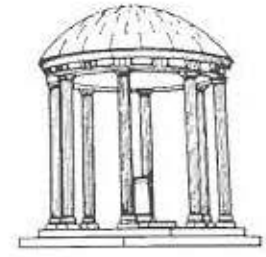
CLOCS Operating System Reference Documents

TR88-023

May 1988

Bill O. Gallmeister



The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

CLOCS Operating System Reference Documents

Bill O. Gallmeister

CB # 3175, Department of Computer Science [1]
University of North Carolina
Chapel Hill, NC 27599-3175

May 6, 1988

[1]This Research was supported by Office of Naval Research Contract N00014-86-K-0680.

Contents

Introduction

CLOCS (Computer with LOw Context Switch time) is an experimental computer system designed at the University of North Carolina at Chapel Hill by Mark Davis and Bill O. Gallmeister. CLOCS is designed to explore the performance issues associated with a machine that can context switch extremely rapidly by virtue of minimal CPU state to save and restore on a context switch. This emphasis strongly influences the design of the operating system, which is built to support finely grained scheduling and dynamic extensibility of the system.

This document collects the papers describing the CLOCS operating system. An overview of the kernel design is first presented, followed by a detailed specification of the entry points to the kernel. Chapter 3 is a brief discussion of scheduling in the CLOCS kernel. The final chapter is an enumeration of the signals used in the operating system.

Chapter 1

An Overview of the Kernel Design

CLOCS (COmputer with LOw Context-Switching time) is a machine being designed at the University of North Carolina at Chapel Hill, by Mark Davis and Bill O. Gallmeister. CLOCS is an experimental system, both hardware and software, created to explore the consequences of a design that permits extremely rapid context switches. The CLOCS Operating System is designed to exploit the unique features of the CLOCS hardware to meet specific performance and qualitative goals: real-time responsiveness, fair multiprogramming, and dynamic reconfigurability. This paper describes the most basic part of the machine's operating system – the CLOCS Kernel.

While the CLOCS kernel is only the lowest layer of the operating system, it provides the necessary building blocks to meet the design goals of the system as a whole. This document emphasizes the overall concepts that relate to these goals, deferring more detailed kernel descriptions to [12]. Section 1 discusses the goals of the system. Section 2 provides an overview of the strategies used to meet these goals. Descriptions of the modules of the CLOCS kernel are given in section 3.

1.1 The Goals of the CLOCS Kernel

1.1.1 Real-Time Response

A major objective of the CLOCS Operating System is to provide real-time response, meaning that processes must be able to respond to events, generated by software or hardware, within a specified (and assumed small) amount of time.

Real-Time Systems are Difficult. Real-time response is hard to achieve in operating systems, because not only must the answer be right, it must be delivered on time. Like most software, typical multiprogramming operating systems run with little regard for external, real-world time. In designing a real-time system, the software designer must pay close attention to the amount of time taken in all sections of code – asymptotic order notation will not suffice! The designer must assure that interrupt response times are bounded, must support guaranteed scheduling and completion by external time, and must carefully analyze the timings of interacting parts of the system to assure that the timing constraints of the system are met. In sum, real-time constraints make programming *harder in general*[20], because they add a whole new dimension – the time dimension – to the problem space being explored.

1.1.2 Fair Multiprogramming

Real-time response has been achieved in other systems, but usually at the expense of fairness – the processes requiring real-time response are treated preferentially to other, non-real-time processes. The second design goal of the CLOCS operating system is that it provide fair multiprogramming for all processes. A scheduling algorithm is called "fair" if all processes are given equal consideration by the scheduler at all times[8]. Fair multiprogramming is difficult to reconcile with real-time capability, since real-time processes may have special requirements – they may need to be scheduled more often, or perhaps allowed to run longer, in order to have any value whatsoever! Reconciling "fair" scheduling with demands for real-time response is discussed in detail in [10].

1.1.3 Dynamic Extensibility

Software is a malleable substance, and quite often software systems are altered "on the fly" as they are being used: functional modules are added to, and subtracted from a running system *as it is running*. This is especially true in real-time programming, where the programming is often associated with some unique data collection device that must be specially driven[17]. Small, frequent changes to software components should not require recompiling and rebooting the operating system. Therefore, the CLOCS operating system must expand and contract dynamically as it runs. This allows new drivers or specially expanded functionality to be added to the system as needed, removed when the machine resource is better spent elsewhere, or changed when it is wrong.

A second reason for dynamic extensibility is the advantage of programming an application on the target machine for the application[18]. Programming on the target machine requires that the machine support a full development environment, but such an environment is only useful when the system is being developed. When a production system is running, a full development environment is just baggage. It must be possible to link in the capabilities of a full-featured operating system on demand, then jettison them when they are not required.

1.1.4 The CLOCS Machine

In a multiprogramming system, processes are frequently context-switched, i.e., the running process is stopped, its state saved and another process started. Machines with large amounts of state in their processors have historically achieved better rates of throughput, but they also context-switch more slowly than machines with less state. In the past, throughput of a single process has been the metric for gauging a machine's performance, but as multiprogramming systems become more common, throughput of multiple, concurrent processes is increasingly important. Context switching speed is an important component of multiprogrammed computer performance.

The CLOCS project is studying the tradeoffs between single- and multi-process throughput involved in the design of a system – both hardware and software – which targets fast context switching as its major performance metric. Since the novel design of the hardware has influenced the kernel design, a short overview of the hardware is in order.

The CLOCS CPU

To switch context, a machine must store all internal registers and replace them with new information. In order to allow fast context switches, the CLOCS machine has only one register, called the state word; storing it and reloading its contents takes exactly two instructions.

Because there are no other registers, the CLOCS operation set is small – there is no need for load or store operations, and the lack of registers also makes for fewer addressing modes. This dramatically simplifies the instruction set: CLOCS supports only 20 different operations!

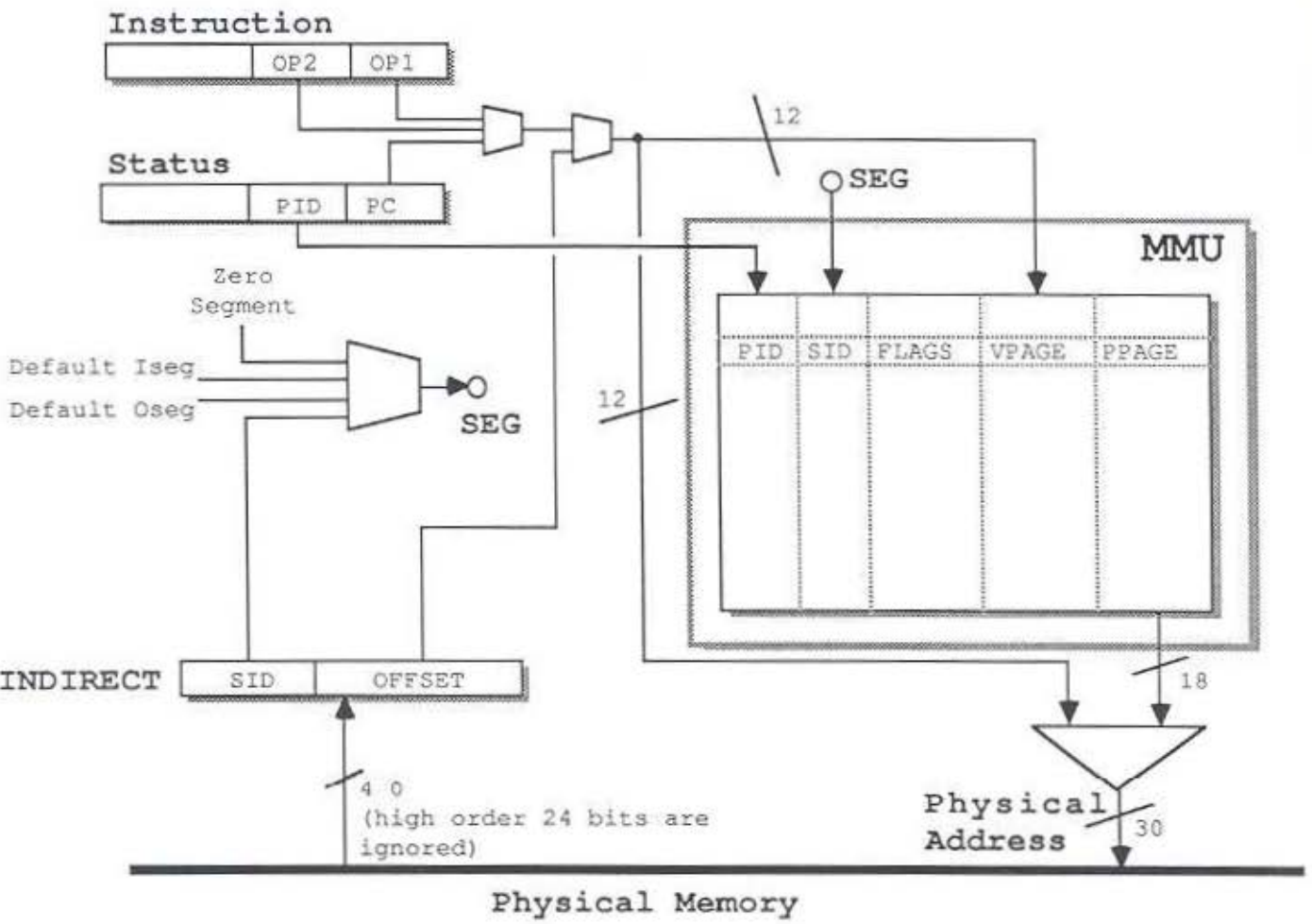
This minimal amount of CPU state impacts the programming model for the machine. The bare minimum information is stored in the state word: a process ID, the program counter, and flags, including the current interrupt mask. A great deal of process state, such as stack and frame pointers, is normally maintained in a machine's registers. In CLOCS, this state is kept in well-known memory locations.

The CLOCS MMU

Real-time systems, and increasingly, general-purpose computing systems must run hundreds, if not thousands, of processes concurrently. Virtual memory has proven to be an important and useful tool for building reliable multiprocess systems, due to the separation and protection it offers. We feel that virtual memory is vital to the reliability of multiprogrammed systems. Therefore, CLOCS supports segmented, paged virtual memory with its MMU. A process ID, stored in the state word, uniquely determines a set of segment and page mappings in the MMU; changing this hardware process ID changes the MMU as a side effect. Although most addressing is assumed to be in one of two default segments (one for instructions and one for data), processes can address data in any segment using extended addressing modes.

The MMU is organized as a single large table, supplying process ID, segment number,

virtual page, physical page, and protection bits in a single tuple. The MMU is an associative memory, and the hardware does not enforce any ordering of the tuples. Since process ID does not determine a fixed number of segments, processes can access an arbitrary number of segments, including segments shared with other processes. The flexible layout of the MMU allows easy memory sharing between processes, but also allows inconsistency. For instance, process ID + segment ID + virtual page number do not functionally determine a unique tuple, making it possible to have two contradictory mappings in the CLOCS MMU! The memory management software must ensure that the MMU remains consistent.



The CLOCS MMU

Event Handling. Events (traps and interrupts) are handled by vectoring; an event vector is a state word that is loaded into the CPU when the associated event occurs. The CLOCS machine provides 1024 separate vectors, half for traps and half for interrupts. This large number of vectored events speeds event handling because the software doesn't need to work as hard to figure out which event occurred. That information is largely implicit in the event vector itself.

The architecture of the CLOCS machine and its MMU are described in a number of papers [6,4,3,5]. Readers interested in detailed architectural descriptions are referred to these papers.

1.2 Achieving the Goals of the CLOCS Kernel

The CLOCS kernel uses a few simple strategies to meet its goals. The general strategies are described below; the next section gives more specific details on the kernel itself. Together, these strategies provide the necessary building blocks for achieving the goals of the whole system.

