Wayne –

I think this will be fine to give to Uncapher if he asks for an overview document. Probably should print up a special copy with the edits I've marked on pp 1, 16, 19, 25. Then have someone stamp PRELIMINARY DRAFT all over it.

@S:W/MMP/AMC/

```
PAPER:9SYM      THURSDAY  MARCH 20, 1975  8:11:17
PAPER:9SYM      THURSDAY  MARCH 20, 1975  8:11:17
PAPER:9SYM      THURSDAY  MARCH 20, 1975  8:11:17
PAPER:9SYM      THURSDAY  MARCH 20, 1975  8:11:17
PAPER:9SYM      THURSDAY  MARCH 20, 1975  8:11:17
```

THE BCC 500 COMPUTING SYSTEM: AN INTRODUCTION AND OVERVIEW

1.0 INTRODUCTION

This document describes the architecture of the BCC 500 system, stressing its multiprocessor structure and the effects of this structure on the organization and implementation of the BCC 500 operating system. Details of the system's hardware components and descriptions of various aspects of the user machine provided by the system are found in other documents and papers [[redacted] list of references]. In order to give the reader some orientation, however, the remainder of this section is devoted to a short account of the system's origins. Brief mentions of its intended uses, its user machine features, and its hardware structure are given in Appendix I.

1.1 Background

The architecture of the BCC 500 system was largely determined by a design team working under Project GENIE at the University of California, Berkeley in 1968 [another list of old Genie references, 6700 documents, Van Tuyl's thesis, etc.]*. The Project's goal at that time was to develop a basic, cost-effective resource-sharing system structure which could serve as the nucleus for a number of operating systems tailored to specific applications areas.

----

*The Project was associated with several commercial organizations in a joint venture to produce what was then termed the SCC 6700.

In early 1969 a number of individuals from Project GENIE joined with others to form Berkeley Computer Corporation (BCC) to develop remotely-accessed resource-sharing computing systems. The system architecture concepts were carried over from the Project. The specific 500 hardware and software were developed in the period January 1969 through July 1970 as a prototype for the company's subsequent line of systems. BCC was unable to secure sufficient funding to establish a viable market based on the concept of centralized resource-sharing systems and was forced to terminate operations in March 1971. The 500 system was operated, however, for demonstration purposes and for BCC's own programming development activities from mid-1970 to the company's demise.

In 1972 the company's hardware assets -- including the 500 prototype -- were purchased by the University of Hawaii, and the equipment was dismantled and moved to Honolulu, where it was refurbished and reassembled by THE ALOHA SYSTEM Project to be used as a research and computing tool. The system became active again in February 1973, and it has been used since that time by the Project, by other ARPA contractors (via the ARPA network) and in classes of the Department of Electrical Engineering.

2.0 SYSTEM ARCHITECTURE

The simplest view of the 500 system architecture is shown in Figure 1. In the figure we see six processors clustered around a common memory. The multiprocessing nature of the structure derives from the concurrent use of the processors in support of a common task and the use of a common memory for interprocessor communication. The common task supported by this computing structure is, of course, the provision to each system user of a ficticious, powerful *virtual machine* which is the target of systems programmers as they prepare compilers and other large sub-systems for the convenience of the users. Providing and controlling the virtual machine -- system resource allocation -- is one of the more prominent roles of an operating system on any type of computing structure.

Figure 1. BCC 500 System.

The six processors are built almost identically. In the system they are individually dedicated, however, to various roles. The nature of this dedication and its implications are among the more interesting aspects of the system architecture. Two of the processors have enhanced hardware capability and are used to implement the more visible aspects of the virtual machine; they are called accordingly CPUs (although the term is not entirely apt in this context). The other processors are dedicated to various tasks of resource allocation.

2.1 Processor Assignments

Figure 2 shows the names of the 500 system processors and suggests their assignments.

Figure 2. BCC Central-site Processor Assignments.

