THE DESIGN OF A VERIFIABLE OPERATING SYSTEM KERNEL

by

THOMAS WAYNE LOCKHART

P.Eng., British Columbia, 1975

B.S., Massachusetts Institute of Technology, 1972

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

Department of Computer Science

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

November 1979

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of COMPUTER SCIENCE

The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1W5

Date 16 November 1979

Abstract

The design and implementation of an operating system kernel is described and justified. The kernel implements processes and primitive operations on processes. A variety of operating systems can be implemented by processes executing in the environment the kernel provides. Processes communicate via messages. Several facilities found in other kernels, such as memory management and device handling, are not included. Instead, the kernel provides operations that allow these services to be implemented by processes.

The ease of reasoning about (i.e. the verifiability of) both the kernel and the systems it supports is of primary concern. The kernel's verifiability is enhanced because: it has been kept as small as reasonable; all kernel operations are indivisible; the semantics of the kernel operations are simple; and, the kernel does not depend upon any processes for its correct operation. The verifiability of systems the kernel supports is enhanced because they can be modularized such that each module can be verified individually. This is possible because the kernel eliminates many process interdependencies.

An implementation of the kernel that supports a multi-process test system lends empirical support to the design. Although a complete operating system has not yet been constructed or verified, the results to date are promising.

Table of Contents

List of Figures

Acknowledgement

I wish to thank the people without whom this thesis would not have been written. Much credit belongs to my supervisor, Dr. David Cheriton, for suggesting the topic, many lively and productive arguments, and much constructive criticism. Many ideas, including susceptibilities, arose out of discussions with Steve Deering, and the rest of the coffee club too! Thanks are due to Dr. Sam Chanson and John Demco for reading and commenting on the penultimate draft. Thanks are also due for the financial support of NSERC (formerly NRC) and my father, Dr. H. B. Lockhart, without which I would have been working for a living, not writing this thesis. Finally, I would like to acknowledge my wife, Michele Lockhart, for her inspiration when I was dry, and for her moral support (and occasional threats) when I needed them. Despite the help of these people, the responsibility for the inevitable mistakes is mine.

Chapter 1 Introduction

This thesis describes the design and implementation of an operating system kernel. The kernel extends the base machine architecture to provide an environment suitable for supporting message-based, multi-process structured operating systems. Chapter 2 describes the kernel operations in detail and chapter 3 justifies their selection. Some interesting aspects of the implementation are given in chapter 4, and chapter 5 presents our conclusions and suggestions for future research.

1.1 Objective

The objective of this work is to explore techniques for building verifiable operating systems. A central idea is that of structuring systems as a kernel and a set of cooperating processes, all of which can be verified independently. Verification involves increasing confidence in system properties, through understanding and convincing arguments. This research does not explore verification techniques, but rather how to structure operating systems so that existing verification techniques can be applied to them. Most operating systems are so large and convoluted they are impossible to verify. Confidence in these systems is usually based on operational experience and testing, not understanding. Advantages of verifiable design include: maintainability, robustness of software, and security as well as increased confidence in the system.

In addition to being verifiable, other properties are of

interest. The systems should be able to meet real-time criteria for responding to external events within a constant maximum time. Their designs should be machine-independent for portability. A level of functionality similar to UNIX, Thoth or other advanced operating systems must be possible. And finally, they must be implementable with sufficient efficiency to be useful.

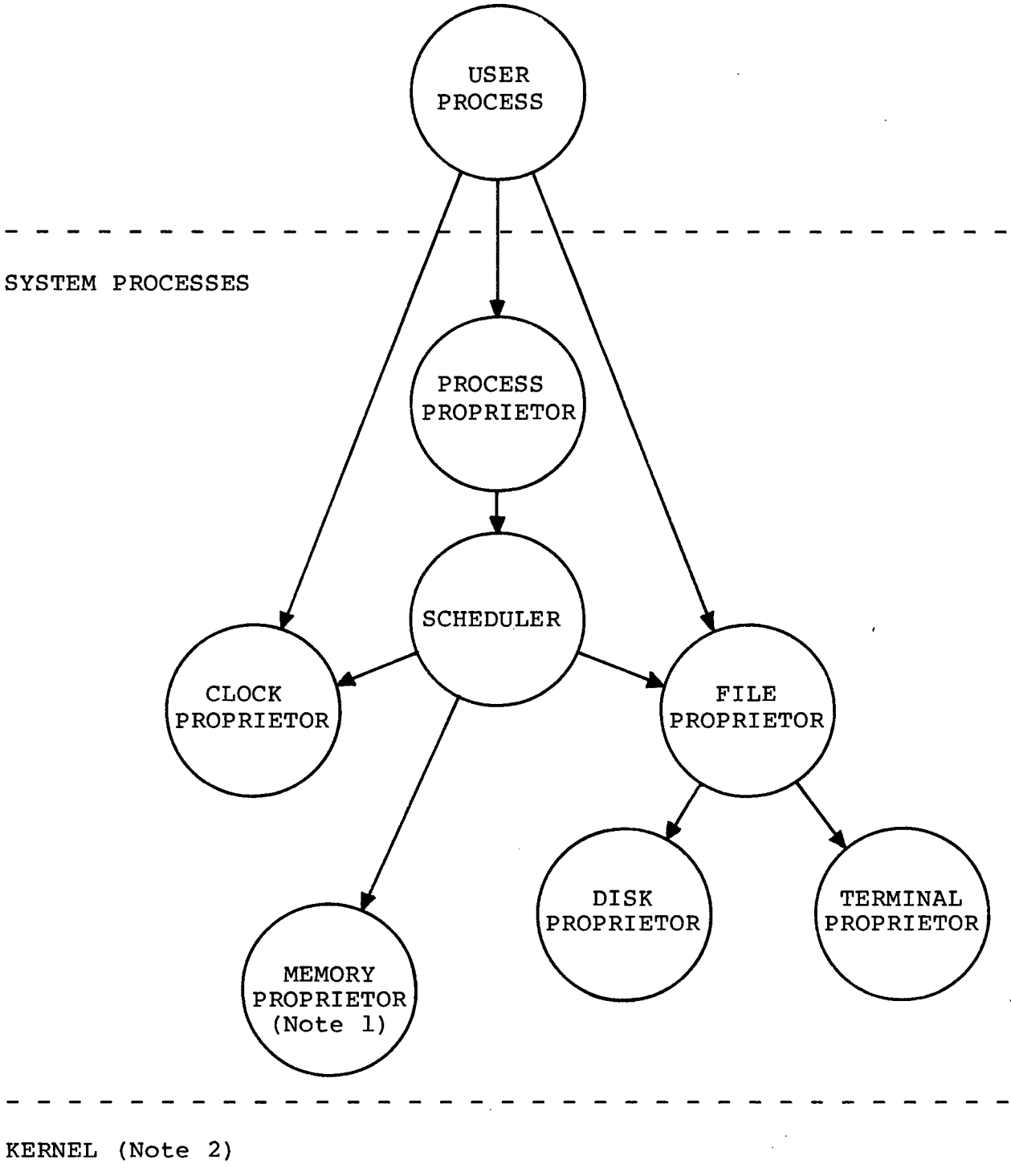
The difficulties of understanding an operating system came to the author's attention when studying the kernel of the Thoth system. Even though Thoth is highly modular in nature, the correct operation of its kernel depends upon the correct operation of the system processes, which in turn depends upon the kernel. Thus understanding Thoth's kernel requires understanding considerably more than just the kernel. In his Ph.D. thesis, Cheriton [1979a] discussed the problem this circularity causes for the verification of Thoth. Solving this problem was the original objective of this work. This was extended, however, when it was discovered that careful kernel design would not only make the kernel independent of the system processes, but would also allow the system processes to be designed so they could be verified individually.

1.2 Strategy

"Divide and conquer" is the basic strategy for making the system verifiable. If the system is structured as a number of small modules that can be considered individually, its verifiability is greatly enhanced. The required independence can

be demonstrated with a dependency graph. The dependency graph of a system includes a node for each module, and directed arcs from each node to the modules it depends on for its correct operation. If this digraph is acyclic, then there are no circular dependencies. Thus, the correctness of each module can be demonstrated independently by assuming the correctness of all modules it depends on. When all modules are verified, the system has been verified. The only global property that needs to be demonstrated is that the dependency graph is acyclic. This acyclic dependency notion is first attributed to Dijkstra [1968b].

Processes plus the kernel constitute the modules of the systems the kernel supports. Independent modules cannot share an address space, thus the kernel must have its own address space and it must allow the processes to have distinct address spaces as well. Occasionally it is advantageous for closely cooperating processes to share an address space, however they then constitute a single module. Figure 1 shows the dependency graph of a typical system the kernel could support.



Note 1 - All processes depend upon the memory proprietor.
Note 2 - All processes depend upon the kernel.

FIGURE 1: TYPICAL SYSTEM DEPENDENCY GRAPH

The criteria that guided the design of the kernel are based upon this basic strategy, plus the strategies of keeping the kernel small and simple. Two sets of criteria were employed, as the kernel design involves two activities: designing the environment provided by the kernel; and, implementing this environment. These criteria are discussed in detail in chapter 3, however they are summarized here to clarify our strategy and objectives. The criteria that guided the design of the environment include:

1. The kernel should allow the systems it supports to be modularized with an acyclic dependency graph.
2. The kernel should not provide facilities that can be provided efficiently by a process.
3. The kernel operations should have simple well defined semantics.
4. It must be possible to implement the environment within the implementation criteria.

The criteria that guided the implementation of the kernel include:

1. The kernel operations should be indivisible.
2. The maximum time for the kernel to respond to external events should be small and bounded by a constant.
3. The design should be machine independent.
4. The implementation should be efficient.

Only the first of these implementation criteria is motivated by verifiability, the others reflect additional desired system properties.

1.3 Kernel Definition

Several definitions of operating system kernels are found in the literature.

> "The kernel is a primitive monitor that implements objects and indivisible operations on these objects as an extension of the basic machine architecture."
> [Cheriton 1979a](Thoth).

> "The kernel is the only program in DEMOS which executes in monitor mode so that memory protection between tasks can be enforced."
> [Clifford & Montague, 1978](DEMOS).

> "The lowest layer, the kernel, provides basic services such as interprocess communication, process dispatching and trap and interrupt handling."
> [Lycklama & Bayer, 1978](MERT).

> "Essentially the monitor is a software extension of the hardware structure, which makes the RC4000 more attractive for multiprogramming."
> [Brinch Hansen, 1969](RC4000).

> "A security kernel is a small, isolated part of the operating system, built at the lowest level, which is generally intended to contain all the security enforcement code."
> [Popek & Kline, 1977](UCLA-VM).

> "The kernel that is referred to here is defined as all programs that implement or affect access control of any kind, discretionary or non-discretionary."
> [Schroeder et. al., 1977](Multics).

In this thesis, the kernel is defined as the lowest level of software. It is independent of all other software, and implements multiple processes by abstracting the hardware interrupt and protection mechanisms. This is a synthesis of the concepts of a multi-programming kernel or nucleus, and a security kernel. It does not "contain all the security enforcement code", but it does contain the lowest level of this code. The kernel has complete control of the security features

of the hardware, access to the devices (including the file support devices), and the memory. The kernel can be thought of as a primitive operating system, providing a set of low-level, primitive "system calls" that can then be used to implement a full operating system.