1.2.1 Obtaining Real-Time Responsiveness

Obtaining real-time responsiveness is the single largest goal of the CLOCS operating system, and its realization requires the most work. Each module of a real-time system must cooperate in order to achieve the performance goals of the system. The modules of the CLOCS kernel work together in the following ways.

Uninterruptible Path Lengths Are Short

If any process requires long uninterruptible periods of time, then real-time performance becomes hard to achieve: rapid response to an event cannot be guaranteed because some process may be just starting a long section of uninterruptible code. The UNIX[1] system, for instance, has a hard time doing real-time processing because it is monolithic, and processes running in the kernel can take many milliseconds to complete. In contrast, the CLOCS operating system consists of short, uninterruptible *paths* through the kernel, connected by sections where interrupts are allowed. At these "checkpoints", rescheduling of the processor can occur, allowing rapid response to events.

Processes Can Run To Completion

CLOCS allows a process to indicate when it must run to completion in order to guarantee that it will finish its real-time work. When a process is allowed to run to completion, it cannot be preempted until it allows itself to be.

More is Stored; Less is Computed

Alan Jay Smith, of Berkeley, has said that any program can be made five times as swift to run, at the expense of five times the storage space. While his numbers may be questioned, his premise may not: programs can be made faster by precomputing and storing results. Where the tradeoffs can be made, the CLOCS Operating System achieves faster execution by using more elaborate data structures. For instance, the data structures used by the scheduling algorithm are optimized to speed the choice of which process to run next.

[1] UNIX is a trademark of AT&T Communications

Small Modules Speed the Kernel

The CLOCS kernel is built from small, effective modules that provide simple abstractions: virtual memory, processes, and interprocess communication. These smaller, more modest modules run faster than megaliths because they do less. Since the kernel can be dynamically extended and contracted, enhanced function can be built on top of the kernel as required by a particular application. Meanwhile, the modest scope of the kernel allows it to run swiftly.

1.2.2 Combining Responsiveness and Fairness

The second important goal of the CLOCS operating system is to combine real-time responsiveness with fair multiprogramming. Scheduling heuristics typically attempt to provide one sort of behavior, either fairness or real-time responsiveness. The CLOCS scheduling algorithm, in contrast, takes both goals into account.

New Scheduling Ideas

Scheduling is often implemented using a priority-based scheme in which a single number denotes a process's "value". The priority can be manipulated according to the process's behavior[8]. Priority-based scheduling provides fair scheduling behavior for non-real-time processes. Unfortunately, the value of a real-time process is not a static quantity, and may vary in a time-dependent, *not* process-behavior-dependent fashion. Thus, priority schedulers have a difficult time supporting real-time tasks. In contrast, real-time systems often practice deadline scheduling, where processes are scheduled in order of shortest deadline first. Variants of the deadline scheduler abound, but all of them schedule processes strictly based on their deadlines. Deadline schedulers do not try to be fair, and in fact *will not* schedule a process without a deadline – i.e. a non-real-time process – unless there are no real-time processes ready to run.

Any scheduler that targets only a single dimension (time, priority, etc.) will fail at scheduling some other class of processes. By providing more information pertaining to the scheduling problem, the scheduler can make more informed choices about which processes must run at any given time. Elaborate scheduling algorithms have been designed to more accurately model process values, and therefore schedule them better, where better is defined by the objectives of the particular scheduling algorithm. In some complicated systems, as many as five numbers have been used to denote the time-varying value of a real-time process[13].

In the CLOCS system, a unified process value model is used, denoting each process's value and its deadline, along with indications of how long the process will need to run, whether there is any value in running the process past its deadline, and whether the process should be allowed to run to completion. These attributes allow more delicate scheduling decisions and are sufficient for proper scheduling of the majority of processes. Dynamic manipulation of the quantities further enhances the system's responsiveness.

1.2.3 Achieving Extensibility

The third goal, achieving extensibility, requires the ability to add and subtract software components on a running system, much as fault-tolerant computer systems allow hardware to be added and removed dynamically. To solve this problem, the interface between the parts of the system must be clean and well-defined, facilitating fast, simple changes that allow dynamic interprocess communication. Breaking the connections and eliminating components must also be easy. Finally, calling a module that is not present must not result in catastrophic failure of the operating system!

Object-Oriented Design Provides Clean Interface

The object-oriented paradigm provides a partial solution to the extensibility problem. In the CLOCS kernel, each object, or "manager", communicates with the other managers and the user processes through a simple interface. Each manager makes specific *entry points* available to the entire system; other processes may only call the manager using those entry points. The manager can also remove the entry points. Calls to non-existent entry points are treated as errors, which can be treated by loading the required module, initializing it, and trying again.

Policy-Mechanism Separation Allows Functional Extension

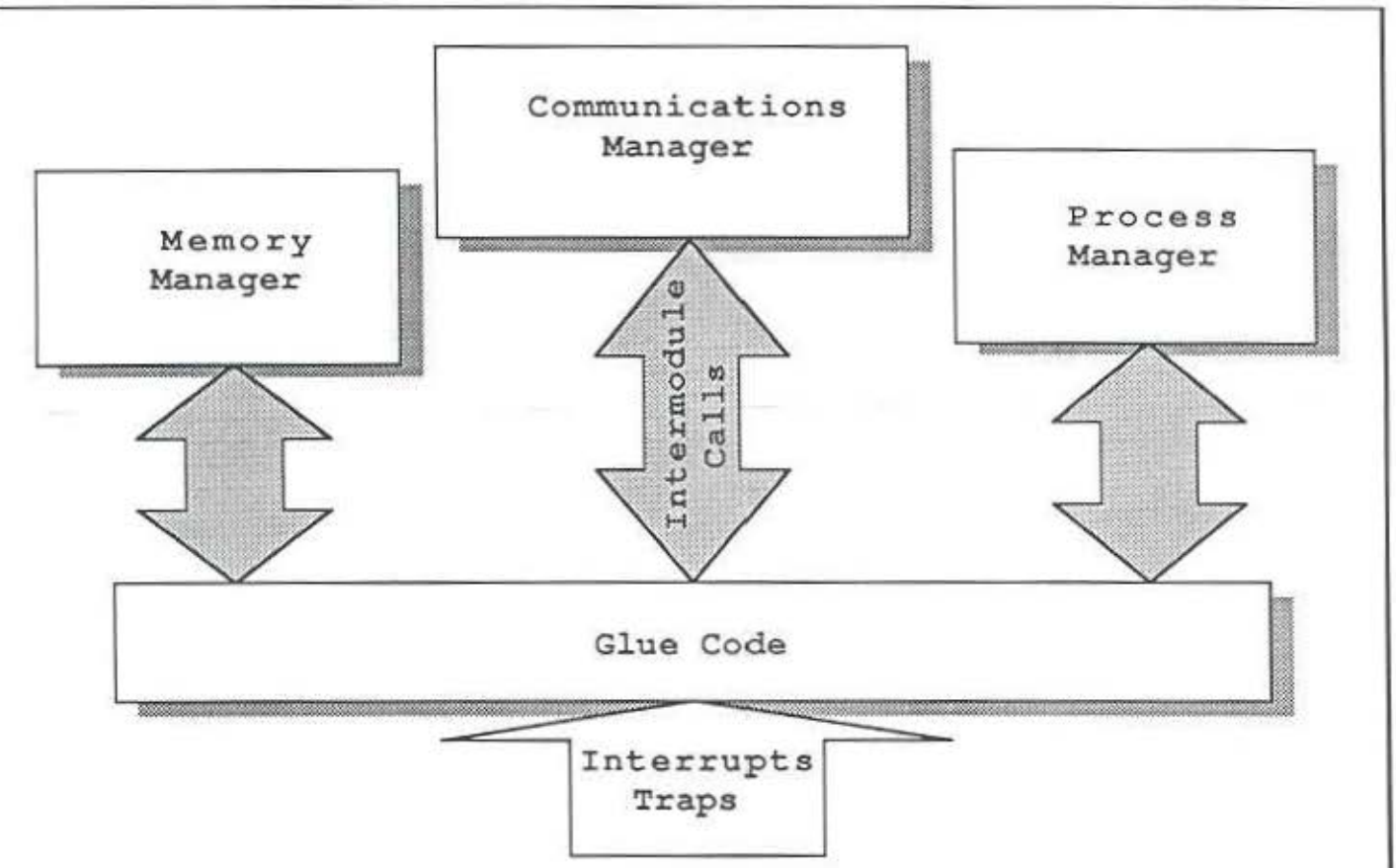
The object model is a necessary, but not sufficient condition for extensibility. If the semantics of the underlying software layers do not allow higher layers to function properly, then extending the kernel becomes impossible. CLOCS supports *policy-mechanism separation*: the lower layers of the kernel cannot implicitly decide policy for upper layers. For instance, the Memory Manager does not make any decisions based on which process is calling it, because it is up to the Process Manager to make process-related decisions.

1.3 Kernel Module Descriptions

The CLOCS system is organized as a set of four modules, each of which implements an abstraction or service. This hierarchical approach to design offers clean, modular interfaces and smaller, easy-to-understand software packages[9]. Four modules make up the kernel of the CLOCS Operating System, each providing basic services on which higher levels will rely. The four modules that form the CLOCS kernel are:

- **The Glue Code:** The lowest layer of the CLOCS kernel is the Glue Code. It handles the details of inter-module communication and exception handling, allowing all other modules in the system to be integrated into a single machine.
- **The Memory Manager:** The Memory Manager handles the CLOCS MMU and provides the abstraction of virtual memory. Virtual memory is necessary for building reliable multiprocess systems because of the protection and separation it offers.
- **The Process Manager:** The Process Manager encapsulates the scheduling algorithm and provides the abstraction of independent processes. The entire CLOCS system is structured as multiple processes, so a process manager is a basic requirement.
- **The Communication Manager:** The Communication Manager provides the abstraction of inter-process communication. Systems such as real-time applications and server applications are often structured as multiple processes communicating in a variety of ways. This paradigm is basic enough to merit support at the lowest levels of the operating system.

The Glue Code provides the most basic level of service, supporting a clean, monitor-like interface between software modules. The other three modules of the kernel communicate using the Glue Code. The Memory and Process Managers are at a slightly higher level than the Glue Code. The Communications Manager is at a still higher level, using the services of the other two managers.



The CLOCS Kernel

1.3.1 How Does This Kernel Meet the System Goals?

Extensibility of the system is supported by the Glue Code, which provides calls to allow modules to make themselves dynamically available to the rest of the system.

The kernel modules run in a request-driven fashion; a call to one of the managers will provoke a short, uninterruptible response. When modules communicate with each other, interrupts may occur, allowing for possible rescheduling. Thus, the uninterruptible paths through the kernel are only as long as the longest path through any particular manager. Since each manager performs simple, small tasks, the paths through them are short, and each call to a manager can be satisfied quickly.

The managers are designed in such a fashion that they store more data than is necessary

in order to avoid time-consuming recomputations. This design style is most evident in the Process Manager, described below. In addition, the scheduler implemented by the Process Manager is designed to meld real-time responsiveness with fair multiprogramming.

1.3.2 A Bottom-Up Description Models Successive Abstractions

The modules of the CLOCS kernel are described from the bottom up, paralleling the successive abstractions provided by each module. Since the complete operating system is not specified, describing the system from the top down is not possible: there is no top!

1.3.3 The Glue Code

The lowest level of the CLOCS Operating System is called the "Glue Code" because the routines and data at this level support the connection of other processes, or modules. Conceptually, this module "glues" the others together. The glue code handles intermodule communication as well as interrupt and trap dispatch. The dynamic extension and contraction of the system is handled from the glue code, and proper access of user applications to the kernel is enforced here as well.