The processors are:

CPUs

The two Central Processing Units are used almost exclusively to execute the code provided by the system's users. The rather significant CPU power required in other systems for the system's own management functions is provided in the 500 system by more specialized processors. It is necessary, of course, for the CPUs to communicate with the other processors, and it is unreasonable to require the user of the system to write such code into each of his programs (especially without error). Hence as a feature of the CPU, code is automatically included into each user's program to do the requisite communication tasks and a number of others which are convenient to the user. This system-provided code is considered to be logically a part of each user's job - called process in our terminology -- and is shared by all processes. It is fully protected by various CPU hardware features; it cannot be inspected or modified by user code, but only entered at discrete entry points implemented as system calls. These calls take on the nature of virtual machine "instructions," checking carefully the user's intended actions and his authority to do them before interacting in any way with the rest of the system.

Beyond the protection mechanisms alluded to above the instruction and addressing capabilities of the CPUs are of little importance to the system architecture. In this system virtually any CPU could be used (assuming hardware compatibility with the memory system and the requisite protection mechanisms).

CHIO

The CHaracter-oriented I/O processor has two main functions: (1) to communicate and manage the terminals which may be connected to the system, and (2) to perform a number of functions related to dynamic buffering of character I/O streams to and from the user processes on the CPUs. Originially the CHIO was designed to communicate with a number of remotely located computers termed Data Communication Computers (DCCs) to which the actual terminals were connected. Thus in connection with (1) the CHIO was to supervise and control the various DCCs, load their local memories, multiplex I/O for individual terminals, acknowledge correct receipt of packets from DCCs, initiate retransmissions to DCCs, etc. In Hawaii it was not necessary to implement a full-scale communications network just to accommodate the local terminals; they were wired directly to the CHIO processor and its activities were modified to deal with the terminals directly. Remote terminals are accommodated by means of the ARPA network, which is interfaced to the 500 system through the CHIO processor which communicates via a wired connection to an IMP port on the network's ALOHA-TIP.

SCHED

The SCHEDuling processor's principal function is to schedule the two CPUs amongst the various active processes. Thus it is responsible for fielding wakeup conditions as they are generated and for making decisions on the order in which processes are passed through the central memory from, say, drum where they commonly reside. This information is passed to the memory management portion of the system for appropriate action. On a different level of activity the SCHEDuler makes final decisions as to which of several processes ready to run in the central memory is assigned to each CPU. This decision is normally made as the process on each CPU blocks. It is possible for the SCHEDuler to pre-empt a running process on a CPU; this does not normally happen, as the algorithms for scheduling are designed to permit CPU tasks normally to dismiss themselves or run to the end of a preset time interval. The SCHEDuler generates its own real-time wakeup conditions.

MMP

The Memory Management Processor is responsible for management of the entire system memory. We define memory here to include the contents of drum and disk as well as central memory. The MMP together with the storage under its control can be viewed as a separate system with a table-driven interface to the rest of the 500 system. The memory system is described more fully in Section 2.3 below. Here we simply point out that the function of the MMP is to get the right information into the right place at the proper time.

SMP

The System Monitoring Processor continuously monitors a number of indicators which are set on the occurrence of some malfunction. It is equipped with a number of special control lines leading to the other processors and has the ability to control these processors as well as monitoring the system's health. The SMP may thus effect automatic crash recovery procedures, in many cases before significant damage to the contents of various system tables has been done. The SMP also contains a special system diagnostic and control routine called SYSDDT. This routine is equipped with an interactive interface via a special terminal to a software or hardware diagnostician who can control individual processors in the system. It contains a full emulation facility, for example, for a CPU. Either of the CPUs, when suspected of being faulty, can be single-stepped by SYSDDT and simultaneously emulated such that after the execution of each instruction the state of the real CPU can be compared with that of the emulated one. This provides the ability to quickly locate CPU faults. SYSDDT is a general-purpose program running from a private memory module attached to the SMP. It permits other diagnostic procedures to be designed on the spot if need be and readily inserted into the processor.

2.2 Realization of the Dedicated Functions

The determining factor for the 500 system architecture was the development of a simple, but powerful microprocessor which is used as the basis of all of the system's processors. The microprocessor, like most, has active registers, scratchpad storage, basic arithmetic and logical processing ability, and an interface both to the central memory and to a private memory. It also has approximately 2K of diode read-only memory (ROM) from which it fetches its micro-instructions. The ROM has a minimum cycle time of about 65 nsec; but the entire processor is operated synchronously with a 100 nsec clock. This means that each 100 nsec a new micro-instruction may be fetched and executed.