More specifically, the kernel provides:

1. many inexpensive deterministic processes;
2. interprocess communication via small synchronous messages, and a data transfer facility.
3. operations for process management, which includes: dynamic process creation and destruction, flexible processor scheduling, memory management, and exception handling;
4. a simple system initialization mechanism;
5. device abstraction, including: limited access, controlled DMA, and buffering for efficient handling of rapid interrupts.

The kernel does not provide memory management, a file system, or high-level abstraction of devices. It does however, as mentioned above, provide operations so that these facilities can be provided by system processes. For example, the kernel provides operations for changing processes' address spaces and execution priorities, but it is up to the system processes to implement swapping, or some other memory management scheme.

1.4 Background

This work extends previous research done on the RC4000, UNIX, Multics, UCLA-VM, MERT, DEMOS, and Thoth operating systems.

The RC4000 system [Brinch Hansen 1969, 1970] introduced the idea of structuring an operating system as a kernel (or nucleus) which implements processes and messages, plus a set of system processes. Our concern for verifiability lead to a number of differences from the RC4000 kernel. For example, our kernel abstracts devices via restricted kernel operations, while the RC4000 system provided external (pseudo) processes. It also provided dynamic buffering of messages, with the possibility for one process to send many messages concurrently, whereas our message-passing is totally synchronous.

The kernel of the UNIX system [Ritchie & Thompson, 1974] is similar to ours in that: it extends the facilities of the host processor to provide a multi-process environment; it is the only software to run in privileged mode and thus have full control of the hardware; it only depends on the hardware for its correct operation; and it has a separate address space. However, it differs in that it implements most of the UNIX environment, including the file system, I/O system, and memory management, which constitutes most of what is typically considered to be an operating system. These differences are due primarily to our desire to build a small kernel that supports a set of system processes, while UNIX uses a very different structuring technique.

The objective of the Multics kernel design project [Schroeder et. al., 1977], to provide verifiable security of an operating system, was similar to ours. The basic stategy was the same as well. They emphasized the acyclic dependency notion and

attempted to remove unnecessary functions from the kernel. Multics is not, however, a message-based multiprocess structured system. Also, the kernel to which they refer is a security kernel, not specifically a multiprogramming kernel as ours is, and thus it implements more of the functions of the system. The scale of their system is much larger than ours as well.

The UCLA-VM system [Popek & Kline, 1977] is similar to ours in that it is primarily concerned with verifiability. It is also structured as a kernel (or levels of kernels) plus a set of processes. The kernels even have similar properties, such as: being the lowest level of software in the system; being the only "privileged mode" software; having address spaces distinct from all processes; being small; and having indivisible operations. These similarities reflect our similar concerns, but are overshadowed by the differences. The environments, or abstract machines, provided by the two kernels are very different. The UCLA-VM kernel includes "security objects" and is primarily concerned with controlling access to these objects. It does not provide general purpose interprocess communication facilities. Because of this, it is unsuitable for supporting systems with different system process structures. This is quite different than our kernel which makes no assumptions about the system process structure. Finally, they make no attempt to exploit the acyclic dependency notion.

MERT [Lycklama & Bayer, 1978] contains a kernel somewhat similar to ours. It is structured as four layers: a kernel, kernel processes, supervisor processes, and user processes.

> "A rich set of inter-process communication mechanisms including messages, events (software interrupts), shared memory, inter-process traps, process ports, and files allow applications to be implemented as several independent, cooperating processes".
> [Lycklama & Bayer, 1978]

On the surface, the strategy of the MERT designers is very similar to ours. The basic structure is the same: a kernel supporting a set of system processes. However, their research objectives were quite different. For example, our criterion of verifiability made their diverse set of interprocess communication primitives unacceptable. They considered the protection/efficiency tradeoff, and opted for efficiency. As our primary concern is verifiability, we opted for protection and have attempted to find ways to be efficient despite the protection.

DEMOS [Clifford & Montague, 1978] is a kernel/process structured system concerned with security. Its kernel is similar to ours in that it implements processes and message-passing. Also, it has its own address space, which is distinct from that of most processes. The major difference is complexity. Its link-based message-passing [Baskett et. al., 1977], which includes dynamic message buffering, is much more complex than our scheme. Also, DEMOS does not exploit the acyclic dependency notion, nor are its kernel operations indivisible.

Thoth [Cheriton, 1979a, 1979c] demonstrated the feasibility of a set of primitives for structuring an operating system as a set of cooperating deterministic processes. Several of these primitives, in particular the interprocess communication

primitives, were adopted by our kernel. Thoth's "kernel", which implements these primitives, is not separated from the system processes. It shares a common address space with the system processes, some of which even have responsibility for maintaining some kernel data structures. This lack of independence is the main difference between it and our kernel. A significant implication of this is that our kernel must provide process management and device manipulation operations, which are not required of the Thoth kernel.

In summary, the basic kernel/process structure which we propose is not new. What is new is our exploitation of the acyclic dependency notion, in a message-based multi-process environment, to achieve verifiability.

1.5 Summary

The objective of this research is to explore techniques for building verifiable operating systems. In particular, the structure of a kernel plus a set of system processes is explored. The interdependencies of system modules are reduced so the system can have an acyclic dependency graph, which implies that each module can be verified individually. The kernel is defined to be the lowest level of software. It extends the facilities of the hardware to provide an execution environment including multiple processes. Most similar systems have not had verifiability as a primary design constraint.

The main contribution of this research is the demonstration that strict rules of inter-module dependencies can be enforced

without impacting efficiency severely. Although "breaking the rules" may be the most obvious method of improving the efficiency of a mechanism, when forced to look, an efficient but legitimate technique can usually be found.

Chapter 2 Execution Environment Provided

The execution environment implemented by the kernel consists of a set of processes, a set of devices, and operations for interprocess communication, process management, and device manipulation. A message-based interprocess communication scheme is used, including operations for reading, writing, and synchronously exchanging message buffers. This scheme is augmented by a data move operation. The process management operations can be used by authorized processes to: create and destroy processes; control process attributes; and handle exceptions. Authorized processes use the device manipulation operations to control, communicate, and synchronize with timing, character, and block devices.

A process deterministically executes a subset of the host machine's instruction set enhanced with the kernel operations. The attributes of a process include:

1. id - an unique identifier used to refer to the process. Id's have two fields, an index number, and a cycle number. Index numbers are used for efficient access to per-process data, because they are unique for all existent processes. Zero is a special invalid id, used as an error indication and to indicate an unspecific process for some operations;
2. machine state - a copy of the host processor's registers used by the process. This is a machine-dependent property. For example, for the TI990, a machine state includes three registers: the status register, ST; the program counter, PC;

and the workspace pointer, WP;

3. map - a specification of the main memory accessible to the process, (its address space). This is a machine-dependent property, which, for the TI990, is a 6-word vector specifying the base and limit of each of three memory segments;
4. priority - an integer used to determine the priority with which the process is allocated the processor;
5. status - indicating the capabilities (for restricted kernel operations and device access) and susceptibilities (to process management operations) possessed by the process;
6. violation - an indication of the nature of the last exception caused by the process.

The kernel allocates the processor to (dispatches) the process with the highest priority that has been ready for the longest time. A process's priority is also used to determine its interruptability on machines that have selective interrupt masks. The higher a process's priority, the less interruptable it is.

The kernel provides a protection scheme based on capabilities for executing restricted operations, and accessing devices. Associated with each restricted operation and with each device (or group of devices) is a capability. The status of a process indicates what operations and devices it possesses the capability to use. It also indicates the susceptibilities of the process, which determine which process management operations may affect it. Susceptibilities are used to prevent the system processes from depending on each other for their correct

operation.

Some exceptional conditions are detected by the kernel. When a process causes one of these error situations, (e.g. by attempting to access data outside of its address space), it is arrested, which means that it is coerced by the kernel into sending to an exception handling process. This exception handler will then decide what to do about the exception, which could include restarting the offending process. Only conditions that are not expected to occur under normal circumstances result in being arrested. Less serious exceptions, such as replying to a non-existent process, result in an error indication being returned by the kernel operation.

The kernel abstracts the devices of the system only to the degree necessary to allow system processes to efficiently and safely access them. Devices are classified into three types: timing, character, and block. Operations are supplied for manipulating each of these types of devices. In addition, general-purpose operations allow direct access to the control registers of most devices.

The following sections describe the kernel operations in detail.

2.1 Interprocess Communication

Processes synchronize and exchange small amounts of data with each other via messages. Each process has one message buffer, which is a small (8-word) vector in the kernel's address space.

Processes read, write, and synchronously exchange their message buffers with other processes using the following operations.

.Write_msg(msg)

copies the vector at msg in the invoker's address space to its message buffer. An exception occurs if the vector specified is not in the invoker's address space.

sender_id = .Read_msg(msg)

copies the invoker's message buffer to msg in its address space, returning the id of the process that sent (or replied to) the message. An exception occurs if the specified vector is not in the invoker's address space.

.Send_msg(receiver_id)

blocks the invoker until its message buffer has been received by the specified receiver, forwarded zero or more times, and replied to. The invoker awaits a reply. The replier's id is returned by the .Read_msg operation, described above. An exception is caused if receiver_id is invalid.

.Await_msg(sender_id)

blocks the invoker until a message is received from the specified sender, unless the message has already been sent, in which case it is received immediately. If id is zero, the sender is unspecified, and the earliest message sent to the invoking process is received. The sender can be determined with the .Read_msg operation described above. An exception occurs if sender_id is an invalid id other than zero.

sender_id = .Receive_msg(sender_id)

is the same as .Await_msg except that instead of blocking, it returns zero if the message has not been sent, otherwise it returns the sender's id. If sender_id is invalid, an error indication is returned.

ok = .Reply_msg(sender_id)

unblocks the process specified by sender_id after exchanging message buffers with it. An error indication is returned if id is not valid. An exception is caused if the process specified by sender_id is not awaiting a reply from the invoker.

ok = .Forward_msg(sender_id, receiver_id)

causes the process specified by sender_id to send the invoker's message buffer to the receiver specified by receiver_id. The receiver receives the invoker's message buffer as if it had been sent by the sender. The invoker does not block. If either sender_id or receiver_id are invalid, an error indication is returned. If the sender is not awaiting a reply from the invoker, an exception occurs. A process can forward a message to itself, which has the effect of putting the message to the end of its incoming message queue.

ok = .Move(n_words, src_id, src, dst_id, dst)

moves a vector of n_words from src in the address space of the source process specified by src_id, to dst in the destination process's (dst_id) address space. Both processes must have eligible priorities, indicating they are in memory. If either id is invalid, or either process has an ineligible priority, an error indication is returned. Both processes must be awaiting reply from the invoker or be the invoker, otherwise an exception

occurs. An exception also occurs if either vector specified is not in its respective address space.

An example of a typical interaction may clarify how these operations are used. Assume there are two processes: a proprietor process that performs service in response to requests; and a requestor process that requires the proprietor's services.