Intermodule Communication

To call an entry point in another process, the caller pushes the process ID and entry point number of the called process on its stack and traps to the Glue Code. The Glue Code checks the calling process's right to call the entry point and, if permitted, makes the call. If the specified entry point does not exist, then an error indicator is returned. Notice that three processes are involved: a caller, the kernel (in the persona of the Glue Code), and the called, or server, process.

The Glue Code supports intermodule communication by enforcing an explicit interface for module access. A module, or process, makes entry points available to other processes by calling the Glue Code and specifying the address of the entry point and the permissions associated with it, i.e. who may call the entry point. The caller associates an entry point number with the entry point address, insulating other processes from the need to know specific addresses within another process. A process may also remove an entry point it has previously made available.

A process containing an entry point will be at some point in its execution when the entry point is called. Entry point calls are handled as if a signal had occurred: the entry is "serviced" by the called process, which then returns from that entry to whatever processing it was doing prior to the call. Meanwhile, the called module is blocked. In addition, while the server process is servicing an entry call, new calls to its entry points are blocked. This is done to prevent simultaneous access to a single process by other processes, possibly resulting in inconsistencies.

Traps and System Calls

Inter-module communication traps are one use for traps, but all other traps are to the glue code, as well. This includes exceptions, such as page faults and divides by zero, and system calls, which are performed as intermodule calls from user processes to the kernel process. For all traps, the Glue Code must save the state of the trapping process before jumping to the appropriate service routine.

Interrupts

Interrupts (external events caused by things like I/O devices or power failures) are also handled by the Glue Code. Although the main bulk of interrupt processing is handled by the kernel proper, the state of the machine prior to the interrupt must be saved, and this is the job of the Glue Code as well.

Humble Access

"Humble Access" is a term for limiting a process' access to privileged operations. Processes can ascend to privileged mode only at specified locations in the code. At these locations, the access rights of the calling process are checked, and its "humble" request for privileged service is granted or denied. Since the CLOCS Glue Code provides the only entry method to other modules, it can and does enforce humble access by checking permissions before permitting entry point calls.

Dynamic Relinking

The abstraction of entry points to other processes allows for easy dynamic relinking of modules, since the relinking is handled through a central location, the Glue Code. As an added advantage, calling a nonexistent entry point is treated as an error and not a catastrophe, so calling modules can be programmed to recover from ill-configured software. This robust, dynamic relinking capability provides the extensibility required by the CLOCS Operating System.

1.3.4 The Memory Manager

Virtual memory is a requirement for building reliable multiprocess systems because of the separation, protection and ease-of-use a virtual memory system offers. The CLOCS MMU provides the raw material for implementing efficient, protected virtual memory; however, it must be carefully managed by software to avoid inconsistencies. The Memory Manager has responsibility for maintaining correctness of the MMU and of physical pages of memory. It keeps track of those segments, physical and virtual pages, and process identifiers (PIDs) which are in use.

Interface to the MMU

As sole access to the MMU, the Memory Manager must also provide efficient, fast access to the hardware. The size of the MMU, 2^{18} words, is too large for the Memory Manager to search linearly; so the Memory Manager constructs software structures atop the MMU to allow swifter access to specific entries.

Segment Allocation

Two different calls allow a process to allocate and deallocate segments. When allocating, the memory manager determines a free segment and assigns it to the calling process, but no mention of that segment is made in the MMU, because there is no memory yet associated with it. When pages of memory are actually allocated within the segment, then the MMU is modified. When a process frees a segment, the segment is removed from the MMU for the process, and if no other process is using the segment, it is returned to the free list. It is an error for a process to try to free its primary instruction or data segments, which are the ones it requires to run in.

Page Allocation

Processes allocate and free virtual pages within an already-allocated segment. The calls specify the starting page and a number of pages to allocate or free. Errors are returned if the process tries to allocate a virtual page that it has already allocated, or if it tries to free pages that are already free.

Page Sharing

An additional call in the memory manager maps pages of memory from one process into another process. This call does not enforce any sort of protection between processes, but the call can only be made by the kernel itself. The mechanism for sharing memory is required by the Communication Manager, which enforces the policy of shared memory by calling the Memory Manager in the "right way".

1.3.5 The Process Manager

Processes are a basic unit of computation. Increasingly, applications ranging from database systems to resource servers to entire operating systems are being constructed as multiple processes which communicate to achieve the goals of the system. This paradigm offers conceptual simplicity as well as increased reliability and fault tolerance. Processes require support at the lowest levels of the kernel, since the higher levels of the CLOCS system will themselves be structured as multiple processes. The CLOCS Process Manager provides the abstraction of processes and encapsulates the process scheduling algorithm. It also manages process creation, destruction and state changes. Although context switches are done by the Glue Code, actual processor allocation and dispatch is performed from within the Process Manager.

Definition of Process

CLOCS defines a process as simply "a schedulable entity"[8]. A process is just a thing that can be scheduled for execution. A process is named by its Process Control Block (PCB), a data structure which contains control information about the process: its last recorded state, what memory it has allocated, its priority and urgency, and so forth.

The operating systems literature mentions two sorts of processes: heavyweight and lightweight processes. The CLOCS MMU supports one kind just as easily as the other, and the Process Manager makes no distinction between the two.

Heavyweight Processes. Heavyweight processes are processes which execute in their own protected address spaces. They are slower to context-switch because they require a full swap of machine state, including, possibly, some MMU contents and some physical memory.

Lightweight Processes. In contrast, lightweight processes have less baggage of their own. Multiple lightweight processes inhabit the same shared address space. Lightweight processes can switch between one another very rapidly because the MMU and memory state required for each is identical and need not be changed.

Difference Between Heavyweight and Lightweight. In the CLOCS machine, there is little difference between heavyweight and lightweight processes. Because the CLOCS MMU contains enough state to cover all of physical memory, memory-resident heavyweight processes will be as easy to switch to as lightweight processes. However, if the memory required for a heavyweight process is not present, then the disk must be accessed, and more time will be required for switching context. Since the CLOCS kernel at this stage does not specify any disk, swapping, or other higher-level concerns, this distinction will not be discussed any further. It is sufficient to note that processes can exist in shared or private address spaces, or even in some combination of shared *and* private space.

Creating Processes. Processes can create other processes. The creating process gives two segment numbers, which become the default instruction and operand segments of the new process. Scheduling parameters and starting address are also specified. The process, when created, is ready to run and is scheduled as soon as feasible.

Destroying Processes. A process can destroy itself, and the kernel can destroy any process. When a process is destroyed, its memory is freed and returned to the memory pool if no other processes are using it, and its PCB is made available to new processes. The process is removed from scheduling consideration.

Changing Process States. Between the time it is created and the time it is destroyed, a process will repeatedly switch between the *running*, *ready*, and *blocked* states. At any given time, only one process is *running*. Either it is using the processor or the kernel is running on its behalf. Processes that could be running, but have not been allocated the processor yet, are called *ready*. Processes that cannot be run because they are waiting for something are called *blocked* processes.

Scheduling

Changing process states, and the decision of which ready process becomes the running process, is called scheduling. In order to achieve both real-time performance and fair multiprogramming, the CLOCS kernel supports an elaborate scheduling system.

"Just In Time" Scheduling. The scheduling algorithm exemplifies a concept that has become popular in manufacturing and inventory control technology called "Just In Time" scheduling. In this method, processes that have to complete by a certain time are scheduled to run at the very last minute. In the warehouse, this leads to reduced inventories and a more efficient operation. In the CLOCS Operating System, by fitting non-real-time execution into the cracks not occupied by real-time tasks, "Just In Time" scheduling provides better response times to non-real-time processes at little or no cost to the real-time processes.

Priority, Urgency, and Quantum. In most multiprogramming operating systems, scheduling is based on priority. Processes have a single attribute, their priority, that determines their importance relative to all other processes. The most important processes always go first. While priority-based scheduling is conceptually simple and easy to implement, priority alone cannot adequately reflect the nature of the scheduling problem. A priority does not state explicitly *when* a process should run; that decision depends on the priority of all the other processes in the system. Thus, it is tricky and unreliable to perform time-based scheduling using only priority.

For instance, a process may not be very important, but may need to run very soon lest it lose all value. Should that process's priority suddenly be raised to enforce its rapid running? If so, how high? And how will it be lowered again? How low? And what if some

other process, which doesn't need to run at any particular time, can be run before the other process absolutely *has* to be run? These problems can only be expressed clumsily (if at all!) using a single priority number. Because there is no way to state the programmer's desire that a process run within a certain deadline, systems are created with process priorities balanced together like a house of cards to provide proper responsiveness. The smallest change in the system or in the environment can bring the house of cards tumbling down[13].

Another sort of scheduling algorithm is deadline scheduling in which processes have deadlines by which they must complete. The process with the closest deadline runs soonest. Deadline scheduling has two significant problems. First, it fails to schedule non-real-time processes (that is, any process with no real-time constraints on its scheduling) since they have no deadline other than "as soon as possible". In many systems, real-time processes, such as data acquisition and physical control tasks, coexist in a machine with non-real-time processes, such as user queries into the database being produced by the real-time application.

More importantly, deadline scheduling fails badly when the processing load exceeds the processor capability since it continues to schedule and run processes that cannot possibly meet their deadlines, either because the deadline is too close or already past[13]. By wasting processor time on processes that will have no value, the deadline scheduler allows *more* processes to become too late; these are scheduled in turn, causing *still more* processes to become late!

The CLOCS Operating System uses three numbers to schedule its processes. A *priority* reflects the process's importance in the scheme of things. *Urgency* is the time, measured in clock ticks, by which the process absolutely must run. *Quantum* is the estimated time, again in clock ticks, the process will take for the run. Only real-time processes have Urgency, because real-time constraints on their operation are made. All other processes are called non-real-time processes. The quantum is used as a time slice in the case of non-real-time processes; for real-time processes, the quantum is taken literally and is used to determine exactly when the process must run.

Blocking. When the running process executes a kernel call that requires it to wait for some event, such as an interrupt or receipt of a message from another process, it is said to block. Blocking is the kernel-level mechanism used to implement all process waiting in the CLOCS Operating System. When the running process blocks, it is removed to the nonready state and a new running process is chosen. The Process Manager supports calls to block processes in a multitude of ways, but these entry points are not callable by user processes. User processes call other kernel modules, which block the user processes in constrained and well-known fashions.