Each word in the ROM contains one micro-instruction. There are 90 bits in each word, however, so that very little encoding of the bits in the micro-instruction is used. This, together with the existence of several busses connecting the various storage and processing elements, permits up to three or four operations to be done in a given micro-instruction. Thus the processor is theoretically capable of bursts of computation of up to 20 or 30 million operations per second. In practice this speed is not attained for long durations, as it is not always possible to cram three or four operations into every instruction and the processor must wait for memory responses when it references central or local memory. Its average rate of processing is consequently somewhat less, depending on the frequency with which it accesses memory.

It was decided early in the 500 system design to place the dedicated functions of each processor directly into the processor microcode. This gives a distinctive cast to the system, as it means in effect that the bulk of the operating system exists in the processor hardware (more precisely, the firmware). Clearly, the processors operating in this mode have high capability (their instruction bandwidth is high since they refer to memory only for data). But the operating system algorithms are difficult to change, as the ROMs can be modified only by removing and inserting diodes.

The difficulty of changing the microcode may be viewed as a beneficial constraint. It is the same constraint that requires a hardware designer to exercise extreme care and regularity in his designs, yielding results which are more nearly correct and more maintainable. Yet, it is probably unfair to press the analogy too far. Thus, in each processor there is found microcode which, in more classical fashion emulates the instruction set of a simple, conventional processor. This microcode goes to memory for instruction fetching for the emulated processor; and thus each processor can execute conventional software.

In practice, both techniques are used. Placed directly into firmware are those functions which are fundamental, clearly-known, time-consumptive operations. In the software are those functions which are not so well defined initially or which are most subject to change. The emulated processor has an instruction to call microcoded subroutines directly, and the microcode can also start up and stop the emulated processor. Thus one can take two views of this hybrid approach: the view that the microprocessor emulates a standard processor which contains an extraordinary set of additional instructions (the microcoded subroutines) to help it perform its dedicated tasks, or the view that the microprocessor performs its tasks directly from microcode, some of which is simulated by the standard processor due to the necessity to change or parameterize it. We prefer the lattter view, but either results in the same end: the time-consumptive operations are executed directly from microcode while that portion of the algorithms most subject to change is kept in software. This software resides in local memories for those processors so equipped; it is otherwise found in dedicated areas of the central memory. Similarly, data storage required for the sole use of a give processor is kept in private memory, whereas central memory is used for that storage which must be shared between processors.

2.3 The Memory System

Figure 3 shows the principal components of the memory system. The processors access only the central memory (CM), shown in the figure as a four-port memory (up to four memory transfers can occur each memory cycle). Three of the ports are used for processor access, and the fourth is externally multiplexed to accommodate the collective transfers of the rotating memory devices. Because of the large number of user programs being accommodated it is necessary to transfer large quantities of information rapidly between CM and drum and disk. One viewpoint which can be taken of the memory system is that the rotating memory -- primarily drum -- constitutes the system's main memory and that the CM is but a window on this memory through which the CPUs are permitted to access user programs subject to usual scheduling considerations. With such a point of view (see Figure 4) we see the desirability for continuous transfer of information through the CM.

Figure 3. BCC 500 Memory System.

In Figure 4 we note that user programs (more precisely, processes) found in the CN may be classed into one of four categories:

1) incoming processes (partially loaded)

2) fully loaded processes (ready to run)

3) active processes (being run by a CPU)

4) outgoing processes (partially unloaded).

Clearly the individual processes do not move physically through the CM as Figure 4 suggests.

Figure 4. Conceptualization of the Memory System operation.

They are placed into randomly available page slots by the ~~AMC~~ and mapped into their proper position in the logical address space by page maps associated with each CPU. To facilitate the handling of pages in central memory, the data format on the drums and disks was designed so that pages are the only unit of information treated as addressible entities within the memory system.

Figure 4 illustrates how the size of CM is minimized by the rapid swapping: the CM need be large enough only to hold the two active processes plus enough additional space to "buffer" the processes under transfer, i.e., to ensure that with good probability there is always a ready process to which to assign a CPU when its current process blocks. (The system actually deals with <u>working sets</u> of pages -- pages of a process on which all memory references are localized during an active quantum of a process.) The amount of CM used to buffer incoming and outgoing processes is kept low by dynamically allocating space on the drums. The ~~AMC~~ writes out process pages at any sector address currently available and under the read/write heads; similarly it reads in processes in the order in which a process's pages happen to come under the heads.