The standard form of a proprietor process is:

```
.Proprietor()
  {
    \ Initialization.

    repeat
      {
        .Await_msg( 0 );     \ From any process.

            \ Proprietor may be blocked here.

        requestor = .Read_msg( request );

            \ Performs the requested service.
                \ This might involve moving data to or
                \ from the requestor using .Move.

            \ Builds the reply.

        .Write_msg( reply );

        .Reply_msg( requestor );
      }
  }
```

The requestor requests service from the proprietor by: building the request, writing it into its message buffer (with .Write_msg), and sending it to the proprietor. Sending blocks the requestor until its request has been serviced and replied to. When the requestor resumes execution, it reads the reply from its buffer (with .Read_msg), and proceeds.

2.2 Process Management

The kernel provides operations that authorized processes can use to: create and destroy processes, control the capabilities of processes, schedule the processor, manage main memory, and handle exceptional conditions. These restricted operations may only be invoked by a process whose status indicates that it has the capability for the operation.

The first process is created by the kernel when the system is started. It has all capabilities, the highest priority, and the maximum address space (disjoint from the kernel's address space). All other processes are descendants of this root process which has the responsibility to complete the initialization of the system, including the creation of the rest of the system processes.

new_id = .Create_process()

creates a process returning its id, unless the maximum number of processes already exist, in which case it returns zero. The created process has no memory, no status, an ineligible priority, and a null machine state. It is awaiting a reply from the invoker.

ok = .Set_priority(OF id, TO priority)

sets the priority of the specified process. Priorities of a process are used by the kernel to allocate the processor. Priorities range from zero, the highest, to a small positive constant (currently 16), which is the lowest eligible level. There is also an ineligible level, indicated by any larger

priority, from which processes are not dispatched.

priority = .Get_priority(OF id)

returns the priority of the specified process.

ok = .Set_status(OF id, FROM status)

sets the status of the specified process from the vector specified by status. The status of a process indicates: what restricted operations it is capable of performing; what process management operations it is susceptible to; and, which devices it can access.

ok = .Get_status(OF id, INTO status)

copies the status of the specified process to the vector specified by status.

ok = .Set_map(OF id, FROM map)

sets the machine-dependent address map of the specified process from the vector at map in the invoker's address space. The map specified is checked by the kernel to ensure that it does not overlap with the kernel or the device control registers. If it does, the map is not changed, and an error indication is returned. An error indication is also returned if the specified process's status indicates that its map cannot be changed (i.e. it is not susceptible to .Set_map).

ok = .Get_map(OF id, INTO map)

copies the address map of the specified process to the vector at map in the invoker's address space.

ok = .Set_machine_state(OF id, FROM state)

sets the machine state of the specified process from the specified vector. The kernel ensures that the new state cannot violate the kernel (e.g. for the TI990, the status register, ST, indicates non-privileged mode and user map). If id is invalid, or if the process is not susceptible, an error indication is returned. An exception occurs if the specified process is not awaiting a reply from the invoker.

ok = .Get_machine_state(OF id, INTO state)

copies the machine state of the specified process to the specified vector. An error indication is returned if id is invalid.

ok = .Set_violation(OF id, TO violation)

sets the violation indication of the specified process. An error indication is returned if id is invalid. An exception occurs if the specified process is not awaiting a reply from the invoker. Note that this operation does not cause an exception to occur for the specified process, it merely changes the violation indication.

violation = .Get_violation(OF id)

returns the violation indication of the specified process. An error indication is returned if id is invalid.

ok = .Destroy_process(id)

destroys the specified process by: invalidating its id, releasing its kernel-managed resources, clearing its message buffer, and causing an exception for all processes blocked on it, by putting them in the pending arrest list (see

.Next_violator below). An error indication is returned if the specified process is not destroyable, as indicated by its status.

id = .Next_violator()

returns the id of the next process in the pending arrest list (processes that had been blocked on a destroyed process). The process specified by id is caused to await a reply from the invoker. If the pending arrest list is empty, zero is returned.

Without intending to limit the flexibility of the system designer, it is envisaged that the capability for each restricted operation will be limited to a few processes. For example, the .Create_process, .Destroy_process, and .Next_violator operations would be used exclusively by a "process proprietor" process, (and the root process). All other processes would send requests to the process proprietor to create or destroy processes. .Set_map would be used by a "memory manager" process, and .Set_priority by a "scheduler" process. .Set_status is such a powerful operation, that its use should be restricted to the root process while it is setting up the system processes. .Set_machine_state and .Set_violation are considerably less powerful operations, as the affected process must be awaiting a reply from the invoker. Thus more system processes might use them. For example, proprietor processes could arrest a faulty requestor by setting its violation indication, and forwarding it to the exception handler. The query functions, such as .Get_machine_state, are even less critical, as they do not affect the correct operation of

processes.

2.3 Device Manipulation

The devices abstracted by the kernel are classified into three broad types: timing, character, and block devices. Real-time clocks and timers are examples of timing devices. Terminals are character devices and disks are block devices. In addition to the specific operations for these types of devices, there are general purpose device operations. Each device has a capability associated with it which a process must possess to use the operations to manipulate the device. The particular characteristics and configuration of devices in the system are maintained in a device descriptor table in the kernel's address space.

General Purpose Operations

Most devices have control registers. Most have the ability to interrupt the processor to signal an event. The kernel provides general purpose operations for reading and writing device registers, and for synchronizing with device events. These low-level operations do not determine the actions of the device, simply the effect upon its registers. These operations cannot be used to control devices that can affect the kernel, (i.e. unmapped direct memory access, DMA, devices).

count = .Await_event(FROM device)

returns a count of the number of times the device has interrupted since it was last awaited. If the device has not

interrupted yet, the invoker blocks until it does and then returns zero to indicate that no interrupts were missed. The invoker is locked in memory (i.e. its map is not settable) while it is blocked. The invoker is arrested if a process is already awaiting the device.

.Write_bits(n, THRU m, OF device, WITH value)
puts the m-n+1 low-order bits of value into the n-th through m-th, bits of the device's control register.

value = .Read_bits(n, THRU m, OF device)
returns the n-th through m-th bits of the device's control register in the low-order bits of value.

Timing Devices

The kernel maintains a real-time clock in a three word time vector. The first word indicates the number of clicks since the last second while the other two words indicate the elapsed time in seconds. The size of a click is a machine dependent parameter equivalent to the resolution possible with the particular hardware clock, (1/120 second for the TI990/10).

.Write_realtime(time_vec)
copies the three word vector at time_vec in the invoker's address space to the kernel's time vector, thus setting the clock. If the number of clicks specified is greater than or equal to the number of clicks per second minus one, the next click will increment the seconds by one and set the clicks to zero.

.Read_realtime(time_vec)

copies the kernel's time vector to time_vec in the invoker's address space.

.Write_timer(clicks)

sets a kernel-maintained timer to clicks which may be reset while running. The timer interrupts after the specified number of clicks. A process uses the general purpose .Await_event operation described above to wait for the timer to expire.

Character Devices

The kernel abstracts character devices, such as terminals, by buffering the characters exchanged with these devices. These operations block on appropriate buffer conditions.

.Write_character(c, TO device)

puts the character, c, into the device's buffer. If the buffer is full, the invoker blocks until it is empty. The invoker is locked in memory while it is blocked. If a process is already blocked because the device's buffer was full, the invoker is arrested.

c = .Read_character(FROM device)

returns the next character from the device's buffer. If the buffer is empty, the invoker blocks until a character arrives. The high order byte of the returned value is a count of the number of characters missed since the last character was returned. The invoker is locked in memory while it is blocked. If a process is already blocked because the device's buffer is empty, the invoker is arrested.

Other features of character devices, such as baud rate and parity selection are not explicitly provided for. The general purpose operations described above are used to control the device's special features.

Block Devices

Block devices are not abstracted by the kernel, except for restricting their use, and interpreting all addresses relative to a process. Thus the actions the devices take as a result of these operations is entirely device-dependent.

ok = .Write_regs(OF device, FROM regs, WITH id)
writes the device's control registers with the contents of the vector regs in the invoker's address space. All addresses specified by the registers are relative to the specified process. The kernel translates them if necessary. The specified process is locked in memory while DMA is in progress with it.

id = .Read_regs(OF device, INTO regs)
copies the contents of the device's registers into the vector at regs in the invoker's address space. It also returns the id specified by the last .Write_regs of this device.

Processes use the general purpose .Await_event operation, described above, to synchronize with block devices.

2.4 Summary

The execution environment implemented by the kernel extends and abstracts the environment provided by the hardware.

Processes and devices are the basic kernel objects. The set of operations allow for interprocess communication, process management, and device manipulation. Processes are execution instances of programs that deterministically execute a subset of the host processor's instruction set plus the kernel operations. They are referred to by unique id's, and their attributes include: machine states, maps, priorities, statuses (capabilities and susceptibilities), and violation indications. Devices are classified into three types: timing, character, and block devices.

Interprocess communication is via small synchronous messages, augmented with a data move operation. Processes read, write, send, await, reply, and forward message buffers. Senders await replies.

Authorized processes manage processes. They create, destroy, and manipulate the attributes of processes. Processes must possess the capability for a process management operation to execute it.

Access to devices is provided by general purpose operations as well as operations for each type of device. Only processes with the capability for a device manipulate it. Generally, the operations only specify the effect upon the device's control registers, not the action of the device.

The kernel objects and operations provided allow the structuring of the system as a small self-contained verifiable kernel plus a set of system processes which can be independently

verified. Elaboration on this claim and the rationale behind this design are provided in the next chapter.

Chapter 3 Justification of the Environment

This chapter presents and discusses the design criteria that were used, and then justifies the design of the environment in terms of them.

3.1 Design Criteria

The objective of this work is to explore techniques for building verifiable operating systems. Cheriton [1979b] has suggested that the following characteristics make operating systems difficult to verify:

1. large size,
2. complexity,
3. concurrent execution,
4. dynamic behavior, and
5. asynchronous events.

The kernel/system-process structure is directly aimed at reducing the difficulties resulting from these characteristics. Two distinct design activities are associated with the kernel component of the system: the design of the environment it provides; and the design of its implementation. Different criteria are required for each activity, although they are related. The environment must be suitable for supporting a set of verifiable system processes, while the implementation must itself be verifiable.

The following criteria guided the design of the environment the kernel provides.

1. The kernel should allow the systems it supports to be modularized with an acyclic dependency graph.

Modularization is a powerful tool for achieving verifiability. If the modules exhibit an acyclic dependency graph, the verifiability of the system is further enhanced, because the correctness of each module can be considered separately. A direct consequence of this criterion is that the kernel must be independent of all processes. It must reside in a separate inviolate address space. It must not even rely on the existence of any processes (e.g. for exception handling).

2. The kernel should not provide facilities that can be provided efficiently by a process.

In this way, the size of the kernel is kept as small as reasonable. However, this criterion must not be taken to the extreme. Often, features can be provided by the kernel with dramatic increases in efficiency and only trivial increases in complexity of the kernel. An example of this is counting clock interrupts.

3. The kernel operations should have simple, well-defined semantics.

This is an obvious point, but it is very important both from the point of view of developing the system processes correctly and verifying the kernel. Special cases are often the source of program failures.

4. It must be possible to implement the environment within the implementation criteria.

This criterion reflects the relationship between the design of

the environment and its implementation. It has often been necessary to consider how the environment could be implemented in order to allow its selection.