What Do Processes Block On? Blocked processes are waiting for *something*, but how is the occurrence of that something flagged? How is the *something* identified? In the jargon of the operating systems community, processes block on *cookies*. A particular cookie corresponds to some event being awaited by one or more process. The cookie can contain any value, but a particular value specifies a particular cookie. When processes block, they block waiting for one or more of these cookies. Other processes can signal the occurrence of a particular cookie, and all processes waiting for that cookie are notified. Any of those

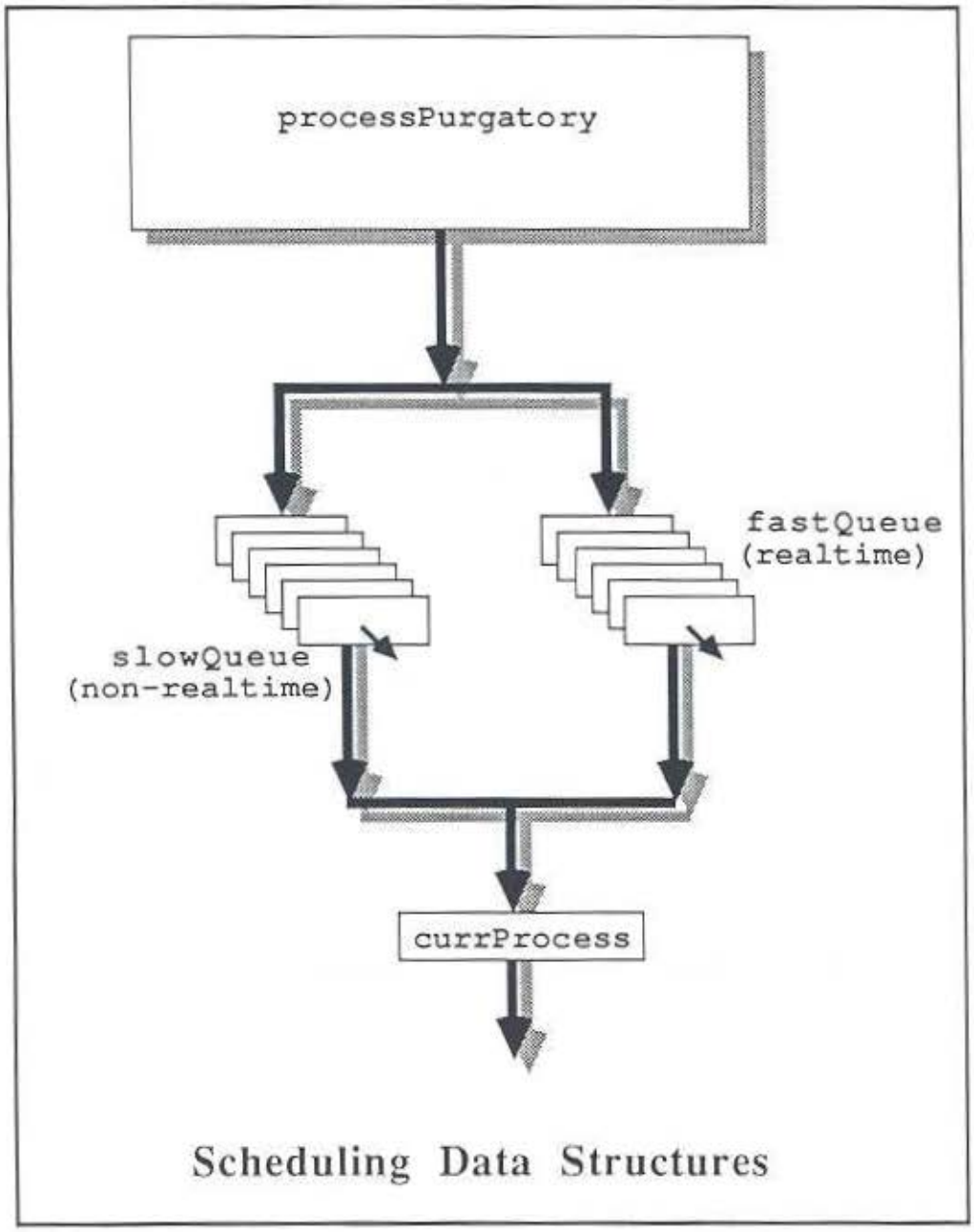
processes which do not need to wait for any more cookies become ready and can contend for the processor. If such a process is a real-time process, its urgency is measured from the time it becomes ready.

Combinations of Cookies. Processes can wait on more than one cookie, and they can wait in different ways. A process can wait on the Boolean AND of a number of cookies; in other words, all those cookies must be unblocked before the process can proceed. In addition, processes can block on the Boolean OR of multiple cookies.

Scheduling Data Structures. The scheduling data structures are designed to speed scheduling decisions. Much of the work of scheduling is done when a process is placed in the data structures, allowing the process manager to quickly decide which process should become runnable next.

The kernel must be able to determine rapidly which processes are waiting on a given event because any application consisting of multiple applications is bound to be doing a great deal of process synchronization, and events are the mechanism used for implementing process synchronization. Therefore, the determination must be proportional to the number of processes waiting on that event, rather than proportional to all blocked processes. In addition, the data structure must be multilinked, because processes can wait on more than one cookie at a time. A process control block may be accessed based on any of the cookies it is waiting on.

When a process is moved into the ready state, it is stored in the run queue. Unlike standard run queues, the CLOCS run queue is structured as two priority queues: one queue for real-time processes and one queue for non-real-time processes. The non-real-time process queue is ordered only by priority. The real-time process queue is sorted in reverse order of $(urgency - quantum)$, so that the process which must run soonest is at the head of the queue. In addition, the time by which each process must run is stored in a differential fashion while the processes are on the run queue. Each process has a *threshold* which is interpreted relative to the threshold of the process before it on the queue. This avoids the need to update the whole run queue on each timer interrupt. Within groups of processes that must run by a certain time, the processes are ordered by priority, most important first.



Scheduling Data Structures

Scheduling Algorithms. The heart of the scheduling algorithm is a decision procedure that determines which process to run next; the scheduler may also reorder the scheduling data structures. The scheduler runs whenever a timer interrupt occurs (signaling quantum expiration), or when the running process voluntarily gives up the processor.

First, the real-time queue is examined. The threshold of the first process on the list is

decremented by the last quantum used, and if the threshold goes to zero, then the process becomes the running process. If the threshold of that process has not gone to zero, then the highest priority process is chosen from the non-real-time queue and it becomes the running process.

To determine which quantum is used: if the current process is a real-time process, then its quantum is used without change. If the current process was taken from the non-real-time queue, then the time-slice given it is either its quantum, or the threshold of the process on the head of the real-time queue – whichever is smaller. This guarantees that the process on the head of the real-time queue will be scheduled when its threshold goes to zero.

```
Decrement the threshold of the most urgent process.
head(fastQueue).threshold := head(fastQueue).threshold - lastQuantum;

Determine whether the most urgent process must be run yet.
if (head(fastQueue).threshold == ZERO_THRESHOLD)

    The set of all real-time processes with this urgency must be run now.
    currProcess := dequeue(fastQueue);
    lastQuantum := currProcess.quantum;
    run();
else

    There is still time until a real-time process must be run,
    so run a non-real-time process.
    currProcess := dequeue(slowQueue);
    lastQuantum := min(currProcess.quantum,
                       head(fastQueue).threshold);
    run();
end.
```

Scheduling Decision Algorithm

Real-time processes that finish their runs before their quantum expires can relinquish the processor voluntarily. When they do this, they are re-inserted to the real-time run queue for another run when their urgency indicates it. This provides for periodic processes.

When preemptive rescheduling (a timer interrupt) occurs, the current process must be re-inserted in the run queue. If the process is a non-real-time process, then its priority is decreased and quantum is increased, as in a multilevel feedback queue[8], and the process is reinserted to the non-real-time queue. If the process is a real-time process, then it did not finish its run before its quantum expired, and this is an error condition. If the process should still be run, then it is run for another quantum. Otherwise it is destroyed and its parent is notified. Whether to run a process once its deadline has passed is determined by a switch settable by the process itself.

Run-To-Completion. In order to guarantee timely execution of some critical function, a process may indicate that it is to *run to completion*, or allowed to run without possibility of preemption. If the current process is to be run to completion, then all interrupts are turned off, including the timer. When the process voluntarily relinquishes the processor, then the scheduler determines how much time has passed and reschedules accordingly. If a process flagged for run-to-completion generates an exception, then the kernel regains control. If the process has an error in it resulting in an infinite loop, then the machine will hang. Run-to-completion mode is not to be used lightly!

1.3.6 The Communications Manager

Communication is a crucial component of multiprocess systems. It makes no sense to structure an application as multiple processes if those processes have no way to interact. Therefore, communication must be supported at a basic level in the CLOCS system. The Communications Manager supports the abstraction of interprocess communication, handling the low-level details of mapping pages from one process to another, blocking processes and awakening them appropriately, and copying data to and from process's address spaces.

Three basic communications models are used in nearly all systems: signals, mailboxes, and shared memory. The CLOCS Communications Manager supports all three.

Signals

Signals are the cheapest communication method to implement and use because a signal's occurrence carries little information with it. However, more information can be sent than with UNIX signals.

Delivering Signals. A process signals another process by specifying which signal should be sent and to which process. The process can optionally provide a one word argument which will be passed to the target process's signal handler; this allows signals to be used for passing short messages. It has been shown that small messages comprise the bulk of most interprocess communication traffic[2].

Handling Signals. Processes respond to each signal by invoking *handler* routines. Default handlers exist; their actions range from doing nothing to immediate destruction of the signaled process, depending on the signal. When a process-specified signal handler is in place and the associated signal occurs, the process immediately jumps to the handling routine. Handler routines remain in place until explicitly removed. Most signals can also be blocked without invoking a handler at all. The default actions for the signals, and the signal names, are provided in a companion document[11].

Masking Signals. While the target process is receiving a signal, new occurrences of that signal are ignored, with the exception of the first occurrence of such a signal.

Mailboxes

Mailboxes are the second utility for interprocess communication. Messages sent to mailboxes are of static size, and they must be explicitly retrieved, although multiple processes can share a single mailbox, and a number of messages can be queued up in the mailbox. By specifying different mailbox parameters, various useful communications paradigms can be realized.

Sharing Mailboxes. The discipline for making mailboxes available to other processes is tricky, so, for the sake of simplicity and familiarity and because it works, CLOCS uses the same mechanism UNIX uses for connecting sockets[15,16].

Connecting to Mailboxes. Following the UNIX paradigm, a server process first creates the mailbox, and then places it in a specific systemwide location where other processes can find it. Finally, it waits (blocks) for other processes to connect to the mailbox, at which time there is a circuit and the two processes can communicate. The creating process can also wait for more processes to connect to the mailbox while still allowing communication with and between the already-connected processes.

Implementing Mailboxes. Mailboxes exist in kernel space and the messages stored in them are protected by the kernel. When a process creates a mailbox, it specifies all the attributes of the mailbox: message and queue sizes, and two important behavioral parameters:

- **Stickiness** is a switch determining whether messages retrieved from a mailbox are removed from the mailbox automatically or not. If the mailbox is "sticky" then messages must be explicitly removed from the mailbox; otherwise they are automatically removed as they are received.
- **Behavior-on-Queue-Full** is another switch which determines how the mailbox responds if a process sends a message to it while its message queue is full. If the sending is allowed, then the oldest message is deleted; otherwise, the send operation fails.

Once two processes are connected through a mailbox, they can send and receive messages. A process can block until a message is sent to it, or it can simply check whether a message is in the mailbox without blocking.

Queue and Message Sizes. A mailbox can accept a number of messages, defined at mailbox creation time as the queue size. The mailbox behavior when the queue fills is determined by the behavior-on-queue-full attribute of the mailbox.

Messages to a particular mailbox are all of the uniform size specified when the mailbox is created. The format of the message is not dictated by the kernel.

Different Paradigms for Mailbox-Based Communication. Mailboxes can be made with widely varying attributes: message size, queue size, behavior-on-queue-full, and mailbox stickiness. By varying these parameters, different communications models are supported: these paradigms have been reported to be the communications methods most used in real-time applications[19,14].

Synchronous Communication Without Data Loss. Synchronous communication without data loss is implemented by setting queue size equal to one, and by disallowing sends

to the mailbox when the queue is full. Processes must therefore retrieve any message sent before a new one can be sent. If a synchronous send-reply discipline is required, then two mailboxes can be used: one for the sends, and the other for replies.

Asynchronous Communication With Data Loss. Asynchronous communication with data loss is accomplished by setting the queue size to one and allowing sends to full mailboxes. Thus, if a message is not retrieved fast enough, it is overwritten by the next message.

Asynchronous Communication Without Data Loss. When behavior-on-queue-full disallows sends to full mailboxes, but the queue size is greater than one, the mailbox supports asynchronous communication without data loss. This lets a certain backlog of messages accumulate in the mailbox, beyond which the sends to the mailbox fail.

Asynchronous Communication, Losing Aged Data. Asynchronous communication with loss of aged data is supported by making the queue size greater than one and allowing sends to full mailboxes. The oldest data will then be lost when the backlog (queue size) is exceeded.