The identity of the pages in a given process working set together with their location in the memory system is maintained by the ~~AMC~~ in various resident tables in the CM. (This space is not shown in Figure 4.) The ~~AMC~~ must refer to these tables constantly and to the rotational position of each of the 16 rotating devices in order to keep the flow of pages moving at an optimum rate.

3.0 SYSTEM OPERATION

The purpose of the system is to provide an environment for the execution of user processes which, in turn, typically access and modify user files. Processes and files are thus principal system objects consisting of pages of information contained within the memory system together with additional descriptive information. The process consists of system code shared with all processes (that code which communicates with the management processors), system code which may be unique to the process or shared with a number of other processes, user code, and all variable storage. It also includes a so-called context block containing the CPU state when the process is inactive, unique names of all the process pages, constitution of the present working set of pages (those pages which must be loaded into central memory before the process can be considered for activation), and mapping information. The file consists of those pages holding the file's contents plus directory information, again naming the pages and ordering them within the file.

The files can be created and destroyed (by processes); in the meantime they reside permanently within the system. The place of residence of inactive files is, of course, disk. When files are accessed the memory system moves them first to drum and then to central memory for the actual access. As the file is no longer accessed and room on the drum is needed for new files or processes, the file is moved by the memory system back to disk.

Processes are created by a single system subprocess by user request, usually when he logs into the system. Each process has a subprocess structure to facilitate both the system design and the needs of the user. The structure is a simple, linear one in which each subprocess is the inferior of at most one other subprocess and in turn is (immediately) superior to at most one subprocess. Subprocesses may have their own memory spaces or may share portions or all of memory with other subprocesses. They may communicate by means of messages passed through shared memory or by (software) interrupts. At the end of a computation the process terminates itself or is terminated by the user when he logs out. An option permits the user to log out of the system, leaving his process undestroyed but in a dormant state. The process then takes on the nature of a file, i.e., it takes on a symbolic name and is placed in a directory for later reference. At a later time a similar option at log-in permits the user to re-attach himself to the dormant process or establish a new one.

It has been implied in all of the foregoing what each of the processors does with respect to a process or a file. We state here more explicitly the activities of each management processor relative to processes and files.

CHIO

The CHIO passes character streams between processes and I/O terminals. It also passes files between the system and external storage media such as magnetic tape or special I/O devices such as line printers. The CHIO processor also communicates with the ARPA network.

SCHED

The SCHEDuler selects processes for running and directs this information to the MMP. It also attaches processes loaded and ready to run to CPUs. It does this by placing a pointer to the process context block in a special central memory location and setting a request latch in the given CPU, which then proceeds to load its own state from the context block and resume computation. The SCHEDuler sets in this state a time interval which is picked up by the CPU and counted down in real time, at which time the CPU blocks the process, dumps its state in the context block and notifies the SCHEDuler that it has blocked. The SCHED processor does not deal with files.

MMP

The MMP swaps processes and files or portions of files included in a process working set. It also goes to disk when required to move files or processes to drum, and vice versa. Both core and drum are dynamically allocated. The locations of pages in these areas are kept track of in special hash tables in central memory keyed by the page unique names. Disk, on the other hand is allocated in a more static fashion: each page existing in the system has a fixed, permanent residence on disk. The location of each page on disk is determined when the page is created and remains so until the page is destroyed.

3.1 Interprocessor Communication

In a multiprocessor system the processors by definition interact. The next several sections explore the nature of the communication between the 500 system processors. This communication closely parallels that between different modules of a conventional operating system.

Basically the processors communicate by means of central memory, i.e., by changing bits in shared tables. To speed up a processor's having to look through extensive tables for changes, a hardware flag system is provided. This mechanism consists of a small number of latches in each processor which can be set by one or more other processors. The latches -- called _request strobes_ -- are tested and reset by each processor's microcode. Flags in central memory augment this rudimentary facility into a more general one.

