# An Operating System Employing Dedicated Resource Management Processors

Anita K. Jones and Melvin W. Pirtle

The BCC-500 was designed to provide interactive terminal service to customers expecting rapid response to requests for simple computations. The envisioned customers included banks, in which tellers consult and alter bank records using terminals; credit bureaus which maintain a data base of credit information and reference it with queries requiring a minimum of computation; and engineering or accounting firms having 'small' computational needs that could be satisfied by interactive use of a language like BASIC.*

All these applications have in common that they require interactive access, reliable service, rapid terminal response and minimal computation. Most users were to be transaction oriented and would not require a general purpose computational environment for writing, then executing their own programs. For example, a bank teller need only be able to query the bank's data base and to make a few types of alterations to it. In fact, providing the teller with a more powerful computational environment--the ability to create and execute arbitrary programs--might compromise the bank management's control over what actions tellers could take. (Of course, as we will see later a general purpose computational environment is required for both the development of customer programs and system maintenance.)

----------------------

*The system, designed by Berkeley Computer Corporation, was to service roughly 500 interactive users concurrently; hence the name BCC-500. The software, firmware and hardware were designed and built by Berkeley Computer Corporation between April, 1968 and July, 1970. BCC was unable to secure necessary funding to finance final development and marketing of the system. The BCC-500 was sold to the University of Hawaii in March, 1971. A reduced configuration has been running there for several years.

To provide the desired customer services, the BCC-500 was designed to connect many low and medium speed devices (remote and local) to a centralized computer facility. Each terminal user is represented in the central facility by a job which is serviced rapidly enough to sustain consistent rapid terminal response. As usual, the operating system is responsible for managing the computation, storage and communication resources including: collecting and transporting streams of characters which users input at slow and erratic rates and output in infrequent, but more rapid bursts; scheduling and running user jobs for short intervals (10-20 milliseconds per allocation of a CPU resource); and managing the storage media.

Customers were expected to execute what we will call **subsystems**--for example, an inventory recording and maintenance subsystem or a banking subsystem. Thus from one perspective, though the actual customers would sit at terminals executing these subsystems, the 'real users' of the BCC-500 were to be the subsystems and their implementors. Then, certainly, the BCC-500 also needed to provide an environment for the on-line development of new customer subsystems, adaptation of existing subsystems, development of general purpose utility programs and maintenance of the operating system itself. Such development work needed to take place at the same time that transaction oriented customers were using the BCC-500. Of course, development work was not to substantially degrade the service being provided to other customers.

Our objective is to describe the major and the most interesting attributes of the BCC-500. Now that we have characterized the two kinds of users of the system: transaction oriented customers and subsystem developers, we ask what facilities must be supplied to them. First, it should be pointed out that subsystems are user written software that is in no way privileged. Code and data in the BCC-500 exists in one of three concentric protection rings. Ring boundaries protect the code and data within them from the effect of execution in outer rings. The mechanisms for calling between rings, passing parameters and returning to a suspended call site are the only means for execution in one ring to affect another. Only the interior ring, the **Monitor Ring**, in which primitive operating system functions execute has special privileges. Subsystems, including compilers and text editors, execute in the **User Ring**. (It was expected that many of the subsystems, even some of the compilers would be written by or for customers.)

Between the User and Monitor Rings is the **Utility Ring** containing the software which augments Monitor facilities and makes them more convenient to use. For example, the Utility extends the Monitor provided file and directory system by incorporating space management for the buffering of file input and output. It provides symbolic file names in lieu of the brief fixed size names of the Monitor, as well as automatic file name extensions and strategies for searching the directory structure. The Utility also provides a convenient terminal system interface including a command language and input and output character stream buffering.

Both Utility and User Ring programs can directly invoke Monitor functions. The Monitor is responsible for the allocation and exercise of physical resources so that users can work in a cooperative well-behaved community. Monitor facilities not only control, but extend what outer ring programs see of the three major resources; the terminal and communications hardware, the memory media and the CPUs (central processing units which execute user programs). In the remainder of this section we will discuss the Monitor-provided facilities.

For terminal communications, the Monitor implements the notion of a logical information channel. Each terminal is connected to both an input and an output channel through which characters pass. A User or Utility Ring program can invoke Monitor functions to read a string of characters from a channel or write a string of characters to a channel. The Monitor is responsible for conveying the characters along a channel to or from the correct terminal.

The second resource that the Monitor manages is central (core), drum and disk memory. All memory is divided into 6K byte page frames. The Monitor controls all transfer of pages of information between the storage media. The Monitor also provides virtual to physical address mapping so that programs can be written with the convenience of a virtual address space. In addition, provides a primitive file and directory structure maintained on secondary storage. Files are named with fixed size brief (non-symbolic) names. Individual files can be created and entered into directories to be looked up later and transferred between secondary and central memory.

The third resource that the Monitor manages is CPU allocation. To do so, the Monitor implements the notion of a process. Processes appear to execute concurrently so that the multiplexing of CPUs among processes is not directly apparent to users. The Monitor provides operations for synchronization, for resetting the priority of processes, for requesting process scheduling at real time intervals and for specifying the minimum and maximum resources (CPU, terminals and memory) to be guaranteed to a process.

A process has several options for synchronizing its actions with the state of the system and with the actions of other processes. A process can block, awaiting the occurrence of a specific event. One process may wakeup another process by signalling the occurrence of that event. Alternatively, a process can define an interrupt type and specify what action is to be taken when an interrupt of the type occurs. When that interrupt is signalled by the system or some other process, current activity is suspended and the process executes the predetermined action. Processes can use this interrupt mechanism to establish timers based on real or billed compute time.

To assist in maintaining a well-behaved community of processes, the Monitor provides extensive protection facilities allowing:

--one user to permit others to have controlled access to his files

--proprietary procedures to be created and made available to others in an execute only form

--a user to invoke a program which has capabilities the caller does not possess

--the manager of a user group to maintain control over the resources distributed to individual users in the group.

Protection is based on a notion of keys and locks. With every object to be individually protected there exists a lock list, which is a list of key names paired with an access control description defining how the associated object can be manipulated if opened with the key.

Each new user is given a key by the Monitor, to use for purposes like protecting private file objects. A user process may call on different programs to execute on its behalf. Some of these programs require data objects which are private to the program, but which are accessed on behalf of a caller. To implement such protection, a user process may be divided into eight sub-processes, each having a private set of keys. One sub-process may call a second which executes with keys which the caller does not have and cannot obtain. By presenting a key to the Monitor along with the unique name of an object to which it desires access, a sub-process can gain access to the named object. But the Monitor grants access only if the presented key appears on the lock list of that object. The sub-process gains accesses only as described by the access control descriptor associated with the key in the lock list of the named object. Keys and lock lists are manufactured by and maintained only by the Monitor. Keys cannot be forged.