The following criteria guided the design of the implementation of the kernel.

1. The kernel operations should be indivisible.
This restriction simplifies the demonstration that the integrity of the kernel data structures is maintained despite the occurrence of an interrupt. In fact it simplifies the kernel in general because it implies that all kernel operations execute sequentially and deterministically. Thus, sequential program proving techniques can be used. A consequence of this criterion is that the interrupts must be totally disabled during kernel operations.

2. The maximum time for the kernel to respond to external events should be small and bounded by a constant.
This is the real-time constraint. It is necessary because a general purpose operating system must be able to respond to real world events such as synchronous communication line interrupts. Also it is desirable that the kernel be able to support real-time applications such as process control. A consequence of this criterion is that the time for which the interrupts are disabled must be bounded by a constant. The previous criterion required the interrupts to be fully disabled during kernel operations. Combining these implies that the maximum execution time for a kernel operation must be bounded by a constant.

3. The design should be machine independent.

It is not the objective of this work to develop software for a particular machine, but rather to investigate structures that are generally useful for a large class of machines. Thus, the design of the environment and the implementation should not be governed by specific features of any one machine. However, a broad class of the machines of interest provide similar features, and these can and should be taken into consideration.

4. The implementation should be efficient.

The system should be efficient in order to be practical, but decreases in efficiency to improve verifiability are incurred. Where efficiency is of concern, it is usually time and not space efficiency, because of the desire to provide real-time response.

In addition to the preceding criteria, the implementation should adhere to good programming practices. In particular, it should use structured control constructs, use meaningful identifiers, and be well commented and documented. To this end a high-level language is used almost exclusively.

These criteria enhance the verifiability of both the system and the kernel, by reducing the size of the verification units, ensuring that each unit executes sequentially and deterministically, and simplifying the semantics of the units. Systems adhering to these criteria will be useable, convenient, and portable.

3.2 Interprocess Communication

The kernel's synchronous fixed-size messages enable processes to synchronize and pass control information without becoming mutually dependent. Larger and variable amounts of data are handled with the data move operation. Favorable experience with Thoth's similar primitives indicates that this scheme is reasonable.

3.2.1 Messages

The send, await, and reply message primitives are sufficient for process synchronization. Cheriton [1979a] has shown they are at least as powerful as Dijkstra's general semaphores [1968a].

Each process has only one message buffer so the kernel cannot deadlock on message buffer space. If a dynamic allocation scheme is used, the kernel could run out of buffers and be unable to support any further messages. This problem is avoided by using a static scheme, of which one message per process is the simplest. Processes can only use one buffer anyway, because senders wait for replies, and thus have at most one message in transit at a time.

Messages are fixed-size because: they can be implemented easily; they can be statically allocated to avoid the deadlock problem; and, they can easily simulate variable-size messages (in combination with .Move). Statically allocated variable-size messages reduce to fixed-size messages of the largest size. The cost of exchanging messages does not vary with their size,

because only pointers are exchanged. Variable-size messages would permit accessing variable amounts of the message buffer. This can be easily provided with fixed-size messages as well, but its usefulness is doubtful unless messages are large.

Messages are small so that the cost per process is small and many processes can be provided. Messages are intended for control information and small amounts of data. The .Move operation is for larger or variable amounts of data. The current 8-word size was selected because it worked well for Thoth. Most messages do not use all 8 words, so a reduction might be desirable, but buffer space has not been scarce enough to motivate this.

Replies allow the system to be structured with an acyclic dependency graph. The send (and await reply), await (from any process), and reply sequence allows a proprietor (or manager) process to accept and service requests without depending upon its requestors. A replying process does not block because the sender is awaiting the reply (unless it has been destroyed). Thus, a proprietor can return information to, and synchronize with, a requestor without depending on it.

The non-blocking receive allows processes to be independent with respect to timing criteria. If a process must respond within a specified time, it cannot await a message without depending on some process to send to it within that time. For example, a device proprietor that must service periodic interrupts, as well as await and service requests, has this problem. If it blocks awaiting a request, it is depending on

some process to send to it before the next interrupt. If it receives without blocking, it can service the request, if any, and only block for the interrupt. It need not depend on any requestors.

There are other solutions to this problem. A timer process can send back within a specified time. This mechanism works well for slow rates, but requires an extra process and two more process switches per interrupt than the non-blocking receive. Or, the non-blocking and blocking receives could be generalized into a receive with a variable timeout, from zero to infinity. But it is not clear how to implement this without severely complicating the kernel. Finally, strict timing requirements could be met by the kernel itself, which has in fact, been done for timing and character devices.

Forward was included because it is convenient to redirect messages without becoming dependent upon and complicating the new receiver. This ability also permits a process to redirect a message to itself, thus ignoring a request and moving it to the end of its incoming message queue.

The specific receive operation, which is useful when processes are closely cooperating, could be generalized to receive from sets of processes. With this operation, a process could receive from only the processes it was cooperating with, while ignoring other processes. Although this operation would be useful, it is not included because it is not necessary. It can be simulated using forward and receive unspecific. Also, it is not clear how to implement receiving from arbitrary subsets of

processes in constant time.

Reading and writing the message buffer with operations distinct from send, receive and reply allow the latter operations to be indivisible. Previous versions of these operations blocked before moving the message buffer, which violates the indivisibility criterion. This argument applies only to reading the buffer. Writing need only occur before sends and replies, thus it could be integrated with them. The separate write buffer operation was included for the sake of symmetry and because it reduces the average time to respond to external events.

The indivisibility criterion also implies that values cannot be returned from the blocking send and await operations. This is because the return value is not known when the invoker blocks and the unblocker cannot return the value because the recipient (the sender or awaiter) might not be in memory at the time. Thus, the return would have to occur after the blocking process had been dispatched, making the operation divisible.

Processes that use the communication primitives incorrectly are arrested for three reasons. First, it eliminates the need for the invoker to check for error indications. Because the kernel does these checks to protect itself, it can easily arrest offending processes. Second, it ensures that faulty processes are dealt with. The invoker may not check for error indications, either out of neglect, or because it is out of control. Third, arresting these processes gives the system designer more flexibility, not less. Arrested processes are easily restarted

by an exception handling process. But if the kernel ignores these errors, the system has no alternatives.

The kernel does not arrest processes for errors that can arise in normal operation of the process. For example, replying to a requestor that has been destroyed does not cause an exception. The same is true for both the forwardee and the intended receiver specified in a forward operation.

Processes invoking the message operations are arrested under the following conditions:

1. sending to, or awaiting from a process with an invalid id,
2. forwarding or replying to an existent process not awaiting reply from the invoker, or
3. specifying a vector to read or write that is not in the invoker's address space.

The latter two cases are clearly indicative of faulty situations, but the first is controversial. Blocking on a non-existent process is considered an error because the invoker is attempting to become dependent upon a non-existent process. The invoker most likely can not function correctly without it. One exception to this is a "vulture" process that awaits a message from a process that does not send to it, solely to monitor its destruction. This construct, although expensive, could be useful for resource reclamation. None the less, the arrest alternative was selected because it is more flexible. Also only minor changes to the kernel and the processes would be required to revert to the non-arrest alternative.

In summary, the message-passing scheme implemented by the

kernel allows processes to synchronize and communicate with each other without becoming mutually dependent. It is easy to understand, convenient to use, simple and efficient to implement, and cannot deadlock.

3.2.2 Other Communication Mechanisms

The data move operation provides efficient communication of large amounts of data. It also allows variable-size data to be handled naturally, rather than packaged into messages. Its use is restricted to transfers of data between processes awaiting reply from the invoker so other processes do not depend on the mover; its requestors already do. Also, there are no synchronization problems, because at most one of the processes involved, the invoker, can execute. Full read/write access to the requestor's address space may seem excessive, but is convenient for debuggers, exception handlers, and other monitoring processes. This functionality could be provided by a system process using .Set_map to give itself access to the data to be moved. The kernel provides .Move though for several reasons:

1. it is easier, more elegant, and faster for the kernel to do it,
2. it prevents that part of the system from becoming a bottleneck, and
3. it gives low-level processes this facility.

To prevent a mover from being arrested when it attempts to move data to or from a swapped process, the kernel should ensure that the processes are in memory, returning an error indication if

they are not. A minor problem with the semantics of this operation, is that the best the kernel can do is to check that the processes have eligible priorities. They could, however, be swapped and have eligible priorities but this is unlikely. Words rather than bytes are moved, because this is simpler. If it is desired to move an odd number of bytes, this can be simulated by the invoking process, saving kernel complexity. The maximum number of words that can be moved is limited so that the maximum disable time is bounded.

Pipes, first introduced by UNIX, are a convenient mechanism for interprocess communication and synchronization. They can be thought of as one-byte buffered messages, or as shared temporary files with the reader synchronized to be behind the writer. Pipes can be simulated with the kernel's message scheme. Because they are thought of as files (and thus part of the I/O system), they are logically at a higher level than the kernel. For these reasons, pipes were not included.

A potentially attractive alternative to small, fixed-size messages is to use segments of a process's address space rather than message buffers in the kernel's address space. This is impractical with most mapping hardware, and abstracting the mapping hardware more than it already is would complicate the kernel.

Sharing memory segments is another technique for interprocess communication. As the kernel does not determine memory allocation strategy (except for protecting the kernel's memory), it neither provides nor disallows memory sharing. The system may

implement this if desired. Processes that share segments they can modify become interdependent for their correct operation.

In summary, of the non-message communication mechanisms considered, only a data move operation was included. It efficiently alleviates the inconvenience of handling large or variable amounts of data with small fixed-size messages. The other schemes considered, pipes and shared memory, were considered to high-level to include in the kernel.

3.3 Process Management

Process management involves:

1. initializing the system processes,
2. creating and destroying processes,
3. managing main memory,
4. scheduling the processor,
5. controlling protection and security, and
6. handling exceptional conditions.

It is desirable for system processes to implement as much of these management functions as is reasonable so that the kernel is small. Processes cannot directly access the kernel to implement these features without the kernel depending on theses processes. Providing process management operations rather than accessibility, allows the kernel to be independent of the system processes, even though they do much of the process management.

Some of these operations are so powerful they can be used to prevent other processes from working correctly. Thus their use must be restricted. A process must have the capability for one

of these operations (indicated by its status) to be authorized to execute it. Unauthorized processes invoking these operations are arrested, and thus prevented from executing unless the exception handling process restarts them.

This section argues that these operations are simple and efficient, but still allow flexible system design.

3.3.1 System Initialization

The kernel first initializes its own data structures, and then creates one process to initialize the system. Giving the responsibility to initialize the rest of the system to a process eliminates the need to change the kernel whenever the process structure of the system changes. Thus the kernel is independent of the system process structure. One "root process" is also easier to initialize than multiple system processes.

A table driven root process has been written that enables a process to be added to (or deleted from) the initialization procedure simply by changing a table and recompiling this small process. Other initialization schemes are also possible.

One disadvantage of the root process scheme is that the .Set_status operation would not be necessary if the kernel created all processes that had special authorization. However, because .Set_status is trivial to implement, the root process initialization scheme is preferable.