Sharing Memory

The third communication paradigm is shared memory. Shared memory provides the highest bandwidth of data transfer, since data is written instantly to the address space of the sharing processes. Memory can be shared among an arbitrarily large number of processes.

Calls to Support Shared Memory. A process shares its memory with other processes by specifying pages of memory that are available to other processes, subject to access permissions. The segment, starting page, and number of pages to share are given in the call, and the process is blocked until another process requests to share the memory. By specifying the ID of the sharing process, the correct segment number, starting page, number of pages and access mode (read-only or read-write), one process requests shared memory from another process. The pages of memory can be mapped into the requesting process's address space at any location that is not already occupied by pages of memory. The requesting process indicates the access mode it wants for the pages: read-only or read-write. The request is granted or denied based on the permissions stated by the sharing process.

Synchronizing Access to Shared Memory. Access to shared memory must be synchronized using some scheme, such as semaphores or monitors. CLOCS mailboxes can be used to implement semaphores. In addition, the blocking behavior of calls through the Glue Code makes implementation of monitors straightforward, using a separate process for each monitor.

Persistence of Shared Memory. Shared memory is persistent for the life of all of the sharing processes – if the original process frees the pages of shared memory, the shared memory still remains, until the last process is done with it.

1.4 Current and Future Work

This section briefly summarizes the current work being done on the CLOCS kernel and machine, and speculates on future work that may be undertaken.

1.4.1 Kernel Implementation

The kernel as specified is being implemented by a team of students in a software engineering class. The kernel is being built to run on Sun Microsystems workstations under "test rigs", which will allow the function of the kernel to be tested before a simulator for the CLOCS machine is built. When the kernel and simulator are fully constructed, context-switching and other benchmark programs will be run to measure the performance of the entire system against commercially-available machines. If simulation studies indicate merit, then a prototype CLOCS system will be built and used for further experimentation. The kernel will be extended with additional functional modules necessary for running actual applications and the hypotheses of the group will be tested out under real circumstances.

Chapter 2

Kernel Modules Specification

2.1 Overview

The CLOCS project is investigating the tradeoffs incurred in designing an architecture whose major objective is achieving extremely low context switch times. We have designed an architecture, CLOCS (Computer with LOw Context Switch time), which can theoretically switch contexts at a rate orders of magnitude greater than a Sun workstation or VAX minicomputer.

The CLOCS architecture has made tradeoffs in order to achieve such low context switch times. In particular, all operations are memory-to-memory; there is but one register, and there is no specialized computational capability that would require loading/unloading of state information. The CLOCS machine will not provide optimal performance for single-threaded, computationally intensive applications. It is more suited towards applications where events provoke small, fast responses.

The CLOCS architecture makes it possible to drastically reduce the overhead necessary to run multitasking applications. Many of the tasks usually associated with context switching – saving and restoring processor state, saving and restoring MMU state – have been distilled out of the architecture.

2.1.1 Real-Time and Server Applications

As part of this research, we are looking at applications which will benefit from such a machine.

Real-time applications are often constructed as a large number of communicating processes. If a real-time system of this nature is run on a uniprocessor machine, then context switching behavior becomes of critical importance.

Real-time applications, though, are only a special case of a more general class of problems which the CLOCS architecture can benefit. This is the class of systems which:

- are structured as a large number of active processes
- require effective emulation of a multiprocessor

The value of the "large number" of active processes is a fuzzy one; more relevant is the number of processes requiring the processor per unit time. The larger the number of processes requiring the processor in an interval of time, the higher the frequency of context switching will be. The amount of time occupied by context switching rises; beyond some threshold, the processor is spending most of its time simply moving from one process to another.

Examples of other applications that might benefit from the CLOCS architecture are:

- real-time systems
- network disk servers
- communications servers

2.1.2 Operating System Required

The CLOCS architecture is unique in the universe of computer architectures. The TMS9900 is the closest thing to it that we have found.

An operating system provides the abstaction of a virtual machine to the programmer. As such, modern operating systems bring out and make available the features of an architecture. Since no modern architecture is oriented towards rapid context switching on a uniprocessor, we find no existing operating system that will effectively exploit the CLOCS architecture.

We need an operating system which provides rapid context-switching capability, as well as providing the programmability that current operating systems afford.

2.1.3 A Complete Programming System

A programming system is composed of more than a machine and an operating system. Language compilers, debuggers, link editors and a host of programming utilities are all required as well.

The CLOCS project has a cross-compiler for the C language, and work is proceeding on an assembler/link editor suite. However, these tools are secondary, as the CLOCS machine is only a paper architecture at present. When it is built, as a simulator or as metal, a program development environment will be critical. However, this document addresses only the requirements for the operating system.

2.2 The CLOCS Operating System

To achieve minimal context switch times, the CLOCS architecture has removed all possible state from the processor.

2.2.1 Mechanisms for Achieving Rapid Context Switch Rates

The CLOCS operating system will provide rapid context switch rates in the same way: by removing all possible state from the calculations made by the operating system. Alan Jay Smith (?), of Berkeley, has said that any program can be made to run five times as fast, with the side effect of increasing the size of the program by a factor of five. This hyperbolic claim simply means that algorithms can be made to run faster by storing previous results, and in general not computing anything that's been computed before.

This discipline will bear fruit in the CLOCS operating system. Switching context will be accomplished by just loading up a new process ID. State pertaining to processes will be stared for the lifetime of the process in a readily accessible place, with no special movement of data required to make another process active.

2.2.2 Policies for Achieving Rapid Context Switch Rates

Simply by providing a mechanism to perform context switched rapidly, we have not guaranteed that the operating system will switch context rapidly. Also required are policies to support the attainment of rapid context switch rates.

Specifically, path lengths through the kernel, and preemptability of the kernel must be addressed.

Preempting the Kernel

UNIX[1] is an extremely popular operating system among the scientific community. A number of groups have attempted to provide UNIX with real-time capabilities to further cater to the needs of data acquisition and process control applications (VRTX, RTU, POSIX Real-time). The major hurdle encountered by these groups is the monolithic nature of the UNIX kernel. This nature of operating systems is not specific to UNIX, and it makes rapid response to events very hard.

The essential problem is that, once in the kernel for any reason, a path through the kernel must be traced without interruption, or else the integrity of the operating system can be compromised. These path lengths can easily require many milliseconds to traverse. During those times, the kernel may not be pre-empted by a process, no matter what its priority.

[1] UNIX is a trademark of AT&T Communications.

The solution to this problem, of course, is to make paths through the kernel shorter, or alternatively, to segment the paths into component atomic operations, with rescheduling checkpoints along the way. State-changing operations must be atomic; an operating system must perform these actions swiftly to achieve real-time responsiveness.

The CLOCS operating system kernel will perform small, rapid changes to the state of the machine. In between these indivisible operations, rescheduling of the processor may occur. The kernel itself will always be ready to run, and will in fact be run when the urgency of real-time tasks passes.

Specific Policies for CLOCS

To obtain rapid atomic operations, we first separate the functionality of the kernel into modules. Operations within the modules are atomic; in passing from one module to another, rescheduling may occur.

This policy, as a side effect, also permits the expansion of the operating system at a later date.

The modules of the CLOCS kernel each implement a specific abstraction which is essential to the operation of the machine. Three modules are specified to comprise the innermost kernel of the CLOCS operating system:

- Memory Management
- Process Management
- Communications Management

In addition, a small amount of glue is specified to hold the pieces of the operating system together.

2.3 Memory Management

(Abstraction: Virtual Memory)

The Memory manager provides the interface to the CLOCS MMU. Given the physical memory of the machine, it provides the abstraction of virtual memory to higher layers.

Routines are provided to allocate and free segments and pages on a per-process basis; an assitional routine allows changes to the MMU page control bits to support permissions and to allow processes to influence the paging algorithm.

No checking of process access rights is done at this layer – it is strictly mechanism for playing with the MMU. In fact, the memory manager does not know what a process *is* – it simply associates memory with process IDs.

1. **allocatePages:**
   - *PARAMETERS (processId, segmentNumber, starting_page, number_of_pages)*
   - *RETURNS success_or_failure;*
   - *EXECUTION:* May be executed by any process: the PID of the issuing process becomes the processId parameter.
   - Allocates the given number of pages from the free page pool. Updates the MMU for the process identified, so that virtual pages, located in the given segment and starting with the indicated starting page are mapped through to the allocated physical pages.
   - *ERRORS:*
     - *FAIL_BADSEG:* The process doesn't have access to that segment.
     - *FAIL_PAGEINUSE:* One or more of the virtual pages specified are already mapped through to physical pages.
     - *FAIL_NOMEMORY:* Not enough physical memory to satisfy the request.
2. **freePages:**
   - *PARAMETERS (processId, segmentNumber, starting_page, number_of_pages)*
   - *RETURNS success_or_failure;*
   - *EXECUTION:* May be executed by any process: the PID of the issuing process becomes the processId parameter.
   - Frees the given number of pages from use by the process. Updates the MMU, invalidating the appropriate virtual pages in the given segment. If no other processes are using the pages of physical memory, then they are freed back to the memory pool. Freed pages are cleared.
   - *ERRORS:*
     - *FAIL_BADSEG:* The process doesn't have access to that segment.
     - *FAIL_NOPAGES:* One or more of the virtual pages specified are already free.
3. **allocateSegment:**

- *PARAMETERS (processId)*
- *RETURNS segmentNumber;*
- *EXECUTION:* May be executed by any process: the PID of the issuing process becomes the processId parameter.
- Allocates a segment that is currently unused and assigns it to the specified process.
- Used in creating processes, among other things.
- *ERRORS:*
  - *FAIL_NOMEMORY:* No free segment exists.

4. **freeSegment:**
   - *PARAMETERS (processId, segmentNumber)*
   - *RETURNS success_or_failure;*
   - *EXECUTION:* May be executed by any process: the PID of the issuing process becomes the processId parameter.
   - Frees up the specified segment – the process can no longer use it. A side effect is the freeing of all pages currently in the segment.
   - *ERRORS:*
     - *FAIL_BADSEG:* The process does not have access to that segment.
     - *FAIL_PRIMARYSEG:* The process is trying to free one of its primary segments.

5. **freeAll:**
   - *PARAMETERS (processId)*
   - *RETURNS success_or_failure;*
   - *EXECUTION:* May be issued only by the kernel.
   - Frees all segments and pages associated with the process identified.
   - *ERRORS:*
     - *FAIL_BADPID:* No such process.

6. **map:**
   - *PARAMETERS (fromProcessId, fromStartPage, toProcessId, toStartPage, number_of_pages, mode)*
   - *RETURNS success_or_failure;*
   - *EXECUTION:* May be issued only by the kernel.
   - Takes the number of physical pages, located at startingPage in the specified segment of the process named fromProcessId, and maps them into the address space of the process named toProcessId, starting at toStartPage. The pages are mapped in with the given access mode.

7. **getPageStatus:**
   - *PARAMETERS (processId, segmentNumber, pageNumber)*
   - *RETURNS pageStats;*

- *EXECUTION:* May be executed by any process: the PID of the issuing process becomes the processId parameter.
- Returns the permission and page-control bits associated with this virtual page of the specified process.
- *ERRORS:*
  - FAIL_BADPID: No such process.

8. **setPageStatus:**

- *PARAMETERS (processId, segmentNumber, pageNumber, pageStats)*
- *RETURNS success_or_failure;*
- *EXECUTION:* May be executed by any process: the PID of the issuing process becomes the processId parameter.
- Sets the page control bits for the specified virtual page of the process to the contents of pageStats.
- *ERRORS:*
  - FAIL_BADPID: No such process.
  - FAIL_BADSTATS: Invalid stats structure.
  - FAIL_BADPAGE: The specified process does not have access to the specified page.
  - FAIL_BADSEGMENT: The specified process does not have access to the specified segment.