A terminal user is represented by one or more processes in the BCC-500, and it is to these processes that CPU and memory resources are allocated. Perhaps the most straightforward implementation would be to have sufficient processors and sufficient rapidly addressible memory so that each user could be allocated a processor and an (expandable) working set sized memory when he dialed into the system. The speed of the processor and access time of the memory would have to be sufficient to provide whatever was deemed 'reasonable response'. Economics veto such a solution, both at the time the BCC-500 was designed (1968) and today. For example, if each of 500 users required a 60,000 byte working set, 30 million bytes of directly addressible memory would be required.

As usual, the BCC-500 emulates such a solution by allocating both the CPU resources and rapidly (directly) accessible central memory resources to a user process for only a short time quantum before allocating the resources to the next user process. Before a process is allocated a CPU resource, its central memory working set pages are loaded into central memory. A small crucial part of the information which the system maintains describing each process appears in the Process Table. This table is permanently resident in central memory and provides the state information needed to remember the existence of a process for scheduling purposes. A Process Table entry includes the name of the page containing a Process Context Block (which may not be resident in central memory), the process' state (e.g., running, blocked, working set loaded into central memory), the type of event a process is waiting for if it is blocked, and its scheduling priority.

The Process Context Block gives a more complete picture of a process. It includes pointers to portions of the directory structure this process can reference, the various sub-processes and their keys, the current value of the program counter, and the content of the general registers if the process is not running on a CPU. The Context Block describes the process' address space which may include up to 128 6K byte pages. The Context Block also indicates that subset of the pages known to the process that are in its drum working set--i.e. intended to be drum resident whenever the process executes. A subset of the drum working set pages are in the process' central memory working set. It is these pages which are loaded into central memory before the process is allocated CPU resources. (Other tables in central memory indicate where each existing page can be found in the various levels of the memory hierarchy: central memory, drum and disk.)

The two CPUs in the BCC-500 system contain virtual to physical address mapping hardware based on a hardware map table. Each map entry associates a uniquely named virtual page address to the physical location of that page in central memory. Each CPU reference to central memory causes a lookup in the map to find the physical address associated with the process' virtual address. This map is loaded on demand. The unique name of a page and central memory location of any page actually resident in central memory at a given moment is recorded in a Core Hash Table (CHT).

Whenever a CPU references a virtual page for which there is no entry in the map, the map loader (implemented in CPU microcode) will first find the unique name of the page in the Process Context Block and then find the central memory address of that page in the CHT and load the map appropriately.

Should the referenced page not be in central memory, the page is added to the current central memory working set of the process, the CPU stores its internal state and the CPU is reallocated to another process. Typically, pages of the preempted process are unloaded. In any case, by the time the process is again allocated a CPU, its working set, including the new page, will be in central memory. (A process can request that a page be added to its working set in anticipation of use, to attempt to ensure that the page will actually be loaded into central memory before its first use.)

It should be clear that the physical resource management functions are sizeable. Consider the time and effort required to preempt a processor and central memory resources from one user process and allocate it to another process. CPU preemption time is very short compared to the time required to preempt central memory resources from one process and subsequently prepare central memory for use by another process. Many pages may have to be transferred between central and drum memory and the associated tables (e.g., the CHT) updated. Thus, as one expects, memory management and processor scheduling have constraints which make them quite different, complicating the resource management problem.

To illustrate the magnitude of the memory management task, we will consider an example system in which each terminal user's process is assumed to have an 84K byte (14 page) central memory working set: 48K bytes (8 pages) of Monitor and Utility pages which are not swapped, and 36K bytes (6 pages) of code and data which are swapped. Each time a process receives a quantum, 71K bytes (12 pages) are transferred (6 pages in and then out). If we assume a 25 ms quantum of CPU resource is given to a process each time it is loaded into central memory, then 40 processes can be given a quantum each second. The BCC-500 has 2 CPUs to execute user processes, so actually 80 processes execute each second. This requires that 1000 pages/second (80 processes x 12 pages/second) must flow through central memory. In this scenario the central memory involved in swapping need not be greater than 24 pages (144K bytes). Not much central memory is required. But this example does not take into consideration the processing needed to support the page transfers. As we will see later the CPUs are assumed to be saturated executing user processes and memory management is provided by a separate processor. Similar scenarios would illustrate that scheduling of processes for CPU allocation so that guarantees of processor resources are met and management of information streams requires processing that is both substantial, in terms of the number of instructions executed, yet must be and responsive.

It should be clear that the Monitor is a sophisticated and powerful system. It is far too large to be implemented as a monolithic entity. In the BCC-500 implementation a micro-programmable processor (upp) is dedicated to the management of each of the three major physical resources. One upp is used to schedule the two CPU's which exeucte only user processes. A second upp performs memory management. Memory is of three types: 386K bytes of central memory directly accessible by all processors and the drum and disk controllers, drum memory and disk memory. The central memory has a high bandwidth connection to the high transfer rate rotating memories. The memory management processor is responsible for causing the working sets of appropriate processes to flow into and out of central memory as illustrated in the scenario above. This flow must be synchronized with scheduling so that 'reasonable' terminal response time can be maintained. A third management processor handles terminal information streams.

It should be no surprise that the resource management portion of the operating system (the three dedicated management upps) is larger (measured in MOPs or actions per second) than the user programs executing on the two CPUs. Certainly it is larger than the remainder of the Monitor, a set of functions which user or Utility programs may invoke to be executed on the CPUs. For convenience in talking about them, we name the three management micro-processors as follows:

| **Function** | **Microprocessor** |
|---|---|
| memory management | MMP (Memory Management Processor)* |
| processor allocation and process synchronization | SCHEDULER |
| character input/output management | CHIO (CHaracter I/O) |
| system monitoring and maintenance | (also in) SCHEDULER |

Where no confusion should arise, we will use the microprocessor name to refer to either the microprocessor itself or the function it performs. Logically the BCC-500 components are linked as shown in Figure 1.

---------------------

*Also called AMC (Auxiliary Memory Controller) in BCC documentation.

**The BCC-500 Hardware**

Before discussing the BCC-500 resource management in detail, it is worthwhile to take a closer look at the hardware configuration managed. The heart of the system is the central memory (core) which has four ports through which requests to the memory can be made. Each CPU is assigned one port. The three management upps are connected to a third port through a multiplexor. The fourth port is used (via another multiplexor) by the two drum and two disk controllers providing a high bandwidth path between central and secondary memory. Each port can accept a request every 100 nanoseconds and can potentially sustain a 30M byte/second transfer rate. The architecture of the central memory is described in Appendix I.

