizational styles" of parallel programs.

- (1) On one end of the scale are jobs that are static in their use of PEs and are subject to real-time constraints. Processes associated with such jobs should not be preempted under any circumstances.
- (2) More important is a class of non-real-time applications referred to above, which coordinate parallel activities on a fixed group of PEs and are characterized by very frequent internal synchronization. As described above, busy-waiting synchronization is appropriate for such jobs. They may be preempted or even swapped out, as long as all of the processes are preempted together.
- (3) Jobs displaying dynamic parallelism are better suited to our original model of process scheduling. Although internal synchronization will still be needed occasionally, it can be adequately managed with the hybrid synchronization mechanism proposed above, even in the presence of chunking. Nonetheless there are reasons to keep related ("sibling") processes executing concurrently: First, there are algorithms whose performance is improved when parallel processes execute at more or less the same rate, that is, when the execution rate of the slowest-progressing process in the spawned set is maximized. Second, the effectiveness of the processor cache will be improved when successive processes executed on the same PE come from the same job, since they are then likely to reuse cache entries for program code⁷.

The structure of the central ready queue is further complicated by this need to recognize groups of sibling processes in swapping and scheduling. However, the original fetch-and-add based notion of bottleneck-free inserts and deletes can be maintained.

5. Kernel Interface

The set of operating system services needed by parallel programs is to a large degree identical to that provided in conventional serial operating systems. We enhance the UNIX kernel interface with a small number of primitives for creation and synchronization of parallel threads of control, and for management of shared and private memory areas. The desire for high performance also has ramifications for other system functions, such as I/O and file system organization, that are not specifically related to MIMD or shared memory systems. Because we are only beginning to investigate these last areas, no further comments about them will be made in this paper.

There are many different ways of structuring parallel programs, and hence the most important goal of the system interface design is generality. Both process management and memory management offer choices for programming language and application designers that involve complex

⁷Here we assume that the cache architecture permits retaining of cache lines across a context switch.

tradeoffs between ease of program design and debugging, and possible efficiencies to be obtained through low-level coding or lower levels of protection; between efficient accommodation of volatile levels of parallelism and efficient processing of I/O or tight internal synchronization; between turnaround time for a particular job and overall throughput for a set of jobs with varying resource requirements; and so forth. As always in operating system design, the kernel must implement a small set of primitives that provide for the widest possible range of user applications.

Almost all of the facilities described in this section are implemented in a prototype UNIX-based operating system at NYU, which is described later in this paper. However, the kernel interface is very much a work in progress. Far more experience is needed in designing languages and programs for shared-memory multiprocessors before we will fully understand the requirements for the programming model in service of parallel programs.

5.1. Process Management

At the most basic level, each thread of control in a parallel program is embodied in a standard UNIX process. We use the term *job* to refer to the collection of processes executing a single parallel program, normally a subtree rooted at a process created by *fork*. Certain operating system functions such as scheduling, shared memory management, and signaling may recognize jobs as well as individual processes.

5.1.1. System Calls The *spawn* system call has already been introduced. It is a multi-way fork that creates n processes in time essentially independent of n (unlike an iterated *fork*, which would require time proportional to n). Spawned processes are full-blown UNIX processes, and they inherit attributes from the parent much like forked processes, with only minor exceptions⁸. The set of child processes created by a single *spawn* is referred to as a *spawn group*.

Arguments to *spawn* include the multiplicity (n), option flags, and, optionally, the location of an array used for reporting of child processes' exceptional termination conditions. Option flags include (1) request for nonpreemptable child processes, essentially a request for n PEs, and (2) request for cactus stack processing (to be discussed in Section 5.2.4). The parent process may obtain exit codes from individual subprocesses, or summary counts representing a histogram of the various termination conditions.

The *spawn* call normally returns inline in all processes, much like *fork*, returning the child's "spawn index" (a number uniquely chosen from $\{1, \dots, n\}$) in each child process and zero in the parent process. Child processes then execute independently until terminating with *exit*.

Mwait ("multiple wait") is a new system call used by a parent process to await the termination of all spawned children. Again, neither *mwait* nor *exit* involve serial operation, as would be the case using the standard *wait* system call, which would have to be iterated. *Mwait* will also return in the event of an abnormal child termination, with the error status suitably reported. Furthermore, *mwait* can be used to test (without blocking) whether outstanding children remain.

A new signal, SIGPARENT, is automatically sent to all processes spawned by a terminating parent. Unlike the traditional situation with forked processes, there is usually no purpose in allowing continued execution of a spawned orphan process. Experience at NYU has demonstrated the need to assist the programmer in cleaning up orphans after an abnormal termination, especially during debugging of parallel programs.

5.1.2. Low-Overhead Parallel Threads UNIX processes are relatively "heavy" objects. Associated with each is a unique memory management context containing private and shared memory areas, a number of open files, and a substantial number of attributes (userids, current working directory, signal actions, etc.). Process creation requires duplicating much of this state and even

⁸Spawned processes must however be distinguished from forked processes, for technical reasons that will become apparent.

context switching can involve considerable overhead, depending on the memory management architecture and other factors. In earlier sections we proposed to reduce the impact of process creation and destruction by pre-spawning processes, and to eliminate the context switching overhead as well by permitting non-preemptable processes. In this last case the program in effect obtains and then schedules a fixed number of PEs; if multiprogramming throughput is not an issue then one might assign all or almost all of the existing PEs to an individual application in this manner.

A natural organization for managing the "assigned" PEs involves user-level threads of control which are scheduled by user code into execution under the UNIX processes which are fixed on the individual PEs. These threads are known herein as *tasks* to distinguish them from UNIX *processes*. The operating system has no cognizance of these tasks; their creation, destruction, synchronization, and resource assignments are accomplished by a layer of software that is part of the user program, a standard library, or the language runtime environment. The "weight" of UNIX processes is no longer a concern, since the processes, once spawned, are entirely static. However, such jobs will be unable to respond efficiently to highly varying demands for service by the usermode tasks; if the user wishes to dynamically expand or reduce the pool of available PEs then process creation or context switching overhead arises.

There are further difficulties with this scheme of "user multitasking" in the UNIX process environment. If a task modifies any aspect of the process state, by (for example) opening a file or changing the current working directory, and that task is later executed under a different UNIX process, its environment will invisibly change. File access will fail or affect an incorrect file, the current working directory will be wrong, etc. These problems appear to be solvable within the current framework, although the details are still under investigation. For example, the kernel interface might be extended to allow access to files opened in other processes within the same job, and in general, system calls (e.g. *chdir*) can be intercepted by a layer of software that insures regular system call semantics are maintained for each task.

From the point of view of the operating system kernel, we have considered all processes to be homogeneous (the *job* and *spawn group* defined above are merely aggregates and have no associated attributes or capabilities). In other work in parallel programming environments the concept of *lightweight tasks* implemented in the kernel has proved popular (e.g., Baron *et al.* [85]). Such tasks are scheduled by the operating system but contain almost no private state; rather, they exist within the resource domain of a process or job. Some of the above difficulties would be alleviated if such objects as opened files were maintained on a job level rather than a process level. Other aspects of the process state semantics would change; e.g. the current working directory could no longer be manipulated by an individual program thread. The overall utility of lightweight tasks is not yet known. It is not clear whether they will enable scheduling of volatile parallel threads with minimal context switch overhead. Further investigation is required in this area.

Memory management issues pertaining to this discussion are considered in the next section.

5.2. User Memory Structure

The operating system must provide one or more segments within a job which permit data to be shared among processes. Here we consider shared memory for storage of data within a parallel program rather than for general inter-process communication. Hence there is no need for memory segments shared among arbitrary unrelated processes, and there may be no need for dynamic creation of shared memory segments other than in the course of program or process initiation. Since arbitrary subsets of the processes constituting a job may wish to share memory, full generality together with full protection would require a large number of shared segments. In a pure global shared memory environment, a simpler, more structured approach appears adequate and natural. Each shared data area is accessible over a subtree of processes; it is created on behalf of the parent process and inherited by all spawned descendants. Thus the number of shared segments visible to an executing process is bounded by the spawn nesting level. When sharable local memory is present, further mechanisms for management of shared segments will be required.

The process image strongly resembles that of traditional UNIX processes. Logical memory segments for shared program text, private data, and program stack remain, though in some cases transformed. We augment these with an intra-job shared data segment.

5.2.1. Object Files Traditionally the text and data segments are initialized by the *exec* system call from the text, data, and bss (uninitialized data) segments of an object (*a.out*) file. In the parallel environment more object file segments are required. The shared data segment is created from shared data and shared bss segments in the *a.out* file. Furthermore, in architectures supporting local PE memory, we may need the capability to specify at compile time program components to be loaded into the local memory. This gives rise to an additional three object file segments for local text, local data, and local bss.

In our prototype operating system we have adopted the Common Object File Format (COFF) from AT&T System V UNIX, although in most other respects our system is based on Version 7 (and in the future 4.3 BSD) UNIX. COFF provides for varying numbers and types of segments in the *a.out* file, and permits the needed flexibility.

5.2.2. Shared Data The shared data segment is created at program initiation (by *exec*), although it may be of zero length. It is inherited by all descendant processes. The segment may be expanded at any time through the new *shbrk* system call, which is usually used via a library parallel memory allocator known as *shmalloc*. The sharing of this data segment is managed much like the traditional UNIX shared text segment, except that it is read-write and exists only within a job.

5.2.3. Cacheability Control Because the shared data segment includes read-write variables, accesses are in general not cacheable. However, there are various circumstances in which certain variables are used, either temporarily or permanently, in a private (accessed by only one process) or read-only manner. A process may dynamically specify the cacheability of such variables. Means will be also be provided for flushing and invalidating the cache as necessary.

5.2.4. Private Data A private data segment is created for each spawned or forked process. Its size is controlled with the standard *brk/sbrk* system call. As in standard UNIX, private data segments are isolated by memory mapping hardware so that even in case of user program error there is no possibility of a private data segment owned by one process being modified by another process. Accesses to the private data segment are always cacheable.

In standard *fork* semantics, data in the private data segment is copied into the new private segment created for a new process. When applied to spawned processes, this policy dictates the following programming language semantics: Private variables replicated for a child process (e.g. an iterate of a parallel loop) are initialized to the current value of the corresponding variable in the parent's private space. It is entirely possible that a programming language will require different behavior, e.g., reinitialization according to an initializer or default value instead of a value propagated from the parent. The *spawn* system call may thus provide an option requesting reinitialization instead of copying of the private segment.

We now consider the impact of user multitasking on the private data segment. An immediate obstacle arises. If the private variables of usermode tasks are allocated in the private data segment, then each time a task moves from one process to another its private variables will have to be copied, or at least remapped, from one private data segment to another. In either case, sufficient overhead is introduced into the usermode context switch to abrogate much of the advantage of this type of program organization. A solution is to place the task private variables in cacheable areas of the shared data segment, so that they are accessible from any process, and rely on the compiler and user code to (1) isolate these private subareas from improper access, and (2) issue the appropriate cache flush or invalidate during the user level task switch. Through avoiding use of the private data segment, most of the task switch overhead has been eliminated. Experience will be needed to determine the dangers of private data areas that are not protected or isolated by the mapping

hardware. Depending on the programming language and compiler, it is possible that debugging of parallel programs will be more difficult than otherwise. Similar issues regarding the private data segment arise in other situations that involve pre-spawning.

5.2.5. Program Stack The stack segment involves some of the same issues as the private data segment. In standard UNIX, the stack is merely a second private data segment, distinguished by the fact that it expands and shrinks automatically. The simplest policy in the parallel environment is to replicate the entire stack segment on *spawn* as is done for *fork*. The stack is thus entirely private and cacheable.

However, only the stack frames created since the last spawn need to be private. Furthermore, sharing the remainder of the stack will permit realization of the scope-based paradigm for variable sharability in structured parallel code, which was discussed briefly in Section 3.1. When parallel constructs are nested, in a block-structured language, the automatic variables declared at each level are allocated in successive stack frames. A natural implementation of the scope rules is to arrange that a parent's stack frame be shared by its child processes, and, in fact, by all those processes constituting the subtree rooted at this parent process. The resulting structure is known as a *saguaro* or *cactus* stack (Hauck and Dent [68]). Private stack frames for active processes are linked to the parent's stack frame. There is no serial overhead in creating or destroying these private frames since concurrent memory requests can be processed in parallel. An option flag in the *spawn* system call is used to cause cactus stack (sharing of existing stack) rather than private stack (copying of entire stack) processing⁹.

When user multitasking or other forms of pre-spawning are involved, the stack segment presents the same problems as the private data segment. The solution, analogously, is to allocate space for all required program stacks in the shared data segment. Usermode code then manipulates the stack pointer register in the course of scheduling tasks, implementing logically private stacks inside the physically shared area. Using cacheable and noncacheable areas as appropriate, a cactus stack can be implemented in this manner. We may now conclude that the only user memory segment needed for such programs is the shared data segment. Private data and stack segments need not even be created. Again, there are potential dangers resulting from the lack of inter-task storage protection.

5.2.6. Local Memory In architectures that support local PE memory as well as global memory, one needs extra logical segments to provide local versions of the text, data, and stack. The IBM RP3 further allows access by each PE to the local memory of every other PE. This is used for message passing and restricted cases of data sharing.

The required kernel interface facilities for support of local memory features are not yet well understood. Explicit allocation of private local memory segments may be provided. Furthermore, the program text and possibly the stack may be "cached" in local memory invisibly to the user. When a local memory is insufficient to service all allocation requests, it may be possible to use the global memory as a backing store, managed with virtual memory techniques. Several additional approaches for utilizing local memory are also under investigation.

5.3. Usermode Synchronization

Both busy-waiting and process-switching synchronization occur among parallel processes or tasks within a user program. Kernel primitives are provided in support of both forms.

⁹A new activation record must be created for each child, so *spawn* must be given the address of a subroutine or code block to be executed by the children. In this case, *spawn* only returns in the parent (after termination of the children).

5.3.1. Busy-waiting synchronization The operating system is for the most part not involved in busy-waiting synchronization code in user programs. No facilities beyond access to shared variables (and perhaps fetch-and-add) would appear to be required to implement such routines. However, two difficulties arise.

- (1) In programs with signal-handling routines, it may occur that a process holding a lock is interrupted by a signal whose handler requires the same lock. Since the signal handler operates within the same process, deadlock will result.
- (2) When busy-waiting locks are used by a collection of preemptable processes, it is possible that the process holding a lock will be preempted and perhaps swapped out while the rest of the processes, still in memory and active, loop unproductively for a substantial period of time.

A new kernel interface feature is used to avoid these problems. The user program can request temporary suspension of signal delivery and/or preemption during busy-wait critical sections. The latter is only a hint, which the kernel may ignore, since it does not affect correctness. It is unreasonable to implement these functions with system calls, which would increase manyfold the expense of a busy-wait synchronization. Instead, the user program sets these temporary modes through specially-designated communication flag variables that are allocated in user memory. The kernel is informed of the location of these flag words with a system call (so as to avoid "magic addresses" encoded into the kernel), and checks the flags at appropriate times. Thus negligible overhead is added to the user's synchronization routines.

5.3.2. Process-switching synchronization Two new system calls, *block* and *unblock*, are added for reliable suspension and reactivation of processes in process-switching usermode synchronization routines. The kernel does not provide semaphores or other coordination facilities directly, but merely manages process status.

The kernel maintains a per-process *pending unblock* flag, which is used to avoid races and deadlock in the user program. The *block* system call atomically tests the *pending unblock* flag, and, if set, clears the flag and returns immediately; otherwise, it suspends the process by entering a blocked state that is distinguished from that set by *pause* or *sigpause*. The user may prevent signals from prematurely awakening the process, as in *sigpause*. The *unblock* system call atomically tests whether the process is suspended by a *block*, and, if so, makes it ready; otherwise, it sets the *pending unblock* flag for the process. The *block* and *unblock* primitives are used in the obvious manner with the proviso that the caller of *block* must be prepared for premature returns. This is because the *pending unblock* flag may be leftover from a previous coordination operation. Code invoking *block* must loop so as to *re-block* if the awaited condition is still not satisfied.

6. Building the Parallel Operating System

Since programs written for an Ultracomputer-like machine will generate hundreds or thousands of concurrent activities, we will encounter a correspondingly high level of simultaneous requests for operating system services. Serial processing of these requests will generate unacceptable bottlenecks on a large machine. Therefore, the kernel must itself be a highly parallel bottleneck-free program.

We have already outlined an implementation of processor scheduling satisfying these constraints. The synchronization primitives and data structures described below in the context of the operating system are equally applicable, with minor implementation differences, in user applications. These algorithms have been implemented in an experimental operating system running on a prototype (8-processor) Ultracomputer. Based on UNIX Version 7, the experimental system is symmetric (i.e., there is no master-slave relationship) as well as parallel. As described earlier, the system incorporates facilities for parallel applications programs. Work has commenced on a 4.3 BSD-based follow-on system.

6.1. Data Structures

All operating system functions make heavy use of centrally stored concurrently accessible data structures made possible by the fact that simultaneous references to the same memory location can be accomplished in the time required for one reference. Here we consider a few structures that have proven useful.

We have avoided pure linked lists, since we know of no bottleneck-free algorithm for deleting items in a linked list. The desirable characteristics of linked lists must be found in other structures. As will be seen, many of these structures use linked lists as subcomponents.

6.1.1. Queues Queues similar to those employed for process scheduling are used by synchronization primitives: The set of processes waiting for a lock or an event are held on such a queue.

The queues used by the operating system are somewhat different from the simple array implementation discussed earlier. We obtain queues of unbounded size by associating a linked list, protected by a semaphore, with each element of the array. An insertion at array element j appends its item to the list associated with element j . The maximum concurrency supported by this structure equals the number of lists, i.e., the size of the array (see Rudolph [82]). A variation of this queue structure in which FIFO ordering is relaxed is also frequently used, e.g., to manage a pool of free items to be allocated.

A similar structure results when hash tables are used to access indexed (dictionary) information. The size of the hash table (number of buckets) is set according to the desired maximum concurrency, usually the number of PEs times a small factor. The buckets are linked lists, protected by readers-writers locks, so that if no updates are occurring, items are accessed with no serialization. An example is given below.

6.2. Memory Allocation

The creation of a new process requires allocation of space for its u-block as well as its private data and stack segments, if any. As with process management, we adopt a self-service mechanism. A number of parallel algorithms for memory management have been designed, including (non-demand) paging, two variants of the Buddy System (Knuth [68]), and a boundary tag method (Knuth [68]). All are parallel analogs of serial algorithms. All, except for one of the Buddy System variants, maintain queues (whereas their serial analogs keep lists) of free memory blocks. Insertions, deletions, and accesses to blocks within a concurrently-accessible queue are at the core of these algorithms; as a consequence we obtain critical-section-free memory allocation.

6.3. Coordination Primitives

An ideal situation for parallel execution occurs when completely asynchronous behavior is permitted, as in some "chaotic" algorithms. Unfortunately, however, it is sometimes necessary to coordinate processes' accesses to shared data structures. In these cases one must be careful to permit as much parallelism as possible. Here we discuss three mechanisms for process coordination that we have used successfully in our kernel, namely *counting semaphores*, *readers/writers locks*, and *events*. When designing such mechanisms, one must specify whether a processor denied permission should (busy) wait, or suspend execution of the current process and switch to another.

6.3.1. Busy-Waiting Synchronization Despite the potential for waste in busy-waiting, there are several reasons for using it, including potentially low overhead and applicability to situations where context switching is inappropriate¹⁰. We have used busy-waiting counting semaphores and readers/writers synchronization extensively and have described algorithms for them before (see Gottlieb *et al.* [83a]).

¹⁰E.g., in the implementation of non-busy-waiting mechanisms themselves.

Semaphores are often used to serialize access to a small partition of a larger concurrently accessible data structure; for example, the individual linked lists used to implement queues of unrestricted size.

Readers/writers synchronization is used naturally in cases where exclusive access is required only infrequently. We also support upgrading a read lock to a write lock and downgrading a write lock to a read lock, and have used the resulting protocols to implement search structures that must support the operation: "search for an item, and insert it if not found". The inode table in our UNIX kernel, for example, is implemented in this manner. Such a structure uses linked lists accessed through a hash table, as described in Section 6.1.1. By performing the search with a read lock, serialization is avoided in many instances including the important case in which many processes search for the very same inode (e.g., the root inode). Only if the inode is not found is it necessary to upgrade to a write lock. The upgrade operation may fail in which case the process goes back to search again (while the one process that succeeds performs the insert).

6.3.2. Non-Busy-Waiting Synchronization Process-switching synchronization is commonly used in multiprogramming systems since it permits a processor to continue performing useful work when the progress of the current process is logically blocked. As in other multiprocessor UNIX implementations (e.g. Bach and Buroff [84], Felton *et al.* [84]), we have replaced the internal kernel *sleep* and *wakeup* mechanisms. For each of the new mechanisms described in this section, when a process must block the PE places it on a queue associated with the condition to be satisfied, and executes the next process from the ready queue. When the condition is eventually satisfied, the blocked process is moved from the waiting queue to the ready queue. Unlike other implementations, we avoid critical sections in many cases by using fetch-and-add and concurrently-accessible queues.

The best examples of non-busy-waiting synchronization come from the area of I/O processing, which takes on special significance for the Ultracomputer, because large numbers of processes can simultaneously perform related I/O operations. For example, searching of important file system directories would be a bottleneck if serialized. Since a group of processes reading such a directory would likely all attempt to read the same disk block; serialization would be devastating.

At a low level, physical I/O devices require serialization; this is easily provided by semaphores. Apparent parallelism can be achieved via in-memory buffer caching. Once a disk block is copied into a memory buffer, it may be concurrently accessed for reading; readers/writers synchronization is appropriate in this situation.

Process-switching synchronization is also used to implement *events*, which are often associated with external occurrences, such as the completion of an I/O operation. At such a time, the event is *signaled*, remaining in this state until *reset*. Additional signals (before the reset) have no effect. Since an event might never happen (consider input from a user's terminal), it must be possible to terminate a process that is blocked on such an event¹¹. We have developed a method of *premature unblocking* for all of the non-busy-waiting synchronization primitives described above. This requires the *interior removal* primitive previously mentioned in Section 4.1.

There is a special difficulty in designing bottleneck-free algorithms for readers/writers and events, because the most natural implementation would require a single process to completely empty a queue. For example, when an event is signaled, all processes waiting for it must be awakened. One solution to this problem is to move the wait queue as a single object onto the system ready queue, where it will be treated in much the same way as an item with multiplicity. Our current implementation lets newly awakened processes "help out" by waking up other processes. This latter approach is less complex than the "queue of queues" method, but requires more time.

¹¹Of course this is actually done as part of signal handling.

7. Conclusion

The forthcoming emergence of highly parallel machines will require that essentially no serial bottlenecks are introduced by either hardware or software. The fetch-and-add coordination primitive provides a simple and powerful means for achieving this goal by permitting programmers to employ shared data structures without relying on critical sections. We believe that parallel operating systems can be built that perform the traditional functions of resource management, scheduling, and coordination using critical-section-free algorithms. The prototype NYU Ultracomputer operating system has been constructed to demonstrate the feasibility of this approach. Results to date are highly encouraging.

One way to facilitate the early use of highly parallel computers is to furnish a simple programming model that permits users to write programs using variants of conventional high-level procedural languages. Further experience is needed to determine whether the operating system primitives that have already been designed will effectively support the parallel constructs needed for such languages. It is especially important that these high-level mechanisms not introduce inefficiencies that would prevent their widespread utility. In particular, using these mechanisms must not significantly reduce the parallelism obtained. A variety of approaches to parallel control and scheduling is being studied for the support of a wide range of parallel applications.

References

- J. R. Allen and K. Kennedy, "PSC: A Program to Convert FORTRAN to Parallel Form", in *Supercomputers: Design and Application*, Kai Hwang, Ed., IEEE Computer Science Press, 1984.
- M. J. Bach and S. J. Buroff, "Multiprocessor UNIX Operating Systems", AT&T Bell Laboratories Technical Journal **63**, no. 8, October 1984.
- R. Baron, R. Rashid, E. Siegel, A. Tevanian, and M. Young, "Melange: A Multiprocessor-Oriented Operating System and Environment", Technical Report, Department of Computer Science, Carnegie-Mellon University, 1985.
- A. Borodin and J. E. Hopcroft, "Routing, merging and sorting on parallel models of computation", *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, May, 1982.
- P. Brinch-Hansen, *Operating System Principles*, Prentice-Hall, Englewood Cliffs, NJ, 1973.
- W.C. Brantley, K.P. McAuliffe, and J. Weiss, "RP3 Processor-Memory Element", *Proc. Intl. Conf. Parallel Processing*, pp. 782-789, 1985.
- W. W. Collier, "Principles of Architecture for Systems of Parallel Processes", IBM Technical Report TR00.3100, March, 1981.
- M. Conway, "A Multiprocessing System Design", *AFIPS 1963 FJCC*, (Spartan Books, New York).
- J. Davies, "Parallel Loop Constructs for Multiprocessors", Tech Report UIUCDCS-R-81-1070, University of Illinois, Urbana, Illinois, May 1981.
- J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations", *Communications of the ACM* **9**, no. 3, March 1966.
- E. W. Dijkstra, "Co-operating Sequential Processes", in *Programming Languages*, F. Genuys (ed.), Academic Press, New York, pp. 43-112, 1968.
- E. Droughon, R. Grishman, J. Schwartz, and A. Stein, "Programming Considerations for Parallel Computers", *IMM 362*, Courant Institute, New York University, Nov. 1967.
- J. Edler, A. Gottlieb, C. Kruskal, K. McAuliffe, L. Rudolph, M. Snir, P. Teller, and J. Wilson, "Issues Related to MIMD Shared Memory Computers: The NYU Ultracomputer Approach", *Proc.*

12 Annual Computer Architecture Conf., 1985.

W. A. Felton, G. L. Miller, and J. M. Milner, "A UNIX System Implementation for System/370", *AT&T Bell Laboratories Technical Journal* 63, no. 8, October 1984.

L. R. Goke and G. J. Lipovsky, "Banyan Networks for Partitioning Multiprocessor Systems", *Proc. First Annual Symp. Comp. Arch.*, 1973.

J. A. Gosden, "Explicit Parallel Processing Description and Control in Programs for Multi and Uniprocessor Computers", *Proc. AFIPS 1966 Fall Joint Computer Conf.*, Spartan Books, New York, pp. 651-660, 1966.

A. Gottlieb, B. Lubachevsky, and L. Rudolph, "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors", *ACM TOPLAS* 5, pp. 164-189, Apr. 1983a.

A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer—Designing an MIMD Shared Memory Parallel Computer", *IEEE Trans. Comp. C-32*, pp. 175-189, Feb. 1983b.

A. Gottlieb and C. P. Kruskal, "Coordinating Parallel Processors: A Partial Unification", *Computer Architecture News*, pp. 16-24. Oct. 1981.

E. A. Hauck and B. A. Dent, "Burroughs' B6500/B7500 Stack Mechanism", *AFIPS 1968 SJCC*, pp. 245-251. Also in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, 1982, pp. 244-250.

K. Kennedy, "Automatic Transformation of FORTRAN Programs to Vector Form", Technical Report 476-029-4, Department of Mathematical Sciences, Rice University, October 1980.

D. Klappholz, "Stochastically Conflict-free Data-base Memory Systems", *Proc. Intl. Conf. Parallel Processing*, pp. 283-289, 1980.

D. Klappholz, "Parallelized Process Scheduling for a Tightly-Coupled, MIMD Machine", Technical Report, Division of Computer Science, Polytechnic Institute of New York, 1982.

D. E. Knuth, *The Art of Computer Programming, v. 1*, Addison-Wesley, 1968.

C. P. Kruskal, "Supersaturated Paracomputer Algorithms", Ultracomputer Note #26, Courant Institute, New York University, 1981.

C. P. Kruskal and A. Weiss, "Allocating Independent Subtasks on Parallel Processors", *Proc. Intl. Conf. Parallel Processing*, pp. 236-240, 1984.

D. Kuck and D. A. Padua, "High Speed Multiprocessors and Their Compilers", *Proc. Intl. Conf. Parallel Processing*, 1979.

D. H. Lawrie, "Access and Alignment of Data in an Array Processor", *IEEE Trans Comput. C-24*, pp. 1145-1155, Dec. 1975.

S. F. Lundstrom and G. H. Barnes, "A Controllable MIMD Architecture", *Proc. Intl. Conf. Parallel Processing*, pp. 19-27, 1980.

K. McAuliffe, Ph.D. thesis, Courant Institute, New York University, 1986, in preparation.

V.A. Norton and G.F. Pfister, "A Methodology for Predicting Multiprocessor Performance", *Proc. Intl. Conf. Parallel Processing*, pp. 772-781, 1985.

G.F. Pfister, W.C. Brantley, D.A. George, S.L. Harvey, W.J. Kleinfelder, K.P. McAuliffe, E.A. Melton, V.A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture", *Proc. Intl. Conf. Parallel Processing*, pp. 764-771, 1985.

G.F. Pfister and V.A. Norton, " "Hot Spot" Contention and Combining in Multistage Interconnection Networks", *IEEE Transactions on Computers*, October, 1985, pp. 943-948.

D.P. Reed and R.K. Kanodia, "Synchronization with Eventcounts and Sequencers", *CACM* 22, pp. 115-122, 1979.

L. Rudolph, *Software Structures for Ultraparallel Computing*, Ph.D. thesis, Courant Institute, New York University, Feb. 1982.

J. T. Schwartz, "Ultracomputers", *ACM TOPLAS* 2, pp. 484-521, 1980.

J. Wilson, Ph.D. thesis, Courant Institute, New York University, 1986, in preparation.

A UNIX(tm) SUBSYSTEM ON THE CRAY TIME SHARING SYSTEM (CTSS)*

Karl Auerbach
As consultant to:
ZeroOne Systems
2431 Mission College Blvd
Santa Clara, CA 95054
Auerbach#K%MFE@LLL-MFE.ARPA

Robin O'Neill
National Magnetic Fusion
Energy Computer Center
Lawrence Livermore National Labs
ONeill#R%MFE@LLL-MFE.ARPA

ABSTRACT

A UNIX(tm) subsystem has been constructed for the Cray Time Sharing System (CTSS.) The subsystem provides CTSS users with many System V facilities. UNIX processes are created by sub-partitioning a CTSS user process. The UNIX file system has been extended to permit UNIX data files to be stored in CTSS files or on a mass storage system. Each user of the subsystem has his own copy of the UNIX kernel and UNIX root filesystem. Major directories (/bin, /lib, etc) exist as read-only, mounted file systems, shared by all UNIX users.

THE NMFEEC

The National Magnetic Fusion Energy Computer Center was established in 1975 by the Department of Energy. The NMFEEC provides large-scale computational support to the Magnetic Fusion Energy community and the large energy research community.

* This work was performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under contract W-7405-Eng-48 and with partial support by NASA under contract NAS2-11065.

UNIX is a trademark of AT&T Bell Laboratories.

The center presently supports nearly 4,000 users.

MFECC's central computer resources include the following:

Cray 2 --	200 Mflops
CPUs:	4
Memory:	512 Mbytes
Disk:	9.6 Gbytes

Cray XMP --	140 Mflops
CPUs:	2
Memory:	16 Mbytes
Disk:	9.6 Gbytes

Cray 1S --	35 Mflops
CPUs:	1
Memory:	16 Mbytes
Disk:	7.8 Gbytes

Cray 1 --	35 Mflops
CPUs:	1
Memory:	8 Mbytes
Disk:	6.6 Gbytes

The Crays are supported by a mass storage subsystem containing 820 Gbytes of storage in nearly 900,000 separate files.

CTSS

All of the Crays at the NMFEEC operate under the Cray Time Sharing System (CTSS.) CTSS was developed at the Lawrence Livermore Labs and has spread to approximately 20 installations.

CTSS is unlike other time sharing systems. Users are isolated from one another by strong partitions. Files are typically not shared between users. The demand for sharing is satisfied by the current CTSS mechanisms and UNIX-like sharing would not be considered desirable.

The CTSS file system contains a "public" area in which sharable files are placed. Such files may be modified only by users or programs with the requisite trust level.

Private files are destroyed ("purged") on a regular basis. Users are required to move permanent files to a mass storage subsystem. Public files are immune from the daily purging.

CTSS itself imposes no structure on file contents. A CTSS program perceives a file as a stream of 8-byte words.

A user may have up to five "suffixes." Each suffix is equivalent to a terminal session. A user may easily switch between suffixes. A user may initiate programs or commands under each suffix and allow them to proceed in parallel. An active suffix contains a chain of one or more CTSS processes having a parent-child relationship.

The memory of a CTSS process consists of a single contiguous address space containing both instructions and data. There is no means to protect the instructions from self-modification. CTSS system calls give the program the means to control the upper bound of its memory space.

CTSS is a swapping, not a paging, system. Process images are written into a user file called a "dropfile." In addition to being a place to store swap images, the dropfile acts as a checkpoint, allowing restart of a program.

All CTSS terminal I/O is asynchronous, half duplex, and with no type-ahead capabilities. Terminal input is echoed and consolidated into line images by front-end processors. There is no means for a CTSS program to perform full-duplex or character-at-a-time terminal I/O.

ARCHITECTURE OF THE CRAYS

For most programs, CTSS maps the differing architectures of the Cray 1, Cray XMP, and Cray 2 machines into a single abstract machine type. However, to improve performance, CTSS-UNIX takes advantage of the special capabilities of each machine when possible.

The memory management hardware of the Cray machines uses simple base and limit registers. The Cray 1 has only one set of these registers and thus does not support the existence of pure text segments. The Cray XMP, on the other hand, has two sets and permits a splitting of instruction and data spaces, much like the PDP-11/70, thus allowing the normal UNIX sharing of pure text.

The Crays do not use segmented or paged memories. All memory segments must be physically contiguous. On the Cray 2, with 512 megabytes of main storage, this is not a particularly severe constraint. However, on the other machines, memory is a prime resource which must be carefully managed and conserved.

The Cray XMP and Cray 2 support multiple processors sharing a common memory. For the present, allocation of processors is left to CTSS. In the future, CTSS-UNIX may undertake to perform its own processor allocation and scheduling.

Interrupts and context switches are relatively expensive operations on the Cray machines. The amount of process context which must be saved is large. On the Cray 1 and XMP the hardware context is 5120 bytes. On the Cray 2 it can be an order of magnitude larger.

OBJECTIVES OF CTSS-UNIX

CTSS-UNIX is not intended to be a perfect emulation of UNIX. Rather, the purpose is to deliver UNIX-like capabilities as an extension of CTSS.

One specific objective is to provide UNIX shells and many of the standard UNIX utilities. It was recognized from the outset that tools requiring full-duplex or raw-mode terminal I/O would not be feasible.

Another goal is to use the UNIX subsystem as a means of imposing a hierarchical view upon the flat CTSS file space.

It is very important that data files developed by UNIX or CTSS utilities be easily shared between the two environments. In addition, it is desirable that the UNIX environment allow the execution of CTSS commands and programs.

Finally, with the recent emphasis by Cray Research on UNIX as the native operating system for some of their machines, it is felt that a UNIX program execution environment will promote the interchange of software with other Cray sites, such as the Numerical Aerodynamic Simulator (NAS) at the NASA Ames Research Center.

UNIX PROCESS AND MEMORY MANAGEMENT

At the NMFEC a true system V kernel exists as an unprivileged CTSS process. The CTSS-UNIX kernel uses a CTSS system call which allows the kernel to subdivide its memory among multiple, simultaneous Unix processes. For example, the UNIX kernel can load a UNIX process into its memory space, set up tables describing the context of that process, and tell CTSS to run that process. The kernel defines base and limit registers confining the process to a subset of the kernel's own space. When the child process attempts to make a system call, or if a fault occurs, the context of the child process is saved and control returns to the kernel. The UNIX kernel then examines the saved context and processes the system call or the exception.

On CTSS, the level of interactivity quickly deteriorates as the size of the process is increased. To provide a reasonably interactive UNIX system on CTSS there must be a dynamic memory management system to ensure that only the minimum amount of memory is used. For the CTSS-UNIX system, the total memory requirement is adjusted as needed. The system starts with enough memory for itself and two copies of the shell, the low water mark. As UNIX proceeds, there may be some user processes requiring more memory than is currently available. Many others will simply run within the established limits. The CTSS-UNIX system maintains a global system variable, XTRAMEM, that holds the number of words being used beyond the low-water mark. Each user process keeps track of its own portion of this "extra" memory in an entry in the proc structure, P_XTRAMEM. The following is a structured description of the dynamic memory management procedure.

The following procedure occurs whenever a user process calls "exec":

If all other processes were swapped out would there be enough memory to load the new process?

- Yes: Process "exec" normally.
- No: Compute CTSS-UNIX memory to fit process plus N additional words for possible further needs

Is needed memory beyond limit?

- Yes: return error
- No: Allocate memory
XTRAMEM += extra
memory allocated
P_XTRAMEM += extra
memory allocated

The following procedure occurs whenever a user process calls "exit":

Is XTRAMEM > 0?

- No: Process "exit" normally
- Yes: Is P_XTRAMEM > 0?
 - No: Process "exit" normally

Release P_XTRAMEM words
XTRAMEM -= P_XTRAMEM
P_XTRAMEM = 0
Process "exit" normally

A similar procedure exists for the "sbrk" system call. Since memory expansion on CTSS is expensive (CTSS forces the requesting process to be swapped for each expansion) there is always an additional amount of memory added to each expansion in anticipation of future needs (such as further "exec" and "sbrk" calls.) The additional amount of memory to be added is defined by a tunable, global system variable.

When the CTSS-UNIX system wishes to release its extra memory, it attempts to simply release the desired amount from the end of its memory space and return. However, the system's mem-

ory may become fragmented enough that there may not be enough free memory at the end of the space to perform the release. In this case, the system searches for the process that follows the first hole in memory and moves it up, which essentially moves the hole nearer the end of memory. The system then tests to see if there is enough free space at the end to perform the request. If not, the procedure repeats until all the holes are pushed to the end. The extra memory is then released with the remaining memory compacted.