3.3.2 Process Creation

.Create_process allocates and initializes the kernel data structures required for a new process. This operation is restricted so that a system process can: prevent excessive consumption of process descriptors, which can cripple a system; and, ensure that processes are destroyed when their usefulness has ended. An interesting corollary of this is that the kernel is not required to maintain a process descendant hierarchy. These hierarchies, indicating the creator and descendants of every process, are usually found in systems with unrestricted create operations. Their primary function seems to be to facilitate the destruction of a process's descendants when it is destroyed.

The attributes of a new process are always initialized to the same values, so that separate processes can manage these attributes without depending upon all creators. If, for example, a creator could determine arbitrary initial priorities, it could disrupt the processor allocation strategy implemented by a scheduler process.

A new process is created awaiting a reply from its creator, so that it may be initialized and passed arguments. Thus the kernel need not provide this usually machine-dependent feature. A new process may be forwarded by its creator to a third process (such as the requestor of the creation) to initialize. This could be desirable to eliminate a bottleneck if only one process has the capability for .Create_process.

3.3.3 Processor Scheduling

The multi-level priority and dispatching strategy used by the kernel is both capable of responding to real-time events and flexible enough for time-sharing applications. This is possible because there are two levels of dispatching. A micro dispatching strategy, implemented entirely by the kernel, allows dispatching decisions to be made quickly without the intervention of a process. And, the .Set_priority operation allows a scheduler process to implement an arbitrary macro dispatching strategy.

The kernel takes advantage of the hardware interrupt masking mechanism so that the time to dispatch the highest priority process in response to an interrupt only depends upon the kernel and is bounded by a constant. The process with the most stringent response requirement can be assigned the highest priority and thus have a guaranteed maximum response time. As long as this process executes in constant time, other processes can have guaranteed response times as well. If interrupts were not maskable, the response times would depend on all the interrupt rates.

The ability to dynamically change process priorities with .Set_priority gives a scheduler process the power to implement almost any gross (macro) dispatching strategy desired. For example, the processor may be equitably shared among a number of processes by periodically cycling their priorities, each process getting equal time at each level. The number of priority levels is fixed so that both the bounded disable time and indivisibility criteria can be met.

For a process to be swappable, or use virtual memory, it must be suspendable. That is, it must be possible to prevent its execution, even if it is ready. For this reason, the kernel provides an ineligible priority level from which processes are never dispatched. Even without the ineligible level, processes could be suspended by setting their priorities lower than an "idle" process that never blocks. However, the idle process scheme causes extra process switching, and uses resources such as a process descriptor and memory; while the ineligible priority level is easy to implement.

3.3.4 Protection/Security Control

The capabilities of a process determine which restricted kernel operations it can use, and which devices it can access. Its susceptibilities determine the process management operations that affect it. Collectively, the capabilities and susceptibilities of a process constitute its status. Process statuses can be modified by the restricted .Set_status operation. This mechanism allows the implementation of a protected operating system and eliminates many system module dependencies that would otherwise occur.

Two kinds of dependencies arise from the kernel operations themselves. Some operations can prevent the correct operation of the processes they operate on. For example, the .Set_map operation can be used to disrupt a process simply by setting its map incorrectly. This is the first, direct kind of dependency. The second is indirect. If two processes, p and q, can both set

maps, and p must set maps to function correctly, then p depends upon q, because q can change what p has done.

The capability mechanism can be used to eliminate most of these dependencies by controlling which restricted operations, if any, a process can use. However, this mechanism alone is not sufficient to structure the system with an acyclic dependency graph. Consider two system processes, p and q, where p requires q services, and thus p depends on q. If p can destroy q, then q depends upon p, and they are mutually dependent. The susceptibility mechanism can be used to eliminate this source of dependency by determining what operations can be performed on a process. In the above example, q's status could indicate that it was not destroyable (by any process), which eliminates the cycle.

Another approach to this problem is to allow only one process to execute all operations that can cause direct dependencies. This is unacceptable for two reasons: this "super" process would be a bottleneck; and it is not a very modular way to construct a system.

A simple privileged-mode scheme, that restricts the execution of a subset of the operations to a subset of the processes, eliminates dependence (from this source) upon non-privileged processes. This mechanism is provided by the hardware of many systems, and seems to be sufficient to protect the system from users. It does not however eliminate the mutual dependence of system processes.

3.3.5 Memory Management

The capability to set memory maps allows a process to control the use of non-kernel memory. Processes can be given access to any areas of non-kernel memory, subject only to the limitations of the mapping hardware. Thus, separate address spaces for each process, shared segments, teams (processes sharing the same address space), or any other scheme can be provided.

The .Set_map operation allows the full functionality of the mapping hardware, with the exception that no process is allowed to access the kernel's memory. This low-level operation is simple to implement and gives the system designer considerable flexibility. Because address spaces are not changed very often, a process can efficiently manage main memory.

The .Get_map operation is not necessary, but is included because it is simple to implement and eliminates the need for a (memory manager) process to maintain a copy of each process's map.

The kernel must ensure that direct memory access, DMA, (by devices such as disk controllers), cannot access kernel memory or any device control registers. If DMA is not mapped, but can access all of memory, the kernel simulates mapped DMA by translating all requests relative to a specified process's map. This is fairly simple to do, and is also convenient for the processes initiating DMA. In this way, they do not have to be aware of the memory actually being used by the processes involved in the DMA.

A problem exists if the memory of a process involved in a DMA transfer is reallocated before the transfer is finished. The DMA transfer will continue into the reallocated memory. Thus the kernel monitors all DMA activity, and sets the susceptibilities of all processes involved, so their maps cannot be changed until the DMA has finished. This is not necessary for the kernel's independence, but it eliminates the dependence of a memory manager on processes controlling DMA.

3.3.6 Process Destruction

Destroying a process involves two basic activities. The resources allocated to it must be reclaimed, and the processes blocked attempting to communicate with it must be dealt with. This is much more difficult than process creation because the process must be extracted from its environment, which may affect a large number of processes.

The .Destroy_process operation reclaims the kernel-managed resources, which consist primarily of a process descriptor. It does not reclaim any process-managed resources because of the independence criterion. This in turn implies that destruction must be a restricted operation, so the system processes can ensure that the resources they manage are reclaimed. Moreover, indiscriminate destruction has to be prevented. Because destruction is restricted, the system can control it. Two necessary restrictions are imposed by the kernel. Some processes are not destroyable, as indicated by their susceptibilities. This enables the elimination of the dependency loops that result

if a process can destroy the processes it depends upon. Processes not susceptible to map changes are also not destroyable. This last restriction ensures that a destroyed process's memory cannot be reallocated while DMA is in progress with it.

The second activity, dealing with the processes blocked on the destroyed process, is the most difficult and controversial, because there can be arbitrarily many such processes. Both what should happen to these "abandoned" processes and how to accomplish the desired functionality are interesting issues. There seem to be two choices of what to do: arrest the processes or return an error indication to them. The kernel implements the first of these alternatives, because it is more flexible in addition to being easier to implement. When processes are arrested, they are stopped and a system process gains control over them. It can then restart the offenders with an error indication, thus subsuming the other alternative, or it can destroy the offenders, ask an operator what to do, or do virtually anything else desired. This method is easy for the kernel to provide because it already includes the necessary machinery for dealing with other exceptions. A disadvantage of the arrest alternative is that exception handling is more complicated because of the additional type of exception. Also, we have no prior experience with the arrest technique, whereas the direct error return alternative has been used successfully in Thoth.

The problem with implementing either alternative, is that

there is an arbitrary number of processes involved. Thus they cannot be handled individually in a single kernel operation without violating the bounded disable time criterion. Four solutions were considered, three of which require maintaining a list of the processes blocked on each process.

The solution used appends the lists of senders and awaiters of the destroyed process to a special kernel-maintained list. The .Next_violator operation is used to remove processes from this list and gain control over them by causing them to await a reply from the process invoking .Next_violator. The advantages of this solution are that destruction is completed in one indivisible operation, and it can be implemented cleanly. A disadvantage is that it requires multiple invocations of .Next_violator to clean up after a process is destroyed.

The second solution is to append the lists of blocked processes (senders and awaiters) onto the incoming message queue of the exception-handling process, effectively simulating a mass send operation. This technique has many advantages. It is one step, does not require the .Next_violator operation or a special list, and results in immediate destruction. However, it cannot be implemented cleanly in constant time. It requires messages to be received from processes that are not in the sending state (as indicated by their process descriptors). If a better way of thinking about this last difficulty can be found, this would be the best alternative, as it is the simplest and most efficient.

The third solution is to make destruction a multi-step operation, with only a bounded number of blockers dealt with on

each operation. Although this can be implemented cleanly, it requires multiple steps, and destruction would not be immediate, implying a process could be partially destroyed.

The solution which does not require maintaining a list of processes awaiting each processes, is to simply destroy processes, and after each destruction (or periodically) repeatedly use a special operation to check every process, arresting any that are blocked on a non-existent process. This is very inefficient if there are many processes.

3.3.7 Exception Handling

Exceptions are unusual or faulty conditions indicating that execution should not proceed normally. The kernel encounters a number of such conditions. The hardware generates an interrupt when exceptions such as memory errors, illegal instructions, and addressing errors occur. Also, the kernel checks arguments, capabilities, and susceptibilities for exceptions. Some of these situations are always the fault of the process incurring the exception, while others are not. The first group includes such situations as executing illegal machine instructions, addressing outside of an address space, invoking kernel operations without the capability to do so, using bad arguments to kernel operations, replying to a process not awaiting a reply, etc. The second group includes replying to a requestor that has been destroyed, setting a property of a destroyed process, creating a process when there are no more process descriptors, etc.

The kernel uses two different mechanisms to handle these two

cases. When an exception is clearly the fault of the incurring process, it is arrested. This means it is coerced into sending to an exception-handling process, which is then in a position to investigate the exception and determine what action to take. The less serious exceptions are dealt with by returning an error indication, and letting the incurring process determine what to do about it.

The arrest mechanism gives the system considerable flexibility for handling exceptions. It is also fairly simple to implement by taking advantage of the message-passing mechanism. A minor disadvantage of this scheme is that it requires that the system include an exception-handling process, which limits the flexibility of the system designer. However, this is not significant because most systems would include such a process.

Four process management operations are provided to facilitate exception handling (and debugging). The .Get_machine_state operation is included so that an exception-handling process can determine where in an offender's address space to move a return value. It is also useful for a debugger to know the machine state of the process being debugged. .Set_machine_state allows a debugger to transfer control, in the process being debugged, in response to a debugging command. Because the .Set_machine_state operation can be used to make processes non-deterministic (by changing their program counters), it use should be restricted. In fact, it might be preferable to not provide it at all, which would require the initialization of processes machine states as part of the creation operation.

The kernel sets the violation indication of an arrested process, indicating the exception detected. The .Get_violation operation is used by the exception handler to get this information. Instead, the message sent to the exception handler could contain the violation indication, but then information would be lost, and the offending process could not be restarted. .Set_violation was included for symmetry, but could also be used by proprietor processes so that exceptions they detected were handled in the same way as kernel-detected exceptions. That is, when a proprietor detected an exception, it could set the offender's violation indication with .Set_violation and forward the offender to the exception handler.