These three management processors and the two CPUs are variations of the same powerful (15 million operations per second) upp. The CPUs execute about a million of their instructions per second. Each upp has a 2048 ninety-six bit wide control store with a 100 nanosecond cycle time. The CPUs differ from the management processors in having additional hardware to provide an extended arithmetic capability, instruction prefetch and decoding, and a 128 register map used in the translation of virtual to physical addresses. Each management processor has a private memory in which its local data structures and code are stored. Consequently, a management processor references central memory (to access shared data) relatively infrequently. The multiplexed port into central memory permits these references to be rapid so that performance of the resource managers is not degraded.

A diagram of the major components of the BCC-500 appears in Figure 2. Note that though each of the 4 ports into central memory is capable of sustaining a transfer rate of 30M bytes/second, higher traffic is expected across some ports rather than others as indicated by the comparative width of the data paths.*

At this point, it might be of interest to consider the implications of the central memory size and drum transfer rate for the swapping of process working sets between central memory and drum. Recall that central memory is only 386K bytes, far smaller than that in many contemporary systems which provide interactive terminal service for one tenth of the number of users that the BCC-500 was designed to serve. The drum and disk controllers transfer a 6K byte page at a time. Two drums (6M bytes/second each) and two disks (300K bytes/second each) can transfer simultaneously through the port into central memory.

Returning to our memory management example of the 80 processes which are each to receive 25 ms of CPU time, we find that theoretically either one of the two drums is sufficient to transfer the 1000 pages/second to support memory allocation for the 80 processes. (It should be pointed out that though each drum has a 6M byte/second transfer capacity, the MMP cannot exploit the full transfer rate. This is due to the fact that the MMP is transferring pages from selected working sets and may incur rotational delay attempting to transfer just the required pages.)

**The Physical Resource Managers**

In the remainder of this paper we will focus on the three physical resource manages (the SCHEDULER, MMP and CHIO) which provide support for the Monitor. The BCC-500 resource managers have several aspects of their design in common. Each manager is implemented as a collection of **chores** together with a scheduling mechanism which we will call a chore scheduler or **choremaster**. The choremaster is a short simple

----------------------

*A comparison of the planned configuration and the actual configuration running at the University of Hawaii is given in Appendix II.

cyclic program which selects that chore to be executed next. Chores have the attribute that once invoked, they run to completion. Each chore can be viewed as a small finite state machine which, when triggered, causes the state of the resource manager to change. Because chores within the same manager execute indivisably with respect to one another (i.e., chores are not interruptible), they do not need to synchronize their use of data structures private to a manager.

Each resource manager is relatively autonomous, yet, the managers cooperate. This requires that they communicate. Because it is this communication interface which makes the managers interdependent, the interface is kept as simple as possible. Managers user a message sending protocol. Each manager is associated with a set of input message queues which it services. If one manager recognizes the need for a particular manager to act, it sends a message to the appropriate input queue. The manager servicing that queue attends to the message at its own discretion. It is the choremaster which selects a message to be processed and then invokes a suitable chore to process it.

Note that if a message arrives at an input message queue, it will never be considered by the choremaster of the manager which services that queue, until the currently executing chore completes. For the resource managers to be responsive, each chore must be short enough in duration not to degrade upp response to high priority requests which may arrive during its execution. To illustrate this constraint on chore execution time, consider the MMP which must service the disk and drum controllers at least twice for each page transferred between two memory hierarchy levels. A page transfer between drum and central memory takes one millisecond. Consequently, the MMP chores never execute for longer than 1/4 millisecond.

The chores which make up a resource manager also have need to communicate with one another. They use the same message protocol. Thus managers have input queues which are private in that all messages placed in a private input queue originate in the manager itself. The modular structure of the managers that this intercommunication scheme induces should emerge as we discuss the individual resource managers.

## The SCHEDULER

The SCHEDULER's objective is to allocate CPU resources to user processes so as to maintain consistent rapid terminal response and to honor CPU resource guarantees. To meet these objectives the SCHEDULER performs two kinds of functions: synchronization of proceses (which determines whether a process requires CPU resources at all) and multiplexing of the two CPUs among competing processes. Central memory and CPU allocations must be complementary if system resources are to be efficiently utilized. To coordinate the two allocations, the SCHEDULER selects (roughly) the order in which processes are to be given a CPU resource. It then informs the MMP so that central memory working sets of the selected processes can be preloaded into central memory. Later the SCHEDULER actually allocates CPU resources to the (highest priority) loaded processes.

Whether a process requires a CPU resource is recorded in that component of its process state which summarizes the results of scheduling and synchronization actions involving the process. The scheduling component of a process state can have the following possible values:

RUNNING--if the process is currently allocated a CPU and is executing

BLOCKED--if the process has no need of a CPU

CPU-READY--if the process needs a CPU, but is not currently allocated one.

(We will ignore other components of the process state for they are irrelevant for this discussion.)

Conceptually the SCHEDULER performs at least the four operations described in Figure 3. Each operation results in a changed process state value.

To multiplex the CPUs, the SCHEDULER provides the operations:

**allocate** a CPU to a process for some time quantum not to exceed a SCHEDULER determined maximum

**deallocate** the CPU from the executing process. (Deallocation occurs when either the RUNNING process blocks or its CPU allocation quantum has expired.)

The SCHEDULER synchronization operations are:

**block** which deallocates the CPU from the process to be BLOCKED, and changes that process state from RUNNING to BLOCKED

**wakeup** records the reason for the wakeup signal in the Process Table entry for the process to be awakened, and changes its state from BLOCKED to CPU-READY

The SCHEDULER is composed of a collection of chores as suggested by the four operations above, and a choremaster. Each chore is triggered by the presence of a message. The choremaster is a cyclic program which has a priority ordering of all the queues which the SCHEDULER services. Based on the content of these queues, the choremaster selects the next chore to execute.* Communication with the SCHEDULER is via messages placed in the Scheduler Input Buffer. Messages to **block** a process come

---

*The choremaster also permits SYSDDT to run as the SCHEDULER function and SYSDDT share the same upp.

from that process (executing Monitor code to build the message). Messages to **wakeup** a process may be sent by running user processes (via execution in the Monitor), the CHIO or the MMP. Note that the SCHEDULER's **block** and **wakeup** operations are unconditional. It is the Monitor which provides higher level event synchronization.