In an attempt to further increase interactivity, the swap mechanism has been modified. For process swapping, the UNIX I/O request queuing has been replaced with a single CTSS I/O call. It was also found that the "sbrk" call could be enhanced. Previously, "sbrk" expansions would have tried to find enough free space in which to copy the entire process, with its new size. If enough space could not be found, the process was swapped, in order to create it. Instead, "sbrk" now attempts to locate the additional amount of space immediately following the process in memory. If found, the additional memory is simply annexed by the process, without copying or swapping.

The lack of separate data and code segments on the CRAY-1 has led to the consideration of the Berkeley UNIX "vfork" system call in order to avoid copying the entire shell image for every command.

Further attempts to increase the system's efficiency have revealed a problem with the System V "fork/swap" mechanism.

FILE AND DIRECTORY MANAGEMENT

The file system presented to a CTSS-UNIX program or user is essentially that of standard UNIX. Indeed, the file management portion of CTSS-UNIX consists largely of unmodified code from the System V kernel.

Objects representing UNIX filesystems are stored in CTSS files. These files are structured using the standard UNIX data structures (e.g. the "superblock", inodes, etc) and are initialized using the standard "mkfs" UNIX utility. Even the "fsck" utility may be used.

Two new system calls were added to manipulate the binding of a CTSS file to a UNIX special device. In this way it is possible to not only attach CTSS files as virtual disk-based file systems, but also to attach CTSS files as virtual tapes or raw devices for use by "cpio" or other utilities. These calls are restricted to the superuser.

File system mounting works just as in standard UNIX.

The disk and in-core inode structures have been enhanced. An IFCTS flag has been added to the `i_mode` field. When IFCTS is set the associated file exists not within the UNIX file system itself, but rather, is stored within a CTSS file. For such files, the block pointer field of the inode is used to hold various information including the CTSS name of the file.

This works quite well. Most programs do not detect a difference between CTSS-based IFCTS files or "internal" IFREG files. Even executable, "a.out", can be run from IFCTS files. All standard UNIX operations, including linking, work.

A few problems have been observed: When a user program (or the "test" command) examines the `i_mode` field of the inode. To reduce the problem, the "fstat" and "stat" systems calls coerce the `i_mode` flags so that an IFCTS file appears as a regular, IFREG, file with the IFCTS flag appearing outside of the IFMT mask. The fsck program will be modified to tolerate the new inode form and also to allow the existence of files which do not consume any space in the file system.

Within the kernel, IFCTS and IFREG are mutually exclusive. This was perhaps a bad choice. Had IFCTS been created as a modifier to the file format (IFMT) rather than a file format in its own right, then it would have been possible to store other file formats (e.g. UNIX directories) in CTSS files.

By default all regular files are created as IFCTS files. However, the "mode" parameters of the "open" and "creat" system calls have been extended to allow files to override the default. The main use of this facility is to maintain configuration files (e.g. /etc/passwd) and executables (e.g. /bin/sh) which tend to exist as internal, "IFREG" files. This capability may eventually be restricted to the superuser.

CTSS has a maximum file name size of 8 single-case characters. No attempt is made to compress a UNIX file name into this space. Rather, a name is generated from the inode number and user id. This tends to prevent name collisions. However, if a user views his private file space directly from CTSS he will see what appears to be files with meaningless names.

New UNIX system calls have been created to manage the association of CTSS files with UNIX paths. One new call allows a pre-existing CTSS file may be associated with a new UNIX path. When a "text" file (described below) is associated, the CTSS-UNIX kernel opens the file and examines its contents to locate the end-of-text (which, under CTSS, is not necessarily near the physical end-of-file.)

Another new system call allows the inode to be unlinked (assuming the link count is one) without destroying the underlying CTSS file. A third new call can be used to give a meaningful name to an underlying CTSS file.

All file I/O goes through the UNIX buffer cache. CTSS-UNIX uses 4K byte buffers to match the CTSS internal buffer size and disk allocation unit.

Although the kernel of neither system cares about file contents, the CTSS and UNIX utilities have somewhat different means of representing text. CTSS uses the ASCII "US" character as an end-of-line marker and the ASCII "FS" as an end-of-text-file marker. CTSS utilities sometimes compress strings of blanks into a two character sequence.

In order to mask some of these differences in text representation, IFCTS files may be tagged in the inode as being either "text" or "binary." For binary files, the CTSS-UNIX kernel makes no changes to data as it moves between a UNIX process and the disk. For text files the data is scanned and US is converted to NL or vice versa, as appropriate. This translation has a significant overhead because it requires the viewing of each character, a very inefficient operation on the Cray architecture.

No determination has been made how to handle compressed blanks. Since the CTSS representation uses a pair of characters, the kernel would have to always backtrack one character to see whether the preceding character was the escape character indicating compressed blanks. Proper accounting for compressed blanks is necessary for random seek operations.

Each user of CTSS-UNIX owns his own root filesystem. This filesystem has been purged of all unnecessary files to make it as small as possible. The CTSS-UNIX program exists as a CTSS command and is not stored on the root. The standard UNIX directories /bin, /usr, and /lib exist as filesystems stored in CTSS public files. When CTSS-UNIX initializes, it binds these CTSS public files to specific special devices and mounts them as read-only filesystems.

The user's root filesystem contains /etc, /tmp, /dev, and his home directory. Any new directories created by the user are also stored in his root filesystem. Since most files are created using CTSS files as containers, there is rarely a problem of space exhaustion.

Since the user's root contains all the mapping information between UNIX pathnames and CTSS files, it is important that the user move the root and all CTSS files associated with UNIX files to the mass storage subsystem before they are purged. This may be made an automatic function of CTSS-UNIX when a user logs out.

MASS STORAGE MANAGEMENT

For IFCTS files, the inode structures have been extended to also contain an indication whether the file is stored on an on-line disk or on the mass storage subsystem. There is also a

flag to indicate whether the file has been altered since it was last brought-in from mass storage.

It would not be a difficult operation for the CTSS-UNIX kernel to move a file between on-line and mass storage system. However, mass storage operations can involve delays ranging from tens of seconds to many hours. Consequently, when a reference is made to a file which is not on-line, the referencing program will receive an error (and possibly a signal). Then a user-level process may be initiated to bring the file onto on-line storage. Other mass storage operations (such as writing-out an on-line file) will always be performed by user-level processes.

TERMINAL I/O

CTSS-UNIX will remain half-duplex on a line-at-a-time basis. The main impact of this restriction is on programs which use raw mode, principally the full-screen editors.

CTSS already has a full-screen editor which operates, despite the terminal I/O constraints, using an IBM PC as a terminal. This editor will probably be incorporated into CTSS-UNIX.

A low overhead polling mechanism is being tried to allow some degree of type-ahead. Its effectiveness and cost have not yet been evaluated.

PERFORMANCE

We have not yet performed any quantitative performance measurements. But our subjective evaluation indicates that where comparable commands exist between UNIX and CTSS, UNIX is only marginally slower. The performance degradation seems to be no worse than that incurred by UNIX subsystems

which have existed for many years on VAX/VMS.

CTSS-UNIX does have known inefficiencies:

Image copying during a "fork" is expensive and often involves swapping.

Program loading during "exec" generates at least one CTSS I/O request per block (unless the block is already in the buffer cache.) We are considering means to store commonly used executables in contiguous, shared (public) files so that we can load a program with one I/O request.

The CTSS 3400 system call mechanism contains some CTSS administrative functions which slow CTSS-UNIX context switches and system call handling. It is possible that CTSS can be modified to bypass some of these functions when CTSS-UNIX is running.

FUTURE DEVELOPMENTS

CTSS-UNIX will be extended to allow invocation of native CTSS commands and programs directly by the user or by UNIX processes.

Performance improvements will continue to be made. It is expected that CTSS-UNIX can be made nearly as efficient as a native mode implementation, although the throughput will never be as great due to memory and CPU contention with CTSS programs.

CTSS and UNIX may be merged more closely to allow better sharing of programs and data.

The CTSS-UNIX kernel retains multi-user functions. Rather than having a separate kernel and root file system per user, CTSS-UNIX may be expanded to be a true multi-user subsystem.

SUMMARY

The feasibility of a useful UNIX subsystem on CTSS has been demonstrated.

Some unusual mechanisms have been employed to facilitate co-existence with CTSS and to enhance performance. Despite these, CTSS-UNIX presents functionality to users and to programs that, except for full duplex terminal I/O, is quite similar to true System V UNIX.

Considerable work remains to transform CTSS-UNIX into a reliable, efficient production tool and user work environment.

ACKNOWLEDGEMENTS

The original CTSS-UNIX was developed at Cray Research Incorporated, in particular, by George Spix and Dave Slowinsky.

A Unix-based Operating System for the Cray 2

*Timothy W. Hoel
Bruce A. Keller*

Cray Research, Inc.
1440 Northland Dr.
Mendota Heights, MN 55120

ABSTRACT

We have ported System V Unix to run on the Cray 2. A number of changes were necessary to make it run well in our special environment. The motivations for change and the changes themselves are described.

Introduction

Cray computers are quite different from those on which Unix typically runs. The Cray 2 has four tightly-coupled cpu's with a 4.1 nanosecond cycle time, 256 Megawords (64 bit) of central memory, up to 36 disk drives (currently 600 Megabytes each with 32 Megabits/sec transfer rate), and up to four network adapters 50 Megabits/sec each.

The cpus and I/O channels are indeed very fast. We have demonstrated a matrix multiply program running on all four cpus at 1.6 GigaFLOPS (floating point operations per second), and also demonstrated a single process reading data from 20 drives in parallel at 600 Megabits/sec.

The system designer cannot become careless, however, because some things take a long time. It requires four disk drives to store a full memory image and, even when using four parallel I/O streams, it requires 128 seconds to move all the data. Each cpu has a very large register context: eight address registers, eight scalar registers, eight vector registers (each with 64 words), and 16K words of register back-up storage called "local memory". Doing a full context switch from one process to another is, relatively speaking, expensive.

The job mix on a Cray is quite different from that of a typical Unix machine. Cray customers primarily run large Fortran programs which often consume several hours of cpu time. They have traditionally used a batch interface to their machines. Recently, however, customers have become increasingly interested in using an interactive interface to test and debug their programs. They will probably continue to use batch for their production runs. This transition from batch to interactive is one of the reasons that Unix was chosen as the base for the Cray 2 operating system. In order to be successful, the new operating system must not only provide good interactive service, but it must also allow sophisticated users to have full access to the raw power of the machine.

The primary reason for Cray's success as a company can be summed up in a single word: performance. At Cray, compatibility, simplicity, and elegance are considered to be important virtues,

Cray 2 Operating System

but they fall distant second to performance. We have only made changes when we felt they were truly necessary, and then we have tried to integrate them cleanly into the Unix environment.

The File System

In designing the file system for the Cray 2, several requirements had to be met. First, Cray users are accustomed to reading (and writing) data from disk at device speeds ("streaming"). But sometimes that is not fast enough. The term "striping" refers to spreading a user's file across several drives, allocating the next piece of the file from the next drive in rotation. Thus, the file can be read at N times the transfer rate of a single drive. The new file system had to support both "streaming" and "striping". Second, a file's size could not be limited by the size of a single disk drive. Third, bad blocks are not uncommon on the disk drives we use. The new file system had to gracefully and efficiently work around them.

The Cray 2 hardware and firmware is especially good at reading (and writing) tracks of data from the disks. After sensing the rotational position of the disk, it transfers the next sector to the corresponding position in the memory buffer. In this way a track of data can be moved in one revolution plus, on average, one-half sector time of latency.

To take advantage of this capability, large files are allocated and accessed track-at-a-time. A bitmap is used to record available tracks so that as a file grows, a "nearby" track can be allocated to minimize seek time. Since interactive systems typically have many small files, we use a variation of the large block/small block system to avoid wasting excessive amounts of disk space. Files that are eight sectors or less are allocated sector-at-a-time from a free list as in standard Unix.

Our inodes contain eight pointers and a bit indicating small or large format. For a small file these point to eight sectors (4096 bytes each) of data. For a large file the first five point to tracks (currently 18 sectors each) of data. The sixth points to a sector containing 512 pointers to tracks of data. The seventh is double indirect and the eighth is triple indirect.

When a small file grows to be larger than eight sectors, the system automatically converts to large file format. This conversion requires allocating a track buffer, copying data from sector buffers, and sometimes rereading sectors from disk. To avoid the overhead of conversion, a user may request large file format when creating the file.

Unlike the Berkeley 4.2 file system we do not attempt to reclaim the unused space at the end of the last large block (track). Given our environment we prefer to trade off disk space for a simpler, faster algorithm.

A "partition" is a contiguous group of tracks on a disk drive. In addition to the major/minor device numbers, the special node in /dev for a partition also contains other information describing the partition, including the starting track and number of tracks. At mount time and/or device open time this information is copied to system tables for use by the disk driver. Previously this information was hard-coded into the disk driver which made it inflexible.

A file system resides within a "cluster", which is a group of one or more partitions. The partitions of a cluster need not have the same size or placement, but it is expected that they reside on different physical drives. Each partition contains its own super block, dynamic block, inodes, and data space. The super block contains static control information about the partition

Cray 2 Operating System

including the cluster id, the number and location of inodes, and the number and location of bad blocks. The dynamic block contains the free track bitmap, the sector free-list and other dynamic information. The primary reason for splitting the super block into two blocks was that the new information would no longer fit into one. Because the file system is now two-dimensional, block and inode numbers are two-tuples containing the partition number and item within partition.

Previous Unix systems have allowed a file system to span multiple drives by having the disk driver map the Nth block onto the $(N \bmod D)$ th drive, where there are D drives in the group. Although this accomplishes one of our goals and has the virtue of leaving the higher levels of Unix unchanged, it does not allow the sophisticated users in our environment to have the necessary controls over file placement. With our two-dimensional system the user may optionally specify a bit mask at file creation time to indicate which partitions should be used to store this file. A bit mask of all ones would indicate full-width striping for maximum transfer rate to this file. On the other hand if the user had two files and knew that the cluster contained six partitions, file A could be created with a bit mask of 07 and file B could be created with a bit mask of 070 thereby achieving maximum transfer rates for both files and avoiding head contention. If a requested partition is not available, the system uses other space within the cluster and does not return an error to the user.

The standard Unix file system is vulnerable to significant corruption when a directory or inode block becomes unreadable. We have made the file system more robust by redundantly storing critical information in two places, on separate drives if space is available. This "shadow" information is created and maintained by the kernel, but currently we do not use it for on-the-fly recovery. Rather, fsck can rebuild an intact structure in the event of a disk (or system) failure.

Each directory entry contains a primary and a shadow inode number. The shadow inode is a duplicate copy of the primary inode with the possible exception that the block pointers in the shadow inode may refer to a second copy of the data blocks. This facility is used to shadow directory data blocks and indirect blocks. With this design the file system tree structure itself can be completely rebuilt after any single point of failure, even losing a whole drive. Of course the contents of data files may still be lost and must be recovered from some backup medium.

Although it may appear that we have made fundamental changes to the file system, the modifications were fairly localized, thanks to the clean structure of the kernel. The normal user interface is unaffected and the kernel itself is still quite recognizable.

Asynchronous I/O

The user I/O interface in standard Unix is fundamentally synchronous. This design decision simplified the kernel and user programs, and has worked very well in practice. The automatic read-ahead and write-behind of the file system allows many user I/O requests to complete immediately. This design allows a group of users to share a system and all get reasonably good performance.

It is not uncommon, however, for a Cray customer to dedicate a system to a single job for many hours. In this environment it is important to avoid all unnecessary delays. For example, assume a process is doing random I/O to two files which reside on two different drives. In standard Unix there is no way for a process to overlap the seek and latency times for two I/O requests.

Cray 2 Operating System

We have extended the user interface and added two new system calls: `reada` and `writea`.

```
int reada ( fildes, buf, nbyte, status, signo )
int writea ( fildes, buf, nbyte, status, signo )

int fildes;
char *buf;
unsigned nbyte;
long *status;
int signo;
```

The first three arguments have the same meaning as in a standard read/write system call. The file position is always the current position at the time of the `reada/writea`. The file's position is incremented at that time by *nbytes*. If the I/O request can not be satisfied immediately, the request is queued, and normally control is returned to the process. The process may block, however, if it has too many outstanding asynchronous I/O calls or if it is due to be swapped out. When the I/O does complete, the operating system will put a non-zero completion status into *status* and send the signal *signo* to the process. In order to use signals as the notification mechanism, we also had to modify the system call interface for signals to make them more reliable; the standard user interface is now supported by a library routine.

The implementation within the kernel is uniform so that `reada`'s and `writea`'s may be issued on any open file descriptor, including pipes and pseudo ttys, with the expected results. Every device driver must now have a strategy routine. If the request cannot be satisfied immediately, the strategy routine must queue the request and return without sleeping. A fortuitous consequence of reimplementing pipes as a pseudo-device driver was that a new optimization was trivial: if the writer and reader are both in memory, data is copied directly from the writer's buffer to the reader's without going through system buffers.

It is worth noting that the capabilities of this mechanism are quite different from those offered by the `select` system call of Berkeley 4.2. First, since there is no asynchronous notification mechanism, the nature of the `select` call dictates a polling implementation, rather than an interrupt driven one. Second, since there is no way to specify the size of the intended write request, there is no way for the operating system to guarantee immediate success. Lastly, `select` only provides useful information for terminals and sockets, but not for disk files.

Raw I/O

In standard Unix "raw" disk I/O has two different attributes. First, it implies moving data directly between the device and the user's buffer, thereby eliminating the memory-to-memory copy. Second, it implies accessing particular physical blocks of the disk, completely bypassing the security of the file system. Because of the second attribute this mechanism is normally restricted to super-user only, even though the first attribute could be quite useful to users who wish to squeeze every last drop of performance from the machine. We have extended the user interface to allow a process to specify that a normal file should be opened in "raw" mode. The standard restriction then applies that the file must be accessed in n-sector pieces.

Cray 2 Operating System

Networking

Terminals cannot be connected directly to the Cray 2. Rather, they must be connected to a front end computer which in turn is connected to the Cray 2 via a network. We currently use Network Systems Corporation's HYPERchannel, a multiple-access, carrier-sense, collision-detect network with a peak transfer rate of 50 Megabit/sec. We are in the process of adding support for a point-to-point connection to allow fast file transfer between mainframes. Later we may add support for Ethernet.

>From the beginning it was clear that we would have to support multiple protocol families simultaneously, sometimes through the same physical interface. To support early development, we created two simple protocols for file transfer and interactive access nicknamed SEP for simple, effective protocols. Many of our customers currently use a Cray proprietary protocol, named SCP, to communicate between their various front-end computers and their Cray 1 or Cray XMP machines. Since this is an established, proven environment most of these same customers would prefer to continue using SCP to communicate with the Cray 2, at least initially. Of course there is great interest in TCP/IP and it will be one of the first protocol families available. There is also interest in Netex, ISO, and others.

Our HYPERchannel driver is protocol-independent and allows multiple processes to share one or more adapters. Embedded in the HYPERchannel destination address is a logical port number which the driver uses to route in-coming messages to the correct process.

Previous attempts to implement network software outside of the kernel have suffered greatly for lack of asynchronous I/O calls (see "A Retrospective" by Dennis Ritchie, BSTJ July 1978). We used our new asynchronous reada/writea system calls to implement both the SEP and SCP protocols at the user level and found them to be extremely valuable.

To support interprocess communication between daemon and client processes, we have implemented a pseudo tty driver which, internally, is rather like a bi-directional pipe. When possible, data is moved in large blocks and, as with our pipes, if the writer and reader are both in memory, data is copied directly from the writer's buffer to the reader's. Although these optimizations are probably unimportant for user input, we anticipate they will be quite useful for output to graphic work stations and/or bitmap displays.

Multitasking

The standard Unix interface allows a user to easily spawn processes which can execute concurrently on a multiprocessor machine. However, it is not always easy to partition a workload at a macroscopic level into separate processes that can all work together effectively. Therefore, we also want to allow a single user program to execute concurrently on two or more cpus. To that end, we have added a new system call, tfork, which creates a separate process, but one which shares the same text and data space. The Cray 2 hardware supports semaphores which these "Siamese twins" may use to cooperate.

The Fortran Multitasking Library uses tfork and semaphores to create a higher-level abstraction of tasks, locks, and events. When there is nothing for a twin to do, it will first note its intension in a shared table and then voluntarily give up control with a pause system call. When another twin later realizes there is work to be done, it will awaken any sleeping twins by sending a signal.

Cray 2 Operating System

The Initial Port

There have been several stepping stones along the way to having the system we describe here. When we first began this project the Cray 2 did not exist. Therefore, our first goal was to port System V onto a Cray 1 making as few changes as necessary.

Of course we had to rewrite the machine dependent portions of the kernel and write new device drivers, but we also had to make a few significant changes. Unix assumes that the current user structure can be remapped to a fixed address for the kernel. Since the Cray 2 hardware only allows access to a single memory segment, this fundamental assumption is no longer true. This forced two changes to the kernel. First, we had to add a pointer variable called "u" and change all references to the user structure from "u." to "u->". Second, the kernel stack in the user structure is self-referencing, and, therefore, must be used at a fixed address. For our initial port we simply copied the stack to and from a fixed location, knowing full well that we would have to find a better solution to support multiple processors.

Unlike most Unix machines, Crays are word addressable, not byte addressable. Many commands, and even the kernel itself, were not always careful to cast pointer arguments correctly.

Using a Cray 1 simulator running under COS on a Cray XMP, we debugged our initial port. Within three months Unix was up and running native on a Cray 1 with a few ported commands.

With a cross compiler on the XMP and other file system tools, we built a Cray 2 file system on a disk drive, and then rolled it down the hall to the other computer room. Since it took 10 minutes to move 800 Megabytes, we figure our first Cray 2 network had an effective bandwidth of 8 Megabits/second.

To support multiple processors, we no longer copy the kernel stack, but rather use it in place. If the user image is moved in memory, the kernel stack is relocated to the new address. We also had to modify the scheduler to give Unix more than one place to idle. In standard Unix when there is nothing to do, the kernel loops on top of process zero. Since all four cpus can't run on top of the same process, four idler processes were created. The system now only switches to process zero when there is swapping to be done.

Conclusion

At the time this is being written (December 1985), this system is running in-house at Cray and at two customer sites. It is quite stable and the changes we have made seem to be working out quite well.

Ada and the UNIX System

January 17, 1986

Denver, Colorado

Ada, "C", and UNIX¹

Herman Fischer
Mark V Business Systems
16400 Ventura Boulevard
Encino, CA 91436
(818) 995-7671
{ ihnp4, decvax, randvax } ! hermix ! fischer
HFischer@isif.arpa

1. Introduction

1.1 Scope

This report evaluates the usages of Ada², C, and UNIX³ for development and deployment of Defense Department Mission Critical Computer Resources (e.g., operational military applications). It takes the perspectives of industry, borrowing from management oriented views expressed by the Council of Space and Defense Industry Associations (CODSIA), task group 13-82; the perspectives of technical users, as expressed by the Ada, UNIX, and KIT/KITIA communities; and the perspectives of the military user community.

1.2 Terminology

The term, *host* will be used to describe computers, software, and systems used to develop software. The term *target* will be used to describe computers and systems where operational software resides.

1.3 History

Ada, a language, was designed for DoD applications (target systems), where software reliability, maintainability, and reusability were important; *C*, a language, was designed for better implementation of an early assembler version of UNIX (then an Operating System plus early toolset), where fitting the compiler into limited mini-computers was important; and *UNIX*, an

Operating System, was designed as a personal development (host) environment for a Bell Labs "guru" on a very early DEC computer with limited resources.

1.3.1 Ada

The Ada language resulted from a DoD-wide effort, beginning in 1974, to design a common language for Mission critical computer resources software (e.g., embedded applications for "target" systems). Requirements were formalized in a series of documents extensively reviewed by the Services, industry, academia, and foreign military departments. The Ada language was designed in accordance with the final form of these requirements, called *Steelman*. Competitive procurements for four language design teams led to the selection of the "Green" Team's product, which became known as Ada.

Ada was adopted as standard by the American National Standards Institute in early 1983, and is in the ISO (international) standardization process.

The U.S. Army's Ada development efforts expanded in concept from an Ada compiler (only) to an Ada Language System (ALS) environment. Softech is currently delivering initial versions of the ALS for Army contractor use. Though this specific compiling system's implementation is said to be sluggish in performance, Softech personnel believe the performance of later versions will be better.

The Air Force began a competition for a standard integrated host environment. The successful design, by Intermetrics, became known as the Ada Integrated Environment (AIE). AIE was never built as designed. AIE's designers eventually became prime technical contributors to the CAIS (description following), as their own ideas matured.

1. This paper was published as "Ada & C with UNIX in MCCR" by *Defense Science & Electronics* in November and December 1985, and is reprinted with the permission of Rush Franklin Publishing, Inc.
2. Ada is a registered trademark of the U. S. Government, Ada Joint Program Office.
3. UNIX is a trademark of AT&T Bell Laboratories

Ada, C, and UNIX

A significant number of privately-funded Ada compiler developments have been recently validated. Most of these are of average to moderate compiling efficiency, and recent reports show that efficiency of programs generated using some are becoming "decent". There is now even an IBM-PC Ada interpreter, said to be derived from the Army's "Ada Ed", which will be distributed for about \$100.00, and will be competitive in performance to the Basic interpreter which is distributed with PC-DOS and MS-DOS.

During the final design stages of Ada a similar requirements definition effort was begun for environments to host Ada. A document called *Stoneman* identified relationships between the tools in an integrated Ada Program Support Environment (APSE), and forms the basis of Ada tool integration efforts. In 1982 an evaluation team called the KAPSE Interface Team (KIT) was convened to define requirements for interoperability and transportability among KAPSEs, guidelines and conventions to achieve this, and eventually a (set of) standards. The KIT, being an all government entity, was augmented with a team of competitively sought representatives of industry and academia, known as the KAPSE Interface team from Industry and Academia (KITIA).

A KIT/KITIA working group developed a framework of interfaces to achieve tool portability. This Common APSE Interface Set, CAIS, was recommended to become a military standard. Numerous arguments have ensued as to its readiness, though all concerned feel it is appropriate to prototype its concepts and encourage its further evolution and improvement.

1.3.2 The C Language

Early UNIX code was in assembler language. The lack of an HOL for the early versions prompted UNIX's developer to develop *B*, based on a language known as *BCPL*, prior to porting the system from the early PDP models to the PDP-11. After a few utilities were coded in *B*, it became apparent better structuring and data typing would be needed for rewriting UNIX.

A new language was derived from *B*, called *C*, to implement UNIX in a manageable HOL form. The definition of *C* includes only the basic data typing, data and program structuring and operations. By separating systems-level functions away from the language, into the operating system definition, *C* has been relatively resilient to technological changes in the underlying systems which host it. *C*'s operating system support has changed significantly in character, as UNIX has evolved, but the basic language itself has been relatively stable.

While there are large numbers of purveyors of *C* compilers, only a few are known to be efficient or complete. The lack of standards hinders portability among "non-UNIX" *C* compilers. It also hinders consistency needed to ensure that "tricky" language constructs operate identically among the various versions. Some *C* compilers are quite complete and known to generate extremely efficient object code; others lack enough compatibility to deal with code developed on different systems.

The ANSI technical committee X3J11 is chartered to develop a standard for the *C* language, including its libraries (UNIX support functions) and environment. The current schedule calls for publication of a draft in 1985.

A superset version of *C*, called "*C++*", has been implemented, to solve *C*'s problems with data abstraction. *C++* adds to *C* the strong typing, overloading, and polymorphic typing (generics) needed to support programming-in-the-large and *object based* programming⁴.

1.3.3 UNIX

UNIX was originally designed by a Bell Labs programmer, to solve a specific need for a personal working environment. In 1969-1970 it ran on early Digital Equipment computer systems (PDP-7 and -9) which had severe limitations. By 1971 it was

4. B. Stroustrup, "Data Abstraction in *C*", *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, October 1984, pg. 1701-1732,

Ada, C, and UNIX

hosted on PDP-11 computers, which remained the primary host until the VAX series debuted.

The most important achievement of UNIX, according to its originators⁵, is "to demonstrate that a powerful operating system for interactive use need not be expensive either in equipment or in human effort... and less than two man-years were spent on the main systems software [kernel, early utilities, etc]..."

The original versions of UNIX were written in assembly language; however, during the summer of 1973 it was rewritten in C. The HOL version of UNIX was about one third larger than the MOL version, but became the standard. In the larger size was included multiprogramming, reentrant code sharing, and some other features; these improvements more than offset the loss in kernel space.

Once implemented, the UNIX environment grew by contributions of tools which became incorporated into the UNIX distributions. Today "UNIX" is far more than a traditional host operating system, because of the rich collection of environment tools and supporting features which comes with it. UNIX embodies several choices of programmable command interpreters (called *shells*); a configuration management system for software development libraries; an unusual language for pattern matching, report generation, and associative array processing, called *AWK*; several powerful editors and text composition facilities, a robust set of electronic mail and communications utilities; and as widely known, the C programming language and supporting tools for debugging and managing software developments in C.

With UNIX, each added tool appears to the interactive user to become an intrinsic basic command or part of the system. Addition of a spreadsheet program, or of an accounting

program, normally appears to be integrated in the sense that they can pass data amongst each other. This feature, of growth by addition of tools, has been popular with users, because it allows each user to custom-tailor his operating environment to his own needs.

Though there have been a few attempts to use subsets of UNIX in embedded targets (e.g., switching centers originated at Bell Labs), little has been reported in the public literature, and there are no known sales efforts to distribute any UNIX-compatible interfaces for embedded (diskless or operator-less target) applications systems.

1.4 Standardization

Ada, and C with UNIX, both derive their impetus from the perceived needs for standard languages (and operating systems interfaces). While early software developers felt nonstandard languages would help them preserve customer bases (by limiting competition), today's commercial and government marketplaces demand standard software languages, standard operating systems, and interoperable environments. Indeed, UNIX's popularity is due largely to the perceived ability to port UNIX onto nearly any ISA (hardware), and thus be largely hardware independent. One of the key issues identified by the CODSIA task group 13-82 was industry's identifying a need to preserve an environment of *Continuing Competition*, an obvious benefit of standardization. But rather than concentrating on standardization, we should examine the purpose of each of these entities.

1.5 Purpose

This section examines why Ada, C, and UNIX exist.

1.5.1 Purpose of Ada

Ada is a language for expressing computer programs which is particularly suited to DoD (and other users') needs for reliable software which can be maintained by a wide base of programming talent. Ada has features which make it particularly suitable for *programming-in-the-large*, e.g., for software efforts whose total size often exceeds 1/2 million lines of source coding, and requires large teams of programmers,

5. Ritchie, D., and Thompson, K., *The UNIX Time-Sharing System*, Western Electric Co., Inc., October 1981

often characterized by a high turnover in personnel.

Ada defines many mechanisms to help catch erroneous program constructs, and to aid in producing reliable software. Ada also emphasizes two aspects necessary for programming-in-the-large, modularization and interface control (e.g., strong type checking across modules written by different authors).

Ada is intended for both computer programs which must operate in embedded target systems (such as systems with neither disk storage or interactive operator terminals), and those which operate in more traditional environments (timesharing manned systems). It is intended to be useful for both target applications and development (host) tools.

Ada is intended largely for computer programs which are maintained separately from their users. Ada programs must be compiled and linked together before they can be installed (they have *static* linkages). To install new software, a user will have to shut down his operations and load the new program. This is in contrast to most business users, such as banks and DoD MIS users, where software maintenance is done on the same hardware as applications are used. The common practice with Cobol is to use *dynamic* linkages between programs, which permits the using organization to replace or upgrade specific programs without shutting down the entire system to load a newly linked set of software. [CAIS attempts to provide dynamic linkages outside of the scope of Ada.]

1.5.2 Purpose of C

C was originally designed for expressing computer programs which needed to both be compiled and be executed on minicomputers which had severe limitations on hardware resources. C was an attempt to separate the language definition from the run time environment. C models basic operators and operands of most current computers, but is indeterminate with respect to many attributes of the target (run time) environment. A C program is thus inseparable from the definition of the functions

provided by the run time environment (such as UNIX), while more robust languages, such as Ada, define basic resource control, input/output operations, as intrinsic parts of the languages.

C has been used successfully for building UNIX, thousands of small (one to five man) projects, some medium-size projects, and even a few military systems. The author feels C is highly suitable for programming-in-the-small or -medium, but unless augmented with new features, remains difficult or unwieldy for programming-in-the-large. C gives the programmer a significant degree of expressive freedom, and is known to be efficient to execute.

C as a language (e.g., exclusive of UNIX-style system level services) is suitable for both host system tool development and target system applications. However, no standards or guidelines are in common use with respect to "non-UNIX" non-host use of C. Most embedded targets using C have developed their own system interfaces (from scratch), and thus are of very limited portability or reusability.

1.5.3 Purpose of UNIX

UNIX is primarily a host-environment (disk and interactive terminal based) operating system.

UNIX is today a confusing term because it connotes three things to most listeners:

- a specific implementation of an operating system,
- a set of utilities and user services which augment a basic operating system to comprise an environment,
- a definition of a set of standard interfaces.

This confusion is compounded because UNIX appeals to three classes of users, with different viewpoints:

- Software developers
- Embedded system applications designers
- Office and Data Processing users

The UNIX kernel performs low-level input/output, resource control and

scheduling, and provides an intrinsic model of data structures (*file system* model) visible to user-level services.

The user services available through several choices of programmable command interpreters (*shells*) include software development tools, documentation and text processing tools, and electronic mail communication tools. This set of user services is extensible as the user adds tools and programs to his private and the system's libraries (such as addition of spreadsheets and report writers).

The UNIX services available by function call to software programs include functions such as file reading/writing, multiprogramming, and data interoperation with tandem programs. These, too, are well integrated, and user-extensible.

But one should realize that UNIX is not for all embedded targets.

UNIX definitions and services apply primarily to host (development) systems and to target systems with disk storage and interactive users/terminals. UNIX subsets have been (sparsely) applied to embedded applications; usually embedded targets developed with UNIX use special target runtime environments.

Employment of current UNIX coding for embedded targets (those without either disks or interactive terminals) is difficult. Thus use of UNIX in embedded systems would be restricted to a subset of the interface definitions. (This does not, however, restrict UNIX's usefulness as a host for *developing* software for such systems.)

1.6 Issues and Criteria

A number of the issues raised in CODSIA Report 13-82 facilitate the comparison of Ada, C, and UNIX. These are:

- a. *Expensive and Lengthy System Development and Evolution.* DoD systems take long time periods to develop, and are then supported for long periods. Most military systems' lifetimes exceed hardware and software "generations", and must thus be able to respond to changes in evolving hardware technology. Software

must be able to be ported to replacement hardware, and must itself be evolvable as tactics, requirements, and the military environment change.

Related to this topic is the often large scope of military systems' software projects. Often these exceed one half million source lines on a give project, thus casting them as *programming in the large*.

- b. *High Cost and Risk for Nontransportable Software.* Present generation military systems often have justified assembly coded software, and design of project-specific language environments and extensions, because tight control of computing resources was necessary to meet the real time requirements, and the economics of the system. Though language commonality, as a solution to transportability issues, has been addressed for a while (and, indeed, led to the Ada program), DoD and industry are only beginning to address a central aspect of true transportability -- run time software environment compatibility.
- c. *Technology insertion.* Technology insertion traditionally addresses black box hardware replacements. Today it must additionally address retention of software where hardware is upgraded, upgrading of software (both in functionality/operational needs, and in technological sophistication), and more fundamental technology changes (converting from Ada or C to non-Von Neumann implementations such as Lisp and Prolog, or from procedural implementations to knowledge-based implementations).

Two primary concerns are the upgrading of the fielded system and its applications software, and the upgrading of the support software (Language, tools, host OS, etc).

- d. *DoD versus Voluntary Standards.* Ada has taken over ten years to evolve from a concept to a field of validated immature compilers. (Cobol and Ada are, however, proofs that DoD can

make valuable contributions to standardization processes.) C was designed and initially implemented by one person, about when the need for Ada was perceived, and in the same ten years, matured to become a widely accepted systems implementation language. Most college students today have C training, somewhat like the situation ten years ago when most had Fortran training, because it is a mature, conservative choice, and enjoys significant support.

- e. *Preserving competition.* The implementation of DoD-owned Ada compilers (when compared, performance-wise, to compilers implemented by commercial competitors) seems a clear case-point on the need for competition. While both public law and common sense demand that DoD maintains competition in its software supplier community, one need only look at the current experiences with developers of complex software. Developers spurred on by competitive pressures seem to succeed where DoD captive projects take excessive schedules, and provide less capable performance.

2. Comparing Languages to Operating Systems

While it might appear silly to compare a language (Ada) without an operating system to another language (C) in combination with an operating system (UNIX), this is relevant in the case of C for a simple reason: C defines only the syntax and semantics of data manipulation, procedure calling, and control structures, leaving to the programming environment (UNIX) the definition of input/output, resource control, and real-time interprogram communication. On the other hand, Ada defines in the language itself the real-time communications between concurrently operating programs, the resource control features (data storage allocation), and rudimentary input/output (especially as these pertain to target, as well as host environments). Ada does not define sophisticated input/output, sophisticated features for time sharing, data/file system storage models, or other of the higher level

features usually found in host operating systems. Thus UNIX, added to C, provides a significantly larger set of system features than Ada alone has. Of course there is no reason why one could not compare C with UNIX to Ada with UNIX. That would be more fair.

2.1 Host Environments

UNIX represents a host (development) environment, as well as an environment for applications which have similar hardware. For Ada, the KIT/KITIA have come up with a proposed set of software interfaces which provide benefits of tool-level portability and interoperability, the CAIS. (CAIS does not, however, connote a specific "tool collection" as is the case with UNIX.)