The less serious exceptions do not result in arrests to avoid problems such as the exception handler incurring an exception, or processes that the exception handler depends upon incurring exceptions.

Three alternative mechanisms for dealing with exceptions were rejected. Processes incurring exceptions could be destroyed, but this would make it difficult to determine what happened. Exceptions could be ignored (i.e. the kernel operations could return immediately upon detecting an exception), but then errors would be hard to detect. Signals, or software interrupts, could be used to transfer control to a special section of the process's code, but this violates the notion of a process executing deterministically.

Arresting is implemented by coercing messages to a particular exception-handling process. This technique was selected in favor

of two alternatives because it seemed the simplest and most straightforward. Rather than one central exception handler, it might be preferable to specify an exception-handling process for each process, because of dependency problems with the one handler scheme. The other mechanism does not employ coerced messages at all, rather a special list is used to hold all arrested processes, and an additional operation is used to synchronize with and remove offenders from the list.

The issues of exception handling are difficult and deserve more attention than we have been able to give them.

3.3.8 Summary

Process management involves: initializing the system; creating and destroying processes; managing memory; scheduling the processor; controlling security; and handling exceptional conditions. The kernel provides the process management operations so that system processes can perform much of this task. The kernel creates one root process which then initializes the system processes, because this is simpler for the kernel than creating all of the system processes, and allows the kernel to be independent of the system process structure. Process creation does not interfere with other management functions. Two scheduling mechanisms provide speed and flexibility. The kernel's micro dispatching strategy is self-contained and thus fast, whereas the dynamic priority mechanism allows a process to implement any macro scheduling strategy desired. The capability/susceptibility mechanism not only protects the system

processes from user processes, but also from each other. Both are essential for an acyclic dependency structure. Memory is abstracted very little, because this is simple to implement, and it gives the system designer great flexibility. Process destruction is difficult because of the potentially unbounded number of processes that are affected. It is handled in constant time by moving lists of processes. Processes incurring exceptions (faults) are arrested. That is, they are coerced into sending to an exception-handling process which then determines what to do about them, because this is the most flexible option.

3.4 Device Abstraction

The kernel design explores two alternative approaches to device abstraction. The first requires the kernel to do as little as possible, by providing operations for accessing device registers. The second provides convenient yet simple higher-level abstractions of the devices.

The kernel must at least: ensure that devices cannot compromise the kernel; restrict access to devices on a per-device basis; and disable (and/or acknowledge) interrupts that are not being awaited by a process. One way devices can compromise the kernel is via direct memory access, DMA. If a device can access all of memory because its DMA is not mapped, a process controlling the device could access the kernel's memory, which violates the independence criterion. The use of devices must be restricted on a per-device basis to eliminate the indirect dependencies that would result if two processes could

access the same device. The kernel must be able to disable (and/or acknowledge) interrupts, so that if an interrupt is missed, the kernel can prevent the system from going into a tight loop, responding repeatedly to the same interrupt.

The main advantage of the kernel abstracting devices as little as possible is that the kernel can be independent of the device configuration of a particular system, and also independent of the particular device interfaces. Thus none, or only a small amount, of the kernel code would be device-dependent. The environment provided by the kernel would, however, be very device-dependent. But, this would allow most of the device-dependent code to be contained in a few small "interface" proprietors which would implement interface-independent abstractions of the devices, used by the device proprietors. Once the kernel is verified, it would be advantageous not to modify it at all. Even recompilation is undesirable, as compilers have a tendency to evolve. Although simple, new device drivers could be incorrectly added to the kernel by people unfamiliar with it. Another advantage is that the kernel would be smaller if it did not include the code to manipulate device interfaces.

The primary disadvantage of this scheme is inefficiency. Actually its efficiency depends on the hardware mechanisms for manipulating devices. The hardware could be designed so there would be no loss of efficiency with this scheme. It could restrict access to each device separately, ensure that devices could not compromise the kernel (by mapping DMA), and provide a

standard mechanism for disabling (and/or acknowledging) interrupts. But because current hardware does not have these features, all device manipulation has to go through the kernel to obtain the required functionality. This results in all the overhead of kernel calls for each device operation. If these operations are too low-level, the overhead is considerable.

The second approach to device abstraction, to provide a convenient yet simple abstraction of each device type, is more convenient for the operating system. If the kernel does all of the low-level manipulation of the device interfaces, the system processes do not have to. This approach is also slightly more efficient because the operations are higher-level. That is, only one kernel call is required for each logical transaction. Also, it is possible with this scheme to provide very high-level abstractions, with large increases in efficiency. For example, character-oriented devices require service for each character, but they could be abstracted to appear to be DMA devices. The major disadvantage of this approach is that the kernel would contain a significant amount of device-dependent code, and further that the kernel would have to be modified whenever the configuration of devices in the system changed.

Although the kernel has been designed to provide either approach to device abstraction, insufficient experience has been obtained to determine which is preferable.

Devices are classified into three broad types (timing, character, and block) on the basis of the mechanism used to communicate with them. Timing devices typically have no data, or

only a small fixed amount of data, associated with them. Character devices logically handle data a character or word at a time, at a relatively slow rate. Block devices often have high transfer rates with the data bypassing the processor, going directly to or from memory. This operative classification of devices is useful because all devices in a class can be manipulated with the same kernel operations.

3.4.1 General Purpose Operations

The most common and important general purpose operation provided is .Await_event, which is used to synchronize with timing and block devices. It could also be used to synchronize with strange conditions occurring on character devices. The most controversial aspect of this operation is that it returns immediately (i.e. does not block) if an interrupt has occurred since the last time it was awaited. This is however the desired effect in all missed interrupt situations we have encountered. A few examples illustrate this point. First, consider a timer that causes an event when it expires. If a process awaits the event after it occurs, it could wait forever if .Await_event did not return immediately. The same is true for a disk handler if the action it started completes before it is awaited. Even devices that repeatedly interrupt because of real world events rather than in response to internal requests (e.g. input from a communications interface), work better with an immediate return. If an interrupt has been missed, the character may still be recoverable from the interface's buffer. But if the next interrupt is awaited, the character is guaranteed to be lost. In

most instances the immediate return does not even complicate the awaiter, as it handles missed events the same way as normal events. Of course, handlers of devices for which missed interrupts are important have to test this condition, but they have to do something special anyway.

Returning a count of the number of missed interrupts is of questionable usefulness as few situations require thus information. However it is just as easy to provide as the missed indication that is necessary for some devices. Also, if the kernel disables missed interrupts, the count can only be zero or one. It must acknowledge, not disable, the interrupt if the count is to be significant.

Each logical device has at most one event associated with it. This simplifies the semantics of the .Await_event operation. If it is necessary to use .Await_event to independently sychronize with more than one type of interrupt from a single physical device, it must be configured as more than one logical device, and the kernel separates the events. Communication interfaces are the most common case of a device causing multiple types of interrupts (e.g. read request and write request), but these do not need to be configured as two logical devices because they are synchronized with by using the .Read_character and .Write_character operations, not the .Await_event operation. If two or more physical devices share an interrupt level, the kernel separates these into distinct logical events, despite the fact that a process could perform this function. That is, each interrupt level could be associated with a logical event (and

thus a logical device) and a process could await it. This process would then poll the devices connected to the level to determine which had interrupted and pass this information to the proprietor process associated with the device. This was not done because it would cause an extra process switch for each interrupt on a shared level.

The device control operations, .Write_bits and .Read_bits, provide direct access to device control registers. Although these operations are neither aesthetically pleasing nor efficient, they allow the full power of a device to be used by the system processes while the kernel only needs to know how to access its registers, not how it works. These operations are particularly useful for controlling "programmable" interfaces that allow numerous aspects of the device to be changed under software control. Thus the kernel does not need to provide operations to set transmission rates, the number of stop bits, synch characters, or other unusual features of an interface. A disadvantage of these operations is that they could be very difficult to implement on hardware that has strange timing considerations associated with accessing device registers. The author, however, is not aware of any such considerations, other than requirements for minimum delays.

3.4.2 Timing Devices

The kernel extends the functionality of the clock hardware, which (on the TI990/10 and many other minicomputers) provides an interrupt at periodic intervals (120 Hz on the TI). The kernel

maintains the time and provides a variable timer service. This has been done for three reasons. First, it is more efficient for the kernel to process the hardware interrupts because it does not incur the overhead of a process switch on each hardware interrupt. Second, it allows a cleaner system process structure for the clock. Third, real-time clock hardware on some machines provides this level of functionality. Because the kernel abstraction is high enough to exploit this hardware, the design is more portable. That is, only the kernel has to be modified to make use of this more sophisticated real-time clock hardware. The primary disadvantage of this scheme is that it requires the kernel to be more complicated. The amount of code required to implement this functionality is small however, and the significant gains in efficiency seem worth the extra complexity.

The time is represented as a large precision binary number of seconds plus clicks, because this is simpler and more efficient for the software to deal with than Julian time. This is simpler for all software that deals with time, not just the kernel. The program environment library includes routines to convert from the binary representation to the Julian and vice versa.

3.4.3 Character Devices

The kernel provides efficient machine-independent operations for communicating with character devices. The specific details of accessing the device interfaces are handled by the kernel. This has the advantage of allowing processes that use the devices to be interface-independent, allowing the system

processes to be more portable. However, it also results in a less portable kernel. Because giving characters to and getting characters from the interfaces are common operations, a high-level abstraction seems reasonable. Other less common operations, such as setting the rate of a communication line, or determining error conditions are not included. These operations tend to be highly interface-specific, and occur so seldomly that kernel operations specifically for these device operations seem unjustified. The less-efficient general-purpose operations for manipulating device interfaces give the system the ability to use these more esoteric features of the hardware.

Output data is buffered by the kernel to increase efficiency. By providing buffers, the amount of process switching arising from outputting characters is reduced. Because outputting characters is a very frequent operation, its efficiency is of concern. Input data is much less frequent, and is buffered simply to make the synchronization code simpler.

These operations synchronize on full (output) and empty (input) buffers, because this is more convenient for the device-handling processes, and is not difficult to implement. The alternative of asynchronous get and put operations, that return indications that the process should wait before attempting further communication, would be slightly easier to implement, but would also be far less convenient to use, as well as less efficient. The asynchronous scheme would also require the use of a separate synchronization operation (e.g. .Await_event).

Block communication operations are an interesting alternative to these character-oriented operations. Simulated DMA operations would be both convenient and efficient, particularly for outputting large quantities of data, as when listing files. These operations would even be easier to implement than the buffered character operations, either by the kernel or directly by the device interface. For example, a block output operation could specify the number and location of the characters to be output. The kernel would copy these characters into a buffer in its address space, and signal an event when they had been output. Similarly, the kernel could place all input characters into a buffer in its address space, signaling an event on each character arrival, and when a process wanted the characters it would invoke an operation to copy them into its address space. This scheme would make "type ahead" trivial to implement. It would not even be necessary for the kernel to keep buffers if it accessed buffers in the proprietor's address space whenever it serviced the interface. These operations were not included primarily due to a lack of time, and because the character operations were more familiar.