The SCHEDULER maintains several private message queues. The WAKEUPQ contains an entry for each CPU-READY process for which the SCHEDULER has not yet sent a request to the MMP to load. A set of priority queues record the identity of processes which are loaded. It is from these priority queues that the SCHEDULER selects the next process to be allocated a CPU. We expect to find 5-6 CPU-READY processes in these queues during normal operations. More or less would indicate under-utilization of one form or another. Less would indicate that working sets were not being loaded rapidly enough and that CPUs may soon be unproductively idle. More would indicate that loaded processes are occupying memory for a substantial time interval before being allocated a CPU. The memory might be better used by processes which will run before the loaded waiting processes.

Communication between the SCHEDULER and a CPU is via a 'message' placed in a special location. To allocate a processor to a process, the index of the process in the Process Table is placed in this location. CPU firmware loads the state of the process into the processor registers, filling in the hardware map table incrementally as pages are initially referenced. A CPU sends a message to the SCHEDULER indicating that it has stored the state of the process it previously was running into central memory and is idle, by setting an appropriate flag.

**The MMP**

The Memory Management Processor is responsible for the allocation of space in each of the three storage media (disk, drum and central memory), and for the flow of information between the media. Each of the memories is logically divided into 6K page-sized blocks called frames. The MMP maintains a hash table for each memory recording what pages of information currently reside in the page frames. As mentioned earlier, each page of information has a name which is unique to that page over the life of the system.

The MMP responds to requests from two sources. One source is the SCHEDULER which advises the MMP which processes are to be loaded and unloaded. To load a SCHEDULER selected process, the MMP first loads the Process Context Block page (named in the central memory resident Process Table). Then using the Context Block, the MMP loads all central memory working set pages of the processs that are not already in central memory. Processes are unloaded to free up central memory page frames.

The second source of requests to the MMP is the Monitor which, on behalf of a user, may send a message requesting that a page be created or destroyed, that a specific page be moved from disk to drum (for anticipated use) or that a specific page be added to the working set of the currently running process so that it will be loaded before the process first attempts to access it. (By choosing to alter its central memory or drum working set, a process can advise the MMP of where pages should reside for that process to execute most efficiently.)

Another responsibility of the MMP is to service the disk and drum controllers. The MMP places commands to transfer pages directly into controller registers. For example, just before the next drum sector comes beneath the read/write heads, the MMP will place a command to transfer a page to or from a specific page frame in central memory. Eventually the drum operation is complete. Then the MMP checks its results for error, possibly arranging to retry the operation when the sector next is available for transfer. The MMP must update its hash tables with the results of transfer operations.

Like the SCHEDULER, the MMP is composed of a collection of chores, a collection of message queues and a choremaster. The MMP has two shared queues for requests originating in the rest of the system, and has a number of private message queues as well. One of these, for example, is the Cleanup Queue. A Cleanup Queue exists for each device and contains a message indicating what command the controller is currently processing for that device so that it s results can be checked for errors when the command is completed.

The overall flow of control in the MMP is dictated by messages and by device characteristics. As noted earlier the MMP performance is particularly sensitive to the duration of chore executions. The MMP must be available to service the drum controllers within hardware determined time limits. Once a page transfer command is placed in the drum controller registers, the MMP must be available less than a millisecond later to process the outcome (successful or unsuccessful transfer) of the command. For this reason, MMP chores do not execute longer than a quarter of a millisecond. Thus certain activities involve execution of a sequence of chores. Consequently, the MMP uses private message queues so that some chores can terminate by placing a message in a private message queue to signal that a successor chore is to be executed, when the choremaster deems appropriate.

**Adventures of a User Process**

It seems appropriate to illustrate the interaction of the SCHEDULER and MMP. We will describe the events which occur as user process P is loaded and allocated a CPU. We begin by assuming that process P is blocked with its Context Block and central memory working set pages drum resident.

In response to a (message) request to wakeup P, the SCHEDULER sets the state of P to CPU-READY. At some later time the SCHEDULER informs the MMP that P's working set should be loaded when appropriate. The MMP first transfers the Context Block page for P into central memory and then transfers in the central memory working set pages as described in this Context Block.

The MMP then informs the SCHEDULER (by a message, of course) that process P is loaded into central memory. Eventually the SCHEDULER allocates a CPU to P and P executes until the CPU sends a message requesting that process P be blocked or that its time quantum ran out. Note that the SCHEDULER does not stop/interrupt a CPU under normal operations.

The SCHEDULER then informs the MMP that process P can be unloaded. At its own discretion the MMP transfers to the drum any pages (not shared with any other loaded processes) that have be$\overset{e}{\wedge}$modified since they were loaded and possibly reallocates the central memory page frames.

We will now recount this same sequence of events in sufficient detail to illustrate the relative autonomy of the two resource managers and their reliance on message communication. We wish to illustrate not only communication between the SCHEDULER and the MMP, but also message communication between chores entirely within the MMP. This example should also clarify the use and dependency of the upps on shared data structures resident in central memory.

For convenient reference we first list the SCHEDULER and MMP message queues that will be involved. Queues private to one resource manager are preceded by "*".

Scheduler Input Buffer--all messages directed to the SCHEDULER by other autonomous functions in the system (the Monitor, MMP, etc.) pass through this queue.

*WAKEUPQ--contains entries for processes which are CPU-READY (competing for CPU resources), but for which the SCHEDULER has not requested the allocation of central memory resources.

*Priority Queues--each queue contains entries for CPU-READY processes which are loaded. All entries in a specific queue are for processes with the same priority.

SWAPRQ (SWAP Request Queue)--contains SCHEDULER generated requests to load specific processes.

General Request Queue--contains SCHEDULER generated messages stating that specific processes can be unloaded.

*Drum Sector Queues--one per drum sector containing requests to read pages from the associated sector of the drum.

*Cleanup Queues--one per device containing requests to check completion of the command most recently given to the device.

*Context Block Queue--contains requests to transfer to central memory a process whose context block is resident in central memory.

*WRITEQ--contains an entry for each dirty working set page which is to be swapped from central memory to drum.

The following shared data structures reside in central memory:

CHT (Core Hash Table)--contains one entry for each page sized block of central memory. An entry contains the unique name and the state of the page of information currently in the associated page frame in central memory. (The CHT is shared by the CPU address mapping mechanism and the MMP.)

- Process Context Blocks and Process Table entries--record the state of a process, including a definition of the process' working sets. (The SCHEDULER, MMP and CPU share these tables, though were the tables subdivided, some of the sharing could be eliminated.)