2.2 CAIS

In the sense that a portion of UNIX (C library) is required to augment the machine independent portion of C with sufficient functions to be useful in an operating environment, Ada must also be augmented with functions, at least in the host system environment, because it too lacks tool support functions (of the sort provided by UNIX). Two key features absent from Ada are the underlying model of the system level data, and the ability to support dynamic binding (process control). The Common APSE Interface Set, CAIS, defines a *node model* (file system model) and a dynamically bound model for multiple independent programs to inter-react as processes in real time. CAIS augments Ada with some of the (library-level) functions found in UNIX. CAIS does not seek, however, to define all of the tool interfaces found in UNIX, and it does not define any accompanying utility programs, user shells, and the sort of functions expected of UNIX distributions.

Comparing Ada plus the CAIS is equivalent to comparing C plus a subset of UNIX represented by its programmer-level system calls.

Since UNIX is extensible and modifiable, it would also be interesting to compare Ada and C in a modified operating systems environment which consisted of UNIX features with CAIS functionality.

2.3 Target Environments

As mentioned, Ada has facilities adequate for a number of embedded target applications, and needs additional supplemental facilities for target systems which utilize operating systems of the more traditional variety.

C without UNIX has less system resource control and systems-interface facilities than does Ada; however, as mentioned, there is little information on use of UNIX-compatible systems level interfaces in embedded non-operating system-like targets.

A working group, the Ada Run Time Environments Working Group (*ARTEWG*) has been formed to address needs and solutions to the embedded use of Ada.

3. Language Comparisons, Ada and C

This section compares the two languages, Ada and C, the perspectives of the issues identified earlier. Then it attempts to come up with observations on the current use of these languages for mission critical software.

It is fair to state, before examining these issues, that each language has both merits and domains of applicability to defense systems. To summarize to the most abstract level, Ada meets longer term requirements for a language for *programming in the large*, for programming which is understandable by larger crews of less sophisticated backgrounds, and for software which needs sophisticated real-time concurrent multiprogramming. With the same level of view, C is a language of terse powerful expressiveness, which was eagerly adopted by the nation's university-level sophisticates as well as by developers of commercial MSDOS packages like spreadsheet programs, and *Dbase III*, because it allows them to express solutions compactly, efficiently, and with minimal isolation from the underlying hardware. Current applications written in C are very efficient. C compilers are (from some but not all sources) very mature and solid. But, like the highly mathematical language, APL, the religious fervor with which C sophisticates defend it belies the difficulties that larger crowds of "less intellectual"

programmers have with coding produced by the "UNIX Gurus".

It is probably relevant to comment that many (but not all) writers of Ada coding are expected to be prolific commenters (current experience varies from 3:1 comments to code ratios to 1:10 on the light extreme). Most C code from the UNIX operating system is sparsely commented; comments usually note why the code author did something, rather than what he did. Comparing Ada and C may well be a comparison of intellectualism (or, perhaps, elitism), versus programming in the large. Military applications span both spectrums.

3.1 Issues comparison

This section will compare C and Ada with respect to the issues identified earlier.

3.1.1 Expensive and Lengthy System Development and Evolution.

The long term and large size of military systems often contributes to a large staffing requirement, and a volatile (high turnover) staff. Ada has several features which specifically address this problem. These are discussed as a solution to *programming in the large*.

3.1.1.1 Staffing

In the past several years, UNIX has become very popular at universities and colleges. Software department managers report that the large predominance of new-hires is trained in C at the present time. Recent availability of Ada compilers may encourage further training of Ada in the schools. It might be necessary to distribute Ada to universities to ensure a large pool of Ada-capable talent.

Many, however, report that C programs are prone to be hard to understand. The style of some programmers leads to extremely compact code which is more intellectual than straight-forward. In particular, UNIX internal code itself is hard for less intellectual staff to comprehend.

3.1.1.2 Programming in the large

Programming in the large requires support for a very large number of modules, by a large staff. A one-million source lines

project, if modularized into units no larger than 50 lines each, consists of twenty-thousand different modules. Managing such a collection of modules, and ensuring their proper interoperation, is probably far more difficult than designing, coding, and testing the modules proper.

Ada is the first language designed to deal with such large collections of program units. It does this by separating data definitions from procedural coding (common practice but not enforced with C), and by having the capability to verify that each called program unit properly meshes with the caller. It also provides generic program units, which can be instantiated to operate on a variety of data types, to aid in reducing the number of unique program elements.

The recently published C variant, C++, augments traditional C with the same sorts of functionality.

Many other features are required for programming in the large. These include version control, managerial support, analytical tools, and the like. Such features can be provided by programming host tools, either using UNIX features, using user-installed tools on a UNIX system, or using Ada-supportive tools on an Ada host.

3.1.1.3 Enforcing software quality

Ada, as a language, enforces strict rules designed to minimize errant behavior of program code. C, as a language, has significant expressive power which can easily be misused (but, of course, that is avoidable).

Ada is a *strongly typed* language; fields of specific types can only perform desired operations with other fields. When so defined, a base of a triangle can be specified as multipliable by the height, yielding an area. But addition of an area with a height would be disallowed, and caught by the compiler. Furthermore, when one subprogram calls another, the language specifies that data passed between the subprograms must mesh properly in terms of types and variables. C also permits users to define typing, but (except for the very newest C compilers) does not enforce usage of this practice. A companion program to C, known as *Lint*, can be electively used to

check program purity. Because *deLinting* is optional, many C users produce programs which either avoid use of typing or fail to adequately type everything.

Current C compilers do not check for type mismatches and data mismatches between separately compiled programs. Ada does. A frequent cause of debugging difficulties with C results from these types of mismatches. (With the cited example of 20,000 modules in a large defense system, this characteristic is significant.)

3.1.2 High Cost and Risk for Nontransportable Software.

Ada is designed specifically to result in transportable software. Compilers must be validated to provide identical results for a standard test set; users are provided a high confidence of Ada program portability.

C is known for producing programs which have various degrees of difficulty in porting. This often occurs because C compiler limitations differ between targets and compilers. Most frequent complaints about C portability centers on differences between processors in pointer size, and pointer compatibility with integer arithmetic. C programmers tend to use sophisticated pointer manipulation (which Ada prevents). They also tend to define bit fields and hardware dependencies which make programs nonportable. For example, the many 68000 implementations today differ significantly in their memory management implementations. A program written for a virtual environment is unlikely to run on a swapping environment, even if both are UNIX.

Problems which hinder the portability of C programs often relate to UNIX system level services, and differing implementations by vendors. The concept of validation testing of UNIX is only starting to be discussed. With several variants (AT&T System V vs. Berkeley, discussed later), major system services also differ.

The expressive freedom permitted C programmers need not remain unbridled. Use of C's companion, *Lint*, to check for program correctness, new tools to perform intermodule type checking, and enforcement of program style, should contribute to

significantly more portable programs than have been experienced in the past.

It is hard to blame the C language for the woes of software written with dependencies on Berkeley, AT&T, hardware, or other features not present on all C implementations. However, it is relevant to note that tools and methods need to be provided to alleviate the problem.

Similarly, it is hard to blame the language for nonportable embedded systems code (that which executes on targets without traditional operating systems). There have been no known standardization efforts for C run time embedded environments (to compare with the ARTEWG, recently established for the same issues with Ada).

3.1.3 Technology insertion.

During the period of Ada (or C/UNIX) employment on weapons systems, technology will change. Exactly how is unpredictable; however, there are four attributes to examine each language for, with respect to resilience of both applications programs and the language to technology insertion:

1. *Sufficiency*: does each candidate language have features sufficiently abstract to remain meaningful and powerful in the presence of new technology.
2. *Adaptability*: are the semantics of the language features extensible and/or re-interpretable to encompass capabilities and changes caused by technology evolution.
3. *Extensibility*: can the language features be meaningfully and consistently extended to encompass new technology.
4. *Substitutability*: can pieces/subsets of the language easily be replaced with newer technology pieces/subsets when they become obsolescent, without destroying the unity, harmony, or overall balance of the language.

A comparison of Ada and C, more or less for the above issues, follows:

- Ada is a rich and powerful language, incorporating directly real-time

concurrency control features. Questions have repeatedly arisen whether the model chosen by Ada (the *rendezvous model*) is sufficiently flexible to handle the types of distributed processing capable with federations of processors on local area networks. The CAIS, for example, had to define a new form of concurrency basically the same as that provided by UNIX. It is likely that Ada, because it is such a complete language, and complex undertaking, will need to evolve more as technology advances, than will C (because of its more simple provisions).

- C is a compact language, providing primarily data manipulation functions, program control structures, and a simple interprogram interface. In one sense, because the interprogram interface features, concurrency features, and system level features are outside of the language, C might be more resilient to language change than Ada. On the other hand, C developers may well adopt some of Ada's features, such as interprocedure type checking, restrictions on pointer expressive freedom, and the like, in their efforts to "tame" the language and increase its suitability for large projects.

C's implicit reliance on operating system features, for resource control, and features such as concurrency, will hinder transportability of programs as these features change in underlying systems. While C, the language, may be oblivious to such changes, the C programs which result will become as dependent on run time particularities as ISA- dependent programs now are on hardware. This, of course, can be alleviated by standardizing on such features, either through the UNIX interfaces, or through CAIS-like interfaces for run time programs.

3.1.4 DoD versus Voluntary Standards.

Ada standardization has been aggressively pursued by the DoD, with the support of the professional societies, trade associations, and standards associations. C standardization has been only weakly pursued, with an ANSI committee presently working on standardization. Most non-UNIX C compilers

Ada, C, and UNIX

(e.g., MSDOS versions) do not seem to attempt to implement the complete language. Furthermore, though the standards effort is working on the original C language, there are implementations of a C extension to support programming features required for large and reliable programs (C++).

3.1.5 Preserving competition.

There does not seem to be much difference between Ada and C, in regard to this issue. A large number of vendors are implementing both languages. Six to seven vendors have validated Ada compilers; at least that many have competent full C implementations. Initial concerns that Ada's complexity would limit the field of vendors seem to have been ill-founded.

3.2 Current employment of Ada and C

3.2.1 Size and scope of projects

Ada has not yet been employed as a coding language on many projects. The following list shows projects and claimed lines of source coding (project size).

- *CCA*: The AdaPLEX Data Base Management System was done in Ada. About 500,000 source lines.
- *Digital Electronics and Arnold AFB*: Rocket and jet engine test cell sensor control system planner/configurator. 150,000 uncommented (sic) Ada lines.
- *General Electric*: Said to be implementing an advanced Ada work station. Several other Ada projects reported.
- *Informatique Internationale (France)*: Software development effort measurement / monitoring data collection package. 7025 Ada source statements plus 811 comment lines (being expanded at present, target 55% comment lines).
- *Informatique Internationale (France)*: Simulation package for queuing-type network analysis model. 3000 Ada statements in 12,000 Source lines.
- *Intellimac*: reimplemented pre-existing commercial programs into Ada.
- *Intellimac*: AFATDS Data Base Management System, est. size 50,000 source

lines.

- *McDonnell Douglas*: Test case of Ada flown on aircraft.
- *MOOG*: Servo control system, commercial. 4551 Ada statements in 12887 lines of source code and commentary.
- *Northrup*: F20 Simulator translated from Fortran to Ada.
- *Northrup*: F20 Operational Flight Program translated from Jovial to Ada.
- *Raytheon*: Software requirements tracer tool conversion from Fortran. 15,000 source lines.
- *Rockwell Cedar Rapids*: Beech Aircraft Cockpit Display. About 15,000 lines source code in Ada.
- *Telelogic (Sweden)*: Telecommunications autopolling system, for service center; polls PABX'es for diagnostic readouts. About 6500 source lines.
- *Texas Instruments*: Virtual Terminal Package (AIM). 4800 Ada Statements within 22000 lines of source coding.

C is relatively new to be used on military projects. Some examples located are:

- *General Electric*: Said to have several military systems being implemented in C.
- *Litton Data Systems*: LRIP (4 programs for hand held portable military computer). Approximate size 45,000 source lines in C.
- *Litton Data Systems*: LFATDS - US Army Tacfire rewritten in C for Z8000-based "briefcase"-sized portable computer. Approximate size 250,000 source lines.

C has become very popular for commercial projects. A number of widely known projects are said to be either written in C or to be converting major portions to C. These include:

- *UNIX*: Well known portable operating system from AT&T, enhancements from UC Berkeley. K. Thompson (Bell Labs) described an early Bell Labs UNIX kernel as 10,000 lines of C, plus 1000 lines of assembler code. The assembler

coding, he said, included 200 lines for efficiency's sake, and 800 lines for hardware control not (then) possible in C.

The UNIX environment today includes numerous tools, utilities, and products in addition to the operating system kernel. As a completely packaged environment (including the kernel) it has grown from about 100,000 lines of C (in original Bell Labs versions) to a recent count of 435,000 lines in the Berkeley release.

- *Dbase III*: Popular Database for Personal Computers and some UNIX machines.
- *Kermit*: Popular public domain communications program for mainframes, mini's, and micros. Available in portable C version. Includes numerous special provisions to ensure compatibility with the various different UNIX versions (e.g., the communications support features of Berkeley and AT&T UNIXes differ significantly).

4. Operating System Comparisons, Ada and UNIX

Ada is, of course, not an operating system; UNIX is [for disk and terminal based environments]. The reason Ada is compared to C and UNIX is because Ada, being a robust language, carries with the language definition of services which in the past were provided by operating systems outside the definition of languages.

Operating System features found in Ada which also exist in UNIX, as services UNIX provides to C, include interfaces between concurrently running cooperating programs, input/output, resource control (such as memory allocation), and utility functions (such as mathematical routines). Many of the UNIX services which are available to C programs in the UNIX environment do not exist with all C language hosts; for example, on a Personal Computer (e.g., IBM PC or clone) when using Xenix (a UNIX implementation) to compile C programs the programmer has the full resources of UNIX at his disposal, whereas when using MSDOS on the same

hardware, the C programmer is severely restricted with respect to concurrent programming (intrinsic in Ada and UNIX), data file access and naming (intrinsic in UNIX and CAIS, but not Ada or C), and some resource management (intrinsic in UNIX and Ada).

4.1 ALS and UNIX

The original ALS architects claim they were influenced early by UNIX, and disappointed the Army chose VMS instead of UNIX as their host system. (This choice occurred 5 - 6 years ago, before the UNIX "bandwagon" started rolling.) Several presentations of ALS architecture have been made to show UNIX influences. Of course, the present ALS is a unique implementation, and has long since diverged from UNIX in its structure, style, and implementation.

ALS adds to UNIX a library structure for program development, which reflects more modern practices than the UNIX equivalent (Source Code Control System). The ALS library structure, however, is less flexible than the structure proposed by the CAIS.

4.2 CAIS Derivatives and UNIX

The CAIS designers (particularly the KITIA members of the CAIS drafting group) were heavily influenced by UNIX architecture (both in terms of what they liked and what they disliked with UNIX architecture.) CAIS can be said to be a filesystem representing up-to-date features with a UNIX flavor. [I feel strongly that the CAIS provides a model for a new kernel-level filesystem to be implemented inside UNIX.] Marriage of CAIS, within the UNIX kernel, would provide an up to date host for Ada development, and should still be able to preserve compatibility with current UNIX C tools.

4.3 Proliferations of UNIX(es)

There are two UNIX "Camps", and a number of UNIX "clones". There is also strong indication that the incompatibilities between these two are about to soon become resolved. The mainstream UNIX Camps are:

Ada, C, and UNIX

- a. AT&T, presently marketing a version of UNIX known as "System V (five)". AT&T provides four products:
 - i. UNIX source code, with licenses allowing resellers to install and sub-license to third parties,
 - ii. UNIX binary code, with licenses for end users (where AT&T provides the service of compiling, debugging, and maintaining the given system),
 - iii. A System V Interface Definition (known as SVID), provided at no cost as a standard for those wishing to build conforming UNIX-compatible operating systems (clones), and
 - iv. A certification/validation service, provided as a testing service to evaluate an implementors system to ensure that it behaves according to the SVID.
- b. Berkeley, presently providing for a nominal fee source code for enhancements to an early AT&T UNIX version which was known as "version 7". These enhancements include a very popular text editor, a mailing system, and some other utilities which are also now distributed by AT&T, and a kernel, a C-Shell (user interface command language) and several other features which are not distributed by AT&T.

The current Berkeley distribution, known as 4.2, does not support some of the AT&T System V software interfaces. Conversely, many of the AT&T System V compatible implementations lack some of the Berkeley interfaces. Since the incompatibilities are primarily in areas of interactive user terminal (CRT) control, they affect the portability of many tools (e.g., editors, windowing software). Both Berkeley and AT&T representatives have stated that the next release from Berkeley, to be known as 4.3, will support the AT&T System V interfaces. Assuming it occurs, it will be a positive step forward to reduce tool-incompatibility between UNIXes.

Arguments of whether Berkeley, with a modified older version of UNIX, or AT&T, with its current System V, are better, border on religion. Berkeley is legally thwarted from itself upgrading from v.7 to System V by a AT&T licensing technicality which would prevent Berkeley from giving source code away to all takers unless Berkeley itself also purchased source code licenses for the many target machines. (Hopefully release 4.3 will end this incompatibility.)

4.4 Issues comparison for Host Systems

We should compare the use of Ada (without a specific operating system, or simply with the CAIS interfaces) to the use of UNIX for host (disk based and terminal based) applications. This comparison will revolve around the same CODSIA issues previously discussed. One issue we will not treat is security. There is one implementation of a "secure" UNIX kernel (Honeywell's *SCOMP*). There are efforts to ensure that Ada-hosting operating systems (such as those which might include CAIS interfaces) also meet DoD security requirements.

4.4.1 Expensive and Lengthy System Development and Evolution.

UNIX has evolved from a personal working environment to one suitable for larger projects. Ada's suitability for development work tool implementation depends on its host. (There are several Ada implementations on UNIX, and some of these can use UNIX's other tools and facilities. There are others based on other operating systems, such as VMS, which (though not vendor independent) do offer substantial facilities.

CAIS will provide Ada with certain "low-level" tool interfaces which will aid in construction of tools (in the Ada language) which are more independent of their hosting operating system than possible at present.

4.4.1.1 Programming in the large

UNIX, per se, offers several configuration management systems, electronic communications systems, program dependency systems, and similar tools needed in large implementation staff environments.

Ada is presently dependent on (and specific to) the underlying host system when used to

implement tools for these environments.

4.4.1.2 Enforcing software quality

As tool bases mature, both Ada (on any host) and UNIX should develop rich bases of tools for enforcement of software product quality.

4.4.2 High Cost and Risk for Nontransportable Software.

UNIX hosted software is transportable to other UNIX equipped hosts. Ada software for the host environment is yet to be quantified in its measure of portability; this depends to a large degree whether a specific program uses strictly the internal features of the Ada language, the additional proposed standard features provided by the CAIS, or additionally, the features specific to a given host operating system.

Some UNIX software has been written in a manner to make it version specific. Berkeley and AT&T versions have diverged in their forms of interactive terminal control, and some kernel features, in recent years. See "Proliferation of UNIX(es)". This can lead to nontransportability when not specifically addressed at the time of software tool implementation.

4.4.3 Technology insertion.

The UNIX operating system has been a prime example of a program which has successfully undergone technology insertion.

- The kernel has evolved significantly, from a single user minicomputer to Berkeley and AT&T multiuser large minicomputer implementations.
- The user utilities have changed significantly in nature. For example, editors have evolved from line oriented editors to syntax-directed and documentation-oriented products. Command interpreters have evolved to have history and advanced programming features. The communications utilities provide networking capabilities and Arpanet functions.
- The management functions now include several sources of configuration management and project management systems. Several vendors compete with

proprietary products for the DoD supportive marketplace.

Ada (alone) provides no equivalent operating system facilities, so is not comparable to UNIX with respect to this issue.

Both UNIX and CAIS provide low-level operating system features. They are very specific in terms of the file system (node) and process models, and such are potentially vulnerable to changes in data storage technology or distributed network software concurrency solutions. There is little experience to use in evaluating the resilience of these low level features to technology insertion.

4.4.4 Preserving competition.

The UNIX operating system is available from a large number of independent vendors. Implementations fall into two categories, those which are derived from source code licensed from AT&T (e.g., Berkeley and Xenix), and those which are implemented independently with UNIX-compatible interfaces (such as Coherent). AT&T plans to offer a certification/validation service to test interface-level compatibility.

Ada provides no equivalent operating system facilities, so is not comparable to UNIX with respect to this issue.

4.5 Issues comparison for Target Systems

The lack of definition for Ada run-time environment interfaces hampers a reasonable discussion of these topics. Ada can only be considered as a language for systems which require simple (textual) input/output and system-specific other services.

The lack of definition for UNIX-compatible interfaces for other than disk-based terminal-supported systems hampers its discussion for target systems. The use of UNIX compatible interfaces for embedded applications (such as torpedoes or smart land-mines), is as fraught with uncertainty as a similar discussion would be for Ada.

4.5.1 Expensive and Lengthy System Development and Evolution.

See preceding discussion.

4.5.2 High Cost and Risk for Nontransportable Software.

See preceding discussion.

4.5.3 Technology insertion.

See preceding discussion.

4.5.4 Preserving competition.

See preceding discussion.

4.6 Positive features of UNIX

4.6.1 Robustness and completeness of tool set

See 1.3.3

4.6.2 User extensibility

See 1.3.3

5. Recommendations for applying Ada and C†

Until a set of stable and mature Ada compilers is available, and until academia uses Ada as the basic language to train in, there will be difficulties in applying Ada. (The compiler situation, however, has become dramatically better in the last half year. It should be solved for good within the next two years.)

Until C is augmented with facilities and tools to enforce style, provide strong type checking, and support programming-in-the-large (as claimed to be provided by C++), it will remain difficult to complete large scale military projects in C. Worse yet, post development support for such large projects will demand "UNIX Guru" intellectual talent to be successful. The quantity of those sorts of people who will work for long-term projects, or the military, is severely limited.

† These represent the author's current personal opinion. They are likely to change as the many factors cited change and as the products mature.

Thus there appears to be discrete tradeoff criteria to use in comparing C and Ada for military systems application.

5.1 Occasions to use Ada

Ada usage is indicated for large scale developments, to be procured on traditional long-lead schedules. With the rapid maturation of Ada compilers, current problems with compilers should be alleviated early in the long traditional procurement cycles. With the long cycles, personnel training can be accommodated.

As mentioned earlier, Ada's features to support large developments, and to support reliability and maintainability, are important to military operational software.

Ada's usage is also indicated, in the short term, for developments where extreme reliability of the resultant software is important.

Ada should be emphasized for research type projects, especially for the types of IR&D efforts which will lead to products and programs within the time frame for Ada compilers to mature.

5.1.1 Availability to schools

Many attribute the UNIX bandwagon to the "university give-away" AT&T practiced in recent years. To significantly increase the quantity of Ada training, a "compiler give-away" (of a quality product) would be recommendable.

5.2 Occasions to use C

C (with UNIX resource control and system level services) would be recommendable for several classes of applications:

- a. Host system tools, where the host already uses UNIX.
- b. Target systems which need complete disk-based operating systems (such as hard-site command and control systems); with the caveat that the size of individual programs be small to medium, but not large.
- c. Target systems which are embedded (no target operating system) and used no target-level features from UNIX (c.g., a smart land-mine with minimal software/peripheral needs), for which

Ada, C, and UNIX

transportability and reusability concerns are less important than speed-of-deployment.

- d. Target systems to be implemented rapidly in the short term, where use of UNIX and C would reduce risk, even at the expense of post development support, or UNIX change risk later on. A crash-effort weapons system might fall into this category. The large pool of C capable talent and the robustness of the current UNIX hosts are strong points in favor of such needs.

Usage of C will be more recommendable for a larger class of military systems when the C++ dialect becomes more widely available and supported.

5.2.1 Controlling intellectualism in C style

This should be possible through the use of software engineering methodologies and management which limits the practice.

5.2.2 Providing facilities for programming-in-the-large

This should be possible by defining tools and supplemental language features to support inter-module type checking, and other features described earlier. Newer C dialects (with tools to restrict excessive freedom to avoid safe programming) will also be helpful.

5.2.3 Determining which UNIX version to use (BSD vs AT&T)

Many tools are dependent on either BSD or AT&T interfaces. While there is a high probability that BSD eventually will adopt AT&T interfaces, in the near term users will have to contend with incompatibilities.

Current Ada products which are UNIX hosted (Verdix's Ada compiler) are BSD-dependent. DEC has also adopted the BSD interfaces for its versions of UNIX (Ultrix). However, many defense contractors have adopted AT&T versions (General Electric Corp., IBM Corp., Litton Industries). Furthermore, there is a trend towards providing kernels which are AT&T derived (for more modern kernel and driver implementations) yet exhibit both styles of interfaces.

Microsoft (with Xenix) has been successful in that direction and there are indications that the next Berkeley release will do likewise.

Some UNIX-hosted tools have attempted to maintain compatibility with both AT&T and BSD UNIX. C-Kermit is a notable example. The effort which went into inserting that compatibility has practically exceeded the original design and coding effort. To guarantee that future tools are compatible with both is costly, and requires on-site "UNIX Guru" talent. To restrict oneself to the common subset of both is not possible in the area of interactive terminal and screen support.

This dilemma might be resolved in the very near term. Pressure on Berkeley might get them to implement Xenix-like support of both forms of terminal driver interfaces (that is the most incompatible area of conflict). Pressure on AT&T might be similarly rewarded.

6. Recommendations for applying CAIS and UNIX

Recommendations in the operating system and software environment domain need to consider the category of use: host or target. The next two sections provide some recommendations.

6.1 Host Environments

A clear trend in industry is to favor host environments which are supported by multiple vendors. While many of the current validated Ada products run on proprietary (single-vendor) operating systems (such as VMS or AOS), and while many of the proprietary operating systems have significant technical merits, their adoption would leave the DoD still in the position of being dependent on one vendor.

There seems to be evidence that it is difficult to write host system utilities and compiling systems which are truly operating system independent. Two attempts to produce such tools (where portability to entirely different operating systems was expected) have not met success:

- Examination of the Army ALS, an environment said to be designed to be rehosted, shows that it is heavily dependent on host operating system features, and indeed even on a unique language (Bliss) only provided by one vendor.
- The Kermit project attempted to produce a host program in a widely available language (C with UNIX interfaces). A significant portion of the program is nonportable (to non-UNIX C hosts) to other operating systems (even where they do support C).

UNIX offers a way to write host tools and software, where the *same* operating system is available from numerous vendors on many classes of hardware. For the near term, it offers a solution to the problem of host tool portability.

6.1.1 CAIS Interfaces

The CAIS promises to standardize on the lower level host service interfaces. CAIS should bring to Ada a similar degree of portability as UNIX affords to C. At present, however, draft versions of CAIS are subject to technical controversy and need to be prototyped and need to have time to mature.

6.1.2 Berkeley versus AT&T Interfaces

See 5.2.3.

6.2 Run-Time Environments

The extreme lack of standards and directions in run time environment interfaces makes this a Pandora's Box of confusing choices.

6.2.1 Preserving kernel-level UNIX compatibility

Where UNIX is used as a host development system, it would seem beneficial to use UNIX-styled interfaces in target environment kernels. Some technical controversy exists as to the optimality of such an approach. However, it does make the job of software testing, on hosts, and installation, to targets, go smoother. No standards or directions have been established in this area.

6.2.2 Using Actual UNIX kernels For Run Time Environments

UNIX kernels (and operating systems) can be used as is for some target systems, particularly to fill the need to field computers with disks (magnetic or optical) and large scratch memory (disk, bubble, or ram), and interactive terminals for operator control. Progress with multilevel security implementations of UNIX is encouraging, particularly for such environments.

For strictly embedded systems, which do not have host-like peripheral complements, while some UNIX code might be applicable, most observers doubt that the actual UNIX kernel will apply. Research and development of limited-resource run-time executives will need to be conducted to provide data for decisions on this area.

6.2.3 Adding concurrency models to UNIX-style environments

UNIX kernels support a form of concurrency designed for centralized computers with time sharing users. Current directions in both host and target systems is to use federations of processors, either as host workstations, or in targets, for functional decomposition of a system. Approaches to using the UNIX process model in these environments need to be evaluated for military application. There remains a likelihood that new forms of loosely-coupled multiprocessing communication will eventually result in changes to the underlying models in systems like UNIX.

6.2.4 CAIS interfaces for Run Time usage

The use of CAIS, as a set of low level kernel interfaces, in target systems is subject to almost the same arguments as is the use of low level UNIX interfaces. These issues will not be reevaluated here.

The Ada Run Time Environments Working Group is evaluating low level interface needs of Ada programs. Their progress will have to be closely monitored to assess various alternatives which might result from their efforts.

Revision Control Tools and the Ada Program Library

Dick Schefstrom
TeleLOGIC AB,
Regnbagsallen 4
S-951 87 Lulea, Sweden

ABSTRACT

Undertaking the transition to Ada today will to most of us not mean using a full APSE, [DOD80, BUX80], tailored for this purpose. Instead, this transition must be made by using an existing environment, stepwisely integrating its proven tools and concepts in the realm of Ada.

This paper describes one effort taking this approach, where the Unix revision control tool RCS, [TIC85], is integrated with an Ada program library, using a UNIX style library interface.

1. Introduction

Already at the beginning of the Ada project, the importance of the programming environment was pointed out, and a number of ambitious projects and investigations were initiated, [DOI81, THA82, INT82, KIT84]. However, today few APSE's of this spirit are available for production use, and even if they were, existing environments could be chosen for a number of reasons: since they are proven and people know how to use them, something that should not be ignored when planning for an introduction of Ada,

It can therefore be argued, [ICH85], that we, at least for the moment, should take a bottom-up approach to tools and environments, and spend the efforts on careful integration between existing operating systems and the Ada-world. To make it convenient to use Ada in an existing environments, the interfacing to the Ada utilities should be carried out in the spirit of the host environment, and its tools should be reused in the Ada context whenever possible.

The arcs program, described here, is an example of an attempt to do this for the UNIX programming environment.

2. Revision Control in an Ada Environment

Revision and configuration management was given a very important role in the requirements of the APSE, [DOD80], and has consequently been given much thought in different APSE-efforts. Still there are tools, such as RCS, widely in use for such purposes today, which could be reused very neatly in the context of Ada if suitable measures are taken.

The *Ada program library* is very important to notice in this case, since it will always be there, (in one form or another): describing the user's world of Ada units, and their relationships to each other. It is likely that these program libraries will be a main focus of interest, just like file directories are today. Examples of such program libraries are [DDC84], [DEC84], [NAR85]. But, in parallell, users will still want to keep track of successive revisions of source codes, based on files, using tools like RCS, [TIC85], SCCS, [ROC75], or CMS, [DEC82].

If these two tools, the program library and the revision control tool, don't know about each other, we will experience a number of questions...

- * Which revisions/versions of a program unit are actually present in the program library?
- * What is the name of a unit: the Ada unit name as present in the library or the filename as when dealing with RCS?
- * For a particular unit in the program library: where is its "RCS-file"? Is the unit under revision control or not? Has its source code been checked in since it was last modified?

- * How do we interpret concepts defined in the context of program libraries when using RCS, and vice versa. For example, can we check in a configuration? Check it out?

The problem is really that we will have two uncoordinated representations of the same thing: the program; leading to unwanted redundancy and confusion.

Typical revision control tools, such as RCS, are good at keeping successive revisions of a single file together. However, they are not that good at describing system structure and relationships between different units. On the other hand, an Ada program library always explicitly represent relationships between separately compiled units and catch more of the semantics about what is being stored, but usually has no tracking of successive revisions. So, by combining the two we can get the best out of both tools.

3. The Program Library

The work described here is the initial results of an attempt to integrate the UNIX revision control tool RCS with the TeleSoft Ada program Library, [NAR85a], [NAR85b], [SCH85a], [SCH85b]. To give the necessary background, a short discussion of the properties of the program library is appropriate. For a more detailed investigation see the references given above.

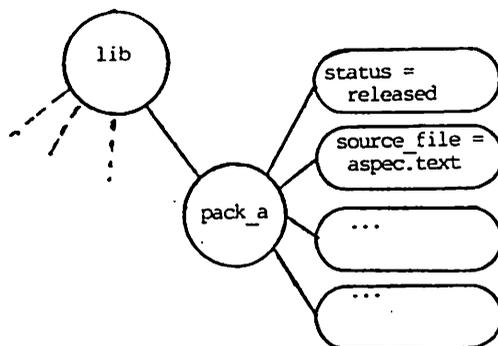
The full typechecking between separately compiled units, as required by the Ada language, implies a close interaction between the compiler and the environment. It leads to the introduction of a file, or "database", holding the descriptions of which units have been compiled earlier, what services they provide, and how they relate to each other. Although each compilation must be performed in the context of a program library, the Ada language definition does not prescribe its functioning in any detail, but regard it as part of the environment. At this point we also get implications for "programming-in-the-large" and software engineering, and interesting criticism has been presented on this subject by some researchers, [BUX85].

Since a lot of information about the software is automatically collected in the program library, in a form suitable for mechanic manipulation, there is a big potential of building new tools utilizing that information. Among the more obvious is an alternative, or substitute, for Unix/Make, [FEL79], system structure, or cross-reference, -generators, and configuration and version management tools. In another paper, [NAR85a], the author argues that the "environment database" envisioned by the original APSE requirements, is best approached as an out-growth of the program library.

The implementation of an Ada program library we have used here, the TeleSoft Ada program library, is composed out of a freely constructable sequence of sublibraries, together constituting a full library. Each sublibrary implements a datamodel describing a directed graph where the edges are typed and the nodes may have arbitrary attributes. Besides being nodes in a typed directed graph, the nodes are, within a sublibrary, arranged in a hierarchy.

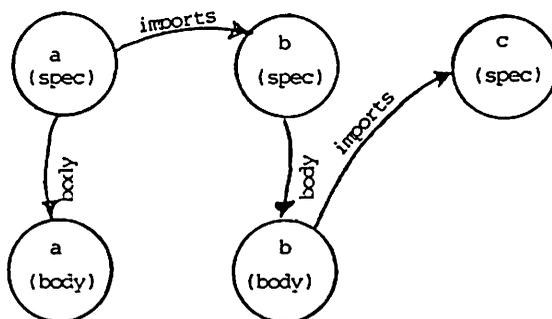
The system is open-ended in the sense that new nodes, edges, types of edges, and attributes, may be introduced dynamically. Further, procedures to do this are available as Ada package specifications, making it possible for different organizations to tailor their own bridges between the world of Ada and their existing environments. We believe that such open-endedness is very important for a smooth transition to Ada.

As an example, a package specification for a package "pack_a" is represented by a node with the same name. To distinguish the specification from its body, (which in Ada has the same name, but is a different compilation unit), it is the child of a special node called "lib"...



To fully name a node, one must specify its path, so the unit above would be called `"/lib/pack_a"`. The rectangular boxes attached to the node represent attributes, which may arbitrarily be attached to any node. The text inside the attribute symbols always has the format `<attr_name>=<attr_value>`.

A small system can then be illustrated...



...where the nodes represent compilation units, and the arcs represent relationships between them. The different labels of the arcs could be viewed as a typing, or categorization, of the corresponding relationships. Different tools, (most frequently the compiler), introduce different types of edges between nodes.

It is a main point that different programming-in-the-large concepts, like consistency levels, configurations, and releases, can be defined in terms of the datamodel used, [NAR85a]: and this is explored in the integration with RCS which is to be described in the following.

Edges may also span between sublibraries, so that what we get could be viewed as a "layered graph". Different layers, (sublibraries), could then be used to hold different subsystems, versions, or act as a mean for sharing of objects between many programmers, [NAR85a]. However, in the examples used in the following we do not explore these properties: to simplify, and since the ideas presented should be clear anyway.

4. Basic Concepts of Arcs

A program library may contain lots of attributes and many different types of edges between nodes. At any single moment, a user is probably not interested in all of that, but rather a subset of the information important for the task at hand: there is a need for an ability to define *views*.

In *arcs*, the basic library listing command, ("l"), takes a list of attribute names, (each signalled by an initial flip, "'"), a list of edge type-names, (each signalled by an initial colon, ":"), and a list of node, (or unit-), names. The given attributes are listed, if present, for each unit reachable by starting at a given unit following edges of any given type, as far as possible. Such a *reachable graph* may be given to each command of *arcs*, with the effect of repeating the command for each node in the resulting graph.

The command...

```
l 'time_stamp 'unit_kind :body /lib/
```

...would therefore list the values of the attributes "time_stamp" and "unit_kind" for all library units, (the ending "/" generates a list of all child units), and for each of these, all units reachable along edges of kind "body". This happens in this case to result in graphs of exactly two units, each containing a specification and its body.

Another example: the command...

```
l 'state :imports :body :parent :subunit /lib/main
```

...would list all units needed for the execution of "/lib/main", and for each such unit, the value of its "state" attribute. (Each listing also points out units being out-of-date, or being missing completely, so that the command can also work as a consistency check).

This facility acts as a basic mechanism for specifying views: the user has complete freedom to operate on, or see, arbitrary subset of the program library contents. To make it convenient to use such views, the *alias* facility may be used: instead of explicitly giving every attribute and edge type-name, the user could issue the command...

```
alias my_view 'state :imports :body :parent :subunit
```

...reducing the previous command to...

```
l my_view /lib/main
```

Arbitrary many aliases may be defined, and each may represent a different view of the program library. *Arcs* also has a large number of "variables" which can be set to customize its behaviour. If we don't want to specify the prefix "/lib/" each time, (to tell that it is the spec of main that is meant), we could set a variable...

```
set default_prefix = /lib/
```

...reducing the command to...

```
l my_view main
```

We could simplify even further by doing...

```
set default_unit = /lib/main
```

...reducing our example command to...

```
l my_view
```

If we'd like to inspect, or edit, the source texts corresponding to some compilation units, the commands "see" and "edit" can be used, respectively. The tools to be used by these commands is specified by setting variables, and the setting...

```
set editor = /usr/local/bin/emacs
set seer  = /usr/ucb/more
```

...would cause "emacs" to be used by the "edit" command, and "more" to be used by "see". The command...