2.4 Process Management

(Abstraction: Processes as Schedulable Entities)

The process manager manipulates virtual pages, associated with process IDs, and provides the abstraction of schedulable processes. The process manager has responsibility for the scheduling of the processor, as well as for maintaining process permissions.

In this module, we create the abstraction of a process, and we talk about processes doing things to other processes. However, notions of *communicating* with other processes are avoided. That is the responsibility of the Communications Manager. E.G., we have a blocking mechanism here, but not an event-signalling mechanism.

In this module, the notions of permissions and UIDs (user IDs) are introduced. User IDs correspond simply to numbers attached to each process. Permissions are granted or denied based on strict matching of UIDs. Two processes with identical UIDs can do things to each other. Processes with non-identical UIDs cannot do things to each other.

As in UNIX, process hierarchies exist. A process that creates other processes is the parent of those processes. Parents can send signals, etc., to descendant processes even if those processes have switched effective user IDs.

If a parent process is destroyed, the children can continue. They are signalled (see the communications manager specification), but that signal can be ignored.

1. **createProcess:**
   - *PARAMETERS (iSegmentNumber, oSegmentNumber, entryPoint, argument, priority, urgency)*
   - *RETURNS process_id*
   - *EXECUTION:* May be executed by any process: the created process inherits the user ID of the creating process.
   - Creates a new process whose primary Iseg and Oseg are the specified ones. Returns the ID of the new process.
   - entryPoint may be set to the pseudovalue ENTRY_FORK, in which case the new process is an identical copy of the calling process; the calling process is returned the identity of the created process, while the created process is returned SUCCESS.
   - This routine suffices to create processes distinct from the creating process (a' la' fork/exec), to create identical but distinct processes (a' la' fork), and to create identical, nondistinct processes (lightweight processes, for which there is no UNIX analogue).
   - *ERRORS:*
     - *FAIL_BADSEG:* Those segments aren't available.
     - *FAIL_BADFORK:* entryPoint was ENTRY_FORK, but the segments specified are not the primary segments of the calling process.

2. **destroyProcess:**
   - *PARAMETERS (processId)*
   - *RETURNS success_or_failure*
   - *EXECUTION:* May be executed by any process: The process id of the calling process becomes the processId parameter to the call.
   - Removes the process from scheduling consideration. Frees all the memory in use by the process. Makes its segments available. Updates the MMU, invalidating the appropriate virtual entries. If no other processes are using the pages/segments of memory, then they are freed back to the memory pool.
   - *ERRORS:*
     - *FAIL_BADPID:* No such process.

3. **setuid:**
   - *PARAMETERS (processId, Uid)*
   - *RETURNS success_or_failure*
   - *EXECUTION:* Can only be executed by the kernel.
   - Sets the effective user ID of the process. Afterwards, the process will have all access rights of that user.
   - *ERRORS:*
     - None as yet.

4. **switchUid:**
   - *PARAMETERS (processId)*
   - *RETURNS success_or_failure*
   - *EXECUTION:* May be executed by any process: the process ID becomes the processId parameter.
   - After a call to *setuid()*, two effective user IDs exist for the process. *switchUid* allows the process to switch back and forth between the two IDs. This allows setting user ID to a privileged mode for a particular operation, then setting it back after the operation, to decrease security holes.
   - *ERRORS:*
     - *FAIL_NOALTERNATE:* No alternate effective user ID exists for the process.

5. **changePriority:**
   - *PARAMETERS (processId, newPriority)*
   - *RETURNS oldPriority*
   - EXECUTION: May be executed by any process.
   - The specified process' priority is changed to the new value. The old value is returned.
   - *ERRORS:*
     - *FAIL_BADPID:* No such process.

- *FAIL_PERMISSION:* The sending process does not have permission to change the other process' priority.

6. **changeQuantum:**
   - *PARAMETERS (processId, newQuantum)*
   - *RETURNS oldQuantum*
   - EXECUTION: May be executed by any process.
   - The specified process' quantum (time slice for running the process) is changed to the new value. The old value is returned.
   - *ERRORS:*
     - *FAIL_BADPID:* No such process.
     - *FAIL_PERMISSION:* The sending process does not have permission to change the other process' quantum.
7. **changeUrgency:**
   - *PARAMETERS (processId, newUrgency)*
   - *RETURNS oldUrgency*
   - EXECUTION: May be executed by any process.
   - The specified process' urgency (time within which the process must be run) is changed to the new value. The old value is returned.
   - *ERRORS:*
     - *FAIL_BADPID:* No such process.
     - *FAIL_PERMISSION:* The sending process does not have permission to change the other process' urgency.
8. **processStats:**
   - *PARAMETERS (processId)*
   - *RETURNS processControlBlock*
   - EXECUTION: May be executed by any process.
   - Statistics about the process are returned, including: priority, urgency, quantum, scheduling state, memory statistics, and so forth.
   - *ERRORS:*
     - *FAIL_BADPID:* No such process.
     - *FAIL_PERMISSION:* The sending process does not have permission to see the other process' statistics.
9. **getProcessId:**
   - *PARAMETERS ()*
   - *RETURNS processId*
   - Returns the process ID of the issuing process.
   - *ERRORS:*
     - None, as yet.

10. **unBlock:**
    - *PARAMETERS (cookie)*
    - *RETURNS success_or_failure*
    - *EXECUTION:* May be executed only by the kernel.
    - Wakes up all process waiting on the particular cookie.
    - *ERRORS:*
      - None, as yet.
11. **sleep:**
    - *PARAMETERS (processId, time)*
    - *RETURNS success_or_failure*
    - *EXECUTION:* May be executed by any process. Only the kernel may put other processes to sleep. User processes can only put themselves to sleep. For all but the kernel, the ID of the calling process must be the processId parameter.
    - The process is sent into the blocked state for the specified time, which is a number of clock ticks. The elapsing of this interval is considered an event like any other event a process may block on.
    - *ERRORS:*
      - *FAIL_PERMISSION:* The process does not have permission to put another process to sleep.
12. **blockOrWise:**
    - *PARAMETERS (processId, cookies)*
    - *RETURNS success_or_failure*
    - *EXECUTION:* May only be executed by the kernel.
    - Sends the process into the blocked state, awaiting unBlock()ing of one or more of the specified events.
    - Note: if the process is already blocked on the OR of some events, these events will be added to the list. If the process is already blocked on the AND of some events, then the call will fail.
    - *ERRORS:*
      - *FAIL_BADPID:* No such process.
      - *FAIL_ANDWISE:* Process is waiting on the AND of some events.
13. **blockAndWise:**
    - *PARAMETERS (processId, cookies)*
    - *RETURNS success_or_failure*
    - *EXECUTION:* May only be executed by the kernel.
    - Sends the process into the blocked state, awaiting unBlock()ing of all of the specified cookies.
    - Note: if the process is already blocked on the AND of some cookies, these events will be added to the list. If the process is already blocked on the OR of some cookies, then the call will fail.

- *ERRORS:*
  - *FAIL_BADPID:* No such process.
  - *FAIL_ORWISE:* Process is waiting on the OR of some events.

2.5 Communications Management

(Abstraction: communicating processes via a number of communications paradigms)

There are four means of interprocess communication which the CLOCS operating system supports: interprocess signalling, events, message passing, and shared memory.

Interprocess signalling consists of the ability for a process to send a signal to another process. Unlike UNIX signals, signal handlers are passed a parameter of type signalMessage, which can convey extra information. Signals may result in a number of outcomes:

1. **Nothing:** a signal can be ignored by a process.
2. **Termination:** a signal can result in the immediate termination of the process.
3. **Handler Response:** a signal can result in a particular action.

The routines supporting this ability are signal(), and handleSignal().

Message passing allows messages of fixed size to be passed among processes. Messages are sent by a process executing the sendMessage() system call, which results in a message being deposited in a mailbox. Processes can wait for messages to appear in mailboxes by use of the awaitMessage() call: they can block awaiting receipt of a message, or they can check for messages without blocking. Mailboxes are created by the mailboxCreate() call. A mailbox is bound to a system-wide location by the mailboxBind() routine; a process may obtain access to a mailbox by using the mailboxAccess() call. For such a call to be successful, the creator and binder of the mailbox (a single process), must currently be executing a mailboxAccept() call. Communication is omnidirectional; any process waiting on a mailbox may receive any message deposited in the mailbox. However, if a mailbox is created sticky, so the messages remain in it until removed, then only one process can access the message at a time. Messages are made available to processes on a first-come, first-served basis; mailboxes can be created, as well, so that messages remain in the mailbox until explicitly removed by a process. All processes using a mailbox are peers; any process can send to the mailbox; any can read from it, and any can remove messages from it[2]. Messages of zero length may be specified as well; this allows mailboxes to be used as semaphores.

The final form of interprocess communication is shared memory. Calls allow a process to make its memory available to other processes; an arbitrary number of processes may share a range of memory. Synchronization of access is the responsibility of the processes, and can easily be done using a mailbox as a semaphore guarding the entire range of memory.

1. **signal:**
   - *PARAMETERS (signalNumber, processId)*
   - *RETURNS success_or_failure*

[2] Directionality of messages might be better for some applications, but breaks the use of mailboxes as semaphores.

- *EXECUTION:* May be executed by any process.
- Like the UNIX kill() mechanism, this routine sends the specified signal to the specified process.
- *ERRORS:*
  - *FAIL_BADPID:* No such process.
  - *FAIL_PERMISSION:* Permission to signal that process was denied.

2. **signalPGroup:**

- *PARAMETERS (signalNumber, processGroup)*
- *RETURNS success_or_failure*
- *EXECUTION:* May be executed by any process.
- Like the UNIX killpg() mechanism, this routine sends the specified signal to all processes in the specified process group.
- *ERRORS:*
  - *FAIL_BADGROUP:* No such group.
  - *FAIL_PERMISSION:* Permission to signal at least one process in the group was denied, based on UID-based permissions.

3. **handleSignal:**

- *PARAMETERS (processId, signalNumber, handlerRoutine)*
- *RETURNS success_or_failure*
- *EXECUTION:* May be executed by any process. The identity of the process executing the call becomes the processId parameter.
- Analogous to the UNIX signal() call, this routine specifies that, upon receipt of the named signal, control should pass to the routine handlerRoutine.
- Three pseudoroutines are allowed as well:
  - SIG_DIE specifies the signal should kill the process.
  - SIG_IGN specifies the signal should be ignored.
  - SIG_DEFAULT specifies the signal should be handled in the default way (either SIG_DIE or SIG_IGN, depending on the signal).
- Signal handlers, as in 4.2BSD UNIX, are retained until explicitly changed.
- *ERRORS:*
  - *FAIL_BADPID:* No such process.
  - *FAIL_BADSIG:* No such signal.

4. **mailboxCreate:**

- *PARAMETERS (messageSize, queueSize, stickiness)*
- *RETURNS mailboxId*
- *EXECUTION:* May be executed by any process.
- Creates a mailbox. The mailbox is not usable until it it is bound to a system-wide location using mailboxBind().
- Messages deposited in the mailbox will be of fixed size *messageSize*.