The chronology of events in the user process' adventure is: 1. A message to wakeup process P is sent to the Scheduler's Input Buffer.

2. The SCHEDULER records data in the Process Table entry for P to notify P of the event which took place. The SCHEDULER changes P's process state from BLOCKED to CPU-READY and places P in its private WAKEUPQ.

3. At some later time the SCHEDULER selects P to be loaded into central memory, removes P from the WAKEUPQ and sends a message naming P to the SWAPRQ.

4. The MMP removes the SWAPRQ message and locates the context block (say, page C) for P using the Process Table entry for P. A request to transfer C from drum to central memory is placed on the appropriate Drum Sector Queue.

5. Drum Sector Queues are serviced as follows:

5.1 When drum sector D is about to rotate beneath the drum read/write heads, the MMP checks to ensure that the transfer should still be made, then places in the drum controller's register a command to transfer a a page from sector D to a newly selected page frame in central memory. The request is removed from the Drum sector Queue D and placed on the Drum Cleanup Queue.

5.2 At termination of the transfer (1 millisecond later) the MMP services the Cleanup Queue request by checking for successful completion. If no errors occurred, the CHT entry is marked to indicate that the associated page from holds the loaded page of information.

6. After C is in central memory, a request to load P is placed in the Context Block

Queue. At some later time the MMP selects this request and scans the core working set description in the context block page C. For each working set page already in central memory, a reference count is incremented. For each page not in central memory, but on the drum, a transfer request is placed on the appropriate Drum Sector Queue. The number of queued transfer requests is remembered to determine when the process is completely loaded. Transfers take place as described in 5 above.

7. After all working set pages for P are in central memory, the MMP records (in the Process Table entry) that P is loaded and sends a message to the SCHEDULER (via the Scheduler Input Buffer) to indicate that P has been loaded.

8. Later the SCHEDULER places an entry for P on the appropriate Priority queue.

9. P will be allocated a processor when it becomes one of the highest priority loaded processes.

10. Eventually P will request (via the Monitor) to be blocked or P's CPU allocation quantum will pass. The SCHEDULER changes P's state appropriately and will (probably) send a message to the MMP (via the General Request Queue) which informs the MMP that, if desirable, P can be unloaded.

11. Eventually the MMP receives this message. If it decides to unload P (to free up page frames in central memory), the MMP considers all pages which reside in central memory because they are in P's central memory working set, but in the central memory working set of no other process which is loaded or currently being loaded. (A reference count in the CHT entry for the page indicates the number of loaded or partially loaded processes whose central memory working set includes that page.) If such a page has not been altered, a copy of it already exists on the drum and the page need not be transferred. If the page has been written into (that is, the CHT entry for

the page records its state as DIRTY), then an entry for the page will be placed on the WRITEQ.

12. Later when a sector D of the drum is about to pass beneath the read/write heads and the associated Drum Sector Queue D is empty (of read requests) so that no read operation can be performed, an entry is removed from the WRITEQ and a command to transfer to drum the DIRTY page is given to the drum controller. When the page is successfully written to drum, the space taken by the old drum copy is released. The CHT entry for the page frame occupied by the just copied page is marked no longer DIRTY. Since the reference count for the page is zero, it is a prime candidate when a page frame is to be chosen for reading in a page. However, the CHT entry still indicates that the page frame is occupied by the just copied page. If that page is in the working set of some process soon to be loaded, then it will be found to already exist in central memory.

We have not narrated the above sequence of events so as to clearly delineate chore boundaries, but many of the chores should be clear. We have not explicitly mentioned the choremaster which selects the next chore to run, based on non empty queues serviced by that choremaster. However, it should also be clear that the policy for queue servicing, embedded in the choremaster, determines the order in which chores are performed and thus the rapidity with which a manager responds to a certain type of request.

**The CHaracter Input/Output Processor**

The CHIO supports full duplex communication between local and remote user terminals and user processes. Each terminal may have independent buffered uni-directional input and output channels related in that input characters may be echoed by being placed in the corresponding output channel.

Each channel has some state which user processes can read and set. The state includes the name of the process to wakeup when the channel needs service. Subsystem code, or Utility code, can read or write channels by making requests via the Monitor. Requests are packaged as messages and then sent to the CHIO. Common CHIO requests include the following:

Readstring(channel, limit) requests the CHIO to read (and remove from the CHIO 'channel' buffers) the next phrase (all characters up to and including the next break character) but no more than 'limit' characters.

Peekstring(channel, limit) requests the CHIO to perform the same action as Readstring without removing characters from the CHIO buffers.

Writestring(channel, string) writes 'string' into a CHIO buffer for the output 'channel'.

As mentioned before, subsystem control over the terminals allocated to the subsystem should not be usurped by the Monitor and/or its supporting CHIO, but enhanced. Echoing is one terminal service which affects the ways in which a program can interact with a user at his terminal. When the user types a character the user's program may wish to echo or surpress echoing of that character, or even to echo another character or sequence of characters in response.

To permit subsystem control of this terminal attribute, part of the state of a channel which the user can set is the break character set. The break set characters are used to delimit at phrase consisting of all characters up to the break character. A phrase is buffered in the CHIO. If a process is to be notified of the arrival of the phrase, a wakeup request (to the SCHEDULER) is generated.

Break character sets may be
--no characters
--all characters
--all non alphanumeric characters
--all control characters, including carriage return

For a terminal with a break set consisting of all non alphanumerics, the characters will be echoed until a non alphanumeric is typed. Though the user may continue to type, no further echoing will be performed, though input characters are buffered. When the user program responds to the break character typed, it may direct that echoing be resumed. Thus console output is a faithful chronological log showing what the program received and sent to the terminal, rather than merely a record of what a user typed.

Initially the CHIO was designed to communicate with concentrators called Data Communication Processors (DCCs).* Each accepts input and provide output to up to 200 low speed (30 characters per second) and a few higher speed terminals.** Besides providing the user interface sketched above, the CHIO together with programs in the concentrator were designed to attempt to maintain an error free communication link over the 4800 baud telephone connection between them. The logical channels are multiplexed over telephone links. A detailed description of the terminal system is given in a paper by Heckel and Lampson.

**System Monitoring**

The BCC-500 system is controlled by an operator who communicates with a terminal connected directly to the SCHEDULER upp which is executing a system debugging program, SYSDDT‡ (in addition to the normal scheduling chores). Commands to initialize the BCC-500 are entered on this terminal and procesed by SYSDDT. After

---

*DCCs are 16 bit microprocessors similar to the other BCC microprocessors, but with a slower cycle time.

