PLEXUS

SysV UNIX System Description

98-40111.0 Ver. A March 15, 1985

PLEXUS

Sys5 UNIX System Description

SysV UNIX System Description

98-40111.0 Ver. A March 15, 1985

PLEXUS COMPUTERS, INC.
3833 North First Street
San Jose, CA 95134
408/943-9433

Printed in the United States of America

CONTENTS

1. INTRODUCTION

1.0.1 Purpose

This document describes the UNIX operating system software. Software is described both functionally (how the user sees it) and specifically (the implementation mechanisms used). Detailed descriptions of software facilities and application add-ons can be found in other documents (see the *UNIX System Documentation Catalog* or *UNIX System Synopsis*). Only those parts of the system typically associated with the operating system itself are described (that is, the kernel, file system, I/O system, and shell). More detailed information about the structure of the code can be found firsthand in the source libraries on your UNIX system. Header files also describe the layouts for many tables and data structures used by the kernel.

1.0.2 UNIX System Overview

In a UNIX System, the operating system oversees the execution of many user programs. These programs seem to execute simultaneously because of the system's ability to time-share the processor among all the programs. Actually, each program will be scheduled (at the appropriate time) to use the processor for a short period of time, to the exclusion of all other programs. This time-sharing makes it possible for many users to be using the system simultaneously. But it also makes some sort of accounting necessary to manage the system properly. The concept of a *process* was developed to allow the operating system to keep track of each program and its use of system resources. The process includes the program executing and information about the various internal parts of the processor affected by the program (such as memory registers, the name of the current directory, the status of open files, information recorded at log-in time, etc.). This concept is described in more detail in Chapter 2.

The UNIX operating system software includes: the UNIX operating system kernel, the "shell" command interpreter, the file system, and various user and system commands. The kernel (comprising from 5 to 10 percent of the operating system software) is the basic resident software on which the entire system relies. It is the only permanently resident part of the system. The kernel consists of system primitives including various facilities to maintain the file system, support system calls, and manage system resources. The shell command interpreter, which is itself a user process executing under control of the kernel, allows the user to communicate with the UNIX operating system. A user invokes processes by issuing commands to the shell. Furthermore, a user can invoke another shell process using the shell command (sh).

1.0.3 File System

The file system of the UNIX operating system consists of a highly uniform set of directories and files arranged in a tree-like structure. File structure is consistent throughout, with all I/O and file accessing handled uniformly. Files can be accessed by a "full pathname" or "relative pathname", have independent protection modes, are automatically allocated and deallocated, and can be linked across directories. The system also allows the mounting and unmounting of any number of file systems of any size anywhere in the tree structure.

1. KERNEL

1.0.1 General

The UNIX operating system kernel is the software on which everything else depends. It maintains the file system, supports system calls, and manages system resources. The software for the kernel is always resident in primary memory (once booted). This section of memory is for system use only and is not swappable.

The primary purpose of the kernel is to control system resources and user and system processes. In the UNIX system, a user executes programs in an environment called a user process. When a system function is required, the user process enters the kernel by a processor trap. During this trap, there is a distinct switch of environment. Beforehand, the process is in the "user mode"; afterwards, the process is said to be in a "kernel mode". In the normal definition of processes, the user and kernel mode are different phases of the same process (they never execute simultaneously). Each user can have many processes in the system simultaneously. Each process is a distinct entity, able to execute and terminate independently of all other processes. In fact, it is not always necessary for the user to be logged into the system while those processes are executing. From a strictly functional standpoint, it can be said there are no "system" processes; instead, there are simply processes in either a user or kernel mode. Each system process (or kernel mode of a user process) has its own stack.

1.0.2 Process Structure

When stored in primary memory, a user process occupies a specific address space. The address space associated with the process has certain access permissions for the user and the kernel. As a process changes mode from user to kernel and back, the access permissions to various structures in that address space change. Due to the different access permissions, those various structures can be visualized as being within either the user or kernel mode. These two modes can then be thought of as different entities, both associated with one process. Thus, certain structures are part of the kernel mode and certain structures are part of the user mode. This is the approach taken in the following discussion. Figure 2-1 illustrates this concept.

The user mode consists of several structures maintained by the kernel mode. These structures (the text segment, the data segment, and the stack) consist of text, data, or information needed by the kernel mode. A user mode has some private read-write data contained in a data segment. The data segment only grows or shrinks by explicit requests. It has two sections: initialized data and uninitialized data. The uninitialized data segment is called the **bss** segment and all bytes in it are set to zero when the process is created. Also associated with the user mode is a stack. The stack section can only grow. It is managed by the kernel during the execution of subroutine linkage instructions.

A process in user mode may execute from a read-only text segment, which is shared by all processes executing the same program. Use of shared text in the UNIX operating system is beneficial to the user and the kernel. The memory savings and swapping efficiencies are significant when large, commonly used programs are shared. Shared text is also useful to interactive programs that tend to be swapped while waiting for terminal input. The main benefit is that if two processes are executing simultaneously from the same copy of a read-only segment, only one copy needs to reside in primary memory. Therefore, less memory space is required.

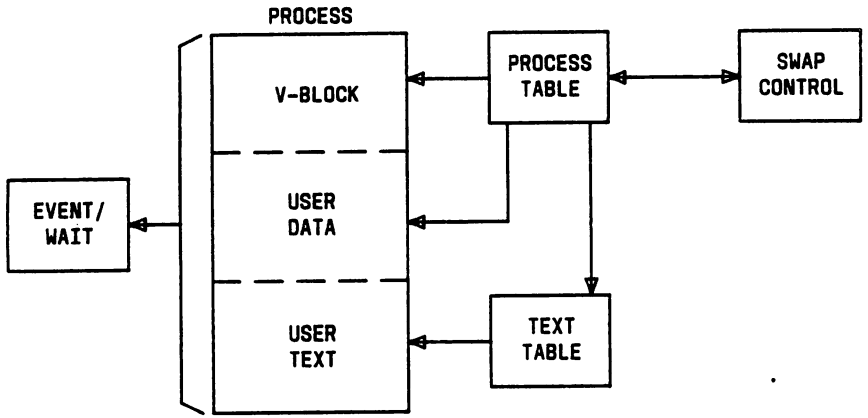

Figure 2-1. Process Control Data Structure

All current read-only text segments in the system are maintained from the text table. A text table entry holds the location of the text segment on secondary memory. If the segment is loaded, that table also holds the primary memory location and the count of the number of processes sharing this entry. If a process needs to execute a segment, the kernel will check the text table for a pointer associated with the shared text (attempting to match the pointer in the table with the pointer being used by the attaching process). This pointer is the i-node pointer discussed in Chapter 3. If a match is made, the count is incremented. Likewise, when this count is reduced to zero, the entry is freed along with the primary memory holding the segment. If the kernel does not find a text table entry with a matching pointer, a new entry is created and the text is loaded from the file system. In the file system, the text is stored in the noncontiguous blocks of a file and must be loaded one block at a time. Once loaded, the text segment becomes one contiguous image that can be copied much more efficiently.

Actually, the kernel maintains two different counts: the number of memory resident processes using the shared segment, and the total number of processes using the shared segment (resident and swapped). The resident count is decremented as each user is swapped out; and eventually, when the resident count reaches zero, the shared segment is swapped. The kernel maintains a flag in the i-node table to indicate whether or not a current copy of the shared segment is in swap space on secondary memory. If there is a current copy available, the shared segment need not be copied out and the swap merely releases the primary memory. This mechanism works due to the fact the shared text is write protected and the copy being used will not become out of date. The total user count is decremented

as each user detaches from the shared text (no longer is executing it); and eventually, when the total count reaches zero, the secondary copy is discarded. Here again, the kernel maintains a "sticky bit" to indicate whether or not this copy should be discarded. This sticky bit is part of the shared text executable image. Sticky text will only be forced out of memory by specific system calls during such operations as unmounting a file system.

Also associated and swapped with the user mode is a small fixed-size system data segment called the *u_area* or *u_block*. This segment contains all the data about the environment of the mode (such as attributes and interfaces). Examples of the type of data contained in the system data segment are: some central processor registers, open file descriptors, accounting information, and the scratch data area. The *u_area* segment is not addressable from the user mode and is therefore protected. These structures are only accessible to the kernel mode when the process is running. They are mapped into the operating system address space during the switch.