3.2 Processor Interlocking

When more than one processor accesses a data structure it is generally necessary to utilize an interlock mechanism -- a means by which only one processor at a time can modify the data structure into a new state which is meaningful to all processors. (In effect an interlock allows a processor to seize a given data structure, rendering it inaccessible to all other processors until the processor has completed its updating task). In the 500 system special hardware interlocks operating at microcode speeds are provided. These interlocks -- called _protects_ -- consist of eight central latches which may be set or reset by any processor but by only one at a time. That is, if more than one processor attempts to acquire the same protect at the same time or if the protect is already set, a hardware contention circuit resolves the conflict and issues a positive or negative acknowledgement to the appropriate processor. Processors receiving positive acknowledgements to protect requests then "own" the protect (and the data structure associated with it) until they voluntarily relinquish it. Combinations of protect latches are used to increase the number of effective protects beyond eight.

The particular protect mechanism used is greatly simplified by the fact that each processor (and in fact, *every* system component) is operated with a common clock. Thus processors make protect requests in exact synchronism. The hardware contention circuit shifts its notions of contention priority in such a way that each processor is treated with the same priority on the average.

3.3 Illustration of Processor Interaction

Figure 5 shows the processors (except SMP) and some of the more important communication signals between them. Each processor is dedicated to its own task and is designed to operate autonomously, independent of the other processors. No one processor is "in cntrol" of the other processors or of the system (except for, that is, the SMP which exercises its privileges only when a problem develops -- see Section xxxx on restarts).

Figure 5. Schematic of Processor Interaction

Consider the actions of the processors and the communication between them in response to an interactive user typing a command on his terminal. We will assume that at this time his process is blocked for reasons of I/O and that the process is physically located on drum.

1. CHIO

As the user types, his characters are accumulted by the CHIO. (In Hawaii the CHIO also echoes these characters; the DCCs were originally designed for this under control of the CHIO.) As part of its work the CHIO checks each character to see if it is a "wakeup" character. (Wakeup characters are defined for the CHIO by the program the user is interacting with.) Until a wakekup character is received by the CHIO it simply buffers and echoes the characters being typed. While this is going on, only the CHIO gives attention to the user. No other processors -- in particular, the CPUs -- are involved. They are executing code for other users.

When the CHIO receives a wakeup character (say, EOL in our case), it initiates the sequence of events which are required to bring into execution the program the user is interacting with. The CHIO does this by passing a simple message to the SCHEDuler; a pointer to the appropriate process is placed into a small queue.

2. SCHEDuler

The message from the CHIO is retrieved by a dispatcher task in the SCHEDuler and passed to the SCHEDuler's WAKEUP task. This task just (1) records a bit which identifies the source of ~~a~~ the wakeup ~~software interrupt in the user's process~~ and (2) queues the process on a queue called the WAKEUP queue.

An independent task (the SCHEDULING task) later removes the process from the WAKEUP queue and places it on the appropriate one of several SCHEDULER queues. Still another task removes the process from this queue and sends a message to the SWAPPER task in the MMP, requesting that the process be loaded into central memory.

3. MMP

The MMP, in response to the request from the SCHEDuler, sets about the task of reading the process working set of pages into central memory. To do this it will have to create space in the central memory by writing out the pages of previously active processes. When this rather complex action has been completed, the MMP notes back to the SCHEDuler that the process is loaded. Note that during the time the MMP is reading pages of the process into memory, only the MMP is concerned with the process. In particular, again, the CPUs are running other processes.

4. SCHEDuler

Loaded processes become the responsibility of the SCHEDler task called the MICRO-SCHEDULER. This is the module that actually controls the CPUs. It keeps track of the "priorities" of the processes executing on the CPUs and of the processes which are loaded and ready to be executed. When a CPU becomes available for our process (by virtue of its being free or because its current process has lower priority than ours) the MICRO-SCHEDULER hands the process to the CPU and tells it to run.

5. CPU

When the CPU receives the "switch processes" message from the SCHEDuler it picks up the state of our process from its context block and starts executing it. If it is running a process already, it waits until the process is not executing from its system code; and as soon as this is true it stores the state of the CPU into the context block of the process. It then sends a message to the SCHEDuler, letting it know that the CPU has blocked the process it was running. On picking up our process, the CPU then communicates with the CHIO to receive the message typed by the user which evoked all this activity.

3.4 Explicit Communication Between the Processors

In this section we consider the processor communication interfaces in more detail. We consider the various possible pairwise combinations.