**In the University of Hawaii configuration, the CHIO is directly connected to terminals and to the ARPA network through a front-end PDP-11 mini-processor.

‡An adaption of the SDS930 debugger.

initialization SYSDDT goes into system monitoring mode watching for events such as power fluctuation or parity errors in any of the upps.

SYSDDT is capable of signalling system shutdown and system restart using control lines running from the SCHEDULER to all other upps. Microcode in the recipient processors responds to such signals at appropriate points in their instruction execution cycles. Upon receiving the shutdown signal, a processor will halt with data structures in a well defined state. A restart signal will cause the processor to reset internal state and central memory interfaces, then to commence execution from location zero in the microstore.

SYSDDT, whose code is stored in the SCHEDULER's private memory, contains a full CPU emulation facility. Either CPU can be single-stepped under SYSDDT control and simultaneously emulated so that actual and emulated results can be compared. SYSDDT was used for system checkout and is still used when system debugging and maintenance are required.

**Management Processor Intercommunication**

Resource managers share data structures such as message queues, the Core Hash Table and the Process Table. The integrity of these structures must be maintained. The BCC-500 includes hardware to support the required synchronization. Shared structures are partitioned into eight groups. A processor requests permission before referencing data in one or more groups; the hardware grants permission to access each group only to a single processor at a time. A processor which has been granted permission to reference a group is expected to give up that permission rapidly. For example, message insertion or removal takes only a few microseconds, so exclusive access to the group including the message queue involved is required only for that short interval.

The actual implementation of this PROTECT mechanism is to associate with each processor an 8 bit register, one bit being associated with each group of shared structures. A processor can successfully 'set a PROTECT' bit in its register only if no other processor has the corresponding PROTECT bit set in its register. The setting of a PROTECT by a processor marks the beginning of a period during which it has exclusive access to the associated group of shared structures. Resetting the PROTECT marks the termination of that period. Enforcement of this interpretation of PROTECT bit usage and of deadlock avoidance is the responsibility of the processors (i.e. of the programs executed on the processors).

In addition to the PROTECT feature for synchronization, the BCC-500 also provides a hardware implemented feature to speed up determination of whether unserviced messages are pending. Communication queues are expected to be empty much of the time, so a choremaster should not have to poll each of the multiple message queues it services to determine their state. The BCC-500 associates a REQUEST latch with each processor. Whenever a message is inserted in a shared queue, the REQUEST latch of the processor which services that queue is strobed. A set REQUEST latch is interpreted to indicate the existence of an unserviced message. It should be emphasized that strobing a REQUEST latch does not cause an interrupt in the associated processor. Each processor chooses when to test and to reset its own REQUEST latch.

**Benefits of Dedicated Processors**

Using dedicated microprocessors for resource management functions accrues a number of benefits. Probably the most important is the modularity it encourages, even enforces, during system design and development. The BCC design places an (almost) autonomous resource manager in its own dedicated processor with communication via a message protocol. Thus the software module boundaries are the 'same' as the hardware boundaries. Usually designers define module interface conventions for parameter passing and communication, but they are often breached. The message protocol is simple and, we believe, easier to enforce because the hardware boundaries discourage its breach. This reduces the difficulty of managing software development just a little. Initial system checkout, maintenance and performance measurement of the resource managers are all simplified since each resides within a separate processor.

For example, a software error (or a hardware error) can be easily traced to a single processor. Because the code for each manager was designed to be autonomous, test cases can be devised for that function alone, with minimal involvement of the other processors. To checkout a chore or set of chores, one need only initialize data structures appropriately, including sending a message requesting execution of the chore(s) to be tested, then check the data structures for correct results. It was in this way that the MMP was checked out before the CPUs were even wired.

Of course, one processor can be used to monitor another. As noted earlier SYSDDT can be used to emulate the CPUs. In addition, performance measurement of software can be instrumented by one processor monitoring a second. In alternative designs where resource management is performed as an extension of a user process, it is difficult, if not impossible, to determine the intervals over which the resource management code of interest is being executed so that measurements can be taken.

The second major advantage of dedicated microprocessors is their raw speed (up to 20 MOPs in the BCC upps). The BCC-500 operating system design depends on this raw speed, for in several instances, processing power is used to 'replace' another resource, usually memory. As discussed earlier, one could not hope to run 500 interactive users out of 386K bytes of central memory unless the memory were usefully occupied by programs and data a very high percentage of the time. Dedicating an entire processor to memory management is the tradeoff made.

The structure of the resource managing software, in particular the chore structure and the message communication protocol, depends upon adequate processor cycles being available. Both message handling and chore dispatching incur additional cost in processing time over some alternative control and communication structures. It is the processor speed that made feasible running chores uninterruptibly to completion without making chores ludicrously small and without sacrificing rapid response of a resource manager to requests. (Making chores interruptible would, we believe, destroy the simplicity of design of the resource managers.)

Processor speed influences the choice of timing dependent algorithms that can be encoded as chores. For example, the MMP dynamically allocates a page frame on a particular drum sector S while the preceding sector S-1 is beneath the read/write heads. By the time sector S is about to come beneath the read/write heads, the write command associated with the allocation is prepared and then given to the drum controller. First, waiting until the previous sector is beneath the read/write heads to select a target frame is made possible by upp speed. Second, the sophistication of the frame allocation algorithm is limited by the number of instructions that can be executed by the MMP while one page is transferred.

Processor speed permits extending the BCC-500 to add new or augmented facilities. To illustrate we consider the service guarantees made by the BCC system to its users. A user can be guaranteed to receive a number of drum page allocations, a number of disk transfers and a CPU share ranging between a specified minimum and maximum during a period of time. In addition a user can be guaranteed to receive up to a fixed number of terminal lines, a fixed number of processes and a limited amount of uninterrupted CPU time. Few other systems make such guarantees--often because the operating system design and implementation preclude being able to fulfill such promises.

To consider one example, guaranteed CPU service, the SCHEDULER has to continually compare the resources a process receives to what is promised to it and schedule accordingly. The processor cycles are available to make such checks partly because the SCHEDULER is not 'stealing' user CPU cycles whenever it runs. Its algorithms do not have to be as streamlined as possible. In addition, the CPU time measurements are accurate to a finer grain than in conventional systems where time accounting is complicated by the fact that the resource being accounted is required to do the accounting. Most conventional system designs preclude accurately accounting for the servicing of low level interrupts.

Analogous to the SCHEDULER, the MMP checks a user's page allocation credit when a user's process asks for a new page to be placed in its core or drum working sets. A user may not exceed his credit. Similarly the CHIO manages user terminal response so that the user sees a consistent stable system, not one in which response varies significantly with system loading. Providing this consistency also costs yet a few more processor cycles. Thus each resource manager provides more than a resource allocation service, for it correlates allocations to multiple users and to their guarantees so that all guarantees are honored.