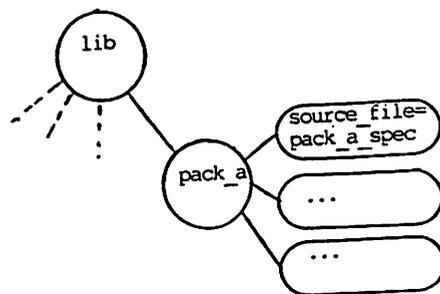
```
edit :body /lib/main
```

...would cause emacs to start up with two windows: one for the spec and one for the body.

Arcs contains lots of commands, and for each command there may be many options and variables affecting its detailed functioning. Presenting them all here is not possible, although some of the spirit should be clear from the examples above. Arbitrary "arcs"-commands, such as alias definitions, settings of variables, and opening of a default library, may be put in a ".arcsrc" file, so that these commands are automatically performed at each invocation of "arcs".

5. Coordinating with the Revision Control Tool

When we successfully compile a unit, a node with the same name as that unit, and some associated attributes, is created, (or updated), in the program library. As an example, if we have the source of a package spec called "pack_a" located in a file "pack_a_spec", the following would be created...



That is, we now have a "lib"-node called "pack_a", which has an attribute "source_file" describing which file the source is located in. Typically, the user is not satisfied with "pack_a" just because it was compiled without errors, so he will repeatedly edit and compile the file "pack_a_spec". However, at some point in time he feels that the current state of "pack_a" spec is something of a milestone, and should be put under more formal control.

So, the units in a program library are of two kinds...

- (1) Milestones, which are revisions considered important and complete, and whose source has been stored using a tool like RCS.

- (2) Editions, which are in a state of development and whose source is not saved: it is just changed to arrive at a new edition or milestone.

Working under UNIX, RCS, (or SCCS), would now be used, "checking in" the file "pack_a_spec"...

```
> ci pack_a_spec
```

...resulting in a new revision, stored in an "rcs-file" called "pack_a_spec,v". However, this is normally not reflected in the program library.

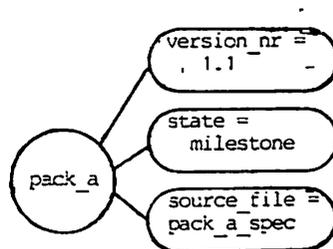
If the program library is to be a central information storage, activities like checking in and checking out must be made to affect it, in a more or less automatic way. To achieve this all the commands of RCS can be given from within "arcs", which let us talk about software in terms of the compilation units and configurations of the program library, and also makes it possible to reflect in the program library what revision control actions have been performed. These commands take the usual rcs-parameter flags as a subset, (prefixed by "-"), but does also take "arcs"-specific flags, (prefixed by "+").

A check in of the current version of pack_a's spec would now look like...

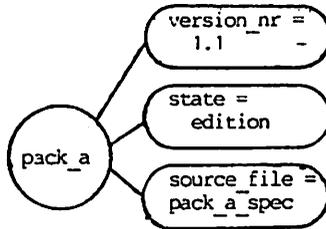
```
ci /lib/pack_a
```

The program then looks in the current program library for a unit "/lib/pack_a", takes the file described by the "source_file" attribute and checks it in to an RCS-file.

To register that what is now in the library is a unit under revision control, a "version_nr" attribute is associated with pack_a, resulting in the following in the program library...



We can now see from the attribute "version_nr" that the unit is under revision control and what version number it has. The "state" attribute tells us that it is the "real 1.1", and not an edition of it. A subsequent compilation of pack_a changes the state to "edition"...



As a way of increasing reliability, the check-in program also checks that the source file has not been changed since it was compiled into the program library, and issues a warning if necessary.

So, by putting some other software around the check-in program we can, in the program library, maintain an explicit distinction between the different states of compilation units, and get a more homogeneous environment with the program library as a main entry into different services.

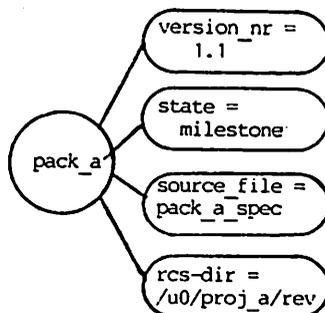
As discussed earlier, we allow different versions of the same unit to coexist in the environment if they belong to different sublibraries. This makes it possible to make full use of both versions without checking in or out, and without any need for recompilations when using another version. However, although these versions can be treated as independent units, there are good reasons for letting them share a single "rcs-file"...

- (1) We get a description of how the different versions relate to each other historically.
- (2) The sources can be stored space-efficiently.

The question that now arises is: how do we locate that common "rcs-file"? As a general rule, a system should not force a user to any particular methodology but should provide convenient defaults. In the case of rcs-files this implies that the user is allowed to provide the name of the directory where the rcs-file is to be located. A typical check-in would then look like...

```
ci +d/u0/proj_a/rev /lib/pack_a
```

That is, the user tells the system that the source of the library unit pack_a is to be checked in at the directory /u0/proj_a/rev. This is also registered in the program library, resulting in the following...



Once a unit has been assigned a rcs-directory, the check-in program looks in the program library for the name of the rcs-file during subsequent check-in's, providing a natural default. So, the user has to decide on the location of the rcs-file only at the creation of the first revision of a unit.

Some information in the program library is now in the form of file names, referring to files in the host file system. To enable the tools to act "intelligently", one must agree on an interpretation of these file names.

Filenames can be of two kinds, absolute or relative. Absolute filenames are no problem: they can be interpreted in only one way, but may on the other hand sometimes generate problems since moving of files becomes harder. Relative filenames make it easier to move files between directories, but require some rule for translating a relative filename into an absolute.

The usual way of using the current directory does not always work in the context of sublibraries, since a sublibrary may be on the same directory during a long time, while the users change their current directories while still using that same sublibrary. To make up for this, the library tools treat relative filenames as relative to the directory of the sublibrary where the names are stored.

Using this idea, the rcs-file is located on the same directory as the sublibrary of the compilation unit, if not explicitly provided.

To automatically repeat the check in operation for a whole set of units, "reachable graphs" can be used. For example, a complete executable Ada program is defined by starting at the "main" unit and following arcs of type "imports", "body", "parent", and "subunit". Using this in the check in command...

```
ci :imports :body :parent :subunit /lib/main
```

...would repeat the check in for each unit in this reachable graph which "need to be checked in". A unit is said to need to be checked in if it has been compiled since the last check in, (state = edition).

Of course, the long list of relation names may be replaced by an alias...

```
alias :p :imports :body :parent :subunit
```

...reducing the command to

```
ci :p /lib/main
```

If different flags need to be given for each unit in a reachable graph, the "+a" flag will cause "arcs " prompt the user at each unit, giving an opportunity to supply different parameters, or to skip the node altogether.

The check-out command works in a similar way. It can be used for single units by giving the name of a compilation unit as in the command...

```
co -r2.1 /sec/pack_a
```

...which would try to localize the rcs-file of the body of "pack_a" and check out revision 2.1 into the current directory. (A "+d" parameter would have caused check out to the directory of the sublibrary, analogous to check in). As in the case with check-in, configurations can be dealt with by giving a reachable graph:

```
co :p /lib/main
```

...would check out, for each unit in the given reachable graph, the latest version on the trunk.

A reachable graph can be thought of as a configuration, where each unit is of a particular version, (as described by the "version_nr" attribute). To be able to, for each unit in a reachable graph, check out the source corresponding to that particular version, the special sign '&' may be used instead of revision numbers in the rcs-flags. Arcs will then, for each unit, substitute the '&' with the value of that units "version_nr" attribute. The command...

```
co -l& :p /lib/main
```

...would therefore check out these sources to the current directory. As previous, the "+a" flag will give the user a chance to provide different flags for each unit. As another example, the command...

```
co -l& +d~ 2/lib/ 2/sec/
```

...would check out, to the directory described by the variable "tilde", (which defines what is substituted for '~'), the sources corresponding to current revisions of all library and secondary units in the second sublibrary in the library list. Of course, instead of "-l&", any of the rcs ways of identifying a revision, (for example symbolic), can be used.

6. Automatic recompilation

As is shown in [NAR85a], the order of the necessary recompilations after a change are very easily computed using the high-level graph-manipulation operations provided by the TeleSoft Ada program library.

However, if the compiler only can compile whole files, the compilation of a unit must be done by compiling the whole file it is located in, possibly generating side-effects because of other units in that same file. It can be argued that the user should not put several units in the same file, if he expects things to work smoothly, but this is in reality a too hard restriction: although the "file" is not suitable as the unit of compilation, it may be a suitable unit of editing, since there may still be users out there having small screens and single window editors.

Instead we made the compiler take not only a file and a library as a parameter, but also a list of unit-names. In the following example...

```
ada -u/sec/pack_a pack_a.text
```

...both the spec and body of "pack_a" are located in the file "pack_a.text", but because of the "-u" parameter, all units in the given file, except the body of "pack_a", are skipped by the compiler.

At any moment the program library contains zero or more units from the same "revision-family", (units whose source is stored on the same rcs-file), and these "representants" act as entries into that revision-family. If some member of a revision family is present in a library, we say that the family is "represented" in that library. If one wants to compile into the library another member of a represented revision family, it can be done by using the "ada" command...

```
ada -r2.3 /sec/database
```

...which, in the example above, would check out revision 2.3 of "/sec/database" and try to compile it into the library. Sometimes this doesn't work, however, since the desired revision maybe "withs" other versions, or other units, than are present in the library. Where the "ada" command fails, the "tli" command can be used instead...

```
tli -r2.3 /sec/database
```

...resulting not in a full compilation, but making the relationships of "2.3" to its environment visible in the library. This gives a convenient way of observing the separate compilation structure of "2.3", thereby pointing out what kind of environment is needed to perform a full compilation. Finally, the "make" command is used to automatically carry out the necessary recompilations after a change in one or more units of a system. As an example, the command...

```
make +c /sec/database
```

...would recompile /sec/database, (if needed), and before that, all the units needed before /sec/database could be compiled. Instead of giving the "+c" flag, which tells make to do the necessary work before the given unit can be *compiled*. We could also have given the "+e" flag, causing all the compilations necessary to make the unit *executable*.

7. Conclusion

During the beginning of the Ada project, there was a lot of emphasis put on the "APSE", the advanced programming support environment, which should accompany the pure language implementation. This APSE was often thought of as something to a large extent built from scratch, using new paradigms, and including new advanced tools.

What is described in this paper is an example of a more incremental/evolutionary approach, where an existing, proven successful, tool, is taken advantage of, stepwisely including it into an Ada programming environment building on the UNIX system.

8. Acknowledgements

Mikael Beckman and Johnny Widen at the TeleLOGIC Lulea office made both conceptual contributions and carried out large parts of the implementation.

9. References

[BUX85]

"Ada: The Language and its Environment", J.Buxton & D.A.Fischer, in Technology and Science of Informatics - Ada Special, North Oxford Academic, 1985.

[DDC84]

"DDC Ada Compiler System Separate Compilation Handler, Functional Specification", Dansk Datamatik Center, Nov 1984.

[DEC82]

"CMS/MMS: Code/Module Management System Manual", Digital Equipment Corporation, 1982.

[DEC84]

"VAX Ada Technical Summary", Digital Equipment Corporation, Nov 1984.

[DOI81]

"United Kingdom Ada Study: Final Technical Report", Department of Industry, London 1981.

[FEL79]

"Make - A Program for Maintaining Computer Programs", S.Feldman, Software Practice & Experience, April 1979.

[ICH85]

"Ada: Ready For Application", an interview with J.Ichbiah, published in Technology and Science of Informatics, Ada Special, North Oxford Academic 1985.

[INT82]

"Computer Program Development Specification for Ada Integrated Environment", Intermetrics, Inc. Cambridge, MA, 1982.

[KIT84]

Common Apse Interface Set, (CAIS)", proposed military standard, v1.4, Ada Joint Program Office, 1984.

[NAR85a]

"Extending the Scope of the Program Library", K-H Narfelt & D. Schefstrom, in "Ada in Use: Proceedings of the 1985 International Ada Conference", Cambridge University Press 1985.

[NAR85b]

"System Development Environments", K-H Narfelt, Licentiate Thesis, Department of Computer Science, Univ of Lulea, 951 87 Lulea, Sweden. 1985.

[ROC75]

"The Source Code Control System", Marc J. Rochkind, IEEE Transactions on Software Engineering, Nov 1975.

[SCH85a]

"Possibilities of Layered Graphs", D.Schefstrom, Memorandum, Dep of Computer Science, University of Lulea, 951 87 Lulea, Sweden. 1985.

[SCH85b]

On Data Organization in Programming Environments", D.Schefstrom, Licentiate Thesis, Dep of Computer Science, University of Lulea, 951 87 Lulea, Sweden. 1985.

[THA82]

"The KAPSE for the Ada Language System", R.M.Thall, ACM/AdaTEC Conference 1982.

[TIC85]

"RCS - A System for Version Control", W.F. Tichy, Software - Practice & Experience, Vol 15(7), July 1985.

Managing Separate Compilation in AT&T's UNIXTM Ada[®] System

G. W. Elsesser

M. S. Safran

T. Tieger

AT&T Information Systems
Summit, New Jersey 07901

AT&T's UNIX Ada System is a rehostable and retargetable compiler, library manager, and runtime system for the Ada programming language. It is currently hosted and targeted on the AT&T 3B2-400 running UNIX System V Release 3. The system's Ada Library Manager (ALM) enforces the Ada LRM[1] compilation order rules, supports the use of compilation units from multiple libraries, and provides an interface for building and maintaining Ada program libraries. It also provides the basis for other closely related tools including a program build facility and a set of administrative tools. The UNIX operating system serves as our Kernel Ada Programming Support Environment (KAPSE). The UNIX Ada System exploits existing UNIX System features to provide the environment facilities imposed by the Ada language requirements. The ALM tools, operating in the UNIX Ada environment, can be viewed as a Minimal Ada Programming Support Environment (MAPSE).

1. Introduction

When presented with the task of designing a library manager, we were not quite sure about what its scope should be. We knew that it needed to manage separate compilation of Ada programs and that it would have to define an implementation of one or more program libraries as specified by the LRM. But how deeply should we delve into environment issues? How comprehensive a database should we provide? Should we rely on using UNIX pathnames for identifying libraries? How much of the UNIX operating system should be hidden from the Ada programmer? Should any protection at all be provided? Were the existing UNIX configuration management tools sufficient or desirable for Ada? (Do we or do we not use *make*[2])? What specific administrative tools should we provide? In short, how much of an APSE did we actually need to develop?

This paper describes the existing Ada Library Manager (ALM) implementation — its database and its software. In the process of presenting its features, we highlight some of the design choices that were made and suggest areas for future work.

2. Overview of the UNIX Ada System

When a system administrator installs the UNIX System, the ALM database is configured. One or more *library indexes* are produced. A library index manages a family of *Ada program libraries*. Each library is a UNIX directory containing a special file, called the DICTIONARY, that keeps track of information about compilation units. Ada source files, symbol tables, object files, and SCCS files typically reside within this directory or among its subdirectories. Ada programs are compiled inside a particular library — the sole place where compilation information may be updated. Compilation units may be imported from other libraries within the same index. Accessible *foreign* libraries are those directories for which the user has read permission.

TM UNIX is a trademark of AT&T Bell Laboratories

* Ada is a registered trademark of the U. S. Government—Ada Joint Program Office

The ALM software is depicted in a block diagram in Figure 1. As can be seen, the choice was made to use a small set of UNIX commands. The *ada* command, like the familiar *cc* command, compiles and links source files into an executable. Compilation completes successfully if and only if all the units on which the specified program depends have already been compiled and stored in the library. Command-line options not recognized by *ada* are passed directly to *adalink* and on to *ld* if necessary. (Care was taken to use familiar option names when possible.) The *adamake* command provides a mechanism for program building. Like the UNIX *make* command, it determines which units need to be recompiled in order to produce a desired target. Like various *makefile* generators, *adamake* examines source files to determine predecessor information. It is able to provide the functionality of both *make* and a *makefile* generator by using a phase of the compiler to scan the source. Both the *ada* and *adamake* commands use a common subset of ALM primitives. The administrative facility is provided by the *alm* command. Since it was believed that the optimum set of functions for this utility would best be determined by users, an early decision was made to design this command so that it could grow gracefully.

We now discuss each of these in greater detail.

3. Program Libraries and Separate Compilation

The LRM[1] specifies that "The compilation units of a program are said to belong to a program library... The possible existence of different program libraries ... are concerns of the programming environment." Should an installation have one or more libraries? If there are many libraries, how do we provide shared access to popular packages? Did we have to resort to copying foreign units? We decided that it was totally impractical to require that all Ada compilations in an installation take place within one library. The UNIX file system's permissions mechanism seemed to be a natural model for providing shared access. This section describes the structure of our libraries and the strategies used to provide consistent access to *foreign* units.

3.1 The ALM Database

With the decision to allow multiple libraries came the question of how to identify particular ones. When compilation units import other units, how can one tell at a glance where they came from? We decided to create a pragma called LIBRARY to pair an import with its location. The UNIX file naming facility offers a way of uniquely identifying libraries. However, embedding hard coded pathnames in source programs did not seem desirable. For example, a user might want to include a package called MATH_PAK from a mathematics library. It would be convenient if that library could be called MATH_LIB and if it wouldn't matter where exactly in the file system MATH_LIB was located. We therefore came up with the notion of a *library alias*. The mapping between an alias and the UNIX pathname, as well as with a small, fixed size internal identifier, is implemented in the *library index*.

The ALM is essentially a database for information about compilation units. Within a particular library, the database is represented in one binary file (called the DICTIONARY for historical reasons). Not using an ascii representation optimizes performance as well as space; startup time is shortened, and references to names can be stored as pointers. Fixed size hash tables are provided for quick access to information about compilation units and source files, to keep track of names, and to avoid storing duplicate copies. The remainder of the DICTIONARY is an extensible heap where lists and strings are stored. The heap is managed by a first fit boundary tag algorithm as described by Knuth[3].

We decided to have a uniform access method for referring to DICTIONARY objects from both the current and imported libraries. Having the current library be memory resident was entirely feasible because of UNIX System V Release 3's paging mechanism. Relying on complete dictionaries for all the imports to also be in memory did not seem reasonable. So we decided to view a DICTIONARY's address space as an abstract data type. Virtual

addresses are provided for all objects. When a higher level routine requests access, it provides an internal library identifier and virtual address to lower level primitives. These primitives hide any representation differences between objects in the home library and those in foreign libraries. They decide whether to return a pointer to a memory location or to use *lseek* to find and fetch an object from a foreign library's **DICTIONARY**.

The data stored in dictionaries can be viewed as a dependence graph. Various traversals provide the following services for the ada compiler and linker:

- A check for whether the Ada compilation order constraints are satisfied for a given compilation unit.
- A topologically ordered list of the names and locations of the symbol tables of all units (in the current and/or foreign libraries) that are needed to complete the compilation of the current unit.
- Cycle detection, based on the three color marking algorithm used by the Ada Breadboard linker[4].
- A list of compilation units in an acceptable elaboration order.

3.2 Concurrency Control

Each read or update to the ALM database is considered a transaction. Like any other database system in a multi-user environment, the ALM must regulate database access to guarantee that transactions are carried out in a legitimate order. Some of the factors that affected our choice of a concurrency control method were:

- The cost of back tracking; i.e. undoing compilations.
- The granularity of read and update transactions.
- The desire to allow shared (concurrent) access to foreign libraries.
- The need to have the basic facilities of the ALM available to other parts of the compiler early in the project.
- The ALM costs in time and space in comparison to those of the compiler.
- The portability requirements of the compiler.

Locking algorithms prevent inconsistent information from reaching other parts of the compiler by blocking transactions that might produce an inconsistent view. We considered several locking and serial validation schemes for the UNIX Ada System. Serial validation was rejected for two reasons. First, the effect of a series of transactions that result from a compilation may not be found to be invalid until just before the compilation is to be completed; with serial validation, inconsistent information could very easily reach other parts of the compiler. Second, the cost of a compilation (the unit of failure) and the percentage of the database affected by a compilation were considered high enough to make serial validation techniques appear unattractive.

Once locking was chosen, the problem of granularity of locking became important. Should locks be placed on the **DICTIONARY** entries of individual compilation units, on libraries, or on the entire system? Should a hybrid locking scheme allow some combination of the above schemes? Because of details of the internal structure of the **DICTIONARY** file, it proved impractical to attempt locking at the entry level. Such locking required far more overhead than could be justified for the gain in concurrent accesses. Locking the entire system (i.e. all ada libraries) was also quickly rejected; it would effectively eliminate concurrent access. Locking libraries turned out to be reasonably easy to implement and has proved adequate for testing needs during development. We look forward to getting feedback on this from users with large projects. Proper division of a project into libraries is

expected to be one of the most significant factors determining the number of concurrent compilations that are possible.

Another aspect of locking algorithms is the choice of lock types. Locking schemes that offer more lock types tend to allow greater concurrency at the expense of additional overhead. Single state locking (access/no access) was rejected out of hand because it unnecessarily limited access to shared libraries. Such an approach would, for example, prohibit a user from getting a report about a foreign library in which another user was doing a compilation. Read/write locking was also considered and rejected in favor of read/modify/write locking. Three state locking (read/modify/write) has the advantage of being easily converted to read/write locking or upgraded to a more sophisticated scheme in the future. In summary, our lock types are

- **READ:** A read lock prohibits users from modifying a library. When a compilation imports a unit from a foreign library, it requests a read lock to prevent that library from being altered before the compilation completes.
- **MODIFY:** A modify lock is used to give a user the exclusive right to alter the current library's **DICTIONARY** file. A modify lock may be held on a library while read locks exist, but not while a modify or write lock is held by another user.
- **WRITE:** A write lock is required before a process can update a library. Write locks are exclusive; they do not share with read or modify locks. A process holds a write lock only while it is actually writing the **DICTIONARY** to disk.

The compiler sets a modify lock on its home library and seeks to upgrade it to a write lock just before it commits its results. It requests a read lock on each foreign library that contains any additional required compilation units. It is possible for a process to "starve" because it is unable to acquire a write lock when many reads on that library are active. If this problem becomes significant, an additional lock type, the intention write lock, can be added to prevent starvation.

UNIX record locking provides the basic primitives that detect deadlock. The ALM terminates a compilation that detects deadlock. The user is also able to set a bound on the amount of time the compiler will wait to get a lock before it gives up.

4. Building Complex Programs in the Ada Environment

UNIX programmers use tools like *make* to keep large programs in a consistent state. Because writing and maintaining the makefiles for a large program is a significant, error-prone chore, many applications provide automated makefile generators. In the process of trying to determine how to produce an automated *makefile* for Ada programs, we realized that all of the relevant compilation unit dependency information existed in the ALM database itself! And if we would also store the specified compiler options for files during compilation, we would have enough information to recompile a file.

Many interesting design issues needed to be resolved before we could complete our program build facility, *adamake*. Should makefiles be generated? What operations should be permitted on foreign libraries? How would source archives be handled? How should *adamake* deal with routines that were not coded in Ada? We touch on these issues as we describe the design of *adamake*.

4.1 Adamake Basics

As Figure 1 illustrates, *adamake* is a command driver which views the Ada compiler as a tool for determining the attributes and dependencies of the compilation units in a source file and for compiling those units. In fact, *adamake* calls the compiler as a subroutine. This tight coupling allows multiple recompilations to be done within a single process - a behavior that *make* does not provide.

Having *adamake* and the compiler operate as a unit has the advantage of allowing the internal data structures that represent the dependence graph to be shared between two logically independent traversals. The traversal made by the ALM (as a service to the compiler) to determine dependent units may be considered to be the 'inner' walk; the traversal made by *adamake* to determine the units to compile is the 'outer' walk. The fact that these traversals generally visit the same set of compilation units should significantly decrease the overhead of using *adamake*. A particular case of interest is the case where a user has provided two versions of the same compilation unit to *adamake*, each in a separate source file. Such a situation, generally the result of a user error, requires great care for a program building tool to handle well.

Large projects generally require some mechanism to regulate changes in source code. *Adamake* encourages the use of SCCS[5], the UNIX system source control utilities, by automatically using the most recent version of source available. It considers SCCS files to be optional and so does not complain if a particular file does not have a corresponding SCCS file.

Adamake is significantly affected by the access policy one chooses for foreign libraries. Our present policy lies on the conservative side of the continuum that ranges from disallowing foreign library access and updates to not distinguishing at all between home and foreign access rights. Disallowing updates in foreign libraries allows a greater degree of concurrent access than do more liberal update policies. We believe that most projects will be structured so that their natural partitioning will not require foreign library updates. Experience in the field will dictate whether or not to shift this policy.

We have discussed *adamake* in the context of programs that are written solely in the Ada language. In a multi-lingual environment, a user of the UNIX Ada system has the option of invoking *adamake* or *ada* from within a standard makefile. Alternatively, object or archive files may be specified on a command line and they will be linked in with the Ada modules. An enhancement under consideration is to provide a pragma which would allow an arbitrary shell command to be specified within an Ada compilation unit. That command would then get invoked prior to the compilation of the Ada source file. Finally, while we do not currently produce a makefile, there is nothing in our approach to preclude adding this option for users who may wish to use some of the macro facilities of *make*.

5. ALM as the User Interface

We have described how the ALM manages separate compilation for the compiler and for *adamake*. The ALM also serves the Ada programmer directly as the mechanism for interacting with Ada program libraries. With the *alm* command, the user can create, destroy, and manipulate the contents of Ada program libraries. Thus, the ALM is also the interface to everyday use of the Ada programming environment provided by our UNIX Ada System. This section describes some of the ALM tools as seen by the Ada programmer and discusses the design philosophy behind the choices we made in their implementation.

5.1 Tools to Manage Program Libraries

The classes of tools provided by the *alm* command are as follows:

- Creating and destroying Ada program libraries
- Translating between Ada library aliases and UNIX System directory names
- Renaming and moving Ada program libraries
- Copying and deleting source file entries
- Reporting on Ada program library contents

- On line help facility

These are an obvious set of needed tools for the creation and manipulation of Ada program libraries. One aspect of the report package, however, merits some discussion. We expect that a given program library will potentially grow quite large over time as compilation units are added and older ones are modified. We took extra care in providing report capabilities that would give crucial information to the programmer regarding compile times, features of the compilation unit and its source file, dependency of the unit on other compilation units, and so on. Reports can be generated easily for any named compilation unit in any program library. Reports can be sorted by time of compilation or alphabetically by name. Source files or compilation units can be selected for inclusion in reports on the basis of the presence of various attributes computed by the compiler or on the basis of compile times before or after a given date and time.

In general, these ALM commands enable the Ada programmer to create and manipulate program libraries without worrying about the details of the implementation of a program library as a UNIX directory structure. In effect, the implementation of an Ada program library is hidden from the user much as Ada hides a private data object from direct manipulation. However, we expect that our users will be very interested in using other UNIX System features. Nothing in our system prevents users from doing just that.

5.2 Who is the Ada Programmer

When we envisioned the user interface for the ALM, we had to consider who the user would be. Because there are few Ada compilers in use in a true software production environment in the outside world, we had to make some educated guesses about users and their needs. We conceived of our users as knowledgeable UNIX System programmers. At the very least, we assumed our users would be experienced programmers who would pick up a working knowledge of the UNIX file system and many user-level commands quickly. Thus, we decided that we would provide tools that looked and worked like other UNIX tools, as much as possible.

5.3 An 'Ada Shell' for Interactive Use of the ALM

We also thought that some users would want to conduct their work more completely in the framework of an Ada-like domain. Thus we provided an interactive mode for the *alm* command. Within this interactive mode, commands are issued by typing keywords associated with the option letters of the command mode. We consider this interactive mode to be a prototype "Ada Shell". Within this shell, Ada library aliases can be reckoned with more directly. For example, we have provided the equivalent of change directory (*cd*) where the argument is the library alias associated with the desired UNIX directory. As with all manner of shells, our Ada Shell is easily escaped for single UNIX command calls, the spawning of another shell, or the invocation of the Ada compiler. The interactive mode also allows the user to execute a list of ALM commands that appear in a file. Finally, within the Ada Shell, users can modify the default UNIX directory structure of an Ada program library itself by changing, for example, the directory for source, object, or SCCS files.

5.4 Tools That Can be Modified and Extended

As mentioned earlier, we recognized some uncertainty in defining the Ada programmer users of our system. Therefore, we decided to be flexible in approaching the form and content of the user interface. First, we designed our tools to be easily modifiable. This was implemented with standard "good" programming practices of top-down design and modular construction of the commands themselves. Second, we decided to approach the problem of user needs directly by seeking out user feedback as early as possible. We did this, and continue to do so through an on-going program of alpha and beta sites. In this manner, we can get an early reading on the clarity of error messages and of the adequacy of our set of tools to meet user needs.

6. Tools for Installation and Administration

The ALM relies on the presence of certain files and shell environment variables that must be made accessible to system administrators, as well as Ada programmers. These need to be provided to user sites in a reasonable fashion. We decided to implement a separate package of tools for Ada system administrators to install and maintain their Ada systems. The end result is *adm_ada* - a program that provides needed support files and administrative facilities for project leaders who would be setting up an Ada site. The program is completely interactive in its nature. It sets shell environment variables and creates default template files to user specification that will structure the Ada environment pervasively. The result is a setup that provides flexibility to project leaders while easing the startup cost to the individual Ada programmer.

7. Future Work

The following list itemizes areas for future work in the library manager.

- Should the ALM database maintain information about compilation units that were not written in Ada?
- Should information be retained about the names and locations of the executables generated by the compiler? Currently, UNIX Ada does not retain such information because executables are frequently moved and or renamed.
- Should we allow updates to foreign libraries? Currently, we take the careful approach of disallowing such transactions. However, the global use of *adamake* to build a target with outside dependencies is potentially limited by this restriction.

8. Conclusions

This paper has described some of the decisions we made as we designed and implemented the Ada Library Manager for the UNIX Ada System. We made the ALM a comprehensive database manager for Ada compilation units, and we made it an effective link between the compiler and the programmer. We provided a mechanism to map between UNIX pathnames and unique Ada aliases to ease the burden on the end user and to provide a partial hiding of the UNIX System implementation. At the same time, our approach did not hinder the use of familiar UNIX tools. We provided a program build facility (*adamake*) that makes optimal use of information already computed by the compiler. Finally, we also provided a set of tools (in *adm_ada*) for administration and installation of the UNIX Ada System.

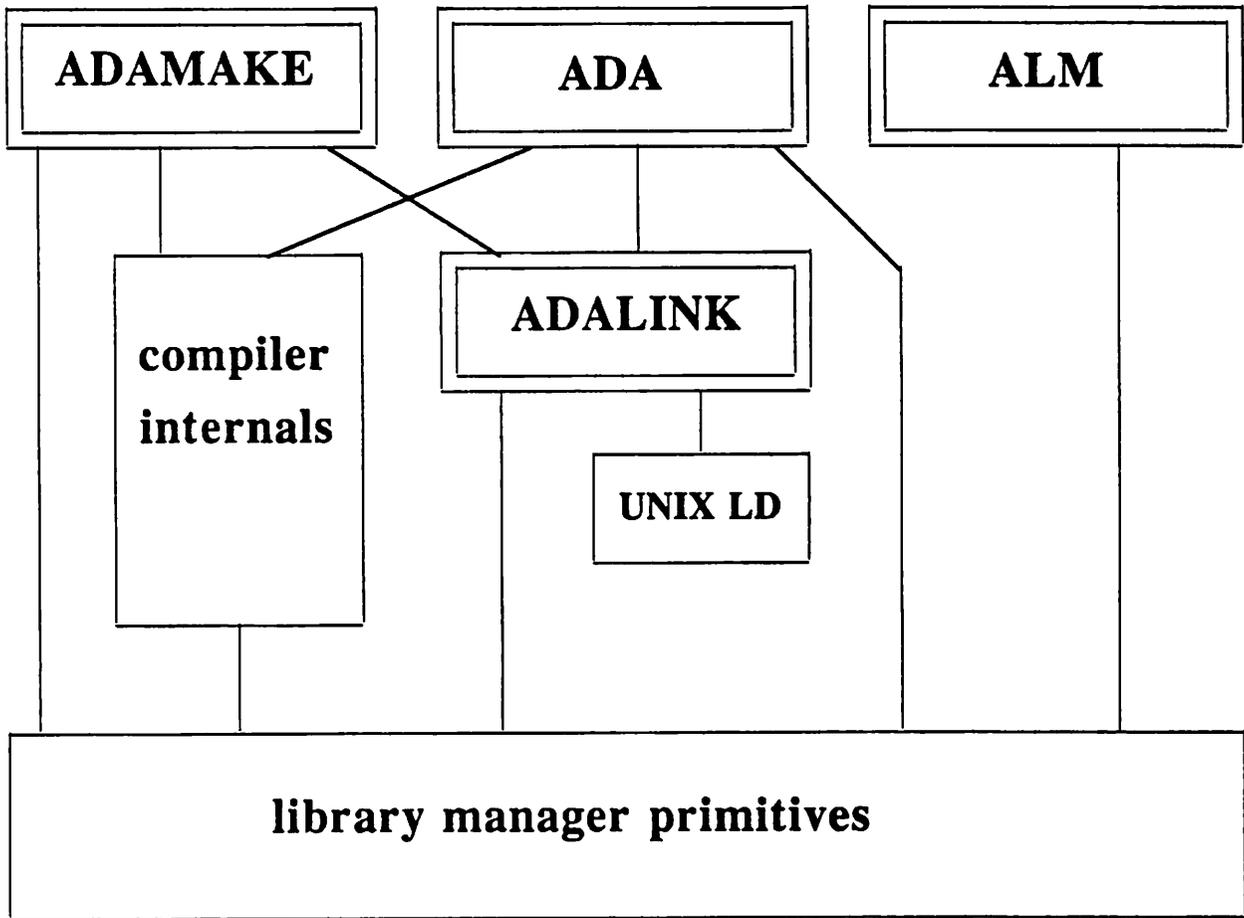
To summarize the effect of these decisions, we feel that the ALM enables the UNIX Ada System to become fully integrated into a UNIX system's family of languages. It makes use of existing operating system and utility features wherever possible and offers simple extensions to the environment as required by the Ada language definition.

References

- [1] "Reference Manual for the Ada Programming Language". ANSI/MIL-std-1815 A. U. S. Department of Defense, February 1983.
- [2] Feldman, S. I. "Make - A Program for Maintaining Computer Programs", Documents for UNIX, Volume 1, Bell Laboratories, January 1981.
- [3] Knuth, D., "The Art of Computer Programming", (1973) Volume 1, Section 2.5.
- [4] Langer, J. "The Ada BreadBoard Compiler: The Ada Linker". Internal memorandum, Bell Laboratories, 1982.
- [5] Bonanni, L. E. and Salemi, C. A. "Source Code Control System's User's Guide". Documents for UNIX, Volume 2, Bell Laboratories, January 1981.

Figure 1.

UNIX Ada Library Management



DICTIONARY files

Ada Source files

Symbol Table files

Object files

SCCS files

Targeting Ada to 68000/Unix

Mitchell Gart
Alsys Inc.
1432 Main Street
Waltham, Mass. 02154

Abstract

We discuss the targeting of an Ada compiler to several 68000/Unix systems. The main focus is on the interactions between Ada - the runtime model, generated code, and runtime system - and Unix. Some problems that are general to the mapping of Ada to Unix, as well as some problems that are specific to the Unix implementations to which the compiler is targeted, are mentioned. Finally, we give some of our experiences as users of the resulting Ada/Unix compiler. We find that for most kinds of applications, Ada fits well on Unix.

1 Introduction

Alsys is developing a family of Ada compilers that are organized into a mostly machine-independent "root Ada compiler" and target-specific code generators and runtime systems. The first compiler in this family (validated December 1984) was a cross-compiler from the Vax to the Altos ACS 68000, a small multi-user Unix System III machine built around the Motorola 68000. Since then we have validated self-hosted compilers for the Sun, Apollo, and Hewlett-Packard HP-200 machines, based on the 68010 and the 68020 and running Unix System V and 4.2 BSD. We are also engaged in a parallel effort towards an Intel 8086/8088/80286 compiler for the IBM PC AT running under DOS. The 68000 target machines that will be mentioned in this paper are the above Altos machine, afterwards referred to as "Altos", and the Hewlett-Packard HP-200, a 68010 Unix System V machine, afterwards referred to as "HP".

1.1 Runtime model and runtime system

In this paper we will discuss the Ada runtime model and runtime system, and their interactions with the Unix targets. The runtime model consists of such issues as how a program starts execution, how its code and data are laid out in memory, how its stack grows, how data is addressed, how such Ada constructs as tasking, exceptions, and input/output are mapped to the target system, and so on. An Ada program that is compiled and linked on Unix represents a mapping of Ada source code to 68000 machine instructions that are generated inline, along with calls to the Ada runtime system (RTS). There are typically two kinds of RTS calls in the generated code: calls that correspond to explicit user program calls for a service in a predefined package such as CALENDAR or TEXT_IO, as well as calls that are implicitly inserted by the compiler into the code stream.

An example of an implicitly inserted call occurs when the Ada semantics dictate that a task is to be activated. A call to the RTS procedure `ACTIVATE_TASK` is generated. A second example is that the compiler may decide that a certain user-declared object, such as an array with size known only at execution time, will be allocated on the heap via an RTS call. This example is interesting because it shows one of the ways that the root Ada compiler is parameterizable to a given target

machine. The decision where each object is to be allocated (global data, heap, or stack) is made based on parameters that differ for different target machines.