3.4.4 Block Devices

Block devices communicate data via direct memory access, DMA. As previously stated, if DMA is not mapped, the kernel must control it directly. If it is mapped, the kernel must control it via the mapping hardware. The .Write_regs and .Read_regs operations allow this control. All addresses are interpreted relative to a process's address space, which guarantees that the

DMA cannot effect the kernel because all process address spaces are distinct from the kernel's address space. An alternative to translating addresses relative to a process's address space is to give each DMA device its own address space, or at least its own map. This would however require some mechanism for specifying this map. .Set_map could not be used for this without complicating its semantics. The mechanism provided is easy to implement whether the host machine maps DMA or not. In fact, it allows the system processes to be oblivious as to whether or not it is mapped.

Control of the actual device parameters, such as block addresses, are not abstracted for two reasons. It is unnecessary for the kernel to do, and simplifies the kernel to exclude it. It also allows the system more flexibility as to how logical block addresses, or even logical devices, map onto physical block addresses.

These operations are asynchronous for increased flexibility. Often one controller is used to interface to several devices. In these situations it is not uncommon for the controllers to allow the devices to be controlled asynchronously. An example is concurrent seeking of disk drives on a common controller. It is still straightforward for a device handler to synchronize with the device by awaiting its completion immediately after requesting the operation.

3.4.5 Summary

Two approaches to device abstraction are considered. The first is for the kernel to remain as device-independent as possible. This would result in a more portable kernel that is insensitive to device configuration changes. It would also allow device dependencies to be localized in a few "interface" proprietor processes. Unfortunately, this approach could be painfully inefficient.

The second approach classifies devices into three types (timing, character, and block) and provides simple yet convenient operations for manipulating each class of devices. This would allow the system processes to be more portable and would also be more efficient than the first approach. It would however make the kernel larger and device- and configuration-dependent. Insufficient experience with these approaches has been obtained to determine which is preferable.

3.5 Chapter Summary

The message-based interprocess communication mechanism was selected because: it is simple and efficient; it allows proprietor processes to be independent of their requestors; and the operations can be indivisible. The other communication mechanisms considered (except a data move operation) were considered too high level to include in the kernel.

The process management operations allow the kernel to be small but still independent of the system processes. Because of

the power of these operations, a protection scheme of capabilities and susceptibilities for these operations is necessary to prevent cyclic dependencies among the system processes.

The design of the device abstraction is complicated by conflicting design criteria. General purpose device operations reduce the device-dependency and complexity of the kernel as well as reduce the number of kernel operations. Higher-level, special device operations provide greater efficiency and simplify the associated device handling processes.

This chapter has attempted to justify the design of the kernel in terms of the design criteria. Empirical support for the kernel is provided by its implementation, which is described in the next chapter.

Chapter 4 Implementation of the Kernel

4.1 Overview

The implementation of the kernel consists of a collection of function and data modules written in Zed [Cheriton & Steeves, 1979d] plus a few TI990/10 assembler modules. The major data structures include process descriptors (PD's), device descriptors (DDs), and ready queues. The kernel functions that act upon these data structures can be classified into six types:

1. operation functions that correspond one-to-one with the kernel operations;
2. interface functions that pass control from processes to the kernel;
3. interrupt functions that execute in response to interrupts;
4. support functions that provide general low-level services such as list manipulation, and process state changes;
5. initialization functions that set up the kernel data structures; and,
6. debugger functions.

Figure 2 depicts the basic structure of the kernel functions.

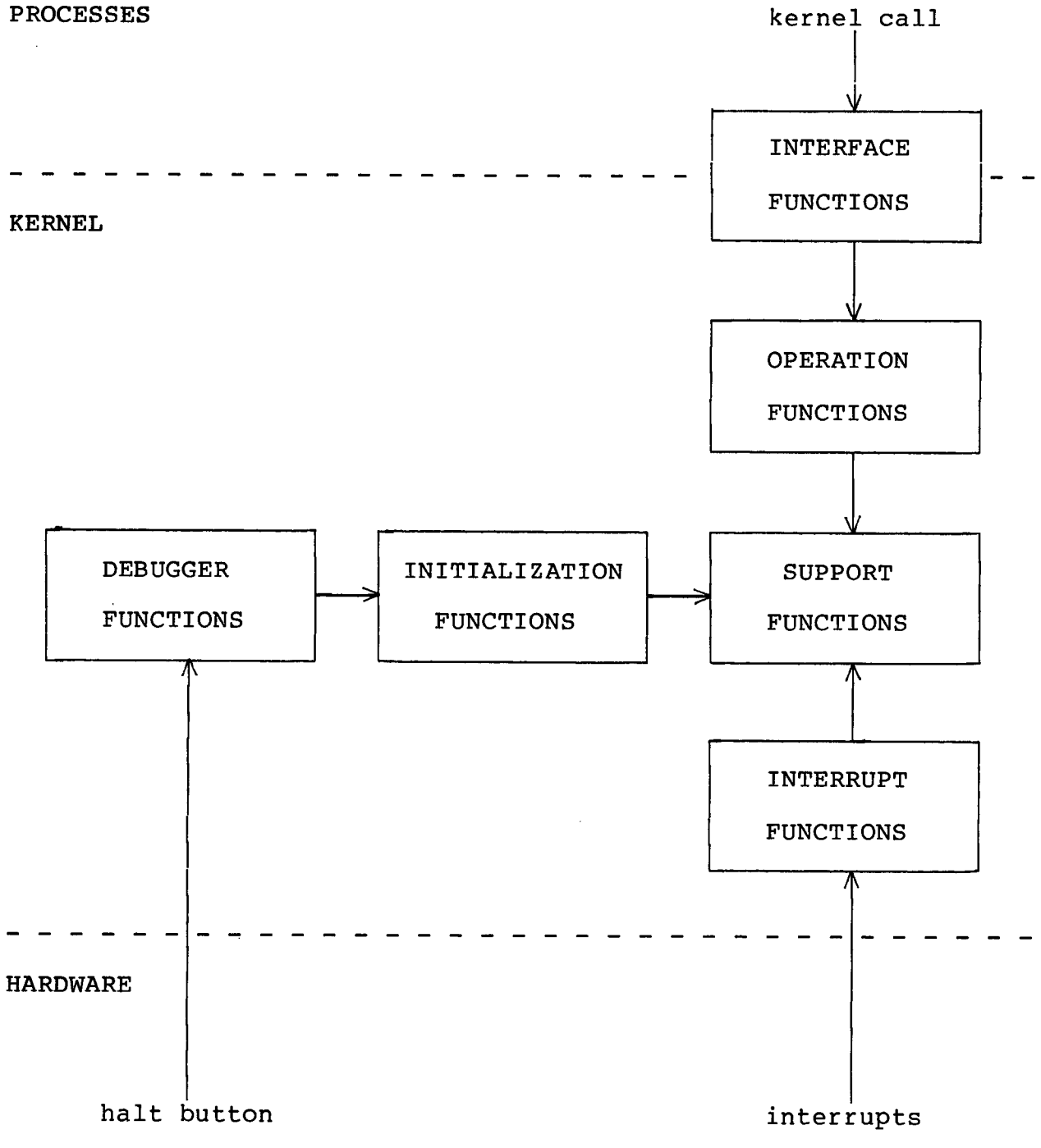


FIGURE 2: THE BASIC STRUCTURE OF THE KERNEL

The kernel is the only module that executes in privileged mode. Thus it has exclusive control over the memory mapping hardware, the devices, and the machine's privileged mode. There are three ways to enter the kernel: kernel calls, interrupts, and the processor's halt button. Processes make kernel calls, devices interrupt, and the front panel halt button transfers control to the ROM debugger/loader. The kernel returns control by resuming execution of the highest priority ready process. The process active when the kernel was entered is returned to unless a higher priority process was readied during the execution of the kernel, in which case it is dispatched.

The kernel executes each operation indivisibly and deterministically, because all interrupts are disabled while in kernel mode, with two exceptions. The power-up interrupt cannot be disabled on the TI990/10, but it should never occur while the system is operating. If there are no processes ready for execution, the kernel enters an idle state and turns on the interrupts. In this state however, it does nothing. It is not yet clear how this will complicate the verification of the kernel.

The kernel has been kept as small as reasonable. The current implementation contains 71 function modules and approximately 2500 lines of source code (of which about 20-30% are blank). Twenty four of the function modules are machine dependent, and only 8 of these are assembler language. The compiled kernel requires less the 8K bytes of code space and 1K + (62 * the number of processes) bytes of data space. The symbolic debugger

requires an additional 8K bytes. These numbers are only intended to give the reader some idea of the size of the kernel. Timing information, such as the maximum disable time and average execution times of important functions (i.e. Send, Receive, and Reply), is not currently available, however it is expected that these are slightly greater than Thoth's [Cheriton et. al., 1979c.

The remainder of this section provides a brief introduction to the Zed language and the TI990/10 processor. The following section describes the kernel data structures and functions, while the last section describes the multi-process test system that was implemented.

4.1.1 The Zed Language

Most of the kernel is written in a machine-oriented high-level language, Zed, which is similar to UNIX's language C. Both are descendants of the language BCPL. Actually Zed is intermediate-level because it supports only word-oriented data types, and has no I/O instructions. However, it does include the standard structured programming control constructs. Unstructured control (e.g. goto) was not used, despite the low-level nature of the kernel. The rather rich set of word operators includes: integer arithmetic, bitwise logical, pointer, comparison, boolean and assignment operators. Pointers are used extensively, but no other data structures are supported by Zed.

A textual substitution mechanism is used to parameterize most of the kernel's constants. Manifest constants, which are

syntactically similar to upper case identifiers, are defined to have string values, which are substituted into the source during the first phase of compilation.

In addition to defining constants, manifests have been used as prepositional comments. Thus manifests such as OF, IN, FOR, etc. are noise words ignored by the compiler. They give semantic case clues for the arguments to a function. For example,

```
.Move( n_words, FROM ptr1, TO ptr2 )
```

is equivalent to,

```
.Move( n_words, ptr1, ptr2 )
```

but hopefully is clearer. These manifest comments appear to be more helpful at the point of function invocation than the point of definition. They have been included primarily as an experiment in style.

Zed has an "escape to asssembly language" feature that was used in the kernel to execute special machine instructions for I/O and control purposes. This feature greatly reduces the amount of assembler language required, permitting the use of functions with Zed control structures and the full power of the machine. The effect of this would be less significant for machines, like the PDP-11, that do not have many special instructions.

A macro feature, similar to that of the C language, would significantly improve the efficiency of the implementation without affecting its understandability. This would be an extension of the manifest facility, that substituted arguments into the textual expansion of a manifest (macro). This would be

useful, as many functions are very simple, but make the code more readable.

4.1.2 The Host Processor

This subsection briefly describes the Texas Instruments [1976] TI990/10 processor that was used for the test implementation. It concludes with a discussion of the hardware features essential to the kernel design.

The TI990/10 processor is a byte-addressed 16-bit minicomputer. Its processor state consists of three registers: a status register, ST; a program counter, PC; and a workspace pointer, WP. WP points to a memory resident array of 16 general purpose registers. PC indicates the location of the next instruction. ST includes condition codes, a privileged mode flag, a map select flag, and an interrupt mask. This small processor state, with memory resident general purpose registers, results in fast context switching.