Finally, tying all these structures together is a process table with one entry for each active process. This process table is the focal point of all kernel access to the structures mentioned before. As long as the process is in the system, it has an entry in the process table. Essentially, removing a process' entry in the table is the final act that terminates that process. Therefore, not all processes listed in the process table need to be running (for example, it could be sleeping). "Sleeping" processes are in user mode in a non-runnable state waiting for some "event" to occur so they can be scheduled again. The process switch mentioned above and the various states of a process are discussed under *Scheduling and Priority*. Each entry contains all the data needed by the system to manage the process. Examples of this data are: the process id, the location of the other segments, and scheduling information. The process table entry is allocated when the process is created and freed when the process terminates. This process entry is always directly addressable by the kernel. The process table is the definition of all processes because all the data associated with a process may be accessed using the process table entry as the starting point.

The kernel maintains a text segment, data segment, and stack associated with the process. These structures are not within the user mode, but rather, contain information that the kernel mode needs to manage the user mode. The kernel stack is saved when switching user processes. Since a process can be (and, in fact, ultimately is) suspended while in the kernel mode, this stack must be saved to allow resumption of the kernel mode when the process is restarted.

1.0.3 Process Creation

A process is the execution of an image; most UNIX operating system commands execute as separate processes. Processes are created (or spawned) by the system primitive **fork**. Overlays, performed by the **exec** system call, do not create new processes (**exec** is discussed later). But, execution of a shell command or shell procedure involves both a fork and an overlay. Command execution is discussed under *Shell*. Figure 2-2 illustrates the sequence involved in executing a program. A typical fork is shown as the first step. The newly created process (child) is a copy of the image of the original process (parent). If the parent process is executing from a read-only text segment, the child will share the text segment. In other words, the child is executing the same program. Copies of all non-resident parts of the parent process (such as the user data and stack segments) are made

for the child process. The child process does receive a new *u_area*; but, most of the variables in it are copied from the parent's *u_area*. A child inherits its parent's permissions, working directory, root directory, open files, etc. This mechanism permits processes to share common input streams in various ways. Files that were open before the **fork** are shared after the **fork**. The processes are informed as to which is the parent and which is the child. Once the copying is completed, the new (child) process is placed on the runnable queue to be scheduled. The parent will then continue running; although, the parent may **wait** for the termination of any of its children. Usually the parent will wait for the death of its child at some point, since this **wait** call is used to free the process table entry used by the child. See the discussion on process termination for more detail.

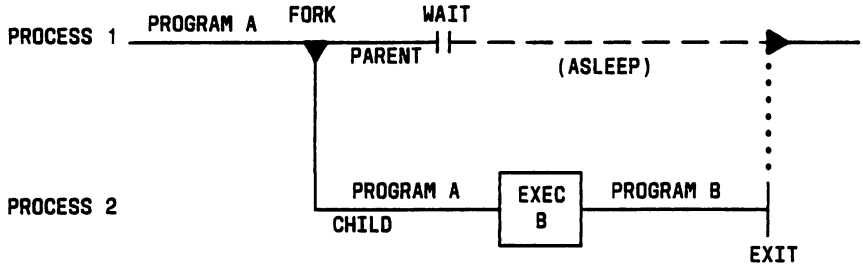

Figure 2-2. Executing a Program

In some cases there will not be enough primary memory to copy the parent. In these situations, the parent will be copied out to secondary memory. The secondary memory copy will be marked as the child, ready to run when it is swapped in. The parent can then continue to run from the primary memory image. This function is similar to what swapper does. However, in this case the primary image is not discarded, but continues to run as the parent. In any case, the child and parent differ in three ways:

- The child will have a different process id.
- The child will have a different parent process id.
- All accounting variables are reset to appropriate values in the child.

1.0.4 Signals

The kernel provides several means by which processes can communicate with each other including pipes, messages, shared memory, semaphores, and signals. Signals are the most frequently used means for a process to indicate the occurrence of some event that may have an impact on another process. There are two specific system calls involved in inter-process signaling: the **kill** system call used to send a signal, and the **signal** system call used to specify how the signal will be handled. Signals interrupt the normal flow of control in a process. They can affect both the user and the kernel mode. If a process is in kernel mode, the signal will not interrupt the process at least until **sleep** is called. While a signal will interrupt the kernel mode only at certain points, the user mode must be prepared to handle a signal at any time. A process must prearrange how it will handle signals via the **signal** system call (the default action is termination). There are two categories of signals, those generated externally, such as *break* from a terminal, and those generated internally (a process fault). Both types are treated identically. There are several ways a signal can be generated, some of which are:

- A user mode attempting to write into protected memory.
- An error during a system call.
- Some condition raised at the controlling terminal of a process (such as *break* or *hangup*).
- An explicit system call to **kill**.
- Expiration of the alarm clock timer or the generation of the trap signal during process tracing.

Signals do not directly affect the execution of a process; but rather, request that the process take some action. The occurrence of the signal is recorded in the process table entry of the receiving process, and is later recognized and acted upon by that process. During the posting of a signal, if it is sleeping at a low enough priority, the receiving process is made runnable without a call to **wakeup** (that is, **swtch** is called directly). This call to **swtch** puts the process on the run queue so that it can be scheduled, and thus, find the signal. It has not called **wakeup**, however, and is really still a sleeping process. If the signal is to be ignored, no action is taken and the process continues sleeping. Signals are posted when they occur, and are handled when the receiving process finds them. It is possible that the signal will not be found until the completion of a system call, the occurrence of a process fault, or the resumption of a preempted user mode. When the process finds a signal, execution may be interrupted immediately; or, if the process is sleeping with a low enough priority, it may prematurely return from **sleep** as shown above and branch directly to some signal-handling routine.

Signals are represented by integers. Each type of signal is associated with a specific integer; for example, the **hangup** signal is the number 1. The signal number is used as an index into the signal array in the receiving process's *u_area*. This array contains the addresses of the appropriate signal-handling routines (assuming the user mode has defined these routines). If no routine has been defined, the entry will be 0 or 1. If the value is one, the signal is set to be ignored; and if zero, the default action will be taken. When the signal is recognized (and if it is not to be defaulted or ignored), the associated address replaces the program counter value saved in the user mode. The original program

counter value is placed on the stack in the user mode. When the mode resumes, the handler routine will be accessed first. Finally, when the handler is finished, it will return using the program counter value on the stack. A signal may be sent to a process by another process, from the terminal, or by the UNIX system itself. For most signals, a process can arrange to be terminated on receipt of a signal, to ignore it completely, or to catch it and act appropriately. For example, an **INTERRUPT** signal may be sent by depressing an appropriate key on the terminal (delete, break, or rubout). The action taken depends on the requirements of the specific program being executed. For example:

- The shell invokes most commands in such a way that they stop executing immediately (die) when an interrupt is received. For example, the **pr** (print) command normally dies, allowing the user to stop unwanted output.
- The shell itself ignores interrupts when reading from the terminal because the shell should continue execution even when the user terminates a command like **pr**.
- The editor **ed** chooses to catch interrupts so that it can halt its current action (especially printing) without allowing itself to be terminated.

A child process inherits the actions of the parent for the defaulted and ignored signals. Caught signals are reset to the default action in the child process. This is necessary since the address linkage for signal-handling routines specified in the parent are no longer appropriate in the child.

1.0.5 Scheduling and Priority

Process scheduling allows many processes sharing one Central Processing Unit (CPU) to be synchronized. It is accomplished with the sleep/wakeup mechanism. This mechanism allows coordination and optimization of the actions of some unknown number of processes. All active processes will be either sleeping (non-runnable and waiting on some event), on the run queue, or actually running (only one process at a time can run). Any process may be resident (in primary memory) or swapped (in secondary memory) at any point in time. The run status and residence status is maintained from the process table.

The **sleep** call involves two steps: first it records which event will wake up the process, and then calls the **swtch** primitive. Likewise, the **wakeup** call involves first, reading the event set by **sleep** and then calling the **setrun** primitive. While **swtch** makes a process non-runnable, **setrun** will make a process runnable. Under certain situations, both can be used directly without calls to sleep/wakeup, but usually are not. When a process recognizes that a needed resource is unavailable, it will call **sleep** and wait for an event indicating that resource is available. Included in this call is a priority value to be used by the scheduler when the process wakes up. Its relative value to other priorities in the system is a measure of the importance of the associated resource. As an example, the structure used for process tracing is deemed less important than system buffers. Thus, a process needing a buffer will call **sleep** with a higher priority than another process that just went to sleep while tracing. The process selected out of the run queue to be switched to will have the highest priority. The only action of an event is to change a set of processes from the sleep state to the wakeup (or runnable) state. The sleep/wakeup software in the kernel is like a co-routine linkage. At any time, all but one process has either called **sleep** or is waiting on the run queue to be scheduled. The remaining process is the one currently executing. When that

process (or possibly an interrupt routine) recognizes that some resource has become available, it will call **sleep** and signal the event associated with the resource. Any process waiting on that event will wake up and be placed on the run queue. This reserves that event for any particular process. The event indicates the resource is available. But, if one process takes the resource before some other runnable process wanting that resource can be scheduled, the lower priority process will have to return to sleep and wait on the event again. By having processes which are not runnable wait for events, process execution can be synchronized. The sequence that occurs when **sleep** or **wakeup** is called is shown in Figure 2-3.