1. CHIO - CPUs

In these bi-directional exchanges the system code running on a CPU is always the initiator. At the command of a user program, the system code requests the CHIO to accept some characters for delivery to a terminal, to deliver to the CPU characters which the CHIO has received and buffered, or to change the state of some terminal parameter such as echo strategy, wakeup strategy, etc. The system code puts its request, together with any associated data, into the CHIO's request buffer (a block of words in central memory); sets a "request waiting" flag associated with this buffer (a bit in CM), and sets the CHIO's request strobe latch to let it know it should look at its request buffer. If the nature of the request is such that no reply or response from the CHIO is expected, this completes the interaction. The system code returns control to the user code that called it. The CHIO procedes at its leisure (although very quickly) to perform the requested operation. When it finishes, it resets the "request waiting" flag. For those requests to which the system code in the CPU does expect a reply (such as one which asks for the delivery of any characters which may have come in from a given terminal), the system code waits while the CHIO performs the request. The CHIO puts its response in the same message buffer and resets the "request waiting" flag to let the system code know that the response is ready. The system code then delivers the response and possibly the characters to the user code.

2. CHIO - SCHED

This communication is only one way: from the CHIO to the SCHED. When the CHIO finds that it has collected a complete message for a process, i.e., when it gets a wakeup character from a terminal, the CHIO sends a notification to the SCHED. It does this by placing a short message containing the process' ID into the SCHED's message input buffer. This buffer is also used by the other processors in communicating with the SCHED to communicate a variety of things. A "wakeup message" from the CHIO must thus contain more than just the ID of the process for whom the CHIO has collected a message. It contains an "opcode" which means that the message is a wakeup message plus a specification of the reason for the wakeup message.

The CHIO sends wakeup messages to the SCHED for other reasons also. Among them are

- the input buffers allocated to a process are (nearly) full

- the output buffers allocated to a process are empty enough that the process can proceed to do more output

- a special control character(a QUIT or ESCAPE character) has been received from the process' terminal.

The SCHED's message input buffer is used by all the system processors. It is protected from simultanous access by means of one of the hardware PROTECTs. Thus, whenever a processor wants to reference this buffer, it first acquires the associated PROTECT, makes its reference (which only requires four or five memory references), and then releases the PROTECT. After placing a request in the buffer, the processor sets a request strobe latch in the SCHED to let it know that there is a message for it.

3. CHIO - MMP

These two processors have no need to communicate with each other.

4. CPUs - MMP

Message exchange between these two processors is always initiated by the system code in a CPU making a request on the MMP. Messages from the MMP to the CPU are always responses to such requests.

What the two processors talk _about_ are pages of memory. Each page in the memory system has been given a unique name at creation time. Pages are always referred to by name. For example the system code in the CPU makes such requests as

- create a new page

- destroy a page

The MMP has three separate message input "ports." Each port corresponds to a major task structure in the MMP. One is for the SWAPPER, one for a direct I/O capability (not a normal user's facility), and one is a utility task.

To make a request on the MMP, the CPU system code first constructs a request in the form expected by the MMP, then copies the request onto the end of a queue at the appropriate port, and finally request strobes the MMP to let it know there's a message waiting for it. Clearly, the copying operation is done under a PROTECT.

Some requests require no response, and the system code in the CPU is finished once it has done the request strobe. Other requests initiate actions which require explicit responses and some of these call for immediate responses while others, taking longer to process, produce a delayed response.

In the immediate response case, the system code waits for the MMP to reply in a communication area dedicated to this purpose. In the other case, the system code blocks the CPU process. When the MMP has the response ready it will put its message in an area local to the process from which the originating message came, and send a message to the SCHED telling it to wake up the process because there's a message for it. In due course this will re-start the blocked system code, which will then go and read the MMP's message.

5. MMP - SCHED

The MMP sends messages to the SCHED in the same way for the same basic reason the the CHIO does, i.e., to let the SCHED know that some event of interest to a process has occurred. Thus messages from the MMP to the SCHED are normally requests to wake up a specified process. Usually, some small amount of data for use by the process accompanies these messages.