Another benefit of dedicating speedy processors to resource management functions is that algorithms can be extended to perform redundant encoding and additional state checking to enhance system reliability. For example, message queues are implemented as doubly linked lists so that at no time are both links to a message broken, unless the message is being inserted or deleted. As another example, each page in the BCC-500 has a unique name recorded with the page on disk storage. The MMP checks that the recorded unique name is the expected one whenever the page on disk is read or written. This check enabled quick isolation of several errors during system checkout. Since normal operations began, it has also been the first sign of certain types of hardware failure. In addition, unique names allow reconstruction of the disk hash table (which describes the disk contents) if it is lost.

Besides expanding an algorithm to perform its function in a way that enhances reliability, entirely new functions can be added when the processor cycles are available: for example, system load monitoring and audit trail recording for accounting and maintenance.

Security is a third advantage of the system organization that dedicates processors to relatively autonomous functions. Resource management code and private data are isolated in a processor inaccessible to any other processor. Microcoded resource management chores lessen the possibility that code can be destroyed, as does isolating management code in memory which user processes can never access. Hardware interfaces together with the 'thin line' communication channels in the form of message queues provide a natural boundary against penetration. It seems much more unlikely that a would-be penetrator could use the message mechanism to induce the resource manager (on a different processor) to inspect or change critical information in an unwarranted way, than if management code were executed as an 'extension' of the user process executing on the same processor(s).

Many potential security flaws are eliminated by only loosely coupling resource management code execution with the execution of user code. For example, a fairly standard system penetration technique is to change data which a management routine expects to remain constant during the execution of that routine. In conventional systems resource management is usually performed as an extension of the user process--i.e., the user directly invokes a management service causing suspension of execution at the user's call site. The designer or implementor of the called management routine may make (erroneous) assumptions about data at the call site not changing (e.g., it may change as a result of a concurrent i/o operation). When communication is via messages between programs known to execute in parallel, programmers are not tempted to make such assumptions and to encode subtle, even timing dependent, errors.

## Caveat

We have written this overview of the BCC-500 from a vantage point of five years of perspective which includes experience with the BCC-500 implementation. We have taken some liberties with our description, but have essentially been faithful to both the design and the implementation.

The terms 'chore' and 'choremaster' do not appear in any previous BCC-500 literature and were not in the parlance of the designers. The chore-based design is very clear in the MMP, but is not so cleanly visible in the SCHEDULER and the CHIO.

The message protocol is used by all resource managers to communicate directly with one another. And the message protocol is used by the CPU-executed Monitor code to communicate with the resource managers. Queues, however, are not implemented homogeneously. For example, queues in which messages are essentially the names of processes are sometimes implemented as a list, linked through a field in the process table entry for the process. (Jack--Correct?)

There are several data structures, in particular those which are currently shared by the SCHEDULER and the MMP (e.g., the Process Table), in which most sharing could be eliminated. Reducing sharing to a minimum would substantially increase the autonomy of the resource managers.

**Summary**

This paper presents a system design based on a number of ideas and its seems worthwhile to restate succinctly the motivation behind the major system design decisions.

The development of computer systems is always influenced by economics. When the cost of computer system development and maintenance is dominated by hardware costs, system designs reflect the designer's goal of optimizing hardware usage. Technological advances have caused the decrease of hardware cost so that software development and maintenance costs dominate. System designers should now attempt to minimize the software development and maintenance cost by designing systems that are as simple as possible to understand, implement and maintain.

We illustrated that to provide the kind of service BCC desired, the Monitor was responsible for providing a vast set of facilities. Given the premise that processing power is cheaply available in comparison to other resources*, it is reasonable to employ system organizations in which as many components as possible are in a form easily understood by people, yet which perform economically. We have argued that one such orgainzation is to isolate large functions such as the resource managers and

---------------------

*With the evolution in technology, this is generally, though not always, true.

to implement them as autonomous entities which depend on, i.e., communicate with, the remainder of the system using a very simple, but well-defined protocol.

In the BCC-500 each resource manager is autonomous. Ideally it has a single source of input, the message queues. Its influence on the other system components is mostly in terms of the messages sent. Each manager can be considered independently for the purpose of understanding how it works, implementing and maintaining it.

Dedicating a processor to each resource management function enhances both the apparent separation (in the eyes of someone trying to understand the workings of one manager) and the actual isolation (in terms of the physical location of private data) of the manager. Because processors are dedicated to specific system functions, there is sufficient processing power to employ algorithms which would be unacceptably expensive in other system designs. Such algorithms can be used to guarantee resource allocations to the users, to enhance reliability of the system and to optimize deployment of more expensive resources such as memory.

**Acknowledgements**

We would like to thank Jack Freeman, Wayne Lichtenberger, Lee Cooprider and Larry Flon for their criticisms and suggestions on early drafts of this paper.

Many people contributed to the design of the BCC-500 operating system design and supporting hardware. It seems appropriate to name them. A major single contributor was Butler Lampson. Other designers and implementors who made significant contributions include Larry Barnes, L. Peter Deutsch, Rick Dove, Jack Freeman, Paul Heckel, Wayne Lichtenberger, Mel Pirtle, Rainer Schultz, Charles Simonyi, Roger Sturgeon, Chuck Thacker, Steve Tulloh and Bob Van Tuyl.

## Fast Memory
Appendix I

The BCC-500 **central memory** consists of a **fast memory** backed up with a **core memory**. The processors and drum and disk controllers directly reference only the fast memory, which is physically small and located near the processors and controllers, but which has very limited capacity. The fast memory will accept up to four requests (one on each of the four ports of central memory) each 100 nanoseconds, and will process each request in 200 nsec. Therefore, the central memory can process up to 4 x 10↑6 requests per second. However, because of its limited capacity, a (hopefully) small number of requests will not be satisfied. In this event, the processor or controller is obliged to resubmit the request.

The central memory is implemented as eight double-word, interleaved modules. Each module consists of a 1 microsecond core module and a six (double-word) register fast memory module*. Each double-word in core memory is permanently bound to a unique address. The assignment of fast registers to memory addresses dynamically changes in response to requests. Since assignment of a fast memory register to a memory address must precede a successful fetch or store to memory, proceessors and controllers can issue prefetch and prestore requests to apprise the central memory that a fetch or store to a specific address is imminent. These requests (prefetch or prestore) are satisfied if the fast memory succeeds in assigning a register to the double-word address named in the request in preparation for the coming fetch or store.