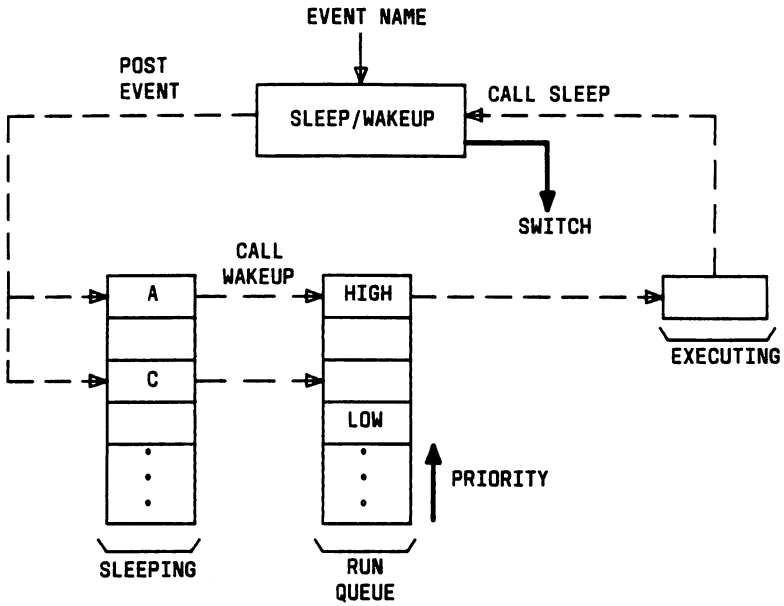

Figure 2-3. Sleep/Wakeup Sequence

An event is represented by an arbitrary integer. There is no memory associated with that event. Signaling an event on which no process is waiting has no effect. Similarly, signaling an event on which many processes are waiting will wake all of them up. No notion of quantity can be signaled via the event mechanism. For example, processes that want memory might wait on an event associated with memory allocation. When any amount of memory becomes available, the event will be signaled. All the competing processes will then wake up to contend for the new memory. In reality, the swapping process is the only process that waits for primary memory to become available. If an event occurs between the time a process decides to wait for that event and the time that process enters the sleep state, then the process will wait on an event that has already happened (and may never happen again). But since processes are switched in the kernel mode by explicit calls to **swtch** and due to the ability to delay servicing of interrupt requests, this never becomes a problem. The event cannot "occur" until after the process enters the sleep state.

Which of the many possible processes is to run next? Associated with each process is a priority. The priority of a system process is assigned by the software issuing the wait on an event. This is roughly equivalent to the response that one would expect on such an event. Disk events have high priority, interactive terminal events are low, and alarm events (performed at a certain time of day) are very low. All user process priorities are lower than the lowest system priority. This means kernel processes will always run before any user process. Also, the priority of kernel processes are not affected by CPU usage. Therefore, their priority remains constant unless a return to user mode is made, in which case the algorithm below will be used to calculate priority. User process priorities are assigned by an algorithm based on the amount of recent compute time consumed by the process. This priority is computed as follows:

priority = p_cpu / 2 + nice + base

The p_cpu element is incremented by the system clock (60 or 100 times per second, depending on the hardware) at regular intervals during process execution. Thus, the value of this element represents CPU usage. To ensure that it accurately represents recent CPU usage, its value is divided by 2 every second. The priority number is also recalculated each second. A process that has used a large amount of compute time in the last real-time unit is assigned a low user priority (a large number in the above formula). Because interactive processes are characterized by low ratios of compute to real time (that is, the amount of CPU usage in each 1-second interval is small), interactive response is maintained without any special arrangements. The nice and base elements provide a mechanism to adjust the priority of certain processes (for example, to prevent long running, CPU-bound processes from adversely effecting the rest of the system's workload).

The scheduling algorithm simply picks the process with the highest priority (or lowest number), thus picking all system processes first and user processes second. Hence, all other things being equal, CPU bound user processes should be scheduled round-robin with a 1-second quantum. Owing to the negative feedback characteristic of the scheduling algorithm, if a process uses its high priority to hog the processor, its priority will drop. A high-priority process waking up will preempt a running, low-priority process. At the same time, if a low-priority process is ignored for a long time, its priority will rise. But, regardless of its priority, a swapped out process will never be scheduled until it is again resident.

So, how is a running process preempted by another process with a higher priority? A process switch takes place: the image of the current process is replaced by the image of the new process. The actual switching of the processes occurs while in the kernel mode. Thus, when the switch is requested (for example, by an I/O device interrupt or system clock interrupt), the first thing to happen is a mode switch to the kernel mode. This is not the same as the process switch, which may involve a number of mode switches. First, the user mode is saved and includes the general registers, program counter, processor status word and some memory management information. This is the same information that is saved during a system call, which also involves a mode switch to kernel. After the switch to kernel mode, the kernel mode is saved in the process's *u_area*. The next process is now selected by the aforementioned algorithm. Then the environment of the new process is restored; and finally, the new process resumes in the kernel mode. Since this new process was previously saved in the kernel mode, it restarts precisely where it stopped and the entire call to **swtch** (the new process called **swtch** when

it was preempted) is effectively just a delay.

1.0.6 Swapping

One separate process in the kernel, the swapping process, simply swaps user processes in and out of primary memory. Swapping out simply means the image is copied to secondary memory and the primary memory it occupied is freed. Swapping in involves allocating primary memory for that process and reading its segments into primary memory where that process will compete for the central processor with other loaded processes. The swapping process is created at system initialization time for this one purpose. It determines which process is to be swapped in or out, based on information in the process table. There are two specific algorithms to the swapping process. First, which of the possibly many processes that are swapped out is to be swapped in? This is decided by secondary storage residence time. The one with the longest time out is swapped in first. Also, which of the possibly many processes that are loaded is to be swapped out? Processes that are waiting for slow events (that is, not currently running or waiting for disk I/O) are picked first, by age in primary memory. The other processes are examined by the same age algorithm, but are not removed unless they are at least of some age. This results in a delay in the swapping of "younger" processes and prevents thrashing. Thrashing is a situation where newly created processes are swapped out prematurely, only to be swapped in again moments later. Unless this is prevented, excessive processor time is spent needlessly swapping processes.

The major data associated with a process (the user data segment, the *u_area* segment, and the text segment) are swapped to and from secondary memory, as needed. Memory layout is illustrated in Figure 2-4. All segments are paged; thus, each segment occupies only certain pages in the memory space. These pages are virtually contiguous, but are physically noncontiguous. The size of the page in bytes depends on the processor the system is on. The system maintains a *freelist* of all pages not in use. When a process grows, new memory pages are added to its address space as needed. The process data is not copied, and any previously allocated pages remain the same. If enough free pages are not available, the process will be swapped to secondary memory. The pages it previously occupied will be freed.

As stated under *Process Creation*, when there is not enough primary memory to fork a child, a copy of the parent will be made in secondary memory without swapping out any other processes. This takes place as follows: The parent will link to the **xsched** queue, signal an event indicating it is ready to be "swapped", and then directly call **swtch**. The event will cause the **xsched** process (referred to as the second swapper) to wake up and swap out the process on its queue. This swapping process remembers which process is swapped out, so it can restart it later. This is necessary because the "swapped out" process does not sleep on some event, but instead relies on **xsched** to place it back on the run queue when the copying is finished. The "second swapper" process demonstrates how primitives (such as **swtch** and **setrun**) can be used directly to avoid the overhead associated with generalized mechanisms (in this case, sleep/wakeup). Here, the fact that only one process is involved makes it possible to bypass the general sleep/wakeup mechanism.