- Up to *queueSize* messages may be deposited before buffers are exhausted.
- If the *stickiness* parameter is MAILBOX_STICKY, then messages sent to the mailbox are retained in the mailbox until a process explicitly removes them. If the parameter is MAILBOX_NONSTICKY, then messages are removed from the mailbox as they are received by processes.
- If the *retain* parameter is TRUE, then sending messages to that mailbox when the queue is full will not be successful. If the parameter is FALSE, then sending messages to a mailbox with a full queue will result in the oldest message being deleted.
- *ERRORS:*
  - *FAIL_SIZETOOBIG:* Message size specified is too large.
  - *FAIL_QUEUETOOBIG:* Queue size specified is too large.
  - *FAIL_NOMEMORY:* Out of physical memory.
- *analogue of UNIX socket().*

5. **mailboxBind:**

- *PARAMETERS (mailboxId, systemAddress)*
- *RETURNS success_or_failure*
- *EXECUTION:* May be executed by any process.
- Binds the mailbox to the specified system address.
- *ERRORS:*
  - *FAIL_PERMISSION:* Another mailbox has already been bound to that system-wide location.
  - *FAIL_BADMAILBOX:* The specified mailbox is invalid.
- *analogue of UNIX bind().*

6. **mailBoxAccept:**

- *PARAMETERS (processId, mailboxId, flags)*
- *RETURNS success_or_failure*
- *EXECUTION:* May be executed by any process. The identity of the process executing the call becomes the processId parameter.
- The process is blocked until some other process executes a mailboxAccess() call, at which point the processes both have access to the mailbox.
- If the call specifies MAILBOX_UNIQUE, then the mailbox will be duplicated when a connection is made, and communications will proceed through that mailbox, leaving the original mailbox free to accept more connections.
- *ERRORS:*
  - *FAIL_BADPID:* No such process.
  - *FAIL_PERMISSION:* Another mailbox has already been bound to that system-wide location.
  - *FAIL_BADMAILBOX:* The specified mailbox is invalid.
- *analogue of UNIX listen()/accept().*

7. **mailboxAccess:**
   - *PARAMETERS (processId, systemAddress)*
   - *RETURNS mailboxId*
   - *EXECUTION:* May be executed by any process. The identity of the process executing the call becomes the processId parameter.
   - Allows the specified process access to the mailbox which is accessible through the specified system address.
   - *ERRORS:*
     - *FAIL_BADPID:* No such process.
     - *FAIL_BADMAILBOX:* No mailbox is currently bound to the system address.
     - *FAIL_CONNREFUSED:* Connection refused by the creator of the mailbox.
   - *analogue of UNIX connect().*
8. **sendMessage:**
   - *PARAMETERS (processId, mailboxId, message)*
   - *RETURNS success_or_failure*
   - *EXECUTION:* May be executed by any process. The identity of the process executing the call becomes the processId parameter.
   - Sends the included message from the named process to the mailbox. The process must have bound the mailbox to a system location earlier.
   - *ERRORS:*
     - *FAIL_BADMAILBOX:* The specified process has not placed the mailbox in a connected state by either the mailboxAccept() or the mailboxAccess() call.
     - *FAIL_BADPID:* No such process.
   - *analogue of UNIX send().*
9. **awaitMessage:**
   - *PARAMETERS (processId, mailboxId, messageBuffer, timeout)*
   - *RETURNS success_or_failure*
   - *EXECUTION:* May be executed by any process.
   - The specified process is blocked until a message is sent to one of the specified mailboxes, or until the timeout period is exceeded. All specified mailboxes must have been bound to system-wide locations earlier.
   - *ERRORS:*
     - *FAIL_BADMAILBOX:* The specified process has not bound some of the mailboxes to system-wide locations by either the bindMailbox() or the attachMailbox() call.
     - *FAIL_BADPID:* No such process.
     - *FAIL_TIMEOUT:* No message was received within the timeout period.
   - *analogue of UNIX recv(BLOCK).*
10. **checkMessage:**

- *PARAMETERS (processId, mailboxId, messageBuffer)*
- *RETURNS success_or_failure*
- *EXECUTION:* May be executed by any process.
- The specified process retrieves a message if one is present in any of the mailboxes; the process does not block, though. All specified mailboxes must have been bound to system-wide locations earlier.
- *ERRORS:*
  - *FAIL_BADMAILBOX:* The specified process has not bound some of the mailboxes to system-wide locations by either the bindMailbox() or the attachMailbox() call.
  - *FAIL_BADPID:* No such process.
  - *FAIL_NOMESSAGES:* No messages were present.
- *analogue of UNIX recv(NON_BLOCK).*

11. **shareMemory:**

- *PARAMETERS (processId, segmentNumber, pageNumber, numberOfPages, permissions)*
- *RETURNS success_or_failure*
- *EXECUTION:* Can be executed by any process: the ID of the process executing the call provides the processId parameter.
- Makes a range of memory available for sharing by other processes. The process dos not block, but rather, is sent a signal when the memory is actually shared with another process.
- Permissions include read, write, and share, for each of processes in the same group, and for all other processes.
- If a page of memory is being shared by multiple processes, then the page is not released until the last process sharing the memory releases the page.
- The process blocks until another process requests access to the shared area.
- *ERRORS:*
  - *FAIL_BADPID:* No such process.
  - *FAIL_ALREADY:* Some of the pages specified are already being shared.
  - *FAIL_BADSEGMENT:* The process does not have access to that segment.
  - *FAIL_BADPAGE:* The process does not have access to one or more of those pages.
  - *FAIL_INVALID:* The permissions given are bogus.

12. **mapInMemory:**

- *PARAMETERS (processId, sourceProcess, sourceSeg, sourcePage, targetSeg, targetPage, numberOfPages, accessMode)*
- *RETURNS success_or_failure*
- *EXECUTION:* Can be executed by any process: the ID of the process executing the call provides the processId parameter.

- The process requests to map the memory of the target process, in the specified segment and page, into its own address space at the specified location. Permission is requested to read or write.
- Once a process has mapped in another page's memory, it can release the memory by freeing it as if the memory belonged to the process.
- *ERRORS:*
  - *FAIL_PERMISSION:* Permission denied.
  - *FAIL_ALREADY:* Those pages are already present in the process.

2.6 Glue Code

Abstraction: Objects

The glue code is the lowest level of the kernel code. It is the means by which the different modules communicate with each other. The glue code performs the following functions:

- Handle intermodule calls and traps
- Support process use of entry points
- Determine, at each intermodule call, whether the calling process can make the particular call.
- Allow rescheduling and preemption.

2.6.1 Intermodule Communication

Intermodule communication is done through traps (system calls). The calling process specifies the target process and the entry point and traps into the glue code. The glue code determines whether the call can be made by that process. If it can, then the glue code simply context-switches to that place. If the call cannot be made, then the glue code returns an error to the calling process.

Processes make their entry points avaiable to *all* processes by notifying the glue code via the *entry* call to the glue code.

2.6.2 Traps and Interrupts

The glue code handles some traps (its own system call traps and intermodule communication traps); but the majority of traps and interrupts will vector directly to the appropriate handler. For instance, timer interrupts vector directly into the process manager for rescheduling service.

The glue code permits kernel preemption implicitly because it often runs with interrupts enabled. Interrupts are disabled when the glue code is processing its own service calls (*entry, unEntry, unProcess*); at all other times interrupts can occur. Specifically, on calls from one kernel module to another, interrupts can occur. Kernel modules themselves always run with interrupts disabled.

When an interrupt is to vector directly to a user routine, the glue code may well note the fact and adjust scheduling parameters accordingly – especially if the data structures permit constant-time updates.

as well, a special calling paradigm should be adopted, wherein the process ID of the calling process is made an implicit parameter to each externally available call. This facilitates interprocess communication by making sure that the information is always present.

2.6.3 Glue Code Calls

- **entry:**
  - *PARAMETERS (entryPoint, entryNumber, permission)*
  - *RETURNS success_or_failure*
  - *EXECUTION:* May be executed by any process.
  - The entryPoint, an address in the text space of the process, is made a valid entry point for intermodule communication. It is addressed with entryNumber. The permission parameter is used to specify whether any process can call this routine, or whether it is limited to just the kernel. Only the kernel can limit its entry points.
  - *ERRORS:*
    - *FAIL_PERMISSION:* The process tried to limit access to the entry point illegally.
    - *FAIL_ALREADY:* This is an entry point already (the entryNumber is already in use).
- **unEntry:**
  - *PARAMETERS (entryNumber)*
  - *RETURNS success_or_failure*
  - *EXECUTION:* May be executed by any process.
  - The specified entry point is made invalid as an entry point for this process.
  - *ERRORS:*
    - *FAIL_ALREADY:* The entry point is already invalid.
- **unProcess:**
  - *PARAMETERS (processId)*
  - *RETURNS success_or_failure*
  - *EXECUTION:* May be executed by any process. If the process is not a kernel process, then its ID must match the processId parameter.

- Removes all entry points for the specified process.
  - *FAIL_PERMISSION:* A non-kernel process tried to invalidate another process' entry points.

Chapter 3

Scheduling: Algorithms and Ideas

3.1 Requirements

The scheduling algorithm must meet slightly different requirements from other, more standard scheduling algorithms.

3.1.1 Fine Granularity of Scheduling

One objective of the scheduling algorithm is to allow process scheduling to occur with a finer granularity than normal.

3.1.2 Fair Multiprogramming

Part of the reason for desiring finer granularity of scheduling is to allow realtime processes to run at the appropriate time, while still letting non-realtime processes get to the processor.

3.2 Definition of Process

Many definitions of processes have been proposed. We do not need to get involved in the philosophical issues of what a process exactly is.

In this document, the terms "process" and "task" are used interchangeably.

3.2.1 The "Schedulable Entity"

For the purposes of this document, we refer to a process as "the schedulable entity", after Deitel[7]. A process is defined simply as one of the things we are scheduling. Specifically, process is denoted by a data structure called a process control block, or *pcb*.

Attributes of Processes

Processes, as schedulable entities, have a number of attributes that control exactly how they are scheduled.

- **Priority:** Each process has a priority, reflecting its importance relative to other processes.
- **Urgency:** Each process also has an urgency, which determines how quickly it must be run after becoming ready. A realtime process is denoted by the fact that it has a urgency that is greater than zero. Non-realtime tasks have urgency zero.
- **Interaction of Priority and Urgency:** Priority and urgency are not treated exactly orthogonally. Urgency takes precedence. Processes requiring urgent execution simply must be run. When no process' urgency demands running, then the priorities are examined to see which task is the most important.

  It is not clear whether non-realtime tasks should receive preferential treatment when scheduling by priority. This scheduling algorithm will schedule non-realtime tasks preferentially; the realtime processes will be scheduled by priority only when no non-realtime tasks are ready to run.
- **Quanta:** Each process also has a quantum associated with it. This determines how long the process may be run for, before the operating system will interrupt and reschedule.

  A non-realtime task's quantum is varied according to its scheduling behavior, as described below. A realtime task's quantum is varied as well, but in a different way. The scheduler wants to provide as much time as the realtime process needs to do its job. The quantum is interpreted as the estimated run-time of a realtime process.
- **Threshold:** Urgency and quantum are values that don't change as the process moves towards scheduling. A realtime process, when it becomes ready, should be run in (urgency - quantum) clock ticks. This value is stored as the process' threshold. The threshold is the value that is actually varied while the process is enqueued for the processor. When the threshold goes to zero, then the process must run.

  All time-related quantities, including urgency, quantum, and threshold, are stored in units of clock ticks for ease of computation.
- **RunToCompletion:** A realtime process may need to be run to completion whenever it is run. If so, then the runToCompletion attribute should be set. If it is set, then the process will be run with all interrupts masked.

3.2.2 Non-Realtime Processes

Non-realtime processes proceed with no particular urgency, or deadline; they are scheduled solely on the basis of their priority and their quantum.

3.2.3 Realtime Processes

Realtime processes posess urgency as well as priority; they are scheduled first by their urgency, and second by their priority.

3.3 Scheduling Data Structures

The scheduling time must be made as small as possible to meet CLOCS goal of rapid context-switching time. The time to determine which process should run next can be reduced by some clever use of data structures.

3.3.1 Purgatory

Processes that are not runnable are stored in a data structure called purgatory. Processes in purgatory are blocked on some combination of events, either the AND of events or the OR of events. One of these events can be a clock time event: when the specified amount of time passes, the time event has occurred.