One objective of central memory is to maintain a value for each address. Ideally,

----------------------

*There are only two fast registers per module in the University of Hawaii configuration.

whenever both a double-word in core memory and a fast register are bound to the same address, both should contain the same correct value. Thus part of the fast memory logic maintains consistency between core memory and fast memory. For example, if as the result of a request to store value V at address A, the value V were stored in a fast register assigned to A, then to maintain consistency fast memory logic will copy the value V to the core word addressed as A. Such an update takes 1 microsecond, the core cycle time.

To help maintain consistency efficiently, a state is associated with each fast register. Besides including the address to which the register is currently assigned, the state specifies whether core or the fast register, or both, have the correct value for the assigned address. In addition, the state records whether the current assignment was made as the result of a fetch (or prefetch), a store or a prestore. The fast memory logic responds differently to the three cases. For example, in case of a prestore, the contents of the core word associated with the assigned address does not need to be brough from core, as it does for a fetch or prefetch.

Although central memory data paths are a double-word wide, processors and controllers request only a single word at a time. The state also includes a 'hold' attribute. 'Hold' can be specified in any request. It indicates that the requestor will imminently make another request involving the same double-word. Thus the fast register assignment to that double-word should not be revoked until the imminent request is successfully completed. (Processors and controllers are responsible for indicating when the hold should be released.)

The hold is particularly useful for drum and disk controllers who access memory sequentially. For example, consider the requests which a controller for a swapping device makes to memory. When swapping out a page, the controller reads words sequentially. It knows far ahead of time which words are to be fetched and can issue prefetch requests ahead of time the time it needs to fetch the actual data values. The controller also specifies a hold in the first fetch request so that the data it needs will remain in fast memory until the contents of both words have been fetched. The second request does not specify hold so that the register assignment can be broken. A similar scheme is used for stores: if a processor or controller is going to store a double-word, a hold is specified in the first store request so that the consistency logic does not attempt to copy the value to core until the second half of the double-word is also stored.

The state of a register also includes priority information. The fast memory logic will attempt to satisfy high priority requests, even at the expense of low priority requests, so long as no data values are irretrievably lost. This is particularly useful when data is being transferred to or from an io device for which a high penalty must be paid (e.g., an extra disk revolution) if the data is not fetched from or stored into memory by a certain time. Both the register assignment and the consistency logic take priority into account.

As in other parts of the BCC design, central memory processing power (the fast memory logic) is used to optimize a scarce resource (the fast registers) so that the effective access time of central memory is greater than that of core memory which comprises the bulk of central memory.

**Hardware Description***
**Appendix II**

**Component Planned Configuration Hawaii Configuration**

Central Memory up to 256K 24 bit words 128K words (8 modules) (8 modules) 4 ports: peak transfer rate of 30M bytes/sec/port (average transfer rate of 16M bytes/sec/port) 96 fast registers with effective 32 fast registers 100 nanosecond access time

Microprocessors 100 nsec cycle time (2 CPUs, CHIO, 2048 96 bit microwords SCHEDULER, MMP) 128 word scratchpad memory 20 MOPs in bursts

Drum and Disk Permits 2 drum and 2 disk 1 drum and 1 disk can Controllers transfers simultaneously transfer simultaneously

Drum maximum of 8 drums 2 drums 6M byte capacity each 6M bytes/sec maximum 5M bytes/sec maximum transfer rate transfer rate 33.3 msec rotation

Disk 393M byte capacity 1 disk with dual positions, 6 heads/24 bit transmission e.g., 2 logical disks, 50 ms rotation each half size 300K bytes/sec maximum transfer rate

Data up to 200 terminals each none, however Communication less than 300 baud (1) HP2100 interfaces to Processor 4800 baud link to CHIO tape unit and LPT (2) PDP-11 interfaces to ARPAnet IMP

Ignore this
follow the formatting
commands.

---

*As specified by Professor Wayne Lichtenberger, University of Hawaii.

## References

[Boehm] Boehm, Barry W., "The High Cost of Software," **Proceedings of a Symposium on the High Cost of Software**, September 1973.

[Freeman, Davidson] Freeman, Jack and John Davidson, BCC500 Protection Mechanisms, ALOHA System Task II Technical Report R-3, University of Hawaii, May 1974.

[Habermann] Habermann, A. N., **Operating System Principles**, Science Research Associates, 1976.

[Heart et al.] Heart, F. E., S. M. Ornstein, W. R. Crowther and W. B. Barker, "A New Minicomputer/Multiprocessor for the ARPA Network," **AFIPS Proceedings**, Vol. 42, 1973.

[Heckel] Heckel, Paul C. and Butler W. Lampson, **The BCC Terminal System**, to be published.

[Hoskyns] John Hoskyns and Co., Ltd., **Implications of Using Modular Programming**, Guide No. 1, Hoskyns Systems Research, Inc., 600 Third Ave., New York, New York, 1973.

[IBM] IBM Virtual Machine Facility/370: Command Language Guide for General Users, File no. S370-36, Order no. GC20-1804.

[Jensen] Jensen, E. Douglas, "A Distributed Computer for Realtime Control, **Second Annual Symposium on Computer Architecture Conference**, Fainesville, Florida, December 1974.

[Jones, Wulf] Jones, Anita K. and William A. Wulf, "Towards the Design of Secure Systems," **Software Practice and Experience**.

[Lampson] Lampson, B. W., "Dynamic Protection Structures," **AFIPS Fall Joint Computer Conference** 1969, 27-38.

[Lee] Lee, Wrenwick K., The Memory Management Function in a Multiprocessor Computer System - A Description of the BCC500, ALOHA System Task II Technical Report R-2, University of Hawaii, September, 1974.

[Lichtenberger] Private Communication.

[Organick] Organick, E. I., **The Multics System: An Examination of its Structure**, M.I.T. Press, Cambridge, Mass., 1972.

[Wall] Wall, Charles F., Design Features of the BCC500 CPU, Technical Report R-1, ALOHA System Task II, University of Hawaii, January 1974.

[Wall, Freeman] Wall, Charles F. and Jack Freeman, BCC500 CPU Reference Manual, Document BCC/M-1, ALOHA System Task II, University of Hawaii, December 1973.

[Wulf et al.] Wulf, W. A., E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson and F. Pollack, "The Kernel of a Multiprocessor Operating System," CACM 17, 6, June 1974.

[Wulf] Wulf, W. A. (editor), "The Hydra Operating System," to be presented at Fifth Symposium on Operating System Principles, Austin, Texas, November 1975.