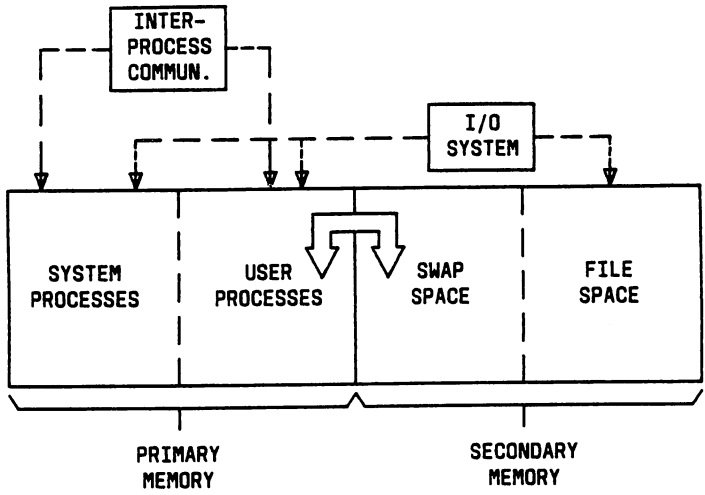

Figure 2-4. Memory Layout

1.0.7 Overlays

A process may **exec** (cause execution of) a file. This consists of exchanging the current text and data segments of the process for new text and data segments specified in the file. The old segments are discarded. Doing an **exec** does not change the process id; the process that did the **exec** persists, but after the **exec** it is executing a different program. Files that were open before the **exec** remain open afterwards. If a program (for example, the first pass of a compiler) wishes to overlay itself with another program (for example, the second pass), then it simply **exec**s the second program. This is analogous to a "goto" in programming. If a program wishes to regain control after **exec**-ing a second program, it should **fork** a child process, have the child **exec** the second program, and have the parent **wait** for the child. This is analogous to a "call."

When **exec** is called, the environment is passed to the new program via the user stack segment of the process. The top of the stack is then modified to appear as though a call to the entry point of that process has occurred, along with three system-supplied arguments. These are the number of command line arguments, the address of the argument strings, and the address of the environment variables (argc, argv, and envp). There are basically 7 steps involved in this.

1. The new program file is located and permissions are checked (to ensure the invoker can execute it). Also, there cannot be another process currently modifying it and it must not exceed the address space allowed.

2. Initial characteristics such as the size and location of the segments and the point of entry is determined.
3. The environment of the old image is saved in swap space.
4. The address space associated with the old image is released.
5. Space is allocated for the new image, any shared text is attached, initialized data is read from the file, and uninitialized data is zeroed.
6. The environment is restored to the stack of the new image.
7. Finally, registers are reinitialized, permissions set appropriately, and the program counter is set to the entry point as mentioned earlier.

1.0.8 Process Termination

A process will terminate for one of two reasons: an explicit call to **exit**, or due to the default action of some signal. As stated previously, after finding a signal, a process looks for some handling routine. If none is found, the process is forced to call **exit**.

If a signal caused the termination, the first thing to occur is an attempt to dump an image of the core. When a process terminates, it can set an eight-bit exit status code that is available to its parent. Even though this code is usually used to indicate success (zero) or failure (non-zero), it can be used in any manner the user wishes. If a signal caused the termination, this exit code is modified to indicate which signal terminated the process and whether or not the core dump was made. Next, resources owned by the process are released, and all signals are set to be ignored. Resources released include open files, working directory, shared text, memory space for the user data and stack segments. The *u_area* and kernel stack are not released until the next process switch occurs. The terminating process is now considered a "zombie" process. All that remains of it is its process table entry; and that is unavailable for use until the process has finally terminated.

Next, the process table is searched for any child or zombie processes belonging to the terminating process. Those children will then be adopted by **init** (that is, their parent process ID will be changed to one). This is necessary since there must be a parent to record the death of the child. The last function of **exit**, before calling **swtch** to bring in the next process, is to record the accounting information and exit code for the terminated process in the "zombie" process table entry and to send the parent the death-of-child signal. It is interesting to note that since the terminated process can never be scheduled again, the final call to **swtch** from **exit** will never return. Sometimes the parent will want to wait until a child terminates before continuing execution. To this end, the parent will call **wait**, which causes the parent to sleep until a child zombie is found (meaning the child terminated). When the child terminates, the death-of-child signal is returned to the parent. Although the parent normally ignores this signal, it will continue to search for child zombies. The terminated child will be found; at which time, the child's exit status and accounting information is recorded in the parent (remember the call to **exit** in the child put this information in the child's process table entry) and the zombie process table entry is freed. Now the parent can wake up and continue executing.

1. FILE SYSTEM

1.0.1 General

One of the most important roles of an operating system is to provide a file system. This file system should be consistent, orderly, and easily accessed. As illustrated in Figure 3-1, the UNIX operating system does this with a tree-like file system composed of directories of files. Using this "tree" structure, files can be attached anywhere onto a hierarchy of directories. To the UNIX system, all files are physically the same (a one-dimensional array of bytes ending with EOF). But the UNIX operating system does keep track of the file type in the file's i-node (discussed under *File System Implementation*). Files are named by sequences of 14 or fewer characters (file names).

1.0.2 File Types

There are three types of files: ordinary files, directory files, and special files. In the UNIX system, files normally reside on a disk. The kernel accesses all three types of files in the same way (remember, to the operating system any file is simply a string of bytes). The user and user application programs must interpret the file appropriately.

Since no particular structuring is expected by the system, an ordinary file contains whatever information the user places in it (for example, English text, source programs, or binary object programs). Any file that is not a directory or a special file is an ordinary file.

Directory files (also referred to as directories) provide the mapping (paths) between the names of files and the files themselves and thus induce a map-like structure on the file system as a whole. Each user has a directory of files. The user may also create subdirectories to contain groups of files conveniently treated together. Although a directory behaves exactly like an ordinary file, since it can only be written by the system, the system controls the structure of directories. The system also maintains several directories for its own use. One of these is the root directory (which may be considered the base directory). Any one of the files in the system can be found by tracing a path through a chain of directories until the desired file is reached. Other system directories contain all the programs (files) provided for general use; that is, all the commands.

Special files constitute the most unusual feature of the file system. Each supported input/output device is associated with at least one special file. Special files are read and written just like ordinary files, but requests to read or write result in activation of the associated device handler rather than the normal file access mechanism. An entry for each special file normally resides in some subdirectory of */dev*, although a link may be made to one of these files just as it may to an ordinary file. So for example: to write on a magnetic tape (mt) device, one may write on a file in the directory */dev/mt*. Special files exist for peripheral devices such as video terminals, disk drives, primary memory, magnetic tape drives, communication links, multiplexers, etc. Of course, the active disks and the memory special files are protected from indiscriminate access by appropriate read and write permissions. There are several advantages in treating input/output devices this way:

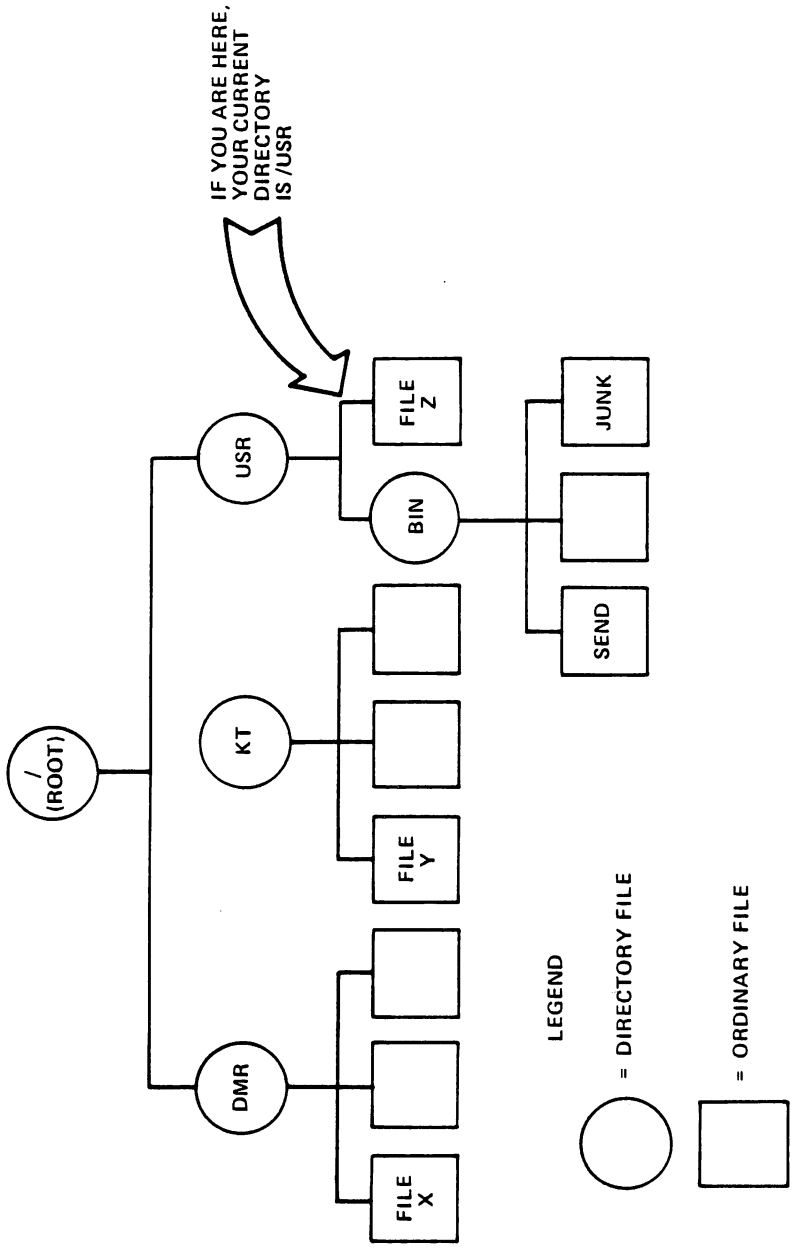

Figure 3-1. Hierarchical File System

(1) File and device input/output are as similar as possible.

(2) File and device names have the same syntax and meaning, so that a program expecting a file name as a parameter can be passed a device name.

(3) Special files are subject to the same protection mechanism as ordinary or directory files.

(4) All I/O is treated uniformly; therefore, the same system calls can be used on all file types.

1.0.3 Pathnames

When the name of a file is specified to the system, it may be specified as a pathname, which is a sequence of directory names separated by slashes, "/", and ending in a file name. This sequence of directories preceding the filename is called a prefix. The UNIX operating system uses certain conventions when reading the prefix: If the prefix begins with a slash, the search begins in the root directory. This is called a pathname. The pathname

/usr/bin/send

causes the system to search the root directory for directory "usr", then to search "usr" for "bin", finally to find file "send" in "bin". The file "send" may be an ordinary file, a directory, or a special file. As a limiting case, the name "/" refers to the root directory itself.

A null prefix (or in fact, any prefix that does not begin with "/") causes the system to begin the search in the current user directory. The simplest form of pathname (for example, "send") refers to a file that is found in the current directory. This pathname allows a user to quickly specify a subdirectory without needing to know (or input) the full pathname. This is just one of several mechanisms built into the file system to alleviate the need to remember pathnames. For example, files can be linked across directories. Therefore, by linking a file to your current directory, you need not supply a prefix when accessing the file. Also, the prefix period, ".", refers to the current directory. This is most useful when copying and moving files. In the above example, the current directory is /usr/bin. And also by convention, the prefix ".." refers to the parent directory (the directory containing the current directory). When a process is created, a current directory and a root directory are associated with that process. These are unique to the process and can be different in other processes. Processes are discussed in the section *Kernel*.

Although the root directory of the file system is always stored on a single device (usually disk drive 0), it is not necessary that the entire file system hierarchy reside on this device. Furthermore, while several file systems can be tied into the system at the same time, not all of them have to reside on the same device (and in fact usually do not). Another file system can be mounted as a directory of the main file system, which in the UNIX operating system is the file system that contains the directory root (that is, "/"). In Fig. 3.1 for example, another branch could be added to the root directory. This new branch could be as large as, or even larger than, the other main branches. File systems can be mounted and unmounted in any place as required. Using the **mount** system request, any directory file can be replaced by a whole new directory tree.

1.0.4 File Protection

Although the access (read, write, and execute) protection scheme is quite simple, it has some unusual features. Each user of the system is usually assigned a unique user identification (ID) number as well as a shared group identification. When a file is created, it is marked with the user ID and group ID of its owner. Also given for new files is a set of protection bits that specify independent read, write, and execute permission for the owner of the file, for other members of the group, and for all other remaining users. The execute permission bit for a directory file is interpreted as "search" permission in that directory. The two highest order bits in the permission field of a file's i-node are the "set user ID" and "set group ID" bits.

1.0.5 File I/O

To read or write a file, it must first exist and then it must be opened. Opening a file makes it accessible to the user process. Various system calls allow the user to create a file, open or close a file, create a directory, delete a file or directory, make a link to a file, etc. Associated with any open file is a file descriptor and file pointer. The file descriptor is an integer used by the user process to identify the file without referring to its name. The file pointer shows which byte in that file is next to be read or written.

Reading and writing within a file is normally sequential. This means that if a particular byte in the file was the last byte written (or read) the next input/output call implicitly refers to the immediately following byte. If n bytes are read or written, the file pointer advances by "n" bytes. Files may be accessed randomly (direct access) by offsetting or positioning the file pointer to the appropriate location in the file with a seek call. Any read or write may be terminated by the file pointer encountering the end-of-file (EOF) condition. The EOF indicates the file pointer is equal to the current size of the file.

The file system maintains no locks visible to the user, nor is there any restriction on the number of users who may have a file open for reading or writing. There are sufficient internal interlocks to maintain the logical consistency of the file system when two users engage simultaneously in activities such as writing on the same file, creating files in the same directory, or deleting open files belonging to the other user. Nevertheless, writing, deleting, or moving files also accessed by another user may very likely create problems for that user when further changes are attempted.

1.0.6 File System Implementation

The UNIX system file system is a disk data structure accessed completely through the block I/O subsystem. A disk is considered a randomly addressable array of blocks. The operating system actually views memory physically as 512-byte sectors and converts the sector numbers to the logical block numbers. This is invisible to the user. The first sector, set aside for booting procedures, is unused by the file system. The second sector is the so-called "super block". See Figure 3-2.

The super block contains a description of the file system (or volume) and includes:

- Size of the i-list and the entire volume
- Free block list and the number of free blocks

- Free i-node list and the number of free i-nodes
- Read-only status
- Mount device, pack name, and file system name
- File system type.

As mentioned previously, several file systems can be mounted simultaneously on different devices. Each mounted file system has a super block and i-list. The i-list is after the superblock. It is nothing more than a list of file definitions. Each file definition is a multibyte structure called an i-node. The UNIX system user accesses a file with a pathname; but, the system itself uses the i-node to do the accessing.

The offset (or index number) of a particular i-node within the i-list is called its i-number. The combination of device name (major and minor numbers) and i-number serves to uniquely name a particular file. To speed up the allocating of i-nodes, an i-node array and i-node list is maintained in primary memory in addition to the information contained in the i-node itself in secondary memory. The list and array work together to provide a buffer of up to 100 free i-nodes that can be allocated and freed quickly and efficiently. When an i-node is allocated, it is removed from the list. When an i-node is freed, it is added to the list. If the list has reached its maximum size of 100 when the i-node is freed, no entry is made. The i-node itself still maintains status information indicating it has been freed. After the i-list and to the end of the disk are located free storage blocks available for the contents of files. These files constitute the file system mentioned previously. Pointers to these blocks are maintained in the *free list array* in the super block. This array has up to 50 pointers to free blocks on the disk. As shown in Figure 3-3, the first pointer in the array points to the *first chain member*. As blocks are allocated, the current size of the array is decremented. When the size reaches zero (not including the first pointer), the next chain member is copied into the array and its size is put into the current array size. As the blocks pointed to by each chain member are allocated, successive chain members are copied in until the last member is exhausted. At that point, all free blocks are allocated (indicated by the first block pointer in the array being zero). As blocks are freed, this mechanism is reversed. The pointer to the freed block is placed in the array and the current size is incremented. When the size reaches 50, the array is copied out into the first (or second, third, etc.) chain member and the current size is reset to zero.

An i-node contains a description of the file it points to:

- The user and group ID of the file owner
- The file protection bits
- The physical disk addresses for the file contents
- The file size
- Time of creation, last use, and last modification
- The number of links to the file; that is, the number of times it appears in a directory.