Access Requirements for Purgatory

Given the occurrence of a particular event, finding and updating all processes in purgatory awaiting that event must happen rapidly. Because a desideratum of realtime systems is that they respond swiftly to events, this update time is more essential than the time for adding a process to the structure. Also, removing a process from the structure must be fast.

Access Methods for Purgatory

A multilinked structure of some sort seems indicated. Perhaps multiple hash tables or multiple trees will prove effective.

3.3.2 Queues

Runnable processes are stored on two priority queues. One queue, called the Slow Queue, stores non-realtime processes. The other queue is used for realtime processes, and it is called the Fast Queue.

Slow Queue

In the slow queue, processes are sorted by priority into levels. Deitel[7] calls this scheme a "multilevel feedback queue". Quanta should be adjustable as process priority decreases.

Fast Queue

In the fast queue, processes are sorted first by urgency, then by threshold, then by priority. Since urgency is a number of clock ticks that decreases which each timer interrupt, it would be expensive to go through this entire list adjusting each urgency by a constant value. Instead, urgency is used as a differential quantity: each process' urgency is treated as a relative number of ticks, not as an absolute. Thus, if the first process has an urgency of 5, it must be run in 5 ticks; if the second process has an urgency of 2, then it must be run in 7 ticks.

Access Requirements for the Queues

The slow queue is accessed on the basis of a process' priority. The fast queue is more complicated. Processes on the fast queue are accessed in order of threshold, then priority. As well, the fast queue is a differential queue, meaning that the threshold of the process at the head of the queue can be modified. In addition, there can be a number of processes all at a given threshold. The total of the quanta for all these processes must be readily available, in order to determine when that set of processes must be run. In addition, the fast queue can be accessed by priority, for *non-deadline-scheduled* tasks (see below).

Priority and Differential Queues

Aho, Hopcroft and Ullman[1] discuss priority queues, but not differential queues. An example of a priority queue use can be found in the 4.2BSD UNIX code for the routines *softclock()* and *timeout()*, which manage the list of tasks to be performed in real time. These can be found in the file *sys/kern_clock.c*; the queue itself is called *calltodo*.

3.3.3 Current Process

The current process must be kept track of by some means, either by PID, by pointer, or explicit copying of the process control block.

3.4 The Scheduling Algorithm

Given the above data structures, the scheduling algorithm is simple.

3.4.1 Use of the Timer

All scheduling breaks are invoked by the timer interrupt. The timer does not interrupt with a predefined Hertz; rather, by setting the timer to go off in a specified number of ticks, the scheduler allows variable quanta and support for deadline scheduling.

3.4.2 Moving Processes Around

Processes are moved from Purgatory onto one of the two queues when conditions for their awakening have been satisfied. If a process is waiting on the AND of some events, it becomes runnable when they all occur. If a process is waiting on the OR of some events, it is made runnable when one of those events occurs.

3.4.3 When a Timer Interrupt Occurs

When a timer interrupt occurs, rescheduling may occur. The algorithm keeps track of how long it has been since the last timer interrupt. This allows the algorithm to update the fast queue.

If the current process is not a realtime process, then it is inserted back into the Slow Queue. If it has used its entire quantum, then its priority is reduced and its quantum increased. If it has not used its entire quantum, then neither its priority nor its quantum are changed.

If the current process is a realtime process, then its priority is decremented, and the Fast Queue is examined to see if there is now a process more urgent and more important. If there is, then that process is run.

This strategy implies that there are times at which a realtime task may not complete by its deadline. This is acceptable in some circumstances and will be discussed below.

3.4.4 Deciding Who Gets to Run, and For How Long

Figuring out exactly when a realtime task must be run gets a bit tricky.

3.4.5 Urgent Tasks Go First

The whole idea behind urgency is that the process absolutely has to run. Therefore, when a process' threshold goes to zero (meaning that zero time remains before the process has to run), then the process is run. Non-realtime tasks are not even considered for running. Tasks chosen to run based on their thresholds are called *deadline-scheduled* processes.

Within a given urgency, there can be multiple processes. They all must finish at the same time. The scheduling algorithm must maintain a total of the estimated time (quanta) for all of these processes, and schedule so that all the processes finish on time.

3.4.6 "Just In Time" Scheduling

However, when a realtime process' urgency and threshold indicates that it does not need to be run just yet, there may be no benefit to running it yet. In that case, the highest priority task is taken from the Slow Queue and run for the minimum of either its quantum, or the time remaining until the most urgent process must be run.

If there are no processes on the Slow Queue, then the highest priority process on the Fast Queue is chosen. Whether the process is chosen from the Fast or Slow Queues, it is referred to as a *non-deadline-scheduled* process in this context.

Missing Deadlines

When a realtime process is running as the current process and a timer interrupt occurs, signalling the end of that process' quantum, it means that the process did not finish its work before its deadline. This is a happening of variable importance. Some tasks may not care about this. Some may require special action. Some may simply die.

The best action in the CLOCS operating system is to send the process a signal whose default action is to kill the process. The process can change that action to be whatever it deems necessary.

3.4.7 Setting Run Times – Interaction of Quantum and Deadline

Once the next process to run has been chosen, the scheduler must determine how long the process can run for. If the new process is a realtime (I.E. *deadline-scheduled*) process, then it is run for its entire quantum, which has hopefully been adjusted to allow it to complete.

If the new process is not a realtime process, then it is run for its entire quantum *only if it can be run for that long without exceeding some realtime process' deadline.* In other words, non-deadline-scheduled processes are allowed to run for the minimum of their quantum and the threshold of the most urgent process.

3.5 Interfaces of Scheduling

The scheduling system is visible only from within the process manager. Timer interrupts vector directly into the process manager, who examines the scheduling state, determines who should run next, and performs the context switch to that process.

Chapter 4

Interprocess Signals

4.1 Overview

Signals in the CLOCS Operating System operate only slightly differently from the signals provided by 4.2BSD UNIX[1] As in UNIX, signals can be caught, ignored, or dealt with in the default manner, which may be either ignorance or process termination. Unlike UNIX, signals can carry communication through a parameter which is passed to the signal-handling routine. The parameter is passed to the signal system call, and appears at the signalled routine as if it were a parameter to a procedure call.

Signals are blocked while a process is executing a system call; also, while a signal is being handled by a process, other signals of the same type are blocked.

The signal names, descriptions, and many of the default behaviors are derived from UNIX signals. More signals can be added as required in the future.

4.2 Signals

1. **SIGHUP:**
   - *(hangup)*
   - Default action: termination.
   - Parameter: none.
2. **SIGINT:**
   - *(interrupt)*
   - Default action: termination.

[1] UNIX is a trademark of AT&T Communications.

- Parameter: ID of interrupting process.

3. **SIGQUIT:**
   - *(quit)*
   - Default action: termination.
   - Parameter: none.
4. **SIGILL:**
   - *(illegal instruction)*
   - Default action: termination.
   - Parameter: address of fault.
5. **SIGMATH:**
   - *(arithmetic exception)*
   - Default action: termination.
   - Parameter: address of fault.
6. **SIGKILL:**
   - *(kill (cannot be caught, blocked, or ignored))*
   - Default action: termination.
   - Parameter: none.
7. **SIGBUS:**
   - *(bus error)*
   - Default action: termination.
   - Parameter: none.
8. **SIGSEGV:**
   - *(segmentation violation)*
   - Default action: termination.
   - Parameter: address of violation (offending address).
9. **SIGPAGE:**
   - *(paging violation)*
   - Default action: termination.
   - Parameter: address of violation (offending address).
10. **SIGSTOP:**
    - *(stop (cannot be caught, blocked, or ignored))*
    - Default action: process is blocked until SIGCONT received.
    - Parameter: none.
11. **SIGCONT:**

- *(continue after stop (cannot be blocked))*
- Default action: process becomes ready again.
- Parameter: none.

12. **SIGCHLD:**
    - *(child status has changed)*
    - Default action: ignored.
    - Parameter: ID of changed child process.
13. **SIGDEADLINE:**
    - *(deadline of realtime process exceeded)*
    - Default action: termination.
    - Parameter: none.
14. **SIGUSR1:**
    - *(user-defined signal 1)*
    - Default action: ignored.
    - Parameter: process-dependent.
15. **SIGUSR2:**
    - *(user-defined signal 2)*
    - Default action: ignored.
    - Parameter: process-dependent.
16. **SIGUSR3:**
    - *(user-defined signal 3)*
    - Default action: ignored.
    - Parameter: process-dependent.
17. **SIGUSR4:**
    - *(user-defined signal 4)*
    - Default action: ignored.
    - Parameter: process-dependent.
18. **SIGUSR5:**
    - *(user-defined signal 5)*
    - Default action: ignored.
    - Parameter: process-dependent.
19. **SIGUSR6:**
    - *(user-defined signal 6)*
    - Default action: ignored.

- Parameter: process-dependent.

20. **SIGUSR7:**
    - *(user-defined signal 7)*
    - Default action: ignored.
    - Parameter: process-dependent.
21. **SIGUSR8:**
    - *(user-defined signal 8)*
    - Default action: ignored.
    - Parameter: process-dependent.
22. **SIGUSR9:**
    - *(user-defined signal 9)*
    - Default action: ignored.
    - Parameter: process-dependent.
23. **SIGUSR10:**
    - *(user-defined signal 10)*
    - Default action: ignored.
    - Parameter: process-dependent.

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[2] David R. Cheriton and Willy Zwaenepoel. The Distributed V Kernel and its Performance for Diskless Workstations. In *9th Symposium on Operating Systems Principles*, October 1983.

[3] Mark Davis. CLOCS Assembly Language Description. 1988. In preparation.

[4] Mark Davis. The CLOCS MMU. 1987. In preparation.

[5] Mark Davis and Bill O. Gallmeister. *CLOCS Architecture Reference Documents*. Technical Report TR88-021, University of North Carolina, Chapel Hill, May 1988.

[6] Mark Davis and Bill O. Gallmeister. CLOCS Cross Compiler and Assembler Language Description. 1987. In preparation.

[7] Harvey M. Deitel. *An Introduction to Operating Systems*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.

[8] Harvey M. Deitel. *An Introduction to Operating Systems*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.

[9] Edsger W. Dijkstra. The Structure of the 'THE' Multiprogramming System. *Communications of the ACM*, 11(5):341–346, May 1968.

[10] Bill O. Gallmeister. Reconciling Real-Time and "Fair" Scheduling. April 1988. In preparation.

[11] Bill O. Gallmeister. Signals in the CLOCS Operating System. 1988. In preparation.

[12] Bill O. Gallmeister. The CLOCS Operating System – Overview and Specification. 1988. In preparation.

[13] E. Douglas Jensen, C. Douglass Locke, and Hideyuki Tokuda. A Time-Driven Scheduling Model for Real-Time Operating Systems. In *Real-Time Systems Symposium*, pages 112–122, December 1985.

[14] Insup Lee and Vijay Gehlot. Language Constructs for Distributed Real-Time Programming. In *Real-Time Systems Symposium*, pages 57–66, December 1985.

[15] Samuel J. Leffler. A 4.2BSD Interprocess Communication Primer. In *UNIX 4.2BSD Manual, Volume2C*, January 1983.

[16] Samuel J. Leffler, William N. Joy, and Robert S. Fabry. 4.2BSD Networking Implementation Notes. In *UNIX 4.2BSD Manual, Volume2C*, January 1983.

[17] Mike Manley. Private communication. March 1988.

[18] Phil Miller. VMS for Realtime? *HARDCOPY*, 76–80, October 1987.

[19] Karsten Schwan, Tom Bihari, Bruce W. Weide, and Gregor Taulbee. High-Performance Operating System Primitives for Robotics and Real-Time Control Systems. *ACM Transactions on Computer Systems*, 5(3):189–231, August 1987.

[20] Niklaus Wirth. Toward a Discipline of Real-Time Programming. *Communications of the ACM*, 20(8):577–583, August 1977.