The major RTS components are:

task kernel	
heap manager	
exception manager	
input/output	(TEXT_IO, DIRECT_IO, and SEQUENTIAL_IO)
CALENDAR	
attributes	('SIZE, 'IMAGE, etc.)
SYSTEM	
arithmetic	(exponentiation, floating point simulation, etc.)

of which CALENDAR, SYSTEM, and the input/output packages are directly callable by a user program, and the rest are called implicitly. The implicitly-called components are grouped into the package ADA_RUNTIME, which is put into a user's Ada library upon creation, but which is not directly callable by a user program (i.e. a "with ADA_RUNTIME" clause is prohibited).

ADA_RUNTIME is a package with an Ada specification. The bodies of the different routines are written mostly in Ada (the parts that are expected to be target-independent), some in C (most of the parts that have to make specific interactions with Unix), and a little bit in assembler (for example the low-level routine that saves and restores machine registers on a task switch).

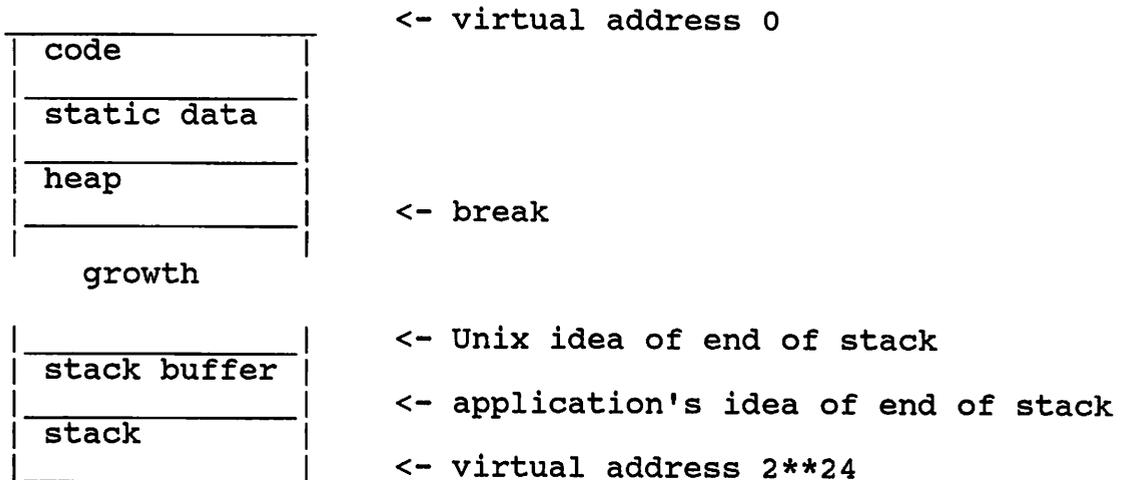
1.2 The rest of this paper

In the next sections of this paper we will first focus on the interactions between the Ada runtime model and RTS with 68000/Unix, in such areas as memory organization, program startup, tasking, exception handling, the heap, and input/output. We will then focus on some problems that arose in targeting the RTS: problems that are general to Unix and problems that are mostly related to incompatibilities between the different Unix implementations. Finally, we give some of our experiences as users of the resulting Ada/Unix compiler.

2 Mapping the runtime model to Unix

2.1 Memory organization

The memory layout that we typically find on target 68000/Unix systems is to have a Unix process with a virtual memory layout from address 0 (Altos) or a small constant (16K on HP) up to a maximum address of either 2^{23} or 2^{24} . Code, then static data, then heap, are laid out at the low addresses, growing toward higher numbered addresses. The stack starts at the top and grows toward lower numbered addresses:



The Unix kernel handles the memory mapping that gives each process this view of its virtual address space.

An Ada program's code, global data, and "environment task" stack are mapped into their respective regions in the above diagram (no big surprise). We would have liked to map Ada static constants into the code section, so that they could be protected when a program is linked with one of the Unix linker's options that makes write-protected code, but we discovered that on the Altos write-protected code is also read-protected and execute-only, so constants had to be mapped into the static data region. (This is the first example, more of which will be mentioned, of a small surprise that impacted our implementation, and that was different between different 68000/Unix implementations.)

Inside these memory regions, the RTS heap manager handles the heap. Stacks and control regions for Ada tasks are allocated inside the heap, and the exception manager uses the "stack buffer". These points are

expanded in the following sections.

2.2 Program startup

The steps of Ada program startup are:

standard C runtime startup	(crt0.s)
C main program	(main.c)
environment task body	
main Ada subprogram	(main.ada, or another name)

The standard assembler runtime executes and calls a small main.c, which is an RTS component. The purpose of main.c is to establish signal catchers for the Ada program, to save argc, argv, and the environment pointers in static variables so that they can be accessed later from the special Ada-callable package UNIX_ENV, and then to call the "environment task". The latter is our name for the environment around the main Ada subprogram. It contains calls to elaboration code for all of the library packages in the application, which according to Ada rules must be elaborated before the main subprogram. Finally, the main Ada subprogram is called. It must be a parameterless procedure in our implementation, but it can access its command line and environment via, for example, UNIX_ENV.ARG_VALUE(1), which returns a string or raises an exception if there were no command line arguments.

Return from the main Ada subprogram ends execution. We provide the possibility for a program to return an integer status result to the Unix shell via UNIX_ENV.SET_RETURN_CODE. The default value returned is 0, corresponding to the Unix shell's convention for "no error".

2.3 Tasking

A major question in targeting Ada tasking is the mapping between an Ada task and a target operating system process. In general, Ada tasks can share variables, as well as passing variables to one another via entry call parameters. Because of the lack of shared memory between Unix processes on System III, as well as the limited interprocess communication primitives (pipes and signals), we chose to map an entire Ada program, possibly containing several tasks, onto one Unix process.

When a task is created its task control block (TCB) and stack are allocated from the heap. The default size of a task's stack is currently 16K bytes, but this may be overridden by either a pragma inside the Ada program or via a command line switch. Once allocated, a task's stack has a fixed maximum size and cannot grow (unlike the "environment task", whose stack can be extended by Unix). Each task's stack resembles that of the environment task, with a small stack buffer at the end.

One task is running at a time, and the others are idle. Each idle task contains a copy of the 68000 registers inside its TCB. A task switch is accomplished by copying the current registers into the TCB of the switched-out task, then restoring the registers of the switched-in task from its TCB, in a small assembler RTS module.

When a task other than the environment task is running, the stack pointer (address register A7) and frame pointer (A6) indicate a "stack" that was allocated within the heap, rather than the Unix idea of the stack. An end-of-stack pointer is kept to insure against stack overflow.

Equal priority tasks may be switched with time-slicing or not, as a user option. Time slicing, and also delay statements, are handled by setting a Unix alarm, catching the resulting signal, and transferring control to the task kernel, for a scheduling decision.

The task kernel has a lot of work to do. It handles decisions about scheduling, task activation and termination, and many other matters. For task rendezvous (entry call, accept, and select statements) it must manage queues of waiting tasks. The organization of the task kernel, as well as the rest of the RTS, other than the parts that interact with Unix, are beyond the scope of this paper.

2.4 Exception handling

Ada requires that many compile-time and runtime checks be performed to insure the consistency of a program. If a check fails at runtime, an exception is raised. Exceptions are supposed to occur only rarely, in error situations, so the goal of an Ada implementation is to perform the checks with as little time and space overhead as possible for the case where the check is OK

and no exception is raised. The code for a check is usually the machine-code equivalent of:

```
if check fails then
  raise predefined_exception;
end if;
```

and the code generator must make the most efficient code to perform the check, as well as the most compact code to do the "raise". A call to an RTS routine to do the raise would take a 6-byte instruction, but some 68000 trap-style instructions are only 2 bytes, so we have used these instructions.

Unfortunately these instructions are notoriously non-portable between 68000/Unix machines, as the following chart indicates:

Trap-type instructions		
	Altos	HP-200
trap#0	system calls	system calls
trap#1	SIGIOT	SIGTRAP*
trap#2	no-op	SIGILL
trap#3	no-op	SIGEMT
trap#4	no-op	SIGILL
trap#5	no-op	SIGILL
trap#6	no-op	SIGILL
trap#7	no-op	SIGILL
trap#8	no-op	SIGFPE
trap#9	no-op	SIGILL
trap#10	no-op	SIGILL
trap#11	no-op	SIGILL
trap#12	no-op	SIGILL
trap#13	no-op	SIGILL
trap#14	SIGTRAP*	SIGILL
trap#15	no-op	SIGILL
*debugger		
divs by 0	no-op	SIGFPE
divu by 0	no-op	SIGFPE
trapv (V clear)	no-op	no-op
trapv (V set)	no-op	SIGILL
chk (OK)	no-op	no-op
chk (fails)	no-op	SIGILL

but we chose to use them because the resulting code is so much more compact.

The 68000 has 3 very useful instructions for performing checks: TRAPV (trap on arithmetic overflow), DIV (a check for 0 dividend is performed automatically), and CHK (trap if a register is < 0 or > a specified value). Unfortunately these instructions are masked by the Altos Unix kernel, presumably because they aren't useful for C programs. We plan to use them on other 68000 machines.

These trap-style instructions cause Unix signals, which are caught by a signal catcher (written in C). For an exceptional trap the catcher reads its (the catcher's) return address to see where the signal occurred, reads the trap instruction to see what kind of Ada exception, and calls the exception manager. For an alarm signal, the signal catcher calls the task kernel.

Because there is a stack buffer (a piece of the stack reserved for RTS execution) the signal catcher has room to run when the stack overflows. If the environment task is running, it tries to extend the stack, returning to the application if successful. If there is no room to extend the environment task's stack, or if another task's stack overflows, the signal catcher has enough space to execute code that raises STORAGE_ERROR in the user's program.

2.5 Heap and I/O

For both the heap manager and the input/output packages our basic approach has been to use the Unix system calls (SBRK for more memory; OPEN, READ, and so on to access files) but to bypass the standard C library functions (malloc and the stdio routines). For the heap we wanted a slightly different organization than malloc, with special treatment for Ada collections (groups of heap objects pointed to by the same access type) grouping all objects in a collection so that they can be freed together. Ada I/O (especially TEXT_IO) has to perform functions that are different from printf.

There is a very nice correspondence between the Ada and Unix notions of STANDARD_INPUT and _OUTPUT (in fact the Ada designers borrowed these concepts from Unix). There is no Ada notion that corresponds to the Unix standard error file, so we occasionally use this file

for RTS error diagnostics.

3 Major problems in targeting to Unix

3.1 Tasking

The ideal target operating system for running an Ada program would provide a 2-level set of primitives similar to "program" and "task" rather than the Unix 1-level "process". All of the data and code within a program would be shared by the tasks of the program, but each task would have a stack that grows independently, with the growth managed by the operating system.

For intertask communication, the Unix System V primitives (semaphore and message) seem to be sufficient, but we haven't investigated this closely. System V also allows zones of data memory to be shared between processes, so we are tempted by the idea of a 1 Ada task/Unix process implementation on System V. This still seems difficult because static data, heap, and stack memory all potentially must be addressable by all of the tasks in a program, for example if a task entry parameter is passed by reference, or if an entry parameter is an access type.

The Ada manual recommends that a delay statement have a precision of at least 20 milliseconds. HP allows this, but Altos implements only a 1 second precision for the alarm clock.

3.2 Input/output

When several tasks are inside one Unix process, and one task blocks on an I/O operation, it is desirable to have only that task be blocked, not the entire Ada program. There is a workaround to this problem that is complicated to program. It involves using the System III FCNTL call to set non-blocking I/O. This solution is far from ideal because a program doing non-blocking terminal I/O crashes under the ADB debugger.

A Unix I/O operation that fails returns the reason for failure in a static variable named ERRNO. If a task performed an I/O operation, was switched out and then back in, and then read ERRNO, it could get the value that had been set by another task. The solution, in a program that uses tasking and I/O, is expensive: a

runtime mutual exclusion mechanism can surround each I/O call.

The Ada I/O packages provide function NAME which returns the full pathname of an open Unix file. If an Ada file is open to a Unix terminal, NAME should return "/dev/tty", whether the terminal is open for input or output, but there is an ACVC test that requires that NAME(STANDARD_INPUT) and NAME(STANDARD_OUTPUT) be different. The author considers this to be a bug in the ACVC test suite, not a Unix problem.

3.3 Incompatibility

To a compiler writer, compatibility between target machines means compatibility at the machine code level. Unix standardization efforts have focussed on the system call level and the C library level (the same call, with the same parameters, should have the same meaning on different Unix implementations). At the machine code level, we have found many annoying differences between 68000/Unix machines.

Two examples, write-protected code and trap instructions, have already been mentioned. Another examples is the assembly language itself! HP supports the Motorola assembly language (e.g. "move.w #6(a0), d0"), but Altos supports a PDP-11-style assembly language ("movw a0@(6), d0"). This means that RTS components written in assembler must be translated.

A few more incompatibilities:

- the context of a signal (where on the stack registers are saved) is needed by the signal catcher to see where and why it was called. This information is difficult to find in any manual, and differs greatly between machines.

- the method to extend the main stack is different on each system we have tried.

In addition to the C source standards for calling library functions, it seems desirable to have another level of standard that would say: "Version N Unix, running on machine architecture X, has assembler conventions that: instruction 1 has ... effect instruction 2 has ... effect " etc. This would be an extra level of Unix standard (a Unix sub-standard?) that

would move toward binary code compatibility, a desirable feature that doesn't exist now.

4 Experiences using Ada

At Alslys we are Ada users as well as Ada implementors. Our compiler and most of our toolset are written in Ada. We are starting to have a significant amount of experience as Ada users on VAX/VMS, 68000/Unix, and 80286/DOS. Ada is proving to be a good language with which to do our work as programmers. Ada programs execute fast on the target machines, and are proving to be highly portable between target machines.

The compiler consists of around 700 compilation units and 250,000 lines of Ada source code. The Ada facilities for program structure (public specifications, private implementations, and trees of subunits) have been tremendously helpful to us as software engineers, in organizing this large software project. The HP, Sun, Apollo, and IBM PC AT compilers are now "bootstrapped" or compiled through themselves. Because we were careful to isolate the system-dependent parts (such as the COMPILER_IO package) the root compiler's source code has proved to be portable without too many problems.

4.1 Portability

Programs that we have written in Ada are proving to be very portable between the Vax, the 68000, and the 80286, running under the 3 different operating systems. The Ada Program Viewer is an Alslys tool that allows a user to browse through the source modules of an Ada program, "zooming" in and out to different levels of detail. The Viewer is written in Ada (about 9000 lines) and porting it from the 68000 to the 8086 required only a change of one package body that did screen input/output and windowing, no other changes to the program source.

We learned from the Viewer experience that Ada alone doesn't guarantee good software engineering. The first port of the Viewer, from the Vax to the 68000, was difficult because the I/O and windowing code was distributed around the program. Once the program was reorganized to encapsulate these actions into one package, the second port, from the 68000 to the 80286, required no changes outside this package body.

4.2 Efficiency

Applications written in Ada are proving to execute faster than Pascal, and at around the same speed as C, on the Unix and DOS target machines. At press time, figures for the 68000 machines were unavailable, but the following benchmarks compare the efficiency of generated code on the IBM PC AT:

PC AT Benchmark results
=====

Perm	Towers	Queens	IntMM	MM	Puzzle	Quick	Bubble	Tree	FFT	Ack
Alslys Ada, no checks:										
2.31	2.25	1.81	5.10	9.87	34.21	1.39	4.78	1.64	11.85	23.34
Alslys Ada, checks:										
3.34	3.51	2.75	6.09	11.09	39.81	2.81	7.18	1.92	13.46	31.00
Lattice C, large model:										
2.86	2.31	4.11	12.31	40.48	12.91	5.82	1.81	3.63	61.29	37.02
Lattice C, small model:										
2.41	2.26	1.31	2.53	29.28	13.12	2.53	1.81	3.30	52.78	35.37
Turbo Pascal:										
4.56	4.4	2.96	3.52	79.58	14.61	3.07	4.23	7.14	121.33	72.72

Time totals:

	no f.p.	floating point (MM + FFT)
Ada (no checks)	76.83	21.72
Ada (checks)	98.41	24.55
C (large model)	82.78	101.77
C (small model)	64.64	82.06
Turbo	117.21	200.91

Times are in seconds, on an 8MHz IBM PC AT. The benchmarks are:

- Perm: Generate all permutations of 7 integers 5 times (recursive)
- Towers: Solve Towers of Hanoi (14 discs, recursive)
- Queens: Solve 8 Queens problem 50 times
- MM: Multiply 2 40x40 matrices of 32-bit floating point numbers
- IntMM: Multiply 2 40x40 matrices of 16-bit integers
- Puzzle: A compute-bound puzzle program.
- Quick: Quicksort 5000 integers

Bubble: Bubblesort 500 integers
Tree: Binary tree sort of 5000 integers
FFT: Do 256-point Fast Fourier Transform 20 times
Ack: Compute the Ackerman function ack(3, 6)
 10 times.

These figures are not meant to be a precise measurement of the relative quality of compilers, but merely to show a general result: that we are able to confidently claim that Ada is able to generate code that is at least of comparable efficiency to these other languages.

For the floating point tests, Ada uses the 80287 floating point co-processor. We think that C also uses the co-processor, but that Pascal does its computations in software. We used Revision 2.15 of Lattice C and Version 2.00A of Turbo Pascal for the tests. In Lattice C, the "small model" allows a 64K byte limit for code, plus 64K combined global data plus stack plus heap. "Large model" doesn't have these restrictions. The Ada results can therefore be most fairly compared to the C "large model" results.

5 Conclusions

An Ada compiler (and related tools) fit nicely into the Unix environment, and are useful for writing applications. We are enthusiastic about our experiences as users of Ada.

Unix is an excellent host system for Ada program development.

Unix is an excellent target system for running Ada applications that don't have heavy real-time constraints, and that don't use tasking "too much". We seem to be able to produce Ada code that executes as efficiently as C. We are happy with the correctness of the tasking implementation, and an application with a few parallel tasks and an occasional task switch runs very efficiently.

As a target for a real-time Ada program, or a program that makes very heavy use of tasking, the standard time-sharing versions of Unix have inherent limitations.

6 Acknowledgements and trademarks

Jacques Sevestre headed the 68000 codegen team, and helped a lot with the preparation of this paper.

The C and Pascal versions of the benchmark suite were given to the author by John Hennessy.

Ada is a registered trademark of the United States Government (AJPO). *Unix* is a registered trademark of AT&T. *VAX* and *VMS* are trademarks of Digital Equipment Corporation. *IBM* and *IBM PC* are registered trademarks of International Business Machines. *Lattice* is a registered trademark of Lifeboat Associates. *Turbo Pascal* is a trademark of Borland International.

A Comparison of UNIX † and CAIS System Facilities

Helen Gill

Rebecca Bowerman

Chuck Howell

MITRE Corporation
1820 Dolley Madison Boulevard
McLean, VA 22102

1. INTRODUCTION

The Ada [®] Joint Program Office (AJPO) has supported several efforts aimed at designing and implementing an Ada Programming Support Environment (APSE) and at designing a standardized set of system calls, the Common APSE Interface Set (CAIS). The original purpose of the CAIS was to serve as a common kernel-level interface for two APSEs being developed for the DoD: the Ada Language System (ALS) and the Ada Integrated Environment (AIE). However, the AIE effort was reduced to a compiler and debugger development, and the ALS encountered serious schedule slips. There has been a proliferation of environments used for Ada software development in the DoD community. As a result, the charter of the CAIS has evolved to a more general operating system capability, increasing the complexity of hosting it on any existing operating system.

1.1. Purpose

The purpose of this analysis was to determine the extent to which the CAIS accommodates the needs of tool writers. UNIX, an operating system and set of tools originally developed at Bell Telephone Laboratories, is a programming support environment that supports a variety of languages, including Ada, C, Pascal, FORTRAN, and LISP. UNIX is one of the most widely used APSES.

1.2. Scope

The scope of this analysis is limited to a comparison of the system calls in the Berkeley 4.2BSD version of the UNIX operating system to the Proposed MIL-STD of the Common APSE Interface Set (CAIS), as specified in [CAIS]. The focus of this analysis is the tool writer's perspective. That is, the analysis examines those areas where there might be an increase or loss of capabilities to the tool writer when using Ada and CAIS system calls as opposed to Ada and UNIX system calls. The comparison is between UNIX system calls and CAIS facilities because the system calls represent the fundamental capabilities of UNIX and provide the basis for UNIX library-level calls. Discussion of CAIS capabilities beyond UNIX is not included in this paper.

1.3. Report Organization

This report begins with a brief background that explains the need for a comparison of this type (Section 2.0). Although time and space do not permit a complete introduction to the proposed MIL-STD CAIS, a brief description is given. The technical issues section (section 3.0)

* UNIX is a trademark of AT&T Bell Laboratories

[®] Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

presents a broad categorization of UNIX system calls and addresses the comparison of UNIX with the CAIS within each of these areas. Section 4.0 summarizes the analysis. The Appendix presents corresponding UNIX system calls and CAIS calls or supporting features in tabular form.

2. BACKGROUND

This section provides a brief history of the DoD efforts to build Ada programming support environments that led to the design of the CAIS and a high-level overview of the CAIS.

2.1. History of the CAIS

In 1980 the Army awarded a contract to SofTech to develop an Ada compiler known as the Ada Language System (ALS). The Army later expanded this effort to include a variety of software support tools, host dependent services, and a database. Although no special programming environment is needed to use the Ada language, it was commonly believed that an integrated set of tools would speed the acceptance of the language. As a result, in 1980 the DoD published the *Requirements for Ada Programming Support Environments* [STONEMAN], the "Stoneman" document. Soon after the publication of Stoneman, the Air Force awarded a contract to Intermetrics to build the Ada Integrated Environment (AIE), a programming support environment based upon Stoneman.

The existence of multiple DoD-sponsored APSEs threatened to undermine the benefits of commonality, which was the primary goal of the Ada program. As a result, the Kernel APSE (KAPSE) Interface Team (KIT), a tri-service organization chaired by the Navy under the guidance of the Ada Joint Program Office (AJPO), was established in late 1981. Here, the term "kernel" refers to the host-dependent code that supports the system call interface, roughly analogous to the Chapter 2 calls provided by UNIX. The objective of the KIT was defined by a Memorandum of Agreement signed by the Deputy Under Secretary of Defense and the Assistant Secretaries of the three services. It stated that the KIT was to define a standard set of interfaces meant to ensure the interoperability of data and the transportability of tools between conforming APSEs. The KIT soon became responsible for a variety of Ada-related activities and formed a subgroup, the Common APSE Interface Set (CAIS) Working Group, to focus on defining the standard set of interfaces.

Meanwhile, the scope of the implementation of the AIE was reduced; KAPSE level services are not included. Work by the CAIS Working Group continued and began expanding from a common set of system interfaces to more general operating system capabilities. In September 1983 a public review was held for version 1.1.2 of the CAIS. A major criticism raised in the public review was that although the CAIS had reached detailed specification, no requirements document existed. Although the Stoneman requirements document mentions a CAIS-like capability, it does not contain the detailed requirements for one.

CAIS 1.2 and Technical Note 1.0 were available on request in June 1984, and CAIS 1.3 was released for public review in August 1984. An analysis similar to the present one was begun under AJPO guidance. Following further revision, the proposed Military Standard was released in January 1985. It includes a draft policy statement that restricts use of this version of the CAIS to prototyping efforts. The proposed MIL-STD CAIS is the version that is treated in this analysis. In November, 1985, the CAIS version 2.0 contract was awarded, with SofTech and Compusec the contractor team.

2.2. Overview of the CAIS Specification

Ideally, all APSE tools would be implementable using only the Ada language and the CAIS. The CAIS specification defines a set of Ada package specifications and their intended semantics. The scope of the CAIS is limited to interfaces to those system services traditionally provided by an operating system that affect tool transportability. Interoperability of data has not yet been addressed. In support of transportability, the CAIS defines the concept of a "node model." The CAIS document states that a CAIS implementation is to act as a manager for a set of entities that are files, processes, and organizational structures. Ada packages that

support management of nodes and relationships, structures, files and devices, processes, attributes, and access control are defined. Additionally, the CAIS specifies a generalized list utility package.

2.3. Overview of the CAIS Node Model

The CAIS model uses the notion of a "node" to represent information about an entity (such as a file, directory, process, or device). The entities represented by these nodes have properties and may be interrelated in many ways. The CAIS identifies three different kinds of nodes: structural, file, and process. File nodes have contents corresponding to ordinary files. Special kinds of file nodes are used to represent devices and to support interprocess communication. Structural nodes represent users and can be used to represent arbitrary structural information. A typical use of structural nodes with file nodes would be for creating file system directory-like structures. Another use of hierarchical node structures is for the representation of groups or roles for access control. A process node represents the execution of a program.

The CAIS node model uses the notion of a "relationship" for representing an interrelation between entities and the notion of an "attribute" for representing a property of either an entity or a interrelation. Some attributes and relationships are predefined; user-defined attributes and relationships are supported, as well. There are two kinds of relationships: primary and secondary. Primary relationships are restricted; each node has only one primary relationship pointing to it, although an arbitrary number of primary relationships may emanate from a node. A node may have an arbitrary number of secondary relationships emanating from it and pointing to it. Relationships can be used to build conventional hierarchical directory and process structures as well as network-like structures.

In the CAIS model, there is a single system level node with user (structural) and device (file) nodes as children. Process trees, hierarchically related collections of process nodes, are subordinate to user nodes. A user may have more than one process tree, or job. Structural and file nodes may be created as children of nodes of any kind to which the creating process has appropriate access rights.

3. TECHNICAL ISSUES

This comparison of the UNIX system calls to the CAIS began with a broad categorization of UNIX system calls and a systematic mapping of 4.2BSD UNIX operating system services to functional equivalents in the CAIS where such existed. This mapping provides a starting point for a UNIX tool writer to determine how to use the CAIS for support of tools written in Ada. An analysis of the mapping uncovers higher level areas where the differences between the CAIS and UNIX have a significant impact on tool development. The technical areas addressed are:

- File system structures
- Input/output and device control
- Access synchronization and control
- Process management
- Interprocess communication
- Error detection, recovery, and diagnosis
- Clock and timer management
- Resource control and accounting
- System administration

For each technical area, the discussion includes the support provided by UNIX and the CAIS, and the significance to the tool writer. (For a related analysis, comparing features of Version 1.2 of the CAIS with System V calls, see [MITRE]). The Appendix contains tables showing the mapping for each technical area. A UNIX system call appears in more than one table when it is significant for more than one area.

3.1. File System Structures

File system support depends upon what is to be included, such as files and devices, upon facilities available to the user for structuring interrelationships among these entities, and also upon the facilities for managing information about them. Table 1 of the Appendix contrasts UNIX and CAIS file system facilities.

3.1.1. UNIX File System Structures

The UNIX file system achieves simplicity and uniformity by limiting what it contains to files and by organizing them into hierarchical structures. UNIX file system objects are ordinary files and "special" files (e.g. devices). Directories are just files containing pointers to other files. All UNIX files are referenced uniformly, using a sequence of identifiers of directories that must be traversed to reach the file of interest.

Although UNIX provides a basically hierarchical system, there is a capability for linking from directories to existing files, permitting construction of directed graph structures. Hard links are evaluated immediately and have essentially equal status with the original, hierarchical link through which a file is created. A file is deleted when the last directory link to it is deleted. Symbolic links, intended to allow linking across (possibly unmounted) file systems, permit delayed evaluation of references to files.

3.1.2. CAIS Node Model Structures

The CAIS model includes nodes: file nodes, structural nodes, and process nodes. These can be combined in various ways to form more general file system structures, using relationships to form directed graphs on nodes and attributes to represent meta-information about both nodes and relationships. Naming is uniform in the CAIS, as well. A CAIS node is referenced by a sequence of identifiers corresponding to the sequence of relationships that must be traversed. As with UNIX, the benefits of uniformity are realized. As an example, a node can be accessed in order to read its attributes or relationships without concern for the kind of node it is.

The CAIS distinguishes between primary and secondary relationships. Primary relationships define strictly hierarchical structures, established when nodes are created (as with UNIX), while arbitrary graph structures can be constructed using secondary relationships. Renaming changes the primary link to a node and is restricted to file and structural nodes. Secondary links in the CAIS are analogous to UNIX hard links in that they are evaluated immediately and track a node that is renamed. There is no CAIS analogue of the UNIX symbolic link.

Primary links are used to enable secure node deletion. When a UNIX file is deleted, write access to the last directory in the path through which it is accessed is required, and that is the directory from which the link is removed. However, when a CAIS node is deleted, its unique primary relationship is removed, and it is unobtainable even though secondary links to it may exist. Access to write relationships is required to the parent (source of the primary relationship) of the node, even though the node to be deleted may be accessed through a secondary relationship. Secondary links to the deleted node become dangling references.

The types of nodes upon which relationships are incident is not restricted. However, UNIX-style directory structures may easily be constructed using CAIS structural nodes in place of UNIX directory files. Instead of a UNIX file having pointers to other files as its contents, the directory is represented using relationships that emanate from the CAIS structural node. CAIS relationships emanating from a node are designated uniquely by a relation name and a key.

3.1.3. Implications for the Tool Writer

For the tool writer, the issue is one of usability — the simplicity and uniformity of UNIX versus the generality and added support for file system concepts offered by the CAIS. It is apparent that the expressiveness added by relationships and by node and relationship attributes can be exploited by APSE tool writers. The CAIS provides a number of facilities for manipulating and navigating relations among nodes, and for managing attributes. An obvious benefit to

the tool writer is that these facilities make it easier to maintain explicit knowledge about the interdependencies and properties of nodes. For example, a structural node might contain a collection of Ada source files for a system under development. There are various ways that these files might be related (e.g., compilation order dependencies, membership in various subsystems, version differences). Without the use of secondary relationships, these different relations among the files may be recorded only in the mind of the system designer, in documentation, or possibly in other files. The increased accessibility of processes as entities, with relationships and attributes that can be referenced by tool writers, is also promising.

The differences in the linking of system structure are important, but there is no loss of capability, and greater control over node existence is available to the tool writer, through the CAIS. However, the use of links in the CAIS is less uniform than in the UNIX file system and linking to a node does not guarantee that a tool can traverse the node (i.e. that it will still exist) as it navigates the node model as UNIX hard links do.

3.2. Input/Output and Device Control

The input/output issues affecting tool writers involve the ease with which I/O services can be used, the variety of useful services, the degree of control afforded for devices versus the control needed, and support for correct use of services. Extensibility of the I/O services supported is important. Table 2 of the Appendix contrasts UNIX and CAIS file and device I/O facilities.

3.2.1. UNIX Input/Output

Generally, UNIX I/O services are presented through a single interface. The user views files as files regardless of the devices they exist upon. The common interface is employed by the programmer for all I/O calls. The device driver translates I/O calls to the appropriate device-specific command sequences, using underlying block and character device disciplines. If greater control of a character device is needed, user calls to *ioctl* with parameters appropriate for the device provide it. UNIX also offers additional levels of abstraction for accessing and manipulating certain device classes at a higher level through specialized services (e.g. curses, sockets).

Under UNIX, extending I/O services is a matter of adding device drivers for new device groups. Device-specific command sequences are simply passed through to the new driver via *ioctl*. Thus the UNIX tool writer can exercise direct control over special device features. Incorporation of the new device driver can be done by simply relinking the kernel and requires only object level manipulation of the system.

3.2.2. CAIS Input/Output

The CAIS generally presumes a higher level of abstraction; there is little direct control of devices comparable to the UNIX *ioctl* call. The trend seems to be a proliferation of device abstractions, as opposed to the UNIX common model. The CAIS follows Ada in providing a variety of "access methods" for using devices. Operations corresponding to particular element types and access disciplines are packaged together. Although the proposed CAIS specification is weak in this area, it is apparently intended that a file descriptor be used by a single such discipline.

Uniform conventions for extension of the CAIS I/O model are not in place yet. New packages must be added to the CAIS package specification and to the package body. The CAIS includes device abstractions for three kinds of terminals (scroll, page, and form) and for magnetic tapes. Obviously, additional interfaces may be needed for printers, plotters, and various other special devices. The addition of such interfaces was deferred in the CAIS. The CAIS also has no facilities for asynchronous or non-blocking I/O. Pragmatic questions exist about what the mechanism will be for changing or extending the CAIS specification to admit new standard device classes or access methods. That is, will a CAIS control board rule on extensions, or will there be an analogue of Appendix F [LRM] for system dependent or local extensions? To promote handling of such extensions through interfaces that conform to the CAIS approach, source distribution of implementations may be required.

3.2.3. Implications for the Tool Writer

The uniform set of I/O calls constituting the UNIX I/O interface contributes to ease of use by the tool writer. For the CAIS tool writer, the CAIS input/output disciplines correspond to the Ada [LRM] services in both style and semantics, giving access to CAIS file node contents via all the Ada I/O access methods.

The facilities for direct control of devices in the CAIS are inadequate for the needs of tool writers. At the least, the tool writer will need to be able to exploit new hardware capabilities through easy addition of new device interfaces. The recompilation cost of adding interfaces to the CAIS under its present structure, though a one-time cost, is very high.

3.3. Access Synchronization and Control

The important issues affecting access to file system objects are: what synchronization mechanisms are there to protect against conflicting concurrent access? And what control mechanisms are there to restrict the kind of access that is permitted and the users to which access is permitted? Another issue is the degree to which such services are automatic or user-controlled. Table 3 of the Appendix contrasts UNIX and CAIS access synchronization and control facilities.

3.3.1. UNIX Access Synchronization and Access Control

UNIX file access synchronization is managed through an advisory locking mechanism with two classes of locks: shared or exclusive. Requests for locks may be either blocking or non-blocking. Use of the locks is by voluntary cooperation, and access is not prevented by a process that does not use locks.

UNIX access control depends upon user, group, and public (unrestricted) visibility of files, where groups of users are defined by the system administrator. Each file has an user owner and a group owner. Permissions assigned to each of the three roles are drawn from a limited set of grantable rights: read, write, and execute/search.

3.3.2. CAIS Access Synchronization and Access Control

Synchronization of access to nodes in the CAIS encompasses file nodes, structural nodes, and process nodes. Protection against conflicting concurrent access is supported for access intents of finer granularity than UNIX offers. For example, read, write, exclusive write, read attributes, and write relationships are among the intents that may be requested. These intents allow varying degrees of synchronization protection, but they are enforced by the CAIS and are not merely advisory.

The CAIS access control mechanisms are only recommendations, but alternate mechanisms must provide all the specified interface semantics. The CAIS approach to access control is two-fold, consisting of discretionary and mandatory modes, and is compliant with the DoD Trusted Computer System Evaluation Criteria [TCSEC]. First, the CAIS provides discretionary access control based upon the identity of subjects and groups to which they belong. Access can be granted to roles: users, (program) files, and groups. The CAIS does not restrict the namespace for grantable access rights, allowing construction of named sets of access rights based upon a set of CAIS-defined intents. CAIS roles are nodes representing users, programs, and hierarchically structured groups. This form of access control is roughly analogous to the UNIX enforcement of file visibility. The second form of access control supported in the CAIS is mandatory access control based upon subject clearances or authorizations and upon classification of sensitivity of the node to be accessed. Mandatory access control is enforced using a node labeling scheme based upon predefined node attributes for object and subject classification.

3.3.3. Implications for the Tool Writer

The CAIS provides more comprehensive synchronization services to the tool writer. This can be particularly important where separately developed tool sets share some intermediate file nodes. Advisory locking among tools of a set might be employed, yet interference across tool set boundaries could occur. With the CAIS, this is not a problem as long as each tool protects itself against interference. Discretionary access control in the CAIS and UNIX access control correspond very closely, with additional support for grantable access rights and for structuring groups in the CAIS. How useful a mandatory access control mode will be for general software tool development is an open question. However, for DoD software environments, a tool writer building the project-specific tools of an APSE may require both security modes.

3.4. Process Management

For the tool writer, the process management issues include the facilities available: for process creation, for control over and access to created processes, and for controlling inheritance of the environment of the created process. Table 4 of the Appendix contrasts UNIX and CAIS process management facilities.

3.4.1. UNIX Process Management Facilities

UNIX provides a hierarchical process structure with a limited adoption capability and an inheritance by the child process of parent information in the program data space. The *fork* system call creates a child process that is a copy of the current process. The *execve* system call can be used by the child process to execute a program file, transforming the copy process into a new process. The *wait* system call allows a process to wait for termination of a child process (or a signal). The *_exit* system call causes the running subprocesses of the process to be adopted by the initialization process.

3.4.2. CAIS Process Management Facilities

The CAIS supplies an underlying process structure that is hierarchical. Adoption (or renaming) of processes is not allowed, but independent jobs may be created with execution not contingent upon the termination status of the creating process. Process trees persist in the model, even after termination, and remain subordinate to the user's node. This allows delayed access to process results. Spawned dependent processes inherit an "environment" of relationships from the parent process. Inherited are: relationships to standard and current input, output, and error file nodes, relationships to user and device nodes, access control relationships, and current job, user and node relationships.

CAIS process management facilities include calls to spawn a process that executes concurrently with the caller, to invoke a process that completes before the caller proceeds, to create a new job (root) process with independent execution, to suspend and resume named processes, and to wait for an arbitrary process to enter a terminated or aborted state.

3.4.3. Implications for the Tool Writer

A major difference between the CAIS and UNIX is in the information inherited upon process spawning. A UNIX child process inherits the entire environment of its parent. A CAIS child process inherits only a subset of the relationships and attributes of its parent. The tool writer will have to explicitly copy other attributes and relations to the new process node. The tradeoff is one of greater effort versus greater control by the tool writer. The ability to wait for arbitrary processes (not just child processes) to reach termination in the CAIS should be useful to the tool writer. The need for UNIX-style adoption should be obviated by the CAIS facility for creating independently running jobs under a user node. However, an important issue is the overhead entailed by retaining CAIS process trees for terminated or aborted processes. The tool-writer will have to explicitly manage the destruction of trees of terminated processes.