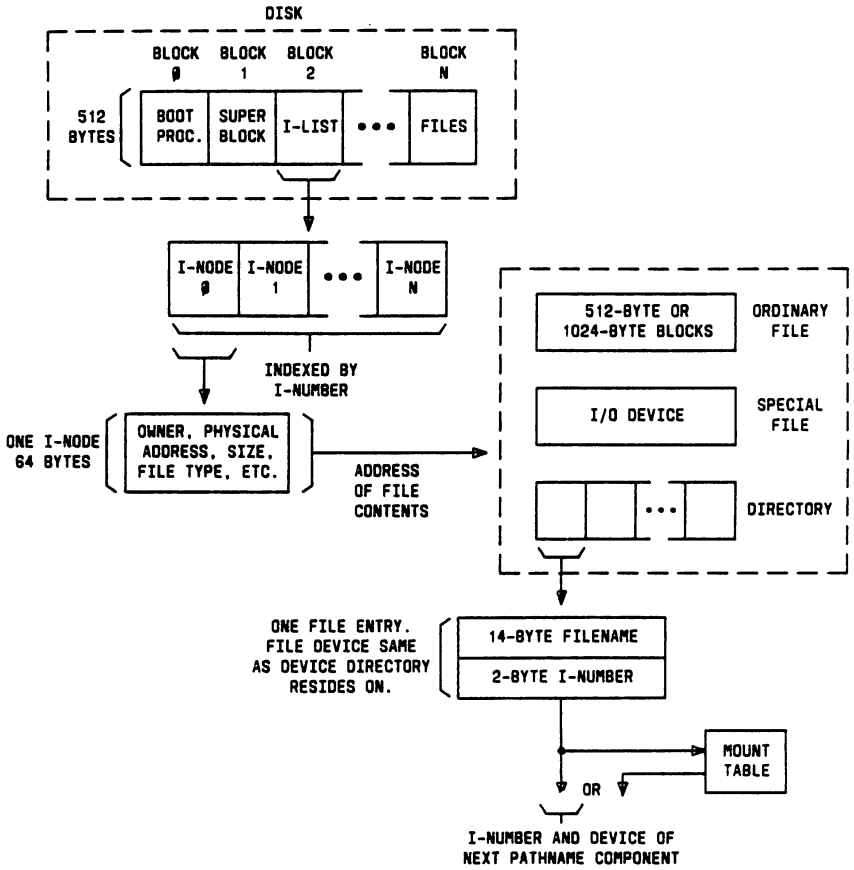

Figure 3-2. I-Node Access Scheme

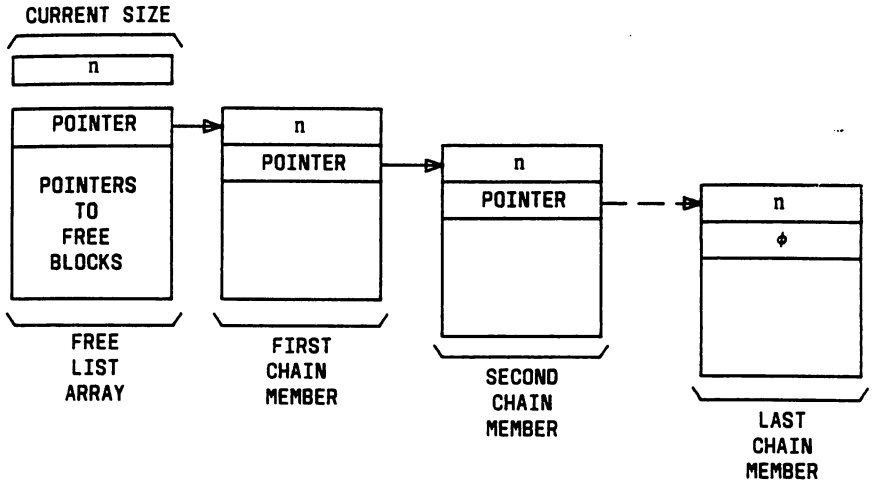

Figure 3-3. Free Block List

The i-node contains 13 disk addresses. The first 10 of these addresses point directly to the first 10 blocks of a file. If a file is larger than 10 blocks, then the eleventh address points to a block that contains the addresses of the next 128 blocks of the file. If the file is still larger than this, then the twelfth address (a double indirect address) points to a block of 128 addresses. Each of these addresses points to another block which then, in turn, points to up to 128 blocks of the file. If the file is too large for double indirect addressing, the file mapping algorithm allows only one more address, which is a "triple indirect" address.

The hierarchical file structure is made possible by using a directory file to maintain the links to the files. These links impart a logical continuity to the entire file system. Both directory files and ordinary files are linked this way. A file may even have links in more than one directory, thus eliminating the need to duplicate certain files that will be accessed from several directories. Because the linking mechanism is the same, a directory can be accessed in exactly the same way as an ordinary file. It contains 16-byte entries consisting of a 14-byte name and an i-number. This i-number is the index to the i-list contained in block 2 of the disk on which the directory resides (unless it is the i-number of the root of a mounted file system). Because the i-node defines a file, the implementation of the file system centers around access to the i-node. See Figure 3-4. The system maintains a table of all active i-nodes. As a new file is accessed, the system locates the corresponding i-node, allocates an i-node table entry, and reads the i-node into primary memory. The table entry is considered to be the current version of the i-node. Modifications to the i-node are made to the table entry. When the last access to the i-node goes away, the table entry is copied back to the secondary

store i-list and the table entry is freed.

All I/O operations on files are carried out with the aid of the corresponding i-node table entry. The accessing of a file is a straightforward implementation of the algorithms mentioned previously. References to the file system are made in terms of pathnames of the directory tree. Converting a pathname into an i-node table entry is also straightforward. Starting at some known i-node (the root or the current directory of some process), the next component of the pathname is searched by reading the directory. See Figure 3-2. The file entry will give an i-number and an implied device (that of the directory). This i-number and device (the i-node) point to the next i-node table entry to be accessed. If that was the last component of the pathname, then this i-node is the result. If not, this i-node is the directory needed to look up the next component of the pathname, and the algorithm is repeated.

When another file system is mounted on the file system hierarchy, that system simply becomes an extension of the current file system. To allow mounting of other file systems (and easily unmounting them), a mount table is maintained. The mount table contains pairs of designated i-nodes and block devices. At each of the i-nodes listed, a file system is mounted (mounted on the device indicated). When converting a pathname component into an i-node, a check is made to see if the new i-node is one of those designated in the mount table. If it is, the i-node of the root of the mounted block device replaces it. The mount device and the root i-node are used to search the next component of the pathname.

The user process accesses the file system with certain primitives. The most common of these are **open**, **create**, **read**, **write**, **seek**, and **close**. The data structures maintained are shown in Figure 3-3. In the u-block segment associated with a user, there is room for some (usually about 20) open files. This open file table consists of pointers to the system file table. The system file table contains a list of all open files and pointers that can be used to access corresponding i-node table entries. Associated with each of these open files is an offset I/O pointer (file pointer). This is a byte offset of the next read/write operation on the file. The system treats each read/write request as random with an implied seek to the file pointer. The user usually thinks of the file as sequential with the file pointer automatically counting the number of bytes that have been read/written from the file. The user may, of course, perform random I/O by setting the file pointer before reads/writes.

With file sharing, it is necessary to allow related processes to share a common file pointer and yet have separate file pointers for independent processes that access the same file. This is necessary to allow the two processes to do I/O on the file at different places in the file. With these two conditions, the file pointer cannot reside in the i-node table. If it did, all processes accessing that file would be forced to use the same file pointer. Nor can it reside in the list of open files for the process. The file table is incorporated in the structure for the sole purpose of holding the file pointer. Processes that share the same open file (the result of **fork**s) share a common file table entry. A separate open of the same file will only share the i-node table entry. Each open will have a distinct open file table entry.

The main file system primitives are implemented as follows. The **open** command converts a file system pathname into an i-node table entry. A pointer to the i-node table entry is placed in a newly created file table entry. A pointer to the file table entry is placed in the u-block open file table for the process. The **create**

command first creates a new i-node entry, writes the i-number into a directory, and then builds the same structure as for an **open**. The **read** and **write** commands access the i-node entry as described above. The **seek** command simply manipulates the file pointer; no physical seeking is done. The **close** command frees the structures built by **open** and **create**. Reference counts are kept on the open file table entries and the i-node table entries to free these structures after the last reference goes away. The **unlink** command simply decrements the count of the number of directories pointing at the given i-node. When the last reference to an i-node table entry goes away, if the i-node has no directories pointing to it, then the file is removed and the i-node is freed. This delayed removal of files prevents problems arising from removing active files. A file may be removed while still open; afterwards, the resulting unnamed file vanishes when it is closed. This is a method of obtaining temporary files.