The SCHED sends messages to the MMP to request the swapping into or out of central memory of the working set of a process. It gives requests to the MMP the same way the CPU does -- by forming a message, appending it to the appropriate one of the MMP's message input queues (under a PROTECT), and then request strobing the MMP.

6. CPU - SCHED

The CPU communicates with the SCHED for various reasons. The SCHED implements a real-time interrupt/wakeup facility for the use of user processes. At the request of such processes, the system code in the CPU calls this function in the SCHED to arrange that the process be awakened and notified at a certain real time. These requests are made by the CPU in the same way that the CHIO and MMP send wakeup message to the SCHED. The CPU seizes the appropriate PROTECT, puts a short message into the SCHED's message input buffer, releases the PROTECT, and request strobes the SCHED.

4.0 DISCUSSION

The use of multiple processors permitted the BCC 500 to be a more capable system than the use of a single processor -- even a considerably faster (amd more expensive) one. The varios processors in the system, while similar in hardware, are dedicated to their assigned functions: three dedicated to system management and two to running the user codes. There are several consequences of this decision which should be mentioned. First, there is no need for complex time-sharing and scheduling of the various processors over the many system and user tasks. The tasks can thus be designed to run to completion, ie.e, with little concern for interruption by higher priority tasks. ~~(Thy is this good?)~~

Putting the system tasks into separate processors has the result that the operating system gets modularized in a "real" way. Modularization is always a problem in any operating system design: the actual choice of modules is frequently difficult, and programmers are often led to "cheat," i.e., to cut across module boundaries against theri own definitions of those boundaries. Although this can be remedied by sufficient self-discipline, having separate processors forces the issue, with the consequence of clean, impossible-to-cheat-on modularization. The modularization is readily comprehensible, even to a technician, say, because he can identify a module (and its function) with a known piece of hardware. This has distinctly beneficial ramifications toward maintainability.

In addition to error analysis the use of separate, dedicated processors in general makes checkout and performance monitoring easier. The modules arebasically independent and are driven by the contents of memory tables by which they communicate and interact. Thus they can be readily tested independently. (Checkout is a serious problem normally in complex operating systems because of the many module interactions.) In the checkout of the 500, for example, the entire AMC function was developed and checked out before the CPUs existed.

Because so much fixed-algorithm computational power exists in the several management processors, it is possible to consider -- within those processors -- the use of algorithms for various operating system functions which are too complex (i.e., time-sonsumptive) to utilize even ov very fast CPUs. Another consequence of the use of multiple microprocessors for system management is the ability to utilize more straightforward coding techniques (since, for example, many "tricks" are used in conventional operating systems for efficiency). The coding is easier to do, is mor easily analyzed, and of course is less likely to fail. Finally, it is possible to include on a practical scale the use of redundant computations for operating system management which make error detection more immediate, give the system more survivability (is this the right term?), and greater fault locatability.

)How do we discuss bringing up the 2nd CPU?)

There are, of course, some drawbacks to the BCC 500 architecture, expecially that of the existing prototype. Mostly these relate in one way or another to the purely read-only nature of the microprocessors' control stores. Because each of the management processors executes unique functions as determined by the (fixed) ROMs, there is no way to continue operation of the system when a processor fails. This does not relate to the basic architecture, ie.e, to the use of dedicated, multiple processors, as the use of writeable control stores would easily permit standby processors to be phased in to replace a given management processor which has hailed.

Bhe BCC 500 system designers were encouraged toward the use of multiple processors by their considerations of the system's potential need for data security. In the earlier example of how the processors executing their separate functions communicate and cooperate, we saw how they utilized common, resident memory tables as their communication medium. The access of any one processor into these tables is highly restricted by the microcode whose behavioral properties are more amenable to study than the more unstructured code of users or systems programmers. The CPU (except its microcoded routines) is generally prevented from inspecting these tables directly -- even in its system mode. Thus the separation of functions into processors has clear ramifications on the potential security of the operating system; it is much more difficult for a penetrator to induce a different processing running fixed code to inspect critical information and thereby cause an abridgment of security. This aspect of the architecture makes it appear favorable for applications in which security is a concern; at least, compartmentalization is a classical (non-computing) means for implementing security.