3.5. Interprocess Communication

The principal issues for interprocess communication is what the modes of communication available to the tool writer are, and what facilities or conventions there are for communicating arguments. Table 5 of the Appendix contrasts UNIX and CAIS facilities for interprocess communication.

3.5.1. UNIX Facilities for Interprocess Communication

UNIX mechanisms for interprocess communication are: signals, pipes, and sockets. Processes can communicate directly through socket and pipe interfaces, and can asynchronously signal each other. The UNIX sockets abstraction is a generalization of pipes, and subsumes them. A socket is a port at which a process may establish a connection and send or receive messages. The *pipe* and *socketpair* calls allow creation of connected ports between processes that are hierarchically related. Additionally, UNIX sockets provide facilities for explicit control over network communication. The UNIX *argv* convention for parameter passing to tools permits uniform handling of process arguments.

3.5.2. CAIS Facilities for Interprocess Communication

Interprocess communication is supported in the CAIS through "queue" file nodes that are similar to UNIX pipes. However, instead of communicating at a socket or port, the queue file node is opened, then read or written. There is no restriction that processes communicating through a particular queue file node be from the same process tree or in any other relationship. Copy and Mimic queues can be created that are initialized from or echoed by file nodes explicitly coupled with them. Signals are not supported in the CAIS. Explicit support for process management and interprocess communication across multiple hosts is deferred in the CAIS; thus, there are no explicit facilities for networking. There is also no established convention for argument passing in the CAIS.

3.5.3. Implications for the Tool Writer

The importance of asynchronous signaling to the tool writer in managing notification of processes of external or severe error conditions seems critical. The lack of this mode of interprocess communication in the CAIS appears to be a serious drawback. The *fifo* (UNIX socket and CAIS queue) communication modes both seem adequate for general tool-tool communication for processes running on the same processor. The additional linkages to files in the CAIS will be useful for some tools (e.g. for logging). Distribution is an important issue, though a deferred one in the CAIS. It remains to be seen how far transparent distribution of the CAIS on a heterogeneous network (under the present interface) will be possible and what additional control interfaces may be needed by toolsmiths. Finally, a uniform convention for access to arguments by processes is needed by tool writers; it might be incorporated into the interface set or provided as a separate standard.

3.6. Error Detection, Recovery and Diagnosis

The issues for error handling mechanisms are: how well they support detection, how exact the possible diagnosis is, and what support is available for recovery.

3.6.1. UNIX Error Handling

UNIX interfaces support error detection, diagnosis, and recovery through two mechanisms. Most system calls have one or more error returns. An error condition is indicated by an otherwise impossible return value (usually -1). A process must explicitly test the value to detect an error. Typically the only information returned is that something went wrong. The external variable "errno" provides more detailed information. In other cases, particularly for fatal errors, signals are used to interrupt the calling process.

3.6.2. CAIS Error Handling

Because the CAIS interface is an Ada language interface, error handling in the CAIS follows the Ada exception paradigm. Detection of an error causes a CAIS exception to be raised. It is the responsibility of the tool builder to provide handlers for the CAIS exceptions that may be raised by a CAIS operation. An exception that is not handled in any scope will cause the program to abort. The Ada exception model is a termination model, and exceptions are not values. Exception handlers are not parameterized. The CAIS exceptions are not very fine-grained, making diagnosis difficult. This is an area of the CAIS that is under review.

3.6.3. Implications for the Tool Writer

The CAIS error handling model, being more consistent with the Ada model, will probably prove easier to use. It also offers more services for detection and recovery than the error value approach. The present granularity of the CAIS error exception offers less support for diagnosis than do the UNIX "errno" values. A major concern is the criticality of software interrupts for managing multiple-process tools under severe error conditions.

3.7. Clock and Timer Management

The issue is whether a common "time" capability will be required by tool writers and what features are needed. Table 6 of the Appendix contrasts UNIX and CAIS time facilities.

3.7.1. UNIX Time Services

UNIX system calls are provided to set and retrieve values from an interval timer and from the system clock.

3.7.2. CAIS Time Services

The CAIS does not include interfaces for clock or timer support. Predefined attributes of process nodes, `Start_Time`, `Finish_Time`, and `Machine_Time`, are maintained by the CAIS implementation and are implementation dependent.

3.7.3. Implications for Tool Writers

Time abstractions are common in configuration management tools, DBMS managers, and "daemon" processes, among others. Portability of these tools across APSEs is compromised in the absence of a common time mechanism.

3.8. Resource Control and Accounting

The issue is whether uniformity in resource management across APSEs is an important tool portability question and what the appropriate resource abstractions should be. Table 7 of the Appendix contrasts UNIX and CAIS resource management facilities.

3.8.1. UNIX Resource Management

UNIX provides resource accounting and a quota mechanism for certain resources. The CAIS includes only minimal support for accounting, specifically, predefined file and process attributes that may be used by an implementation to record file size, I/O transactions, and process times.

3.8.2. CAIS Resource Management

There are few CAIS calls for resource management. Predefined attributes `Start_Time`, `Finish_Time`, and `Machine_Time` of CAIS process nodes can be used in resource management. User attributes and relations could be used for resource control facilities built on top of the CAIS.

3.8.3. Implications for the Tool Writer

Resource management for tool development environments is closely related to the question of project management. It may be more appropriate from the APSE point of view to provide the resource control mechanisms in that context, above the level of system calls.

3.9. System Administration

The issue is whether system administration functions are needed by APSE tool writers. Table 8 of the Appendix contrasts UNIX and the CAIS with respect to system administration facilities.

3.9.1. UNIX System Administration

UNIX provides system calls that allow rebooting of the system, mounting and unmounting of file systems by a system administrator. It also provides interfaces through which the user can access system parameters, such as system page size. A special user id is designated as the "superuser" with unrestricted access rights to system objects.

3.9.2. CAIS System Administration

The CAIS does not include a concept of "CAIS reboot" and does not provide for mountable file systems. Implementation pragmatics, defining minimum support parameters an implementation must provide are specified, but system calls for values characteristic of the underlying implementation are not provided in the CAIS. There is no explicit CAIS "superuser" concept, but a "superuser" access control role could be provided in an implementation.

3.9.3. Implications for Tool Writers

The CAIS explicitly states that certain functions, such as addition of users, are outside the CAIS. This identifies certain aspects of system administration as being outside the purview of the tool writer and not to be included in an interface set for tool support. It is not likely that the omission of these features from the CAIS will impact the capabilities needed for construction of APSE tools.

4. SUMMARY

Although there are large areas of correspondence, UNIX and the CAIS provide differing support to the tool builder in each of the areas of file system structures, input/output and device control, process management, error detection, recovery and diagnosis, access control and synchronization, interprocess communication and networking, time facilities, resource control, and system administration. UNIX appears to provide generally more complete support for input/output and device control, for network communication, for resource control, and for system administration. The CAIS supports more general file system structures, is more consistent (though perhaps more limited) in its error handling model, and offers greater support for access synchronization and control. Process management is better supported by UNIX in some respects, (particularly interprocess communication), and in others (process structuring) by the CAIS.

Beyond the technical comparison given, there are additional issues which will affect the usefulness of the CAIS or any other standard operating system interface. The early and continuing success of UNIX has been promoted by the collection of tools distributed with the operating system. The CAIS has no such tool set. However, collections of Ada tools are being assembled. Our experience so far indicates that hosting existing Ada tools on the CAIS is not difficult. As new tools that exploit the capabilities of the CAIS are built, a more complete evaluation will become possible.

BIBLIOGRAPHY

- [MITRE] Bowerman, Rebecca and Charles Howell, "A Comparison of the UNIX System Calls to the CAIS," MITRE WP-84W00467, The MITRE Corporation, McLean, Virginia, September 28, 1984.

- [ULTRIX] Digital Equipment Corporation, *ULTRIX-32 Programmer's Manual*, May 1984.

- [STONEMAN] United States Department of Defense, *Requirements for Ada Programming Support Environments, STONEMAN*, February 1980.

- [LRM] United States Department of Defense, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983, February 17, 1983.

- [CAIS] United States Department of Defense, *Military Standard Common APSE Interface Set (CAIS)*, (Draft) Proposed MIL-STD-CAIS, January 31, 1985.

- [TCSEC] United States Department of Defense Computer Security Center, *Department of Defense Trusted Computer System Evaluation Criteria*, CSC-STD-001-83, August 15, 1983.

APPENDIX

Table 1. File System Facilities		
BSD 4.2	CAIS	Comments
close	Node_Management.Close	BSD: delete a (file) descriptor. CAIS: delete node handle.
creat	Text_IO.Create, Direct_Io.Create, Sequential_Io.Create, Structural_Nodes.Create, Process_Control.Spawn_Process, Process_Control.Invoke_Process, Process_Control.Create_Job	BSD: create a new file. CAIS: file, structural, and process nodes are created by corresponding packages.
link	Node_Management.Link	BSD: make a hard link to a file. CAIS: creates a secondary relationship to a node.
mkdir	Structural_Nodes.Create_Node	BSD: make a directory file. CAIS: create a structural node.
mknod	Text_Io.Create Sequential_Io.Create	BSD: make a special file. CAIS: create a queue file node for interprocess communication; device nodes are added through a mechanism outside the CAIS.

Table 1. File System Facilities (Continued)		
BSD 4.2	CAIS	Comments
mount, unmount	No equivalent	BSD: mount or remove file system. CAIS: there is no concept of a removable file system.
open	Node_Management.Open	BSD: open a file for reading or writing, or create a new file. CAIS: open a handle to an existing node to allow access to its attributes or relationships.
readlink	No equivalent	BSD: read value of a symbolic link. CAIS: symbolic links are not supported.
rename	Node_Management.Rename	BSD: change the name of a file. CAIS: change the primary (and parent) relationships of a node.
rmdir	Node_Management.Delete_Node	BSD: remove a directory file. CAIS: delete a (structural) node.
stat, lstat, fstat	Node_Management.Kind attributes: Node_Kind, File_Kind, Queue_Kind, Terminal_Kind relationship: Access	BSD: get file status, get file or symbolic link status, get open file status. CAIS: Partially supported through predefined attributes and relationships.
symlink	Not supported	BSD: make symbolic link to a file. CAIS: symbolic links are not supported.
unlink	Node_Management.Unlink	BSD: remove directory entry. CAIS: delete a secondary relationship to a node.

Table 2. File and Device I/O		
BSD 4.2	CAIS	Comments
close	Text_Io.Close Direct_Io.Close Sequential_Io.Close	BSD: delete a (file) descriptor. CAIS: delete a file handle to a file node.
creat	Text_Io.Create Direct_Io.Create Sequential_Io.Create	BSD: create a new file. CAIS: create a file node and return an open file handle to the node.
dup, dup2	No equivalent	BSD: duplicate a file descriptor. CAIS: management of file handles is not explicit; copying is not permitted.

Table 2. File and Device I/O (Continued)		
BSD 4.2	CAIS	Comments
fcntl F_DUPFD F_GETFD F_SETFD F_GETFL F_SETFL FNDELAY FAPPEND FASYNC F_GETOWN F_SETOWN	No equivalent No equivalent No equivalent No equivalent Append_File mode End_Of_File function No equivalent No equivalent	BSD: file descriptor control. CAIS: no copying of file handles. open file handles are not inherited. open file handles are not inherited. no non-blocking I/O. can append to sequential, text files. no SIGIO, must test. no SIGIO uid or gid equivalent. no SIGIO uid or gid equivalent.
fsync	Io_Control.Synchronize	BSD: synchronize a file's in-core state with that on disk. CAIS: forces all data that has been written to the internal file to be transmitted to the external file with which it is associated.
getdtablesize	Not supported	BSD: get process's file descriptor table size. CAIS: file handle space is implementation dependent.
ioctl	package Scroll_Terminal package Page_Terminal package Form_Terminal package Magnetic_Tape	BSD: control device. CAIS: separate packages support specific functionality for classes of devices.
lseek	Direct_Io.Set_Index	BSD: move read or write pointer. CAIS: read or write position can only be directly manipulated from Direct_Io.
open	Text_Io.Open Direct_Io.Open Sequential_Io.Open	BSD: open a file for reading or writing, or create a new file. CAIS: open a file node for access to contents. File node is not automatically created if it does not exist.
read, readv	Text_Io.Get Direct_Io.Read Sequential_Io.Read Scroll_Terminal.Get Page_Terminal.Get Form_Terminal.Get	BSD: read input, scatter read input. CAIS: read file node contents, no scatter read, unbuffered terminal I/O get.
truncate	No equivalent	BSD: truncate a file to a specific length. CAIS: Not supported.
write, writev	Text_Io.Put Direct_Io.Write Sequential_Io.Write Scroll_Terminal.Put Page_Terminal.Put Form_Terminal.Put	BSD: write on a file, gather write. CAIS: write file node contents, no gather write, unbuffered terminal I/O put.

Table 3. Access Synchronization and Control		
BSD 4.2	CAIS	Comments
access	Access_Control.Is_Granted	BSD: determine accessibility of file according to mode. CAIS: check for approved access right.
chmod	Access_Control.Set_Access_Control	BSD: change mode of file. CAIS: create Access relationship from object node to role.
chown	Access_Control.Set_Access_Control	BSD: change owner and group of a file. CAIS: create Access relationships, which can have Grant attribute value of Control.
flock	Node_Management.Open	BSD: apply or remove an advisory lock. CAIS: Intent parameter on Open call is used for synchronization arbitration; not advisory.
getgid, getegid	Create a node iterator over Adopted_Role relationships	BSD: get group identity, get effective group identity. CAIS: roles are targets of Adopted_Role relationships emanating from process node.
getgroups	No equivalent	BSD: get user's group access list. CAIS: no interface for interrogating Potential_Member relation over all groups.
getuid, geteuid	Current_User relationship	BSD: get real user identity, get effective user identity. CAIS: target node is user node upon which process tree depends.
setgroups	Potential_Member relation	BSD: (superuser) set group access list of current user process. CAIS: Potential_Member relation from group structural node to role held by process governs role adoption.
setregid	Access_Control.Adopt Access_Control.Unadopt	BSD: set real and effective group id. CAIS: adopt or unadopt role, with attendant access rights to objects.
setreuid	No equivalent	BSD: set real and effective user ids. CAIS: there is no interface to change the Current_User relationship of a process.
umask	Access_Control parameter of Create subprograms	BSD: set file creation mode mask. CAIS: default Access_Control parameter for node creation is null.

Table 4. Process Management

BSD 4.2	CAIS	Comments
chdir	Set_Current_Node in package Node_Management	BSD: change current working directory. CAIS: change Current_Node relationship of current process.
execve	No exact equivalent (Spawn_Process, Invoke_Process, Create_Job in package Process_Management)	BSD: execute a file. CAIS: CAIS processes do not change into new processes, but all process creation calls execute a file image.
_exit	No equivalent	BSD: terminate a process. CAIS: no explicit call; termination of Ada main program sets process state to Terminated.
fork	Spawn_Process, Invoke_Process in package Process_Management	BSD: create a new process. CAIS: create new CAIS process node. Spawn_Process is asynchronous (fork); Invoke_Process is synchronous (fork, wait).
getpgrp	Current_Job relation	BSD: get process group. CAIS: no process group concept in CAIS; Current_Job target is root process node of process tree.
getpid, getppid	":", Parent relationship	BSD: get process identification. CAIS: pathname ":" denotes current process node; target of Parent relationship is parent process node.
kill, killpg	Suspend_Process, Resume_Process, Abort_Process in package Process_Management	BSD: signals are used to alter process, process group status. CAIS: change status of process and all dependent processes.
profil	No equivalent	BSD: execution time profile. CAIS: not supported.
ptrace	No equivalent	BSD: process trace. CAIS: not supported.
setpgrp	No equivalent	BSD: set process group. CAIS: no concept of process group in CAIS.
syscall	No equivalent	BSD: indirect system call. CAIS: Not supported.
vfork	No equivalent	BSD: spawn new process in a virtual memory efficient way. CAIS: not supported; an artifact of UNIX memory management.
vhangup	No equivalent	BSD: virtually hangup the current control terminal. CAIS: no provision for hanging up terminals is made.

wait, wait3	Await_Process_Completion (Invoke_Process) in package Process_Management	BSD: wait for signal or for child process to terminate. CAIS: wait for process associated with node to terminate or abort.
----------------	---	---

Table 5. Interprocess Communication		
BSD 4.2	CAIS	Comments
accept	No equivalent	BSD: accept a connection on a socket. CAIS: no explicit networking facilities.
bind	No equivalent	BSD: bind a name to a socket. CAIS: no explicit networking facilities.
connect	No equivalent	BSD: initiate a connection on a socket. CAIS: no explicit networking facilities.
dup, dup2	Text_Io.Open, Queue file node	BSD: duplicate a descriptor. CAIS: management of descriptors for multiple-access queues is not explicit.
gethostid, sethostid	No equivalent	BSD: get unique identifier of current host, set unique identifier of current host. CAIS: no explicit networking facilities.
gethostname, sethostname	No equivalent	BSD: get name of current host, set name of current host. CAIS: no explicit networking facilities.
getpeername	No equivalent	BSD: get name of connected peer. CAIS: no explicit networking facilities.
getsockname	No equivalent	BSD: get socket name. CAIS: no explicit networking facilities.
getsockopt, setsockopt	No equivalent	BSD: get options on sockets, set options on sockets. CAIS: no explicit networking facilities.
kill	No equivalent	BSD: send signal to a process. CAIS: no generalized signaling facilities.
killpg	No equivalent	BSD: send signal to a process group. CAIS: no generalized signaling facilities.
listen	No equivalent	BSD: listen for connections on a socket. CAIS: no explicit networking facilities.
pipe	Text_Io.Create Sequential_Io.Create	BSD: create an interprocess communication channel. CAIS: create a file node of File_Kind Queue.
recv, recvfrom, recvmsg	No equivalent	BSD: receive a message from a socket. CAIS: no explicit networking facilities.
select	No equivalent	BSD: synchronous i/o multiplexing. CAIS: Not supported.
send, sendto, sendmsg	No equivalent	BSD: send a message from a socket. CAIS: no explicit networking facilities.
shutdown	No equivalent	BSD: shut down part of a full-duplex connection. CAIS: no explicit networking facilities.
sigblock	No equivalent	BSD: block signals specified in mask from delivery. CAIS: no asynchronous signaling facilities.

Table 5. Interprocess Communication (Continued)		
BSD 4.2	CAIS	Comments
sigpause	No equivalent	BSD: atomically release blocked signals and wait for interrupt. CAIS: no asynchronous signaling facilities.
sigsetmask	No equivalent	BSD: set current signal mask. CAIS: no asynchronous signaling facilities.
sigstack	No equivalent	BSD: set and/or get signal stack context. CAIS: no asynchronous signaling facilities.
sigvec	No equivalent	BSD: software signal facilities. CAIS: no asynchronous signaling facilities. (Also, Ada has no subprogram variables, values.)
socket	No equivalent	BSD: create an endpoint for communication. CAIS: no explicit networking facilities.
socketpair	No equivalent	BSD: create a pair of connected sockets. CAIS: no explicit networking facilities.

Table 6. Clock and Timer Management		
BSD 4.2	CAIS	Comments
getitimer	No equivalent	BSD: get value of interval timer. CAIS: duration parameters allow timeout during node open; no explicit control of timer.
gettimeofday	No equivalent	BSD: get date and time. CAIS: no calendar abstraction provided.
setitimer	Time_Limit parameter of Node_Management.Open	BSD: set value of interval timer. CAIS: duration parameters allow timeout during node open; no explicit control of timer.
settimeofday	No equivalent	BSD: set date and time. CAIS: Not supported; see gettimeofday.
utimes	No equivalent	BSD: set file times. CAIS: support for capturing file creation and modification times is not provided.

Table 7. Resource Management		
BSD 4.2	CAIS	Comments
acct	No equivalent	BSD: (superuser) turn accounting on or off. CAIS: explicit accounting control is not supported.
brk, sbrk	No equivalent	BSD: change data segment size. CAIS: explicit memory management is not supported.
getpriority	No equivalent	BSD: get program scheduling priority. CAIS: priority scheduling is not supported.
getrlimit	No equivalent	BSD: get maximum system resource consumption. CAIS: limitation of resources is not supported.
getrusage	Machine_Time attribute of process nodes.	BSD: get information about resource consumption. CAIS: implementation-dependent record of process execution duration; otherwise no support for resource tracking.
quota	No equivalent	BSD: manipulate user's disk quotas. CAIS: limitation of device or object resources is not supported.
setpriority	No equivalent	BSD: set program scheduling priority. CAIS: Not supported, see getpriority.
setrlimit	No equivalent	BSD: set maximum system resource consumption. CAIS: Not supported, see getrlimit.
setquota	No equivalent	BSD: (superuser) enable/disable quotas on a file system basis. CAIS: Not supported, see quota.

Table 8. System Administration		
BSD 4.2	CAIS	Comments
chroot	No equivalent	BSD: (superuser) change root directory. CAIS: no interface provided for access to system level node.
getpagesize	No equivalent	BSD: get system page size. CAIS: no concept of virtual memory management.
reboot	No equivalent	BSD: reboot system or halt processor. CAIS: no concept of "CAIS reboot."
swapon	No equivalent	BSD: add a device for interleaved paging/swapping. CAIS: no concept of virtual memory management.
sync	No equivalent	BSD: update super-block CAIS: nothing comparable-- this is an artifact of the implementation of UNIX I/O.

SVID As A Basis For CAIS Implementation

Herman Fischer¹
Mark V Business Systems
16400 Ventura Boulevard
Encino, CA 91436
(818) 995-7671
{ihnp4, decvax, randvax} | hermix | fischer
HFischer@isif.arpa

1. Introduction

The Common Ada Programming Support Environment (APSE) Interface Set² (CAIS) is a set of interfaces, defined in Ada³, which promote the transportability of software development tools, and which enhance the ability to move project development databases between CAIS implementations. These interfaces support large scale programming projects, such as are encountered in mission critical Defense Department computer systems work.

This paper examines CAIS as it relates to the System V Interface Definition, SVID (and UNIX⁴ as a particular implementation of SVID). The paper begins by exploring why the CAIS effort exists, its goals, and the solutions it attempts to achieve which are not in today's implementations of "vanilla UNIX". Next, the paper examines the anticipated user community and why it is presumed to want CAIS. The functionalities present in the current version² and the functions left for later versions are identified. Two means of implementing CAIS-like functionality on host systems (such as UNIX) are identified; present implementations of CAIS are categorized and discussed. Finally, a comparison is made between CAIS and the European Portable Common Tool Interface

(PCTE) project, possibly one of the most ambitious and CAIS-like UNIX extensions under way.

2. Goals of CAIS

2.1 Tool Transportability and Interoperability

The primary goal of CAIS is to solve a perceived problem in DoD: a lack of tool transportability and interoperability facilities (1) among defense system support contractors, (2) between contractors and the Government, and (3) between Government entities themselves. (The UNIX aficionado might feel that he has had the answer to these sorts of problems for years; he must be reminded, however, that neither the Government nor its contractors have historically been big fans of UNIX systems, mostly because of the inability to support programming in the *very large* on what were historically small-sized UNIX systems.)

The Government is expected to spend, this year, over \$13.54 Billion on mission-critical computer software⁵ (not including business and accounting applications). At typical rates of expenditures, over 126,000 software people work on over 500 defense projects (supported by many other categories of non-software labor). Several of the projects include software deliveries of the tens of millions of source lines. This code is expected to be maintained for the lengthy lifetime of military equipment; thus the ability to have many teams work on parts of the job, at differing support sites over the

1. Mr. Fischer is chairman of the KAPSE Interface Team from Industry and Academia, and participated in the development of the CAIS.
2. Proposed MIL-STD CAIS, *Common Ada Programming Support Environment Interface Set*, Department of Defense, Ada Joint Program Office, January 1985.
3. Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.
4. UNIX is a trademark of AT&T Bell Laboratories.

5. "DoD Computing Activities and Programs, Ten Year Market Forecast Issues 1985 - 1995", Electronic Industries Association, October 1985.

SVID As A Basis For CAIS Implementation

system lifecycle, is important. (Large projects are often developed by several organizations and maintained by others. This entails a variety of computers and operating systems, and moving the project database/filesystem.)

The CAIS itself focuses on the support of software tools, in development environments. It is not intended to be a real-time or applications-supportive system, though several have suggested that CAIS facilities may apply to non-development applications too.

The CAIS currently defines an advanced filesystem (database), a process model, a security model, some device control, and some access synchronization. The difficult issue of *data interoperability* is a deferred item for the CAIS authors to tackle.

2.2 Defined in Ada

The CAIS is defined in Ada, and is intended to support tools written in the Ada language. Many of its interfaces appear in an *Ada style*, using strong typing, overloaded procedure call selection, and Ada-like packaging. There was no attempt or concern to support previous languages when CAIS was first defined; however, current interest in compatibility with other languages may influence CAIS implementations to support prior-generation languages.

2.3 Evolved from APSE Concept

In the late 1970's, Ada environment research developed the concept of a development environment architecture based on a layered model. Called the Ada Programming Support Environment (APSE), this model is shown in figure 1.

The core of the APSE is the Kernel APSE (KAPSE). It's purpose was considered novel, to encapsulate differing host machine and operating system capabilities into kernels which all had a common interface to higher level tools and user programs. The KAPSE included such general system services as file management, process control, device control, and hardware resource control.

Surrounding the KAPSE in the original models, is the Minimal APSE (MAPSE), a layer with "coding" tools such as editors,

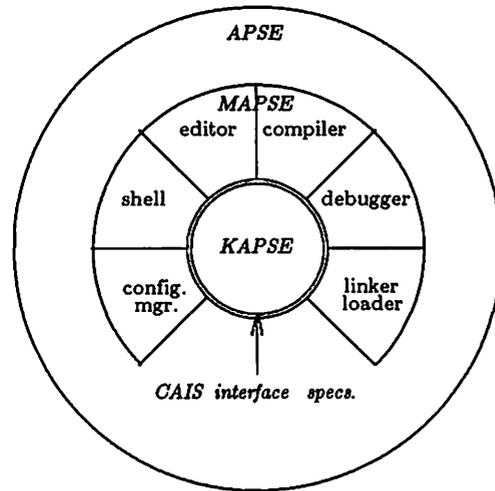


Figure 1. APSE Structure

compilers, linkers and command interpreters. Current thought focuses more on the need for an APSE to support the entire life cycle of software. This extends the support environment beyond coding tools, with such facilities as requirements analysis and design support, test support, management support, and the like. However, in the original model, these features were considered project unique tools and relegated to the APSE layer of the model.

The important contribution of this model is the idea that a kernel (KAPSE) can be defined with standardized interfaces so that low level tools (e.g., language compilers) and high level tools (e.g., project support and configuration management) can be independent of specific underlying hardware and host operating system software.

2.4 Influenced by UNIX

CAIS has been strongly influenced by UNIX. Many defense projects are still hosted on flat-file systems, such as IBM's 370 series operating systems. The CAIS designers felt the need to provide more advanced filesystem support. UNIX was seen as a model for filesystem ideas and for process control. Though UNIX was considered more advanced than many defense project environments in use, it too was perceived to have shortcomings in the area of supporting large projects. This led to the more general *node model* in CAIS.

SVID As A Basis For CAIS Implementation

Fitting UNIX-like process concepts into Ada was not straightforward. Ada implies a tasking rendezvous model, which permits only synchronous parallelisms, and then only when the parallel parts are all compiled and linked together. UNIX, on the other hand, permits asynchronous parallelisms, connected by pipes and other vehicles, where each process lives in its own address space and protected from the other. Resolving these philosophical differences was not easy.

2.5 Why Ada alone (without CAIS) is not enough

With the C language, a portion of UNIX (C library) is required to augment the machine independent portion of C with sufficient functions to be useful in an operating environment. Ada must also be augmented with functions, at least in the host system environment, because it too lacks tool support functions (of the sort provided by UNIX). Two key features absent from Ada are the underlying model of the system level data, and the ability to support dynamic binding (process control). The CAIS defines a *node model* (file system model) and a dynamically bound model for multiple independent programs to inter-react as processes in real time. CAIS augments Ada with some of the (library-level) functions found in UNIX. CAIS does not define all the tool interfaces found in UNIX, and it does not define any accompanying utility programs, user shells, and the sort of functions expected of UNIX distributions.

3. Who will use CAIS

The CAIS will appeal to suppliers, customers, and projects beset with the Government's problems: namely, supporting multiple teams of software tool users on different host products, over a lengthy software system lifecycle. It is unlikely to appeal to developers of strictly single-user products (such as single-user Personal Computer software), developers of products which require hardware lock-in in order to protect their market, or developers of closed-architecture products who feel ease of integration of foreign software products erodes their market position.

3.1 Tools and tool builders

Most tool builders today strive to support a broad base of customers on a broad class of hardware. Indeed, the popularity of UNIX implementations is due to this phenomenon. CAIS carries the UNIX notion of independence further, into the Ada domain, and into a domain of a more sophisticated database capable of supporting programming in the large. CAIS may initially appeal primarily to Government contractors, and to tool builders supplying that marketplace. However, the near equivalence of non-Ada efforts, such as the European Esprit Program's PCTE project, supported by several SVID implementations for the industrial and commercial (non-Government) market, lends credence to the need for CAIS-like system functionality.

3.2 Project environments

Large programming environments need strong configuration management, the ability to support heterogeneous hosts with the same tool base, and the need to support their tool base over a lengthy time period. During the maintenance of the software, one is likely to see four or five hardware generations, and expected reprocurments of support equipment. CAIS makes Ada tools independent of hardware and underlying host OS changes.

4. What's the Present CAIS

4.1 Node model including processes

UNIX supports its users with a strictly hierarchical filesystem. For example figure 2 shows a typical user-oriented hierarchy.

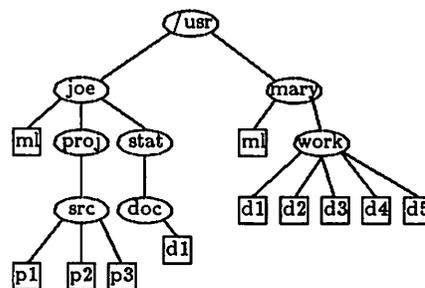


Figure 2. User hierarchy of files

Most implementations of UNIX use the directory structure to support users. Figure

SVID As A Basis For CAIS Implementation

2 shows two users, *mary* and *joe*, each with their independent hierarchy of files, independent of the work assignments and project considerations. For example, joe has a project directory, with a source directory for three programs (and presumably also has binary and test program directories for the same). Another hypothetical directory might include documentation. Whether mary is working on the same project or not, the files under her control would be in her own directory hierarchy.

CAIS supports building secondary networks of relationships, such as project directories with logical connection paths. Shown as curved arcs in the following figure, are a set of links from logical components to an owning project (regardless of which user owns them). Relationships can cover a number of logical connections, such as project ownerships, project version relationships, and the like (in a far more complex manner than in figure 3).

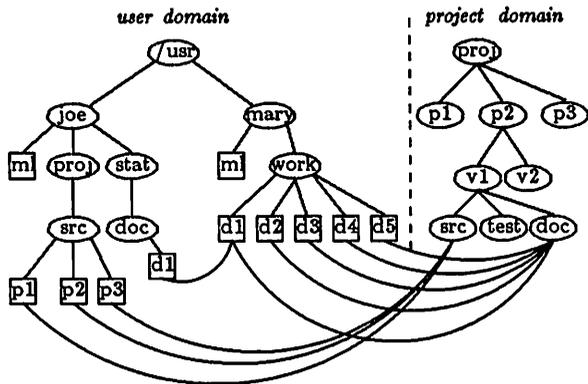


Figure 3. Network of relationships

While present UNIX distributions do not support non-hierarchical linkages or inter-filesystem linkages, some SVID extensions, such as PCTE, provide the same type of support. In general, the model underlying CAIS is one of a set of entities (e.g., tools, users, files) and their interrelationships. These may be depicted as a directed graph of nodes and edges, where the nodes represent file, device, directory, or process objects; and the edges denote relationships.

A database schema for the node model is shown in figure 4.

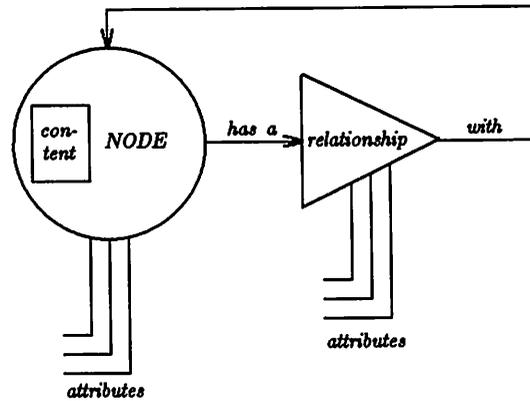


Figure 4. Database schema for CAIS node model

An important attribute of the CAIS is that processes are part of the node model. (This is similar to an enhancement to the experimental Eighth Edition of UNIX which has processes in the filesystem namespace.) With processes as named nodes in CAIS, one can have relationships between processes, between processes and file/directory nodes, and between processes and nodes for the implementation of security access models.

4.2 Terminal and Device control

CAIS defines input/output for the nodes (filesystem), as well as for terminals and tape devices. Terminals are supported as character imaging devices at present. CAIS provides support for three types of terminals: scrolling terminals, page-mode terminals, and forms-mode terminals. Scrolling terminals are basically teletype-like devices which have no cursor control. Page terminals have full screen capabilities and are equivalent to the common ANSI type of terminal (e.g., vt100). Forms terminals display fixed field menus, and receive changes to data fields, similar to some of the IBM 327x style devices.

Only rudimentary device control has been provided. For tapes, the operations provided allow the CAIS to support file creation for transport between CAIS host systems. The interfaces handle labeled and unlabeled tapes.

4.3 Security Model

CAIS provides two kinds of security access control: *mandatory* and *discretionary*. Mandatory controls, equivalent to the conventional hierarchy of UNCLASSIFIED,

SVID As A Basis For CAIS Implementation

CONFIDENTIAL, SECRET, and TOP SECRET, identify the operations of reading, writing, and reading/writing by a classifying node. Discretionary controls, equivalent to the UNIX style of user/group/other read/write/execute bits, limit the authorized access of process nodes (executing programs) (*subjects*), to other nodes (e.g., file nodes) as *objects*. Unlike UNIX, access is not controlled by storing a pattern of bits and maintaining user and group id's. Instead certain relationships are defined to other nodes to determine a node's role. Typical operations such as set-user-id are replaced by a specific process having secondary relationships such as one known as ADOPTED-ROLE.

5. What's Not in present CAIS

CAIS *does* provide many of the equivalent functions of SVID's system calls (UNIX manual chapter two); namely, typical kernel-level system services. In addition, some of the library functions (UNIX manual chapter three) are provided.

CAIS does *not* at present provide a number of deferred items. These include:

- Database Schema and Entity Typing methodology. Currently deferred is a decision whether or not the CAIS should enforce a particular typing methodology and what types of CAIS interfaces should be available to support it. Typing could range from simple schema representation of allowed relationships for classes of node linkages to a comprehensive control of process access to nodes depending on rules.
- Distribution. The existing definition of CAIS is intended to be implementable on a distributed set of processors, but in a manner which is transparent to CAIS interfaces.
- Advanced User Interfaces. The current CAIS does not provide interfaces for the establishment of windows or bit mapped displays.
- Inter-tool interfaces. The current CAIS does not proscribe the formats of data between tools, nor does it provide any interoperability data interfaces. The

equivalent of SVID file formats (UNIX manual chapter 5) has not been determined.

- Configuration management and archiving. The current CAIS interfaces support tools which implement configuration management or archiving, but there is no proscribed underlying model for such tools to follow. In a sense this is similar to the current situation with UNIX implementations, where sites individually determine tools and procedures to follow in this regard. There is an effort under way to expand CAIS to include version control.

6. How to implement CAIS

There are two ways to provide implementations of the CAIS: a native implementation within a kernel (where the CAIS is or becomes part of the host operating system), or a *piggyback* implementation on top of a host operating system or kernel. There are prototypical examples of both forms of implementation at present.

6.1 Kernel implementation

The only project under way which is in this category is the European implementations of PCTE, as modifications to UNIX System V.2 (see section 6.3). The implementations currently do not support Ada or Ada interfaces; however, the "C" interfaces provided will be shown to map cleanly into CAIS services. A CAIS implementation on top of PCTE would use Ada library routines, which translate the Ada interfaces of CAIS into underlying PCTE kernel services. This would not be called piggyback because the low level services in the kernel provide a significant portion of the functionality of the node model, without relying on superimposed user-state software to implement it.

6.2 Piggyback implementation

A piggyback implementation of the CAIS might be schematically shown as in figure 5. When implemented on a UNIX environment, the CAIS implementation exists primarily as user-state coding, generally without any changes to the underlying kernel. Either shared common processes can be used for the CAIS implementation or purely user-

SVID As A Basis For CAIS Implementation

linked coding. Two firms implementing CAIS by this technique are Mitre and Gould.

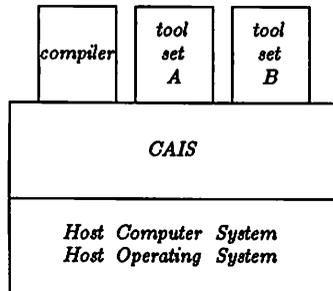


Figure 5. Piggyback CAIS implementation

6.3 PCTE

PCTE will be introduced and compared to CAIS because of two relevant points: it exists as SVID extensions, and it provides a significant part of CAIS functionality in a kernel-level implementation.

PCTE is both an interface set and a prototype implementation.

- As an interface set, PCTE exists as a set of *man pages*⁶, which describe the PCTE node model, transaction processing model, distributed processing interfaces, and user interface primitives (windowing and locator device support).
- As a prototype implementation, PCTE exists as a UNIX System V kernel extension, scheduled for test in 1986. A second implementation, known as Emeraude, seeks to provide a production quality version. The PCTE prototype is part of the EEC Esprit Program, and Emeraude is a French national project.