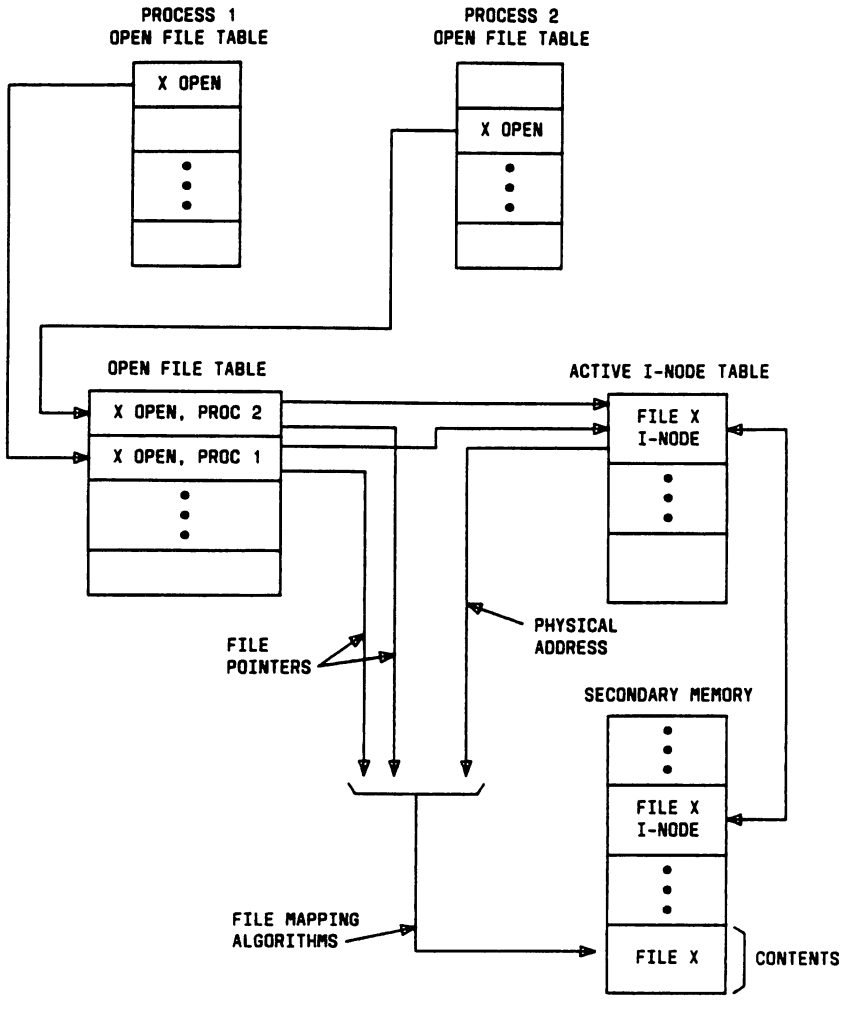

Figure 3-4. Open File Table Structure

1. I/O SYSTEM

1.0.1 General

The input/output (I/O) system is divided into two completely separate subsystems: the block I/O subsystem (structured I/O) and the character I/O subsystem (unstructured I/O).

The user communicates with the peripheral devices by system calls. The system calls to input or output are designed to eliminate the differences between the various devices and styles of access. There is no distinction between "random" and "sequential" I/O, nor is any logical record size imposed by the system. The UNIX system also attempts to eliminate differences between ordinary disk files and I/O devices such as terminals, tape drives, and line printers.

An entry appears in the file system hierarchy for each supported device, so that the structure of device names is the same as that of file names. Not only do the same read and write system calls apply to devices and disk files; but also, the same protection mechanisms apply. Figure 4-1 illustrates the functional layout of the I/O system.

1.0.2 Block I/O

The model block I/O device consists of randomly addressable, secondary memory blocks of 512 (or 1024) bytes each. The blocks are uniformly addressed 0, 1, 2,... up to the size of the device. The block device driver has the job of emulating this model on a physical device.

The block I/O devices are accessed through a layer of buffering software. The system maintains a list of buffers (typically between 100 and 200) each assigned a device name and a device address. This buffer pool constitutes a data cache for the block devices. On a read request, the cache is searched for the desired block. If the block is found, the data is made available to the requester without any physical I/O. If the block is not in the cache, the least recently used block in the cache is renamed, the correct device driver is called to fill up the renamed buffer, and then the data is made available. Write requests are handled in an analogous manner. The correct buffer is found and relabeled if necessary. The write is performed simply by marking the buffer as "dirty". The physical I/O is then deferred until the buffer is renamed.

1.0.3 Character I/O

The character I/O device drivers handle any physical devices that do not fit the block I/O model. This includes the "classical" character devices such as communications lines and line printers. It also includes magnetic tape and disks when they are not used in a stereotyped way; for example, 80-byte physical records on tape and track-at-a-time disk copies. Although an I/O request from the user is sent to the device driver unaltered, the execution of these requests is, of course, up to the device driver. To this end, there are guidelines and conventions to help in the creating of certain types of device drivers.

1.0.4 Device Drivers

Each supported I/O device is associated with at least one special file and one device driver. Special files are read and written just like ordinary disk files, but requests to read or write result in activation of the associated device by the driver. See *File System* for more on special files.

Devices are characterized by a major device number, a minor device number, and a class (block or character). For each class, there is an array of entry points (configuration table) into the device drivers. The major device number is used to index the array when calling the software for a particular device driver. The minor device number is passed to the device driver as an argument. The minor number has no significance other than that attributed to it by the driver. Usually, the driver uses the minor number to access one of several identical physical devices.

The general disk driver makes it possible to use the disk as a block device, a character device, or a swap device. The driver creates a queue of records that contain a primary memory address, a secondary memory address, a count of the number of bytes to be transferred, and a read/write flag. Swapping is done by passing each record to the swapping device driver. Block I/O is done by passing each record to the device driver with a request for a system buffer (in the data cache described above). Character I/O is done by creating a record that points directly into the user address space, and passing that record to the device driver. The driver will insure the user is not swapped during the I/O.

Real character-oriented device drivers use a queue of characters (a character list) to store characters to be transferred. Space for a queue is allocated as characters are added and released as characters are removed from the queue.

A typical character-output device (for example, line printer) is implemented by passing characters from the user onto a character queue until some maximum number of characters is on the queue. The driver then gets the characters from the queue and sends them to the hardware serially. The number of characters on the queue is checked and, as the count falls through some intermediate level, an event (the queue address) is signaled. The process that is passing characters from the user to the queue can be waiting on the event, and refill the queue to its maximum when the event occurs. Reading from an input device is handled in a similar fashion.

Another class of character devices is the terminal class. A terminal is represented by two input queues and one output queue. The device driver software must (in addition to reading and writing to the terminal) be able to handle escape sequences and control characters.