APPENDIX I

1.0 Intended Area of Applications

A goal of BCC was to develop systems which could be used by _computer utilities_, i.e., companies offering appropriately packaged computing power and services by means of telecommunications and remote terminals. The 500 system was intended to serve this purpose particularly where the computing jobs individually require small amounts of computation; that is, the system architecture was _biased_ toward accommodating large numbers of relatively small jobs. Common examples of intended applications were small scientific computations (e.g., small BASIC programs), bank-teller systems, reservations systems, real-estate systems, etc. Many of these systems, while not requiring large amounts of computation, do require a large machine -- large in the sense of file capacity and memory address space. Thus a mini-computer, for example, would be ill-suited to the application, while a large machine might not be justifiable on economic grounds.

[more: separate utilities, wholesaling, guaranteed service, data security]

1.1 User Machine and Operating System

The operating is partitioned into functional areas and is executed on different processors concurrently. These processors are dedicated to their tasks. One type of processor is "dedicated," of course, to the running of general-purpose (i.e., user-specified) code. This processor, called a _central processor_, also executes portions of the operating system, always on behalf of its active user. Those CP characteristics of most interest from the operating system viewpoint are briefly listed below:

- The CPs are designed to be programmed only in higher-level languages. There is no assembler for the CP.

- Virtual memory: The CPs see a virtual memory of 256K words in pages of 2K each.

- Hierarchical, ring-structured protection mechanisms: 3 protection rings designed to accommodate two levels of operating system and one for subsystems and user code. Inter-ring references acquire the protection of the ring being addresses.

- Numerous addressing modes designed to facilitate efficient running of compiler-generated code. Heavy emphasis is made on the use of descriptors for common data structures such as strings, fields, and arrays.

- Comprehensive function call and return mechanism. Copies arguments between environments, creates and stacks environments.

- CP traps directly under user control.

The principal features of the operating system are:

- A monitor, common to all user processes. Contains a bare minimum of calls and is as general as possible. Is intended to be a "kernel" for additional operating system code. Exists in highest protection ring.

- A utility, which can be tailored for individual users. Implements all of the user machine characteristics and executive command language. Runs in middle protection ring.

- Subsystems and user code, running in lowest ring, can call utility or monitor just like they call their own functions, subject to the ring protection.

Finally, the virtual machine seen by user code (the user machine) has the following general appearance:

- A 128K address space

- The usual types of operating system calls.

- Some unusual calls such as:
    - User control over process working set
    - Scheduling decisions like whether to block for I/O.

1.2 Hardware

The system as it exists at the University of Hawaii includes six processors connected to a high-bandwidth multi-port, multi-module 128K central memory; two large drums, two large disk files, with their associated controllers; and centralized control, synchronization, and communication logic. These resources are capable of significant computation. The original system design, however, called for a 256K central memory and allowed for a significant expansion of both drum and disk storage. More importantly, the basic design included a number of remote processors for terminal and communications handling, connected to the central site by a network of communication lines. Three of these remote processors were constructed and brought to Honolulu. They were not included in the present configuration, having been "replaced" in function by a connection through one of the central-site processors to the ARPA network.

All of the processors -- the central-site processors and the remote-site processors -- are implemented around a basic microprocessor designed by BCC for use in the system. The microprocessors used in the central site have 24-bit wide arithmetic units, and each executes at a maximum rate of about twelve million operations per second. The remote-site processors had 16-bit arithmetic units and were slower. Each version uses discrete diode circuit boards for their control memory, which contains 2K x 90-bit words.

The central memory to which the six processors are connected is a four-port, eight-module memory capable of sustaining an average data transfer rate of sixteen megawords per second. Each module contains 8K double-words of one-microsecond core storage and two double-words of active storage together with associated logic. The active storage and the logic associated with it are called the Fast Memory (FM). The FM functions as a temporary repository for data as it passes to and from the core module, and as a gathering point for memory requests for better assignment of available memory bandwidth in the face of contention between the processors and controllers. The FM provides a small amount of look-behind capability.

The drum and disk memory units are interfaced with the central memory by rather simple controllers termed "transfer units." Each drum is equipped for full parallel transfer at the rate of two megawords per second (six megabytes per second). The disk units transfer approximately ten times more slowly, or at about 600 kilobytes per second.

[The Aux Mem is considered to be part of the system "main memory" (as opposed to being I/O units), and the data format ... pages, unique names, etc.]