An additional implementation of PCTE in Ada is scheduled to be performed by Olivetti as a piggyback-styled implementation intended to be portable on a variety of hosts and processors.

6.3.1 SVID Extensions

PCTE implements a physically distributed database of objects, with a logically

distributed kernel. Figure 6 shows how three workstations might share a logical distributed kernel. In this example each workstation has some portion of the database objects physically resident in its own hardware, under the control of its own local kernel, but has transparent access to all other objects of the system-wide (homogeneous) database.

In Figure 6, *UI* represents the User Interface software function of a workstation; *objects* represent database files and attributes stored locally on a workstation; and *IKC prot.* represents the inter-kernel communications protocol.

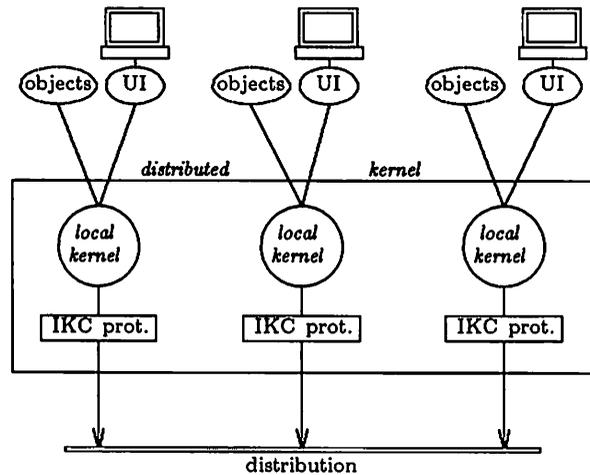


Figure 6. Distributed Kernel and Data Base

PCTE extends SVID V.2 in four logical areas. These are

1. *Basic Mechanisms.* The basic mechanisms' logical components are execution primitives, communications primitives, and inter-process communication primitives. The execution primitives, for process and context management, operate on a transparently distributed environment of heterogeneous workstations. The communication primitives provide the transparent access to distributed objects (replacing SVID filesystem primitives). The inter-process communication primitives implement piping, messages, and shared memory on a transparently distributed environment. One can start a pipeline, where pipe processes are physically separated on different

6. *PCTE, A Basis for a Portable Common Tool Environment*, Functional Specifications, Third Edition, BULL (France) et al., 1985.

SVID As A Basis For CAIS Implementation

workstations, and their objects again on different workstations.

2. *Object Management System (OMS)*. The OMS implements PCTE's equivalent of the CAIS node model. It is an Entity-Relationship model, based on a schema with typed nodes, attributes, and relationships (but without type-checking on process usage of OMS objects). The Schema is partitionable, so that logical views supportive of user or project needs can be implemented, and control of object relationships can be regulated. (E.g., an object program could have a derived-from relationship to a source program but not a mailbox file.) The OMS replaces the entire typical SVID filesystem, providing compatible interfaces so that binary code capability is retained for old programs ported to the PCTE implementation. It also adds support for the node model, relationships and attribute maintenance, and transparent distribution of objects.

The PCTE OMS also provides concurrent access synchronization, both in the form of simple locking and transaction commit/abort support (e.g., rollback of object, relationship, and attribute status to state prior to commit action if a transaction sequence is aborted).

3. *Distribution*. PCTE supports fully transparent process and object distribution. It does this with only two primitives in the entire PCTE definition which explicitly reference network nodes (for explicit starting of a process on a specific workstation in the case where several may qualify for executing a certain process).
4. *User Interface*. The User Interface functions of PCTE implement a overlapped windowing system, using mouse-like locator devices, on bit-mapped terminals. The physical terminal interfaces, in one implementation, with a User Agent function, which interfaces to applications agents for each running process. Processes can either have an active window on

the screen or be iconized (replaced by a symbol). The Applications Agent provides a virtual terminal for the application, so that user-state programs need not deal with window management.

6.3.2 PCTE and CAIS

PCTE is similar to CAIS in a number of areas:

- The node models are nearly identical.
 - The relationship models are very similar.
 - Attributes are handled in a similar manner, though schema typing in PCTE causes some practical attribute handling differences from CAIS implementations without schema support and attribute typing.
 - The Process model can be installed in a similar way. Though PCTE implements processes in the manner of System V.2 (e.g., processes are identified by identification numbers which are integers), there is precedence in experimental implementations of UNIX Eighth Edition to make Processes part of the filesystem "name space". PCTE could either inherit the mechanism of that UNIX version, or it could use a library routine (outside of the kernel) to implement processes as special types of nodes.
- Ada tasks, both on PCTE and on conventional SVID implementations, are expected to be implemented by compiler libraries which place all linked tasks for a given Ada program as a single (or set of) SVID processes. In general, it is doubtful that separate tasks can be represented by independent processes; thus the process model of CAIS can be made to correspond directly to the process model of PCTE and SVID.
- Finally, Ada implemented I/O should be the same on both PCTE and CAIS implementations, because in order to validate a given compiler, one must consistently provide Ada I/O regardless of the underlying host implementation.

PCTE and CAIS differ in several areas which are important to note:

SVID As A Basis For CAIS Implementation

- PCTE supports a concept of schemas, subschemas, and the notion of working schemas. These can be used to restrict the logical view of objects and to control relationship and attribute mapping to objects. Nodes, classes of relationships, and attributes are typed. PCTE does not, however, perform any process to object type checking during execution (there is debate as to the implementability of process to object type checking in real time).
- PCTE supports transparently distributed processing on multiple (heterogeneous) workstations and processors sharing a Local Area Network.
- PCTE provides binary code compatibility with UNIX System V.2 tools; programs which are only obtainable in binary executable forms (and cannot be recompiled or relinked) will operate properly.
- PCTE provides a windowing user interface.
- PCTE provides a SVID-like discretionary security system, which is different from the model in CAIS.
- PCTE has no software provisions for mandatory security. The certification of the SCOMP system provides some hope that a "hardware hack" could be used to implement mandatory security for a PCTE implementation. It is also possible to use the view restrictions afforded by the Schema capabilities to implement some security functionality.

In general the primitives in PCTE can be mapped to the primitives in CAIS and vice-versa. Mappings from CAIS to PCTE are nearly complete, though CAIS lacks some of the functionality provided in PCTE. It is interesting to note, however, that the differences between Ada and SVID style impact the apparent *granularity* of primitive operations. For example, let us compare opening a node handle in the two systems:

The CAIS call to open a node handle specifies a time limit, either a character pathname or a base node and relationship from that base node, and an intent specification. These are one transaction to an Ada program (though they may represent any number of low-level operations in an implementation).

The PCTE equivalent requires several kernel and user library-routine operations: allocating a current object (e.g., the node handle), performing an alarm(time limit), a function, `chrefobj()`, to make the current object equivalent to the path to the node, and a possible lock() operation. These separate operations might be necessary at the "C" interface level if the "C" user wished to perform the same operations as the CAIS Ada user.

Another visible difference between CAIS and PCTE is in the handling of errors. With CAIS, the Ada style of exception raising is used, while with PCTE, the SVID style of error returns is used. Generally most implementors of Ada compilers map the error returns of SVID implementations into exception returns anyway, so this is more a difference of language usage style than an important one. There are a few ambiguities of error return to exception mappings, but these are minor.

Process control primitives differ. For example:

In CAIS a single function call is used to spawn a process.

With PCTE, the equivalent functionality would require a `start()` (of the process), a possible `startact()` (transaction locking primitive), a `crobj()` to create the node model object representation for the process node, a number of `setattr()` calls to set the attributes up for the process, and a possible lock() call. Of course, it is quite likely that a specific CAIS implementation would break a process spawning function call down to a number of subfunctions anyway; however, the user sees a higher level of abstraction of function call. (There is debate as to the value of abstraction granularity in this regard.)

7. Conclusion

This brief report discusses why we have CAIS, how CAIS might be and has been implemented, and how CAIS is very close to the SVID extensions now being implemented. The author strongly recommends SVID as a means of implementing CAIS.

An Overview of the Ada [1] Shell

Lisa M. Campbell, Mark D. Campbell
Advanced Systems Development, NCR Corporation
Engineering and Manufacturing - Columbia
West Columbia, South Carolina 29169

Abstract

In this paper we present ash, a command language interpreter presently being prototyped for the Unix [2] operating system. Ash incorporates many of the features of existing shells while exploring new paradigms made possible through a mapping of the Ada language onto a shell. The most prominent feature of ash is the use of Ada-like control structures, much like the use of C-like control structures in the C shell. Because the Ada language definition includes the concepts associated with multi-tasking, however, the mapping of Ada control structures is more complete in ash. In addition to its command language, ash is designed to facilitate productivity by providing a very flexible interface to the system which is to a large degree user-definable.

Rather than present an overview of ash which would entail a rehash of many of the features of existing shells, we limit our discussion to those features which represent concepts associated most closely with ash. Many of these concepts are existing facilities that have been modified to more closely reflect the Ada philosophy; however, there are several concepts that reflect the influence of Ada on the Unix environment.

1. Introduction

Traditionally the command language of a particular operating system consists of a haphazard collection of features that reflects various aspects of that operating system. Examples of this are Digital's DCL for their VAX/VMS operating system and early versions of the Unix Bourne shell. With the advent of the C shell, the notion of a command interpreter

-
1. Ada is a trademark of the Department of Defense.
 2. Unix is a trademark of AT&T.

changed somewhat from an operating system-based concept to a language-based concept. Constructs like the "shell function" of the current Bourne shell and the control structures of the C shell are easily mapped onto the corresponding C function assertion and control structures.

The delineation between an operating system-based and language-based command interpreter is blurred in Unix because of the strong coupling between Unix and C. The C language is seldom thought of by users as the language as defined by Kernighan and Ritchie but rather as that language with the system and library calls provided in Unix. This coupling increases the ease in which Unix is conceptualized. This is best illustrated by the remark "If you know C [Unix] then you know more about Unix [C] than you realize." often made to users with experience in one but not the other. What we found really striking was the ease with which C users could grasp the fundamental concepts of the Unix shells such that they could begin coding moderately complex shell scripts after only a few hours of study.

While the strong coupling of Unix and C aids users' conceptualization of both, it does not aid those users who have experience in other languages but not C. These users often wish to program only at the application level in a language other than C, but must learn a great deal concerning C in order to effectively use the Unix tools. For this reason we began the design of a language-based command interpreter for Unix that was based on a language other than C. The language upon which we chose to base this command interpreter was Ada because of its scope and its expected range of use. In the tradition of the C shell (csh) and the Korn shell (ksh), we termed this command interpreter the Ada shell (ash).

Ash is designed to offer a cohesive Ada-based Unix shell which will aid Ada programmers and non-Ada programmers alike. Ada-like control structures are provided to facilitate the use of the shell. Command line editing is supported to minimize the number of wasted keystrokes. A fast, minimal help system is also supported. Good facilities from other Unix and non-Unix command interpreters have been incorporated into ash.

2. The Ada Command Language

ACL (Ada Command Language) consists of a collection of control structures and statements that may be used both batched and interactively. Similarly to the way in which the C shell supports a C language syntax, ash features an Ada-like command syntax. Builtin Ada constructs include:

```
:= (assignment)
if <condition> then ... end if
case <variable> is when ... end case
loop ... exit <condition> ... end loop
for <variable> in <first..last> loop ... end loop
while <condition> loop ... end loop
```

2.1. Assignment

Environment variables may be set using the assignment operator, much like the "=" operator of the Bourne Shell. For example, to assign a value to the variable TERM, and the terminal is a vt100, the following ash statement would be necessary:

```
TERM := vt100
```

To check a variable's value, it may be echoed by prefacing the variable with a dollar sign and using the "put" command:

```
put $TERM
```

2.2. The "if" Control Structure

The format of ash's "if" control structure is as follows:

```
if <condition> then
  <command 1>
  <command 2>
  .
end if
```

If the condition is true then "<command 1>" and "<command 2>" (and others if they exist) are executed. An "else" clause may be included to execute other commands if the "<condition>" is not true.

2.3. The "case" Control Structure

The "case" control structure provides for selective control similar to the "case" statement of the Bourne shell. The ash "case" control structure is given below, along with the Bourne shell "case" statement, to allow their structures to be compared:

case <variable> is	case <variable> in
when <value 1> => <command 1>	<value 1>) <command 1>
<command 2>	<command 2>
...	... ;;
when <value 2> => <command 3>	<value 2>) <command 3>
...	... ;;
end case	esac

Ada Shell "case"
control structure

Bourne Shell "case"
control structure

Each value in the "when" part is compared to the value of <variable>, and if they match, then the commands after the "=>" are executed. Variables may be numerical, strings, or regular expressions. If a variable's value matches none of the "when" parts, then no commands are executed except those which follow an optional "when others" clause.

2.4. The "loop" Control Structure

The simple "loop" control structure is unconditional, allowing command repetition with an "exit" command for termination. The "exit" command may specify a condition for exit and may refer to a loop by its labelled name, which may follow the word "loop". The format of the "loop" control structure and an example follow.

loop <loop_name>	loop
...	test_prog > file
exit <loop_name>	i := 'cat file'
end loop	exit if \$i = 2
	end loop

2.5. The "for" Control Structure

The "for" control structure is another type of iteration structure, used for performing a series of commands a predetermined number of times. The "<first..last>" sequence may be a range of numbers, a range of filenames in the current directory (lexicographically ordered), or all filenames in the current directory:

```
for <variable> in <first..last> loop
  ...
end loop
```

To find the pattern "inode" within all C source files in the current directory, the "for" control structure might be used as follows:

```
for i in a.c..x.c loop
    put The filename is $i
    grep inode $i
end loop
```

2.6. The "while" Control Structure

The "while" control structure is similar to the "loop" control structure except that the condition is checked at the beginning of the loop. Its format and an example follow:

```
while <condition> loop
    ...
end loop

while i /= 2 loop
    test_prog > file
    i := 'cat file'
end loop
```

2.7. The Ash Rendezvous

In Ada, intertask synchronization is achieved by "rendezvous" between a task asserting an "entry" statement and a task asserting an "accept" statement. Whichever task issues one of these statements first is queued until the other task issues the corresponding statement. At this point the body of the "accept" is executed while the task that issues the "entry" is queued. After the body of the "accept" has been executed, execution of both tasks resume again in parallel. The queueing may be altered by the "select" and "terminate" statements, which allow conditional and timed "entry" calls. Intertask communication is achieved through the use of parameters in conjunction with these statements.

The "accept", "entry", "select", and "terminate" statements are also used for intertask synchronization and communication in ash (note: pipes are also used, in the same format as in the Bourne and C shells.). Their functions are almost identical to those of the corresponding Ada statements. The closest analogy to these statements in present Unix shells are signals and signal handlers. Note that unlike signals, ash allows the queueing of an "entry" statement.

In Ada, an interrupt is defined as a low-level "entry" which may be handled by an "accept". Likewise, in ash a signal may be caught by an "accept". Because of the nature of Unix signals, it is assumed that all signals act as conditional "entry" calls. This means that the signal is not queued as an "entry" but must have an "accept" queued for it.

2.8. Other Constructs

There are many other builtin commands which are a part of ash, including:

```
> abort <process>           > terminate
> low                       > submit <command>
> delay <time>              > prompt
```

The "abort" command results in the termination of a Unix process. A process may abort any task it has permission to terminate, including itself. The "abort" facility is an instantiation of the "entry" command that sends "SIGKILL" (9) to the specified process. The "terminate" command logs the user off the system, while the "submit" command enables the <command> given as its argument to continue execution should the user "terminate". The "delay" command is similar to the C shell "wait" command, in which the user may specify a specific time for a process to suspend execution. The "prompt" command is used to define the user's prompt. Using the assignment command, the user may record the event number and the date in the prompt. The sequence

```
prompt := "Ada Shell<\!>[\!>]"
```

will yield the prompt

```
Ada Shell<3>[Fri Dec 6 20:52:26]
```

where the event number is within angle brackets and the date is within square brackets. The date will be updated each time a carriage return is received.

As has been outlined in the preceding paragraphs, ash embodies much of the control structure used in the Ada language. Although different from both the Bourne and C shells, differences in basic structure have been kept to a minimum. Ada's rich, explicit syntax allows for a more complete mapping of the language's constructs to ash control structures.

3. Command Line Editing

3.1. Visual Editing Features

Though the standard command shells are powerful, typing errors, even at the beginning of a line, are most easily altered by erasing all characters back to the erroneous one. For example, in the command

```
cpoi -iBcv < /dev/rtp,
```

in which the "oi" of "cpio" should be "io", or

```
cpio -iBcv < /dev/rtp,
```

The user must erase all the way back to "cp", then retype the remainder of the line. The C shell does provide complex

command line editing, but it is not highly visual in nature. On the other hand, ash has features which allow the user to perform interactive command line editing, such as positioning at the beginning or ending of a command line without deleting any characters. Capabilities defined are those which allow the user to go to the beginning or end of a line; to delete an entire line or from the cursor to the end of a line; to move forward or backward by character or word without deleting any existing characters; and to move up and down lines within control structures.

3.2. Renaming Commands

The C shell possesses an aliasing mechanism to enable the user to map commands to personal choices; in effect, an individual may cultivate his own command set. This is also particularly useful for persons who use many different operating systems, who may wish to map the Unix command names to a more familiar operating system's command name set. For example, a user most familiar with the VAX/VMS operating system may decide to use the following aliases:

```
alias    'delete'      'rm'
alias    'show system' 'ps -ef'
alias    'directory'   'ls'
```

Ash uses a similar mechanism, the "renames" command, to assign command synonyms. The following ash statements correspond to the C shell aliases:

```
delete      renames  rm
show system  renames  ps -ef
directory   renames  ls
```

Also, ash supports shell functions similarly to the System V.2.2 Bourne Shell, but the syntax is that of Ada.

3.3. History Mechanism

Similar to the C shell, ash uses a history mechanism to retain a list of a specific number of previous commands and their corresponding event numbers. The C shell does not retain a history of control structures, only their first line. Unlike the C shell, ash's history mechanism is divided into two parts, the line history (lhistory) and the command history (chistory). Chistory keeps a list of previous control structures (the entire structure, not just the first line), while lhistory retains one-line commands. Each history list defaults to a length of 24, the number of lines on a standard character terminal, but may be altered using an assignment command, i.e., "lhistory := 10" or "chistory := 10".

The user may invoke previously-used commands by referring to them by number; typing !38 will invoke "find . -print | cpio -oBcv > /dev/rtp" in the following lhistory list:

```
36      vi test.c
37      cc -c -O test
38      find . -print | cpio -oBcv > /dev/rtp
39      df -t
40      ld -n test.o -o test -lld
```

Also, commands may be recalled by typing the unique portion of command line, such that !l (or !ld, or !ld -n, etc.) will invoke command event 40,

```
ld -n test.o -o test -lld.
```

As with the C shell, the immediately previous command is invoked with a double exclamation (!!). Ash simplifies the mechanism which modifies other previous command lines. For example, if event 38 is to be altered so that the cpio -c option is removed, then the user may enter !38:. The colon indicates that the command is to be brought back to the command line but not invoked. Once the command is again on the command line, it may be easily edited using the command line editing features discussed previously.

4. The Ash Help Facility

The standard help facility of the Unix system is the "man" command, which displays the manual pages for the utility in question. For example, to receive information concerning the "dd" command, the user must either find the "dd" entry in the manual or use the "man" utility. In this example the user would type "man dd" and wait for the information to be displayed. Though an interactive display of the manual pages is extremely useful, drawbacks do exist. Novice Unix users, especially those that simply wish to use the system for high-level application development, usually find the manual pages both terse and difficult. In addition, on many microprocessor-based systems the interval between the time that the "man" utility is invoked and the information is displayed is quite long.

Although the "man" utility is extremely useful, a simpler and quicker utility suffices for the many of the common problems that users experience. This facility is invoked with by typing "grok". (Note: the obvious name for this command, "help", is already used.) The ash help facility acts as a quick reference to help solve many of the less complex problems a user has with Unix utilities. In cases in which a problem may not be solved using the help

facility, the manual pages can still be accessed by using "man".

Ash's help command differs from the "man" utility in a number of other important ways. For example, the "man" utility has one fixed format for its invocation, "man [options for output] <command name>". Conversely, the ash's help is hierarchical in nature, somewhat like the current VMS "HELP" command. If the user types "grok", then he will be presented with a screen of available topics upon which help may be received, and the prompt will move to the bottom of the screen, after the message "Topic?". At this point, he may type in the name of the topic for which he desires information. Upon the choice of a topic, information concerning that topic is displayed. If that topic in turn has any subtopics for which information may be displayed, the same selection process is followed. At any time, the user may either enter "quit" to exit help or "return" to backup one level in the "grok" help hierarchy (except at the top level, where quit and return will perform the same function). An example of ash's help mechanism follows, in which a user desires information concerning the "ls" utility:

Sauron<9>: grok

Available topics for which help may be obtained:

adb	dd	man	QUIT
cc	df	nroff	RETURN
col	ls	uname	

Enter command: ls

The ls utility is used to display the files in the current directory. Options are:

-a	-b	-c	-C	-d	-f	-F	-g	-i	-l	-m
-n	-o	-p	-q	-r	-R	-s	-t	-u	-x	

QUIT

RETURN

Enter option: -l

Displays files in a long format, showing size, owner, number of links, and time.

If the user types "grok ls", then the screen for "ls" appears without first showing the top-level of the help hierarchy. Likewise "grok ls -a" may be typed to receive information concerning that particular instance of the utility's invocation.

5. Syntactic and Semantic Difficulties of an Ada-Based Shell

Basing a shell upon any non-interpretive language presents both syntactic and semantic inconsistencies. The broader the scope of the language definition, the more apparent these inconsistencies become. For example, the lack of tasking support in the C language definition allows that definition to be specified in either the shell or the operating system. Conversely, tasking support is defined in the Ada language definition; thus a conflict arises between the language definition and the operating system definition of tasking.

These conflicts are intensified by the terseness of common Unix shells when compared to the Ada syntax. As an example, the common Unix shell sequences

```
ls
exec csh
```

first causes the "ls" utility to be executed and then causes the current Unix process to be overlaid with the named file, then transfers to the entry point of the file image. The corresponding ash commands take the form of Ada task specification and is given as

```
task ls
task csh is
      pragma EXEC
end csh
```

While the experienced Ada programmer new to the Unix environment would be comfortable with this syntax (after learning the semantics of the EXEC pragma) that programmer would probably soon grow tired of this lengthy utility invocation. Ash supports both types of invocations. Besides aiding the programmer during the first part of the learning curve associated with using any new operating system, the latter syntax should also be quite appropriate for writing maintainable shell scripts.

An example of the semantic difficulties associated with an Ada-based shell is the support of the Ada exception mechanism. Initially it would seem as if Unix signals and Ada exceptions could be mapped one-to-one. Upon closer examination, however, it is obvious that there are crucial differences between the two. In addition, there is the possible mapping of a signal to a low-level entry call. Ash allows both; unfortunately, neither conforms to the exact Ada definition.

6. Ash Status

Ash is still in the prototype stage, with limited in-house distribution. Acceptance of ash has been greatly facilitated by the command-line editing and novel history mechanisms which are not present in the shells currently in use at NCR Columbia (the System V.2.2 Bourne shell and the C shell). The Ada command syntax has proven to be somewhat of a hinderance in ash usage because of the relative inexperience that exists programming in Ada.

Due to the inavailability of Ada on our target machine, "C" has been used as our development language. Given the state of Ada compiler development, we do not foresee changing this language base for some time. Lex and Yacc are used for input tokenizing and parsing, respectively. Terminal independence is attained with Unix "terminfo" and "termcap" library routines for character input and output.

Code size is not yet a consideration at this stage of development; currently the total size of text, data, and bss is approximately 60K on the MC68020-based NCR Tower 32. Performance is quite good on all of the machines upon which ash has been tested: from 10MHz MC68010 to 16.7 MHz MC68020-based machines.

7. Conclusion

Unlike past operating systems, the Unix command interpreter is an application program and can be interchanged with other application programs. This is an extremely powerful feature that has yet to be fully exploited. Much like syntax-directed editors and language-sensitive debuggers, the shell can be constructed to reflect a particular language. This could later be integrated with other language-specific components of a system to provide a cohesive and comprehensive environment. Language-based shell design, while not a panacea, is an important step in the direction of language-based environments on general-purpose operating systems.

In this brief overview of ash, we have highlighted the impact that the Ada language has on shell design and on the Unix environment in general. In addition, we have demonstrated several features that, while not specific to Ada, are still fundamental to the operation of ash. While the feature content of no system and language can be expected to map one-to-one, the functionality of Ada maps remarkably well with the functionality of Unix.

8. References

1. Booch, Gary, Software Engineering with Ada,

- Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, 1983.
2. Bourne, S.R., The Unix System, Addison-Wesley Publishing Company, London, 1983.
 3. Campbell, Mark D., Kira-- An Ada Support Kernel, Master's Thesis, University of South Carolina, May 1984.
 4. Joy, William, "An Introduction to the C shell", Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley; Berkeley, California, December 21, 1979.
 5. Olsen, Eric W., Stephen B. Whitehill, Ada for Programmers, Reston Publishing Company, Inc., Reston, Virginia, 1983.
 6. Pyle, I. C., The Ada Programming Language, Prentice-Hall, Englewood Cliffs, NJ, 1981.
 7. csh, Unix System V manual, AT&T, September 12, 1984.
 8. sh, Unix System V manual, AT&T, September 12, 1984.
 9. Military Standard: Ada Programming Language, American National Standards Institute, Inc., ANSI/MIL-STD-1815A-1983, January 22, 1983.

Implementing Curses in Ada[®]

Karl A. Nyberg

Verdix[™] Corporation
14130-A Sullyfield Circle
Chantilly, VA 22021

Abstract: Prototype Ada packages and associated programs are described that can be used in the management of terminal capability databases and in the development of terminal-independent display software in Ada. These packages and programs are intended to provide the capabilities similar to the UNIX[™] terminfo utilities and curses libraries for the Ada programmer, and are intended to be sufficiently general as to be easily ported to other operating systems.

1. History

1.1. Termcap

Termcap (terminal capabilities) consists of a database describing the various capabilities of display terminals and a collection of routines for accessing those capabilities. It was originally developed at the University of California at Berkeley by Bill Joy while developing the screen editor *vi* (1) [Joy1]. The display routines were reverse engineered from the pseudo-terminal display features of the ITS operating system [Joy2]. Those routines particularly directed toward the optimization of cursor movement were packaged up by Ken Arnold and provided as a separate library, known as *curses* (3X) [Arnold], for use in other display oriented applications.

1.2. Terminfo

Like termcap, terminfo (terminal information) consists of both a database describing capabilities of terminals and a collection of routines for accessing those capabilities. It was developed by Pavel Curtis at Cornell University based upon termcap, but with additional features used (such as insert/delete line) [Curtis1], and optimization of execution time provided by compiling the capabilities database. A *curses* library utilizing the terminfo database and routines was also developed [Curtis2].

2. Ada Implementation

The Ada implementation consists of two parts - a compiler for the capabilities database and a package for accessing those capabilities from their compiled form. Each of these two parts is described in the following sections.

2.1. Capability Compilation

The concepts used in the implementation of terminfo were followed in the Ada implementation. A database is maintained that contains descriptions of terminals in a human-readable format. For each terminal in use on a particular system, that terminal's description is compiled into a separate file.

Ada is a registered trademark of the U. S. Government (Ada Joint Program Office).

Verdix is a trademark of Verdix Corporation.

UNIX is a trademark of AT&T Information Systems.

* The terminfo/curses database and routines have been taken over and supported by AT&T Information Systems in the latest release of UNIX, System V. The work reported herein is based upon the public domain version.

2.1.1. Format of the Capabilities Database

The termcap database was used as the source for terminal descriptions, and the *term (5)* format of the compiled term file was used for runtime execution. The reason for using the termcap database was one of availability. The runtime benefits of having a compiled database for accessing capabilities were sufficient to decide to compile the database. Since terminfo had a structure for compiled databases, it was decided to use that format in order to allow compatibility.

2.1.2. Type Capabilities - package caps

The capabilities list from which all software described herein was developed consists of the three capability types - booleans, integer, and strings. These capabilities are contained in a file (see example below), which is processed by tools to create the capability compiler and the type definitions for the capabilities.

```
--- begin bool
auto_left_margin,    "bw"  "bw"  cub1 wraps from column 0 to last column
hard_cursor,        "chts" "HC"  Cursor is hard to see.
--- end bool

--- begin num
columns,             "cols" "co"  Number of columns in a line
label_width,         "lw"   "lw"  # cols in each label
--- end num

--- begin str
back_tab,            "cbt"  "bt"  Back tab
set_right_margin,   "smgr" "MR"  Set soft right margin
--- end str
```

Once processed, the capabilities list results in a package consisting of an enumerated type, **capabilities**, with subtypes **boolean_capabilities**, **integer_capabilities**, and **string_capabilities**. An example of part of the resulting package appears as:

```
PACKAGE caps IS

TYPE capabilities IS (auto_left_margin, ..., set_right_margin);

SUBTYPE boolean_capabilities IS capabilities RANGE auto_left_margin .. hard_cursor;
SUBTYPE integer_capabilities IS capabilities RANGE columns .. label_width;
SUBTYPE string_capabilities IS capabilities RANGE back_tab .. set_right_margin;

END caps;
```

2.2. Terminal Description Compilation

Once the capabilities list has been compiled, the tools for compiling terminal descriptions are generated. Data structures for storage and accessing the information with the following types and variables is used in the compilation of the terminal descriptions:

```
TYPE arg IS ACCESS STRING;
```

```
TYPE boolean_array IS ARRAY (boolean_capabilities) OF BOOLEAN;
```

```
TYPE integer_array IS ARRAY (integer_capabilities) OF INTEGER;
```

```
TYPE string_array IS ARRAY (string_capabilities) OF arg;
```

```
booleans   : boolean_array := (OTHERS => FALSE);
```

```
integers   : integer_array := (OTHERS => -1);
```

```
strings    : string_array := (OTHERS => NULL);
```

An Ada package and set of routines developed for accessing terminal capabilities in a manner similar to that of the C termcap library is used to read in the capabilities, and produce the compiled terminal description.

2.3. Accessing a Terminal's Capabilities - package terminfo

The Ada package terminfo provides the same interface for the Ada programmer as the terminfo library provides for the C programmer with the following exceptions: `setupterm` works for only a single terminal at the moment - the current `/dev/tty` attached to the process, and `tputs` does not allow the definition of a routine to output a single character.

2.3.1. Available Procedures

The procedures available in the current implementation of terminfo are the procedures `tread` and `tparm`. The procedure `tread` takes as a single argument a string of the name of the terminal to be used. The environment will be searched for a variable `TERMDIR` from which to obtain terminal descriptions, and if not available, the directory `/usr/term` will be searched. The error `NO_ENTRY` will be raised if there is no compiled entry for the terminal string provided. The procedure `tparm` instantiates a particular format string and up to five parameters returning the instantiated string. All terminal capabilities are available through the variables as described above in capabilities compilation.

2.4. Curses Implementation - package curses

The Ada package curses provides essentially the same interface for the Ada programmer as the curses library provides for the C programmer. Since Ada allows default values for parameters, it was not possible to use default values for parameters (*e.g.*, what was previously the macro `addch(ch)` and the procedure `waddch(win, ch)` are now the procedure `waddch(ch : character; win : window := stdscr)`) to reduce the actual number of procedures involved by almost half. In the initial implementation, the procedures for getting and putting formatted data from and to the screen (`scanw` and `printw`) have also been left out.

2.4.1. Data Structures

The major data structure used in the implementation of curses is that of a window. In the Ada version a window is similar to the C structure described both in Appendix B of [Arnold] and in the terminfo curses implementation.

```
TYPE screen_data IS ARRAY (NATURAL RANGE <>, NATURAL RANGE <>)
  OF CHARACTER;
```

```
TYPE first_last_record IS
  RECORD
    first, last : INTEGER;
  END RECORD;
```

```
TYPE first_last_data IS ARRAY (NATURAL RANGE <>) OF first_last_record;
```

```
TYPE num_changed IS ARRAY (NATURAL RANGE <>) OF NATURAL;
```

```
TYPE real_window (firstx, firsty, lastx, lasty : NATURAL) IS
  RECORD
    minx : NATURAL := firstx;
    miny : NATURAL := firsty;
    maxx : NATURAL := lastx;
    maxy : NATURAL := lasty;
    curx : NATURAL := firstx;
    cury : NATURAL := firsty;
    data : screen_data (firstx .. lastx, firsty .. lasty)
           := (OTHERS => (OTHERS => ' '));
    fl : first_last_data (firstx .. lastx)
           := (OTHERS => (nochange, nochange));
    nc : num_changed (firstx .. lastx)
           := (OTHERS => 0);
    clear : BOOLEAN;
    leave : BOOLEAN := FALSE;
    scroll : BOOLEAN := FALSE;
    flags : NATURAL := 0;
  END RECORD;
```

Two major differences between the C implementation and the Ada implementation are evidenced here. First, the array of characters being used to store the information for the screen is now indexed as with the row index being the x axis, and the column index being the y axis. Second, the array indices are now no longer zero based, but based according to the declaration of the minimum and maximum window sizes. This has the benefit that offsets into the array no longer need be computed with each reference, but also precludes the ability to move the window to a different set of coordinates.

2.4.2. Available Procedures

There are two types of procedures available to the programmer in curses - those which interface to the operating system for facilities such as setting the terminal driver characteristics, and those which deal with the various displays. The procedures for interfacing to the operating system are currently implemented as calls to corresponding C procedures through the PRAGMA INTERFACE.

By making `stdscr` the last parameter of a several procedures, the quantity of procedures has been essentially cut in half. Of the remaining procedures, the only really interesting procedure is the refresh

procedure. The current refresh procedure is implemented using the basic redisplay algorithm presented in [Finseth]. As the remaining procedures become developed and debugged, the more advanced algorithm will be employed.

3. Results

3.1. Applications

The curses package is in use in three programs at the moment - a Towers of Hanoi example, a program to display certain mathematical functions, and a reimplementaion of a visual text editor.

3.2. Difficulties Encountered

One of the first stumbling blocks encountered was obtaining a suitable database of terminal capabilities from which to compile. Although it was clear that the terminfo concept of a compiled database would be preferable, the difficulty of obtaining an up to date version of the terminfo data and the immediate availability of an acceptable termcap database resulted in the decision to take the hybrid approach.

Originally it had been hoped to use the capabilities themselves as elements of the enumerated type in the Ada implementation. However two hurdles presented themselves in this area. First, some of the capabilities are the same when capitalization is ignored (as it is in Ada), and second, some of the capability names were themselves already reserved words in Ada (e.g., `in` and `if`). Since the capabilities list comes with a list of programming names (in addition to the terminfo and termcap designated capabilities), it was possible to use these without any difficulty.

Since Ada is intended to be more independent of the underlying operating system than other languages, it was also more difficult to access the operating system for the services necessary for setting the terminal modes. It was possible to write routines in C that performed the necessary actions, which were called from Ada via the PRAGMA INTERFACE construct in Ada.

3.3. Applicability of Ada

Although there was increased difficulty in using Ada to interface to the operating system, in other respects it was as good or better than C in this application. Where C had macros, Ada has default values for parameters. One particularly bright spot in Ada was memory management. The underlying details of memory allocation and deallocation were hidden from the programmer and no unnecessary attention to detail was required, whereas the C implementation required significant attention to the proper allocation of data space through *malloc*.

3.4. Future Directions

It has not been possible to develop any metrics for evaluating the relative performance of the Ada implementation with respect to the C implementation. However, using the profiling options of VADSTM, it has been possible to identify execution bottlenecks in the library, and target their reimplementaion for better performance. The refresh procedure would be an excellent candidate for evaluation of this form.

The new curses system is currently undergoing porting to various other systems, both UNIX (Sun, Sequent, CCI, Apollo) and non-UNIX (VMS) in order to localize system dependencies, and to investigate the portability of Ada code. All systems being ported to will still be developed using VADS.

The termcap database allows the specification of delays only at the beginning of a string capability, while the terminfo database allows delays to be contained anywhere within the capability. It would be preferable to be able to use the terminfo database because of this increased flexibility.

It would be quite interesting, especially on a multiprocessor system, such as the Sequent Balance 8000, to have a version written that uses tasking, and to observe the performance results.

4. Acknowledgements

Appreciation goes to Bill Joy for originating the terminal capabilities database and writing the original routines for working with the database; to Ken Arnold for packaging up curses as a separate library for use in other programs, and to Pavel Curtis for the concept of compiling the capabilities database into the terminfo database and the corresponding version of curses.

5. References

[Arnold] - Arnold, Kenneth C. R. C., *Screen Updating and Cursor Movement Optimization: A Library Package*, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Berkeley, California, 94720.

[Curtis1] - Curtis, Pavel, *The Curses Reference Manual*, unpublished manuscript.

[Curtis2] - Curtis, Pavel, *New Features in Curses and Terminfo*, unpublished manuscript.

[Finseth] - Finseth, Craig A., *Theory and Practice of Text Editors - or - A Cookbook for an Emacs*, Bachelor's Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, 02139, May 1980.

[Joy1] - Joy, William, *An Introduction to Display Editing with Vi*, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Berkeley, California, 94720, June 4, 1980.

[Joy2] - Joy, William, Private Communication.

NOTES

NOTES

BIG PAPER ON UNIX

Presented by Robert P. Bilveu
of the MacNeal-Schwendler Corporation
(213)258-9111

USNIX conference 17 January 1986
Denver Colorado

The title of this paper is Big Paper on UNIX. The idea behind the title is this: if large scale computer systems can be referred to as big iron, then a large scale software package can be referred to as big paper. When I first heard of this session, UNIX on Big Iron, I thought that since MSC/NASTRAN runs on almost every machine that is referred to as Big Iron I wanted to tell about my experiences with UNIX in order to promote an effective compatibility between UNIX, Big Iron, and large scale engineering application software.