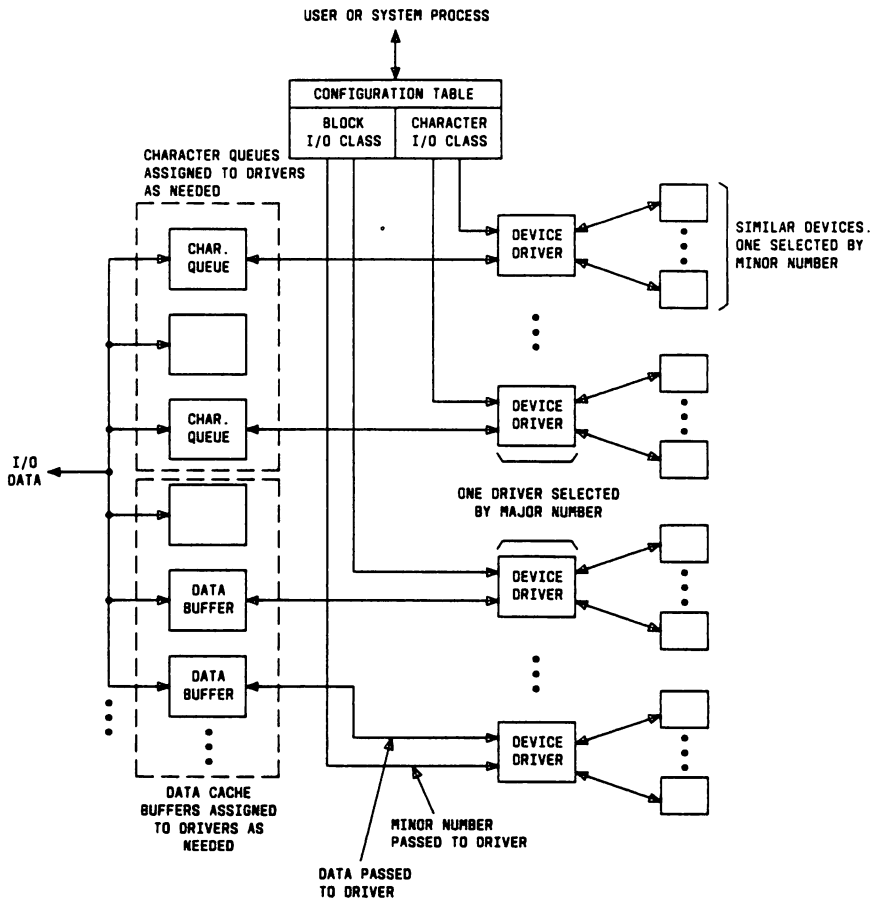

Figure 4-1. I/O System

1. SHELL

1.0.1 General

The user communicates with the UNIX operating system (usually via a terminal) with the aid of a command programming language called the shell. The shell is a command-line interpreter; it reads lines entered by the user and interprets the lines as requests to execute other programs. The shell also provides conditional execution and flow control features. Thus, it is also a powerful programming language (not dissimilar to C-language).

When a user logs into the system and before executing the shell, the login process sets up an execution environment. System default and user-specified *profile* files are read, and then the appropriate shell program is executed. The environment determines (among other things) what commands the user has access to. Normally, a user will log into the system with one process associated with his or her *user name*. This process is the user's shell and will become the parent of all other processes the user creates. Although the UNIX system is set up to allow any program to be used as a shell, the actual shell program is normally used. The basic function of the shell from the operating system's point of view is simply to parent child processes, and a process need not be the shell program to do this. From the user's viewpoint, the shell should accept user input, execute commands. provide output, and access files.

There is a small subset of shell commands that are "built in" and differ from other commands primarily in that no separate process is created to execute these commands. They are part of the standard shell program and include all the execution and flow control constructs. All other commands are either functions, utility programs, or application programs. All commands listed in the *UNIX System V User Reference Manual* can be considered utility or application programs. Execution, termination, and input/output is handled uniformly among all commands (with only minor exceptions as noted in the *UNIX System V User Reference Manual*). The shell provides the flexibility the user needs to make the utility and application programs quickly fit his or her needs. The normal sequence of executing a command involves five steps:

1. The command is parsed by the shell (that is, command and parameter substitution, filename generation, and I/O redirection is done). Certain built-in commands are not parsed.

2. The PATH parameter, or hash table if hashing is implemented, is searched to find the command file (see description under **Commands**).

3. If the file is a compiled machine language file, the parent process (in this case the shell) forks a copy of itself to become the child. The file is then **exec**ed, changing the child from simply a copy of the parent into a totally different image.

   If the file is a shell script, the parent forks a shell as its child to read the file, and the entire sequence begins again as each command in the file is executed by the child (then acting as parent).

4. Next, command line arguments and environment variables are passed to the child process via the user stack. The built-in command **export** determines which environment variables will be passed to the child.

5. The child is now an independent process capable of running, sleeping, or swapping as need be. It will be scheduled when its priority allows.

If the command is a function (functions are available only on certain versions), a new process is not forked; but rather, the command becomes part of the parent process. In any case, the shell expands *, ?, and [] filename patterns by reading directory entries and comparing the filenames to the pattern. In some versions this is done using buffered reads of many entries at once. In other versions, the directory entries are read one at a time. The command substitution is simply an in-line replacement of the command string with the output of the command (interpolation). Parameter substitution requires the shell to search the current environment for the parameter name and interpolate all references to defined parameters with the value of those parameters. A reference to a parameter not defined will result in a null string being interpolated.

1.0.2 Commands

Commands may be read from either a terminal or from a file (which allows commands to be stored for later use). In simplest form, a command line consists of the command name followed by arguments to the command (all separated by spaces). For example:

```
command arg1 arg2 .. argn
```

The shell divides the command line into the command name string ($0) and as many arguments strings as given ($1, $2, $3, etc.). Then a file with the name *command* is sought; *command* may be a pathname including the "/" character to specify any file in the system. The simplest mechanism for finding *command* is to search through a sequence of directories specified by the PATH parameter until the command is found. By default, this parameter specifies the standard system command bins but it can be changed by the user. The search could take considerable time if the command is not commonly used (that is, it's in one of the last directories searched). If *command* is found, it is brought into memory and executed. The arguments collected by the shell are accessible to the command. When the command is finished, the shell resumes its own execution and indicates its readiness to accept another command by returning a prompt character on the input/output terminal.

In some versions of the shell, commands are *hashed*. This simply means the directory containing any particular command is recorded in a table for immediate retrieval when the command is executed. This will speed up the mechanism described above. The location of the command is determined on the first invocation using the **access** system call. Prior to forking a child, that location is saved in the table. The subsequent **exec** and all following invocations of the command will look at the hash table first when trying to determine the location of the command. Since it is still desirable to specify a directory for finding the command, this hashing is not done when the command is specified with a full pathname. Also, since search directories may change or new commands may be added, there is a built-in **hash** command available to clear or update the hash table.

Built-in commands (that is, part of the shell program) are included in the hash table; but, are marked as built-in and will not show up when the table is printed. Therefore, unless a full pathname is used, a built-in command will always be executed in preference to a user-defined command of the same name. In UNIX

System versions without hashing, you can make the shell use your version without needing a full pathname by changing the PATH parameter appropriately.

1.0.3 Input/Output

Programs executed by the shell start off with three open files with file descriptors 0, 1, and 2. When a program begins, is open for writing and is known as the standard output. Conversely, starts off open for reading, and programs that wish to input messages entered by the user read file 0, (known as the standard input). File descriptor 2 starts off open for writing and is known as standard error. File 0, file 1, and file 2 are normally attached to the user's terminal. Input is taken from the terminal and output is sent to the terminal.

The shell is able to change the standard assignments of these file descriptors from the user's terminal display and keyboard. If an argument to a command is prefixed by "$>$," file descriptor 1 will, for the duration of the command, refer to the file named after the $>$. Conversely, preceding a filename with "$<$" will reassign file descriptor 0.

Although the file name following "$<$" or "$>$" appears to be an argument to the command; it is in fact, interpreted by the shell and not passed to the command at all. Therefore no special coding to handle input/output redirection is needed within each command; the command need merely use the standard file descriptors 0 (read), 1 (write), or 2 (error) where appropriate.

1.0.4 Pipelines

An extension of the standard input/output notion is used to direct (or pass) the output from one command to the input of another without the use of temporary files maintained by the user. A sequence of commands separated by vertical bars causes the shell to execute all the commands and to arrange that the standard output of each command be delivered (piped) to the standard input of the next command in the sequence. This "pipeline" will cause the shell to fork a group of child processes (one for each command) with the last command in the pipeline acting as the controlling process for the group. Unless the pipeline is executed asynchronously (in the background), the parent will wait on this controlling process. The exit status returned to the parent is the exit status of this process.

1.0.5 Background Processes

A related shell feature, the '&', allows asynchronous execution of commands and pipelines. If the pipeline is followed by "&", the parent will not wait for the controlling process to exit before prompting the user for the next input. Also, the shell will output the process number of the controlling process to allow the user to track the progress of the command.

1.0.6 Command Lists

More than one command may be entered on the same line but the commands must be separated by one of the following: ';', '&', '&&', or '|'. Furthermore, any of the commands can be a pipeline (actually, a single command can be viewed as the simplest form of a pipeline). By default, the $<$cr$>$ delimits commands in a list. The $<$cr$>$ and ; cause each command to be executed sequentially as follows: As each command in the list is read, the parent will fork a child to execute that command, and then wait for the child to terminate before executing the next command. The && (|) directs the shell to execute the remainder of the list only if

the preceding command returns a zero (non-zero) exit status.

1.0.7 Shell Procedures

The shell can execute commands from a file. The commands are executed sequentially until an EOF is encountered or an **exit** command is executed. This feature allows the user to take advantage of the programming power of the shell. For more information on the shell program, refer to *sh*(1) in the *UNIX System V User Reference Manual*, the *UNIX System Programming Guide*, or the *UNIX System User's Guide*.