I would like to give a brief explanation of what MSC/NASTRAN is, talk about some of the problems we have encountered as we ported our code to some UNIX systems, and discuss how our product looks to the end user.

I What is MSC/NASTRAN?

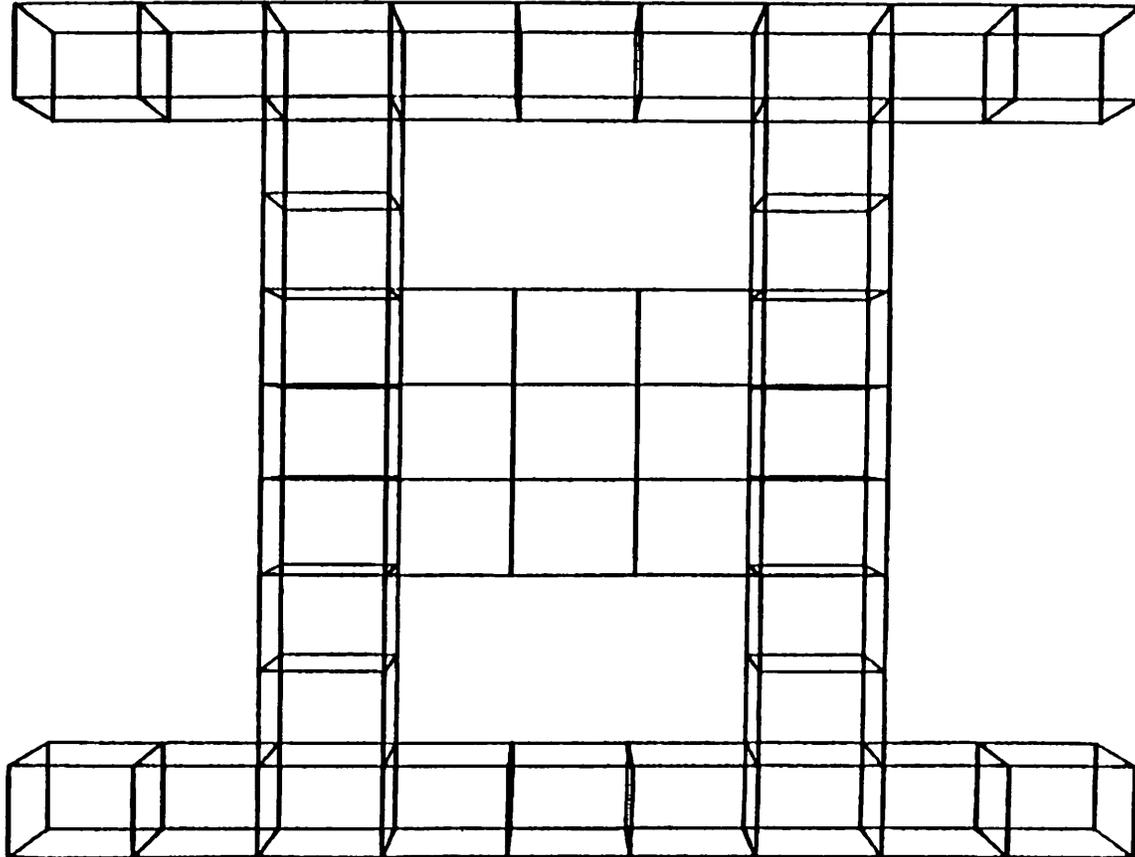
NASTRAN was originally written in the early 1960s under NASA funding. After that project was completed NASTRAN was released into the public domain. The MacNeal Schwendler Corporation took a public domain version of NASTRAN, and enhanced it. We market this improved version as MSC/NASTRAN and continue to make improvements.

MSC/NASTRAN is a general purpose structural analysis program. What this means is that if an engineer can tell our program what a structure (such as an airplane or a car) looks like, and what forces it will be subjected to (such as the vibration due to an engine or motor) MSC/NASTRAN can tell the engineer how the structure will respond to those forces.

A simple example is shown in the following three figures. A pair of square metal tubes are connected by an "H" support. The tubes are going to be twisted, and the engineer wants to know how this structure will respond. MSC/NASTRAN can predict how the structure will respond, but it is up to the engineer to interpret the results. The first figure shows the tubes and the support as separate pieces. The second figure shows the "structure" as it is before any forces are applied, and the third figure shows the structure as it looks after the forces have been applied. (These plots were generated using MSC/GRASP, an interactive pre and post processor that we ((MSC)) market.)

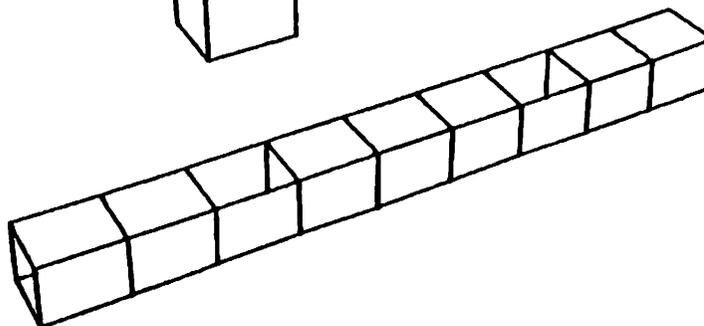
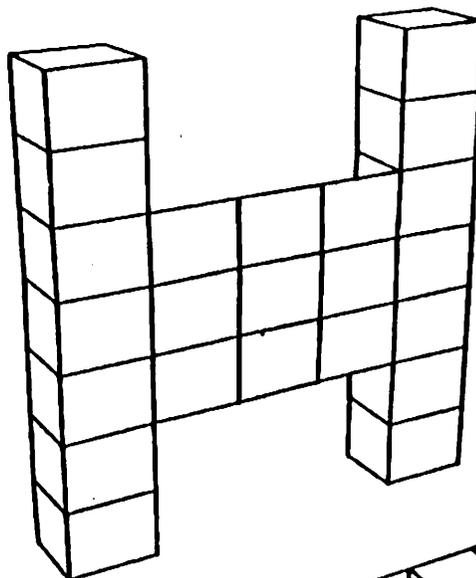
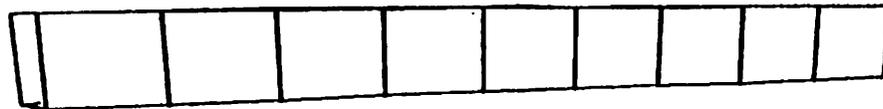
...READY
ESCAPE
...CONTROL RETURNED FROM
THE ALTERNATE FILE.
)

FIGURE - 1 (GRASP DEMONSTRATION, NORMAL MODES)



...READY
ESCAPE
...CONTROL RETURNED FROM
THE ALTERNATE FILE.
>

FIGURE - 2 (EXPLODED HIDDEN SURFACE VIEW)

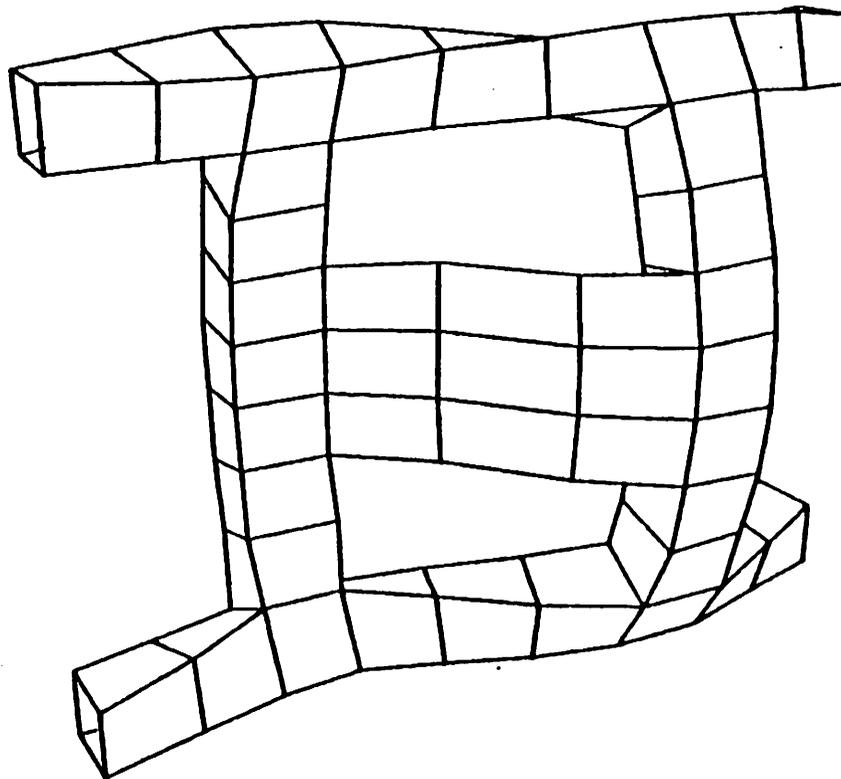


SOLUTION RESULTS FOR
MODE.....4
EIGENVALUE..... 6.00+06
FREQUENCY..... 421.07

MAX DEFORMATION. -1.77
SCALE X,Y,Z... 6.00

...READY
ESCAPE
...CONTROL RETURNED FROM
THE ALTERNATE FILE.
>

FIGURE - 3 (MODE 4 - DEFORMED STRUCTURE)



II UNIX related conversion problems.

Before I discuss the various problems that we have encountered doing ports to UNIX systems, there is an interesting observation that is worth discussing. UNIX was originally designed as a tool for doing operating system research, and is designed around interactive software written in C. MSC/NASTRAN is: an application program, not a part of an operating system; a batch program, not interactive (matrix algebra does not lend itself to interactive processing!); a FORTRAN program, not a C program. With all of these differences it would seem that UNIX and MSC/NASTRAN are not a good match, but user pressure for a 'standard' operating system and application software available on the machine of their choice seems to have made MSC/NASTRAN and UNIX partners.

It is an easy out for me to blame UNIX for the problems I have encountered during our ports, it is also easy to interpret my comments about UNIX as attacks on UNIX. I do not intend this to be the case. I am very excited about the idea of a standard operating system, my comments about UNIX are intended to point out the rough spots between UNIX and engineering software. It is my hope that, as more products port to UNIX based systems, we can learn from these experiences, and that both UNIX and engineering software will evolve into forms that are better for the end user.

There are many ways to organize the porting problems we have encountered. I will present the problems in chronological order.

1. Size. MSC/NASTRAN is written in fortran, and consists of about 2500 subroutines and contains a total of about 500,000 source lines. We like to be able to organize our source code so that each subroutine is a separate file, and all the files are located in the same directory. Some UNIX systems can not handle this many files in a single directory. This makes our maintenance more complex.

We have been able to avoid this problem by splitting our source into 26 subdirectories, based on the first letter of the file name, but this is not a very desirable solution. I would like to see UNIX raise the limit to a much larger number.

2. Fortran. The size and age of our code (some routines were first written twenty years ago) causes problems with EVERY fortran compiler we have ever used (even IBM's H extended enhanced compiler!). (We are the pathological worst case your teachers warned you about!) Many of the compiler related problems we have encountered are self inflicted wounds, but there are some common problems we have encountered with UNIX fortran compilers:

- a. Most non-UNIX fortran compilers produce listings and give cross references for variables and subroutine calls. This does not appear to be a standard feature in UNIX. The argument that such a listing can be generated by separate program is not valid for us: listings are a good place to look for compiler errors (you have to read between the lines, but it is doable), a separate program takes this capability away from me.
- b. The preferred language in most UNIX systems seems to be C. If UNIX is ever going to be a good environment for engineering software, we need a robust fortran compiler.

3. Object code. We use object libraries to help manage our code. It is difficult to maintain our object code as '.o' files, due in part to the restrictions on the number of files in a single subdirectory. (It is also not very user friendly to have to explicitly list all files needed to bind our code.) 'Ar' has been suggested as an object librarian, but it is simply not good enough. It is more concerned with remembering file names, where as an object librarian should be more concerned with entry points. If two subroutines with different associated file names have the same entry point, UNIX will not catch this; at bind time it is unpredictable which entry point will become part of the executable. Object librarians should be designed to work efficiently with the binder, I would like to see UNIX come with a true object librarian.

4. Performance. The performance of the fortran compilers, the object librarian (ie ar), and the binders on UNIX systems tend to be much slower than their counter parts on non-UNIX systems. (The measure of slowness takes into account the manufactures claim of relative performance. For example if a UNIX machine is claimed to be four times as fast as machine V, odds are that their compiler, object librarian, and binder will actually run slower than the corresponding products on machine V.)

5. System calls. An important part of our porting process involves the use of "low level system calls". There are two main problems that we have encountered interfacing with the operating system.

- a. We prefer to be able to call operating system entry points directly from fortran. Most UNIX systems require us to use a C routine to make these calls. The solution to this is to have our fortran code call a C routine that in turn calls the operating system. Normally this "bounce call" is ok, but there are times when this adversely effects performance. I would like to be able to call the operating system directly from fortran.

- b. For a variety of reasons, mostly performance related, we like to manage our I/O at a fairly low level. We like to specify the buffer location, the file name and a file position; we then want the operating system to do the I/O directly between that specified buffer and the indicated file/file location. Our code is able to locate the buffer on what ever bit, byte, word, or page boundary is required; the code is also able to use a transfer size that is right for the I/O device being used. Our experience has been that we can not do I/O at this level (there is usually a system buffer that the data must go through on its way between disk and memory), and that there is, sometimes, a buffer pool that is used. MSC/NASTRAN is set up to do I/O at a low level, and any attempts to restrict our access to this type of I/O, even if the restrictions are "for our own good", will have a negative impact on the performance of our product. I would like to see UNIX give us access to this low level I/O, and I would also like to be able to have more control over any buffer pool. (If we work together the end performance of MSC/NASTRAN will really use the maximum potential of the hardware.)

6. Debug. The last major phase of a conversion project is the debug or Quality Assurance cycle. Due to the nature of our code, the approach that works out best for us is to run a lot of problems through our code in a batch environment and check the results. If a run crashes we do not want to re-run it interactively just to see where it died, we want to be able to generate a trace back at the point of the error (ie a divide by zero or a square root of a negative number), trap the error, print our own "mini dump" and exit gracefully (ie clean up our open files). Some of these runs take hours to run, and use so much file space that it is not practical to run them during the day. It is very important to us to be able to get trace backs in a batch run, and I would like to see UNIX make this a standard capability.

Another capability we need is an interactive symbolic debugger. The features that the UNIX debugger has are sufficient, however the debugger tends to bog down with a program the size of MSC/NASTRAN. A debugger that can really handle a large code makes the conversion so much faster.

III What does MSC/NASTRAN look like to the end user?

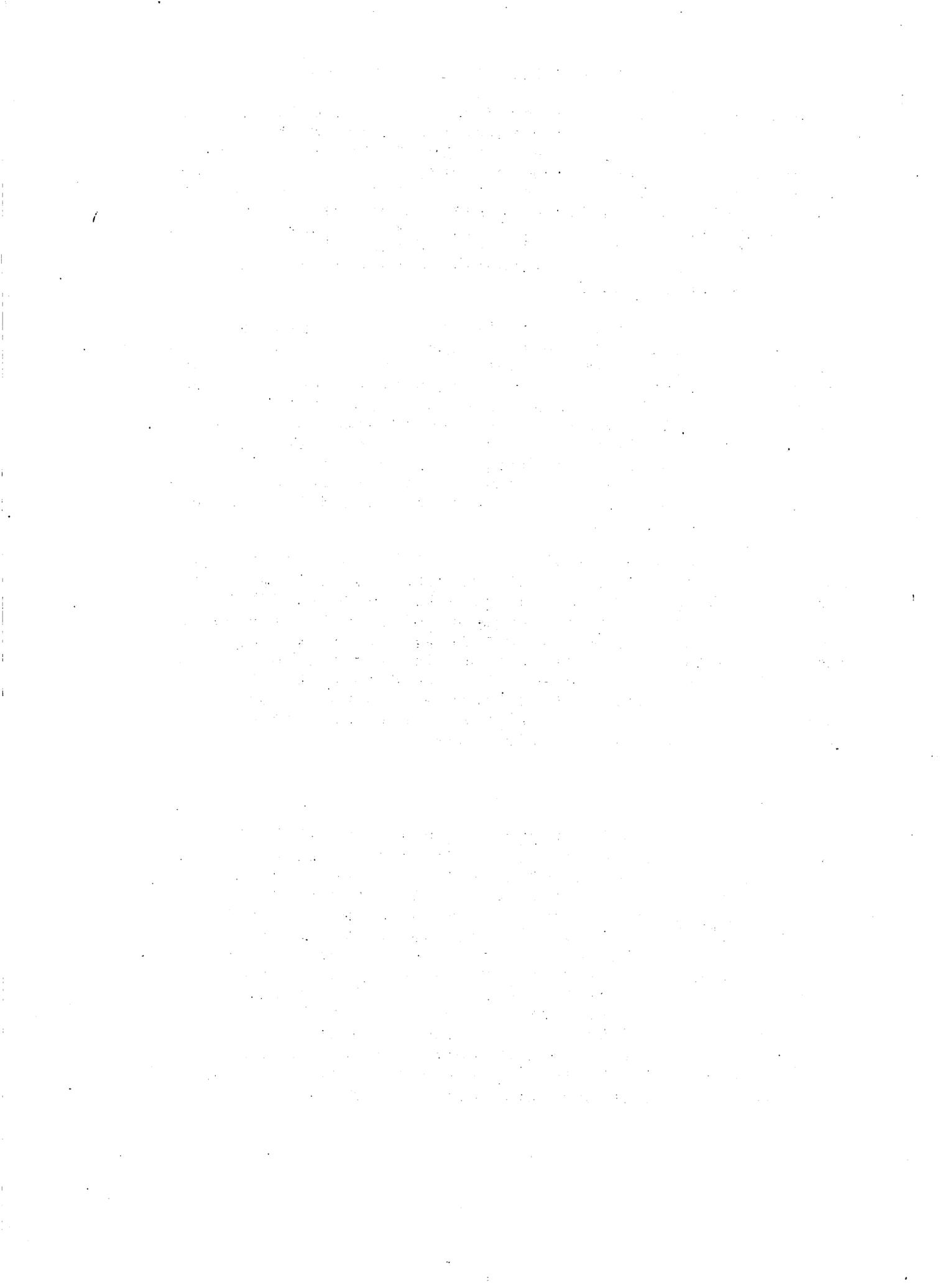
MSC/NASTRAN is a batch program, and in a multiple user environment, background is not the same thing as batch. To truly get efficient resource usage by MSC/NASTRAN, there must be a limit to the number of MSC/NASTRAN runs that are active at any one time, the priorities must be managed, and each run must not be allowed to use the entire machine. Batch queues are the best way to get the most throughput out of a machine that has MSC/NASTRAN as a significant percentage of its load. I strongly suggest that UNIX have a batch queue incorporated into it.

MSC/NASTRAN uses a lot of files: up to 16 data base files, up to 5 user defined alternate output files, up to 3 primary input files, and 2 printed output files. UNIX has the feature of being able to pipe line files, but standard UNIX only can handle one input and one output file. This leaves 24 files that must be re-directed some other way. It would be very nice if UNIX were to expand the pipe line capability in some way, to allow for more than just one input and output file. Our solution works, but it is not a simple one for the end user to go into and modify if he or she needs to change it.

MSC/NASTRAN does a lot of I/O. A file transfer size that is efficient for interactive work is usually much too small for MSC/NASTRAN. An unfortunate design in UNIX is that the file transfer size seems to be a system parameter, and once set is the same for all files. This is a serious blunder in UNIX, and if performance is at all important then different files should be able to have different file transfer sizes. Failure to do this will give end users reasons to reject UNIX on performance reasons, and purchase systems that have better performance.

IV Conclusions.

I am really looking forward to working with UNIX systems for a long time to come. There are some difficult areas relative to running MSC/NASTRAN on a UNIX based system but these problems are not unresolvable. Cooperation between application programmers and the people who set the UNIX standards will result in final products that really solve the needs of the end users. That is our primary goal: my primary product is software for structural analysis, MSC/NASTRAN is my current product; user friendly portable operating systems is your primary produce, UNIX is your current product. As the needs of the end user change, so will the form of our current products; failure to change as our end users needs change will force us out of the market, and I want to stay in the market for a long time!



CONVEX Enhancements to 4.2 BSD UNIX

Robert B. Kolstad, Ph.D. Thomas L. Watson
CONVEX Computer Corporation
701 Plano Road
Richardson, TX 75081
214-952-0351

ABSTRACT

The CONVEX C-1, the world's first affordable supercomputer, uses 4.2 BSD UNIX as its primary operating system. CONVEX's customers demanded performance and functionality enhancements to bring the operating system software's functionality to the same level as that of other supercomputer manufacturers.

This paper details some of the enhancements supported by the CONVEX C-1. Most of these enhancements are for performance or slight functionality improvements that affect portability only somewhat. They include: large real and virtual memory, hardware support for referenced bits, certain "real-time" extensions, shared and locked memory, increased I/O bandwidth and extended features, more thorough accounting and system management utilities, tools for testing and distributing software, advanced language processors, and advanced development tools for those processors.

Introduction

CONVEX's original implementation thrust for 4.2 BSD was that of compatibility. Programs which ran on 4.2 BSD VAX UNIX (and which did not depend on hardware quirks) were to run unchanged on the CONVEX UNIX system. The developers achieved this goal about the time the CONVEX C-1 was announced as the "world's first affordable supercomputer."

Customers of supercomputer vendors have demands and expectations for operating systems which differ from users of minicomputers. CONVEX found that I/O performance and virtual memory performance were issues dear to the heart of supercomputer buyers. In order to attract even more buyers, CONVEX UNIX now contains a number of performance improvements (mostly for I/O and virtual memory) and a limited set of functional additions - implemented in a way which encourages compatibility.

The sections below outline the additions and changes to CONVEX UNIX. These changes usually came about due to larger physical and logical memory, higher scalar speed, and most of all: higher expectations of a computing engine.

Virtual Memory

The CONVEX C-1 solves large mathematical problems that lend themselves to vectorization. The primary applications of a C-1 require very large address spaces - often using dozens of megabytes of physical memory and more than a hundred megabytes of virtual memory. The C-1 hardware architectural improvements provide a more hospitable environment for 4.2 BSD UNIX than the VAX 780, particularly for very large processes:

- 4 GB virtual address space (2 GB System, 2 GB User)
- Up to 128 MB of real memory
- 4 KB page size
- Hardware referenced bits
- Non-contiguous 2-level page tables

Increased Page Size

The larger (4 KB) page size of the C-1 demands less overhead in page fault processing and fewer page faults per process, at the possible expense of more wasted memory and disk bandwidth. The "clustering" of 4.2 BSD UNIX treats multiple virtual pages in the address space

as a single page, in order to gain the advantages above; the C-1's larger page size makes clustering unnecessary. A side effect of the page size choice is that one cannot create a file system on the C-1 with a block size smaller than 4K.

Hardware Referenced Bits

The C-1 hardware provides hardware referenced and modified bits for each physical memory page. Associating these bits with physical pages instead of page table entries allows a cleaner implementation of shared memory (see below.) The C-1 avoids the VAX 780 simulated reference bits and only "steals" pages that have not been referenced recently. The page-reclaim algorithms remain unchanged.

CONVEX Page Tables

The CONVEX C-1 architecture supports hierarchical page tables. Eight segments comprise the virtual address space. Segment descriptor registers point to eight first level page tables, one for each segment. Each first level page table has 256 entries that may point to a level two page table. Level two page tables each have 1024 entries each that point to a 4096 byte page.

On the C-1, the system occupies the lower half of the virtual address space (0x0 through 0x7FFFFFFF). Under UNIX, all processes share the same level one page tables for these lower four segments. Each process is allocated four level one page tables for the higher four segments. The system allocates all four contiguously (since all four fit in page), although it need not do so. It allocates level two page tables from any available free memory page. This scheme eliminates page table expansion swaps which were costly in terms of disk bandwidth and inconvenient when a process was to be locked in memory.

A surprising number of C programs erroneously depend on the fact that the address zero (a NULL pointer) is in user space on a VAX. In the C-1, reference to address zero generates a bus error, since location zero is part of the system space. CONVEX followed in other manufacturer's footsteps eliminating each of these dependencies.

The C-1's vector context includes 8 vector registers - over 8192 bytes of information. Saving this context on each process switch can be very time-consuming. C-1 UNIX exploits the

"vector valid bit" which reflects whether the current process is the one that "owns" the vector registers. All vector operations cause a trap if executed when the vector valid bit is off. The operating system then performs the context switch and resumes execution. When only a small number of people use the vector registers concurrently, this scheme provides a real savings of CPU time.

Even with all these changes and enhancements, CONVEX 4.2 BSD UNIX remains highly compatible with Berkeley VAX 4.2 BSD UNIX.

Real Time Extensions

Users of real-time processing want deterministic response to processing requests. Ways of supporting this desire include:

- Enhancing process prioritization
- Locking programs and data into memory
- True preemptive scheduling with low latency

CONVEX UNIX provides support for the first two schemes above. Traditional implementations of UNIX do not lend themselves to low latency pre-emptive scheduling since a context switch cannot occur while the kernel is servicing a system call.

New Priority Scheme

CONVEX's priority scheme is a superset of the scheme of normal 4.2 BSD. Besides the normal priority values that ranged from -20 to +20, there are additional priorities which extend the range to -64 to +64. These new priorities have special mappings which support both fixed and "variable" priorities.

Traditional UNIX priorities change as a process uses or relinquishes CPU and I/O resources. Processes which are "cpu hogs" find their priority dropping in favor of interactive processes.

The new scheme extends the normal range of priorities to -32 to +32 - these priorities operate just as before. The other new priorities (-64 to -33 and 33 to 64) have special meanings, though. These priorities map onto the extremities of the smaller range. The best extended priority (-64) is not 32 better than the best normal priority (-32). The priorities from -33 to -64 are fixed: they do not change with CPU and I/O demands. Fixed priority -33, though, corresponds to variable priority -16 or so. Likewise, fixed

priority 33 corresponds to variable priority 16 or so. Some high priority jobs with variable priority can have higher priority than some fixed priority jobs (got that?). Some high priority jobs with fixed priority always have a higher priority than jobs with variable priority. The scheme reflects similarly to the other end of the priority scale.

These new priorities allow programmers to specify whether their processes should compete with normal time-sharing or not.

Memory Locking

Real time users do not wish to wait until their programs or data are paged (or swapped) into memory. CONVEX UNIX provides both pre-paging and locking to mitigate this problem.

Two new "magic numbers" direct *exec* to load a program and its initialized data immediately; one of them also inhibits both the page daemon and the swapper from action against the particular process's pages.

Shared Memory Extensions

Sophisticated third party packages (e.g., databases) require shared memory to support tightly coupled cooperating processes. CONVEX shared memory provides a large number (e.g., 32) of memory regions located between the data and stack segments. These regions can be shared among a set of cooperating processes. The shared memory usually contains data but in this implementation can also include shared executable code.

The file system provides the name space for shared memory (the inode pointer uniquely determines the name space in implementation). Three mapping domains (file, swap, and character special device) support different kinds of mapped memory.

When sharing in the file domain, the contents of the mapped file (or a subsegment thereof) appear in each process's virtual address space. Changes (writes) to the space appear in the file automatically. References to locations beyond the length of the file may optionally extend the file. Normal read/write operations can not proceed concurrently with shared operations on that file. The kernel swaps and pages the shared memory directly to the mapped file as system demands require.

When sharing in the "swap" domain, there is no permanent final output file; pages are zero-filled-on-demand and sent to the swap area for paging or swapping.

The "character special device" domain currently supports devices that can share memory directly with a user program (i.e., provide PTE's which multiply map physical memory). The */dev/mem* and */dev/kmem* devices map segments of physical/kernel memory into user space using the dormant *d_mmap* driver entry point in the *cdevsw* table. Future devices could support bit mapped graphics by sharing the map directly with user programs.

Input/Output

The Convex Architecture uses tightly coupled intelligent I/O processors to perform I/O. These processors are connected through caches to the C-1 central memory system.

Currently there are two types of I/O processors. One supports dual Multibuses for connecting standard Multibus cards that interface variety of devices such as disk, tape, and terminal I/O. The other I/O processor type is intended as a high speed (> 40 MB/s) interface to specialized interfaces, such as devices for real time acquisition or very high speed disks.

The I/O processors run a real time executive that responds to interrupts from either the C-1 CPU or device controllers. The I/O processor can command device controllers to send DMA data directly into C-1 central memory (through a cache). The C-1 CPU uses a message based protocol to pass requests to an I/O processor, which processes the request and sends a message back to the CPU that wakes up any processes waiting for completion.

This architecture removes some of the load of I/O processing from the C-1 CPU, at a slight increase in access time to a device (each request to a device passes through both the CPU and the I/O processor).

I/O Bandwidth Improvements

Supercomputing applications need I/O rates of many megabytes per second - the vector unit is capable computing dot products at dozens of megabytes per second rate when running at full speed. Even the improved 4.2 BSD file system was not capable of I/O rates much above 0.3 MB/sec.

CONVEX Unix supports file systems with block sizes as large as 64 Kbytes, thus allowing larger contiguous disk transfers (almost three tracks on an Eagle). Fragment sizes may still range from 1/8 to 1/1 of the block size; this wastes some disk space but provides higher disk bandwidth.

Disk Striping

CONVEX UNIX can map several block devices into a single logical device. This allows a filesystem to be split over an arbitrary number of partitions, with files interleaved across different physical devices. The system manager can change the mapping while the system is up (provided the corresponding file systems are not mounted). The new system achieves bandwidths approaching 4.0 MB/sec with four striped disks and a 64K byte *bsize*, for example.

Since the stripe device is a logical one which interfaces to any other block device, CONVEX conducted the experiment of mapping two tape drives into a single logical tape drive (with twice the bandwidth). In this scheme, sequential records are alternately written to two physical tapes. No use has been found for this high speed application, unfortunately.

The system dynamically adjusts the amount of filesystem read-ahead based on the potential bandwidth of the file system (as determined by the request size and the number of disks "striped" together.) For the highest bandwidth file systems up to twelve 64K byte buffers may be requested in anticipation of future user I/O calls. The C-1 has a very large buffer cache - typically at least ten percent of the physical memory or 12M bytes for the largest configuration (experimental systems have even bigger buffers).

The vector unit copies data at 20 MB/sec between kernel and user space. It also clears pages at a rate of 40 MB/sec for newly allocated disk blocks and zero-fill pages.

Functional Enhancements

CONVEX UNIX supports asynchronous I/O through daemon processes which perform the I/O on behalf of the user process. The daemon and user processes share address space while the I/O is being performed but can be scheduled independently. User processes can then continue while I/O is in progress. The implementation of shared memory has largely obviated the need for

asynchronous I/O, as a set of cooperating user processes can easily perform asynchronous I/O.

The new block tape device driver allows varying physical record sizes, while retaining the desirable qualities of read ahead, write behind, and random read access. The physical record size of a tape is determined automatically for reads when the first record is read.

System Management Enhancements

CONVEX customers include service bureaus that run large number-crunching jobs for which they bill their users. CONVEX customers are also accustomed to having the kinds of system management software they have used in the past on bigger mainframes. This section details some of the enhancements to the CONVEX UNIX system.

Accounting

Some CONVEX jobs run for hundreds of hours and consume vast amounts of memory. The fields in `/usr/include/sys/acct.h` are too small to support the accuracy necessary to bill for CPU and kilo-core seconds. Worse, the "accounting group id" is not particularly practical for some of our customers that have many programmers who each must bill to hundreds of accounts.

CONVEX UNIX now contains an additional "activity id" field in the *proc* structure. This field complements the "group id" field to form a complete billing account. The "group id" field retains all its old meanings. New system calls allow the superuser to change a process's "group id" and "activity id" fields (which means "group id" and the group id list move to the *proc* structure). A setuid program known as *bill* verifies a user's access to an account and changes his shell process's accounting fields accordingly. This scheme has the additional features that the `/etc/group` permissions file remains relatively small and that users billing to one account do not accidentally access another account's files.

A modified version of *sa* coupled with several scripts enable the reporting of cpu time, real time, connect time, lines printed, magnetic tape mounts, disk usage, and other less interesting statistics. Customer programs read the outputs of the new utilities and then do the actual billing. The scripts report their results to files, mailboxes, notesfiles, and line printers. The disk accounting scripts even send polite letters to users

who are over quota asking them to please remove files or get bigger quotas.

System Management

A new program known as "sypic" contains all the windows from "pic" programs (like *ympic*) posted on the network as well as windows which display buffered I/O rates, teletype I/O rates, network data, disk I/O rates, IOP status, and tape I/O rates - a total of 30 windows. Other system management tools include scripts which regularly produce: daily load average charts, monthly UUCP traffic, daily bounced mail reports, system availability plots, disk layout maps, and a weekly list of modem users. The ensemble of all these tools enables system managers to watch the system on both an instantaneous and long term basis.

CONVEX customers required far more sophisticated tape-mount facilities than the "self-mounting" normally used in so many UNIX shops. New *tpalloc* and *tpdeallac* commands provide mount queues, set tape drive protection, communicate with the operator, and log billing information for tape accesses.

A filter between the line printer spooler and the printer itself remembers the last few thousand characters and will happily reprint output in the case of a printer jam.

A batch queue system supports multiple variable-width queues at dynamic priorities. Based on a new file, */etc/queuecap*, the system combines the best of the line printer spooler and the *at* command to provide a facility that works well for CONVEX's customers and internal users.

Tools & Testing

Developers of CONVEX UNIX have several tools at their disposal which aid in implementation and distribution of enhanced UNIX systems. These include: an *adb*-like debugger for UNIX as it runs on the CPU, a similar debugger for the I/O processors, large trace buffers - accessible both from the CPU and I/O processors, a module control system for releasing software, a variety of test suites, a system exerciser, a memory thrashing exerciser, regression test for system timing, and a hardware performance monitor.

The new debuggers greatly ease isolation of system problems. The C-1 architecture has a Service Processor Unit (SPU) which can stop and

restart the central processor (known as the Job Processor or JP). The SPU supports a debugger (known as *jpd*, the job processor debugger) which can not only examine memory in a running kernel but can also insert and service breakpoints and single step execution. Similarly, the I/O processors (68000 based) have a debugger known as *adb68* which performs the same functions for IOPs in a running system. This one-two punch dramatically reduces bug isolation time.

Trace buffers (just like those of Berkeley's 4.2) are bigger in CONVEX UNIX (as much as 8 MB when debugging) and are accessible from both the JP and the IOPs. This feature allows very fine grained performance monitoring for such execution paths as system calls or I/O processing.

The Module Control System (MCS) is a front end for the Revision Control System (RCS) which understands the concepts of "release trees" and "releases". Files define releases in a tree-structured fashion. Powerful MCS commands combined with tree-structured *makefiles* can release any version of an operating system, compile it, and release the object files. This automated front-end reduces errors in distribution tapes in addition to logging all changes to the system under module control.

Test suites and regression tests emerge as very important tools when supporting a large number of machines in the field. We have purchased suites such as those distributed by AIM and Perennial. Each new release goes through extensive regression testing to make sure no new bugs were introduced. The "system exerciser" is the ultimate acceptance test for each CONVEX customer. It contains copies of almost every program which has ever broken the system in addition to diagnostics specifically designed to expose hidden problems. Load averages over 100 are common when running the system exerciser. Similarly, the *thrash* program attempts to thwart the memory system's good intentions by writing pages - and later verifying their contents - in devious patterns. Similar nasty programs exist for each of the language translators (see below).

Finally, a hardware performance analyzer can count and record all kinds of system data, notably the program counter, cache hits, and functional unit status during execution. It samples every clock tick (10 MHz) and summarizes its data when an experiment concludes. A small hook in the kernel allows the performance analyzer to observe a single user processor during

normal timesharing (with the obvious caveats of virtual memory performance).

Language Processors

Language processors are the star of CONVEX's current software family. The optimizing, vectorizing FORTRAN compiler takes loops such as

```
do i = 1, 100
  a(i) = b(i) + c(i)
enddo
```

and transforms them into a simple series of vector instructions (e.g., vector load b, vector load c, vector add, vector store a). External evaluators say that CONVEX's FORTRAN is unsurpassed in its ability to detect and implement vectorization.

A new compiler exploits the same back-end technology used in the FORTRAN compiler to implement what is believed to be the world's first vectorizing C compiler. Similarly, CONVEX recently commenced development on vectorized ADA using the VERDIX company's ADA front-end.

Development Tools

An enhanced debugger (*csd* - the CONVEX symbolic debugger) and a host of profilers complement the language processors. These tools reduce the amount of programmer time required to optimize applications under CONVEX UNIX.

The *csd* debugger is a slightly enhanced version of the old *dbx* debugger - but without all the old bugs! It has all the standard source-program level data display and debugging statements in addition to new data display statements for 64-bit integers.

The profilers include the standard *prof* and *gprof* programs in addition to a new block profiler *bprof*. The *bprof* program uses the hooks required by *csd* to count entries and exits to all the "basic blocks" of source programs. This allows implementors to observe which parts of their programs are actually executed. This data aids in debugging and testing (coverage analysis is now trivial) in addition to optimization (since the profiles exposes which statements are really executed).

Some CONVEX customers have nonstandard or exotic peripherals. The V4.0 release of CONVEX UNIX allows them to write their own device drivers (both the stub in the kernel and the code in the I/O Processors). The customer can use debugging tools to load the IOP from user space during normal timesharing, communicate with the new I/O driver from user space, and even do I/O testing without rebooting the operating system. Once the new driver is tested, the programmer integrates it with the kernel and performs a sysgen. Future kernels may allow tuning of system parameters for greater performance.

Conclusion

The CONVEX UNIX system contains a number of performance and functionality enhancements. Minimal changes to the kernel and utilities maintain a high degree of compatibility with other UNIX systems.

CONVEX's customers are pleased with the functionality of their UNIX systems and perceive them to be growing into mature, commercial operating systems.