The processor executes in either privileged or non-privileged mode, as indicated by the mode select bit of the ST register. Privileged mode is required to:

1. execute the clock control (ckon, ckof), idle, long distance (lds, ldd), load interrupt mask (limi), load map file (lmf), and I/O reset (rset) instructions;
2. access devices connected to the upper portion of the Communications Register Unit, CRU, which includes all of the "internal" devices; and
3. change the mode, map, and interrupt mask in the ST register.

Once non-privileged, the mode can only be changed by an interrupt, trap, or the front panel halt button.

Device interfaces are of two types. Slow devices are connected via the Communications Register Unit, CRU, which can be thought of as a 4096 bit register, with specific bits assigned to each device. Special I/O instructions access this register, either one bit at a time, or in groups of up to 16 bits. The upper portion of the CRU (bit addresses >=$E00) can only be accessed in privileged mode, thus device access can be restricted.

Fast devices, particularly those employing Direct Memory Access, DMA, are connected to the main memory bus and are accessed by referencing a specific group of addresses called the Peripheral Control Space, PCS. Access to the PCS can only be restricted while the processor is using mapl. These fast devices can access all of main memory (even the PCS) because their DMA is not mapped.

Devices interrupt the processor on one of 16 levels. On an interrupt, the PC and WP registers are loaded from a specific two word vector in low memory, the old machine state is saved in the new workspace, privileged mode and map0 are selected, and the interrupt mask is set to prevent interrupts from the interrupting and all lower levels.

The TI990/10 instruction set includes 16 trap instructions. These are similar to interrupts, in that they cause context switches as dictated by a vector in low memory, and select

privileged mode and map0.

The processor's 16-bit byte addresses are mapped into the 20-bit word addresses of the main memory bus. The mapping unit contains three sets of map registers: map0, mapl, and map2. Each map uses 6 words to specify the limit and base of each of three variable size segments. When the mapping feature is enabled, the processor maps all addresses with either map0 or mapl, as specified by the map select bit in the ST register. Map2 is not a general purpose map, but is used exclusively by the long distance instructions. Maps can be loaded by a single privileged instruction, lmf. When map0 is selected, high addresses are always mapped to the PCS and to a ROM which contains a simple debugger/loader. Map0 is selected by interrupts, traps, and the front panel halt button.

The hardware features essential to the kernel design are memory mapping and privileged mode (including traps). They are necessary for the kernel to guarantee its own integrity. It is the only software in the system that executes in privileged mode and map0. Thus it retains exclusive control of the mapping hardware, devices (including interrupts), and privileged mode (including traps).

Some other features, while not strictly necessary, do enhance the kernel's efficiency. These include the rapid context switching allowed by the memory resident registers and load map file instructions, and the improved real-time characteristics allowed by interrupt masking. The workspace architecture is also suitable for supporting stack oriented languages, such as Zed.

The kernel could however, be ported to machines without these features with no structural changes.

4.2 Description

This section describes the major kernel data structures, and each of the classes of kernel functions. It also includes a discussion of process synchronization states.

4.2.1 The Kernel Data Structures

The kernel data structures accessible to all kernel functions include: process descriptors, device descriptors, ready queues, .Active, the free PD list, the "To Be Aborted" list, and some miscellaneous others. Each of these is described below.

Process Descriptors

Much of the implementation of processes involves the process descriptors (PD's). These vectors are defined by the template .PROCESS_DESCRIPTOR, with the following fields:

1. .ID - a one-word identification code for the process. Each id maps to a specific process descriptor, via its index field. An id is only valid if it is equal to the .ID field of the PD it maps to. An id can be invalid for one of three reasons. The .ID field of the PD it maps to is ones-complemented, indicating the PD is currently unused. The cycle number is incorrect, indicating the id refers to a process that has been destroyed already or not yet created. Or, it maps to the first PD, indicating that no PD is allocated for that id. The latter

mechanism has not been implemented, and would require some modification of the initialization of PD's. It would however, permit the allocation of space for a non-integral power of two number of PD's.

2. .PRIORITY - an integer from 0 to the number of eligible priority levels plus one, that determines which ready queue the process enters when it is readied. Thus the priority of a process determines when it is dispatched.

3. .STATE - a word indicating the synchronization state of the process. Every process is in one of the states:
.READY - ready for execution, i.e. not blocked,
.SENDING - blocked awaiting its message to be received,
.AWAIT_REPLY - blocked awaiting a reply,
.AWAIT_MSG - blocked awaiting a message to be sent to it, or
.AWAIT_EVENT - blocked awaiting a device event.
The low order bits of the .STATE field contain an index which indicates the process currently being (or last) synchronized with. See the next subsection for a discussion of the synchronization state transitions.

4. .F, and

5. .B - pointers that doubly link the process into a list or queue. .F points to the PD of the process closer to the tail of the list (forward), while .B points to the PD of the process closer to the head (backward). Each process is in at most one list as determined by its state. Pointers are not doubly-linked to facilitate bidirectional searching, but rather so that the

time to remove known elements from a list is bounded by a constant.

6. .VIOLATION - a word containing a numeric code for the exception which caused the process to be arrested. Exceptions are indicated in this separate field rather than in the message buffer so that no information is lost from the buffer.

7. .STATUS - a two-word subvector representing the capabilities and susceptibilities of the process. Each bit of the first word represents the capability to execute a restricted operation or the susceptibility to a restricted operation. Each bit of the second word represents the capability to access a device. The size of this field can be adjusted to accommodate the number of restricted operations and devices in the system.

8. .MESSAGE_POINTER - a pointer to the 8-word message buffer of the process, which is exchanged by the communication operations. This indirection eliminates the need to move messages between kernel buffers.

9. .INITIAL_MESSAGE_BUFFER - an 8-word subvector, which is the initial message buffer of this process. Because message buffers are exchanged however, this is usually not the buffer of this process. Message buffer space is allocated in this way because it is easier to initialize and it eliminates a vector of message buffer pointers that would only be used during the initialization procedure.

10. .SENDERS - a list header, or two-word subvector consisting of pointers to the head and tail of a doubly-linked

list of processes currently blocked waiting to send to this process. This list facilitates implementation of unspecific receives and process destruction.

11. .AWAITERS - a list of the processes currently awaiting either a reply or message from this process. This list facilitates immediate and complete process destruction. That is, it allows all of the awaiters of destroyed process to be handled as a group.

12. .MACHINE_STATE - a subvector sufficient to store the processor state (i.e. registers) of the process. This information is loaded into the processor whenever this process is dispatched. When a process relinquishes the processor or is preempted, the registers are saved here.

13. .MAP - a subvector specifying the address space of the process. The size and format of maps is machine dependent. For the TI they are six words, specifying the limit and base for each of three segments. The map of a process is loaded into the mapping hardware whenever it is dispatched.

Because of the potentially large number of processes (256-1024), the size of process descriptors (30 words including message buffers) has been kept to a minimum.

Device Descriptors

Device descriptors contain the information the kernel requires about each device. When a device is added to the system, an entry describing it must be added to the kernel.

Device codes map to device descriptors via an index vector, .Pointer_to_dd. The template .DEVICE_DESCRIPTOR defines the following fields:

1. .DEVICE_CODE - a positive integer, less than the number of devices. Device codes are used by processes to refer to devices, and by the kernel to determine the device descriptor. The manifests defining the device codes are included in both the kernel and standard process environments.
2. .ABILITY_FOR - a mask indicating the status bit corresponding to the capability for the device
3. .INTERFACE_TYPE - determines the method used to access the device's control registers. For the TI, this field indicates if the device is a CRU or PCS device.
4. .INTERFACE_ADDRESS - indicates the specific location of this device's registers (i.e. the CRU base address, or the PCS address).
5. .HANDLER - indicates the process (if any) awaiting an event from the device.

There are also additional fields that are used for specific device types.

Ready Queues

For each priority level there is an entry in the .Ready_queue vector, that points to a list of ready processes with that priority. This structure is searched by .Block whenever a process blocks, which happens when a process sends a message, awaits a message, or awaits an event. The penultimate ready

queue never has any processes in it, rather it is initialized so as to stop the search in .Block before it gets to the ineligible level. Thus processes are never dispatched from the ineligible level. This never-empty-level trick moves the completion test out of the search loop in the .Block function.

.Active

.Active is a pointer to the PD of the active process. It is an implicit argument to many of the kernel operations. For example, .Send_msg(id) sends a message from the active process to the process specified by id. It is also used to check capabilities for restricted operations, and to indicate where to save the processor's registers when a process is preempted, and loses the processor. .Active is actually an extension of the processor state, as it must be changed whenever a process is dispatched. When the kernel is in the idle state (i.e. there are no processes to dispatch), it points to the first PD, which is otherwise unused. This last trick, allows the .Dispatch routine to always save the machine registers, even if there is no active process, which saves a test.

Free PD List

The free PD list contains all unallocated PD's. Initially all PD's, except the first process's, are on this list. When a process is created it is allocated a PD from this list, and when it is destroyed its PD is returned to this list. Currently, free PD's are reallocated on a "first come first served" basis. If the recycling of id's was of concern, this list could be ordered

by cycle number, thus stalling the recycling.

"To Be Aborted" List

When a process is destroyed, the processes blocked on it are added to this list. They are removed from it by the .Next_violator operation. All processes in this list are in one of the states: .SENDING, .AWAIT_REPLY, or .AWAIT_MSG.

Miscellaneous

.Kernel_entries is a table of kernel entry points used by the interface functions. The kernel has its own stack which is distinct from the stacks of all processes. .Interrupt_mask is a table used to associate priority levels with interrupt masks. When a process is active, its priority level determines the processor interrupt mask. Other externals used as part of the device implementation include: the kernel-maintained time vector and timer count, terminal output buffers, a spurious interrupt count, and a copy of the front panel display register. The number of 1024-byte blocks of main memory is maintained in .Mem_size.

4.2.2 The Kernel Functions

As depicted in figure 2, there are six groups of kernel function modules: the operation, interface, interrupt, initialization, support, and debugger functions. Each of these groups is briefly described.

The Operation Functions

Each operation function implements the logic of one kernel operation. As with the operations, these functions classify into interprocess communication, process management, and device abstraction groups. Interprocess communication is based upon the synchronization states described in section 4.2.3. Process management involves accessing data in the PD's, and manipulating process lists. The device abstraction functions are somewhat more complicated, requiring interaction with the interrupt functions. The character-oriented functions in particular, share buffers with the interrupt functions. The operation functions only call support functions.

The Interface Functions

The interface functions pass control from the invoking process to the desired operation function. There are three groups of interface functions. The cosmetic functions provide useful combinations of kernel operations. They are also used to insulate processes from changes in the operations. The dummy kernel functions are used to setup the arguments to the kernel operations and execute the trap instructions that pass control to the kernel. Both of these groups of functions actually reside in each process's address space. That is, they are linked in with the code for each process. Thus they are not really part of the kernel at all. They are included however, because they are very closely related to it. The third group, the trap functions are in the kernel's address space. When a trap instruction is executed, the processor passes control to a trap function through the trap (or xop) vector. It then moves the arguments

from the process's stack to the kernel's stack and branches through the kernel entry table to the appropriate operation function. Currently only one of the TI990/10's 16 traps is used because this is the most portable design. The overhead for common operations could be reduced however, if they used individual traps.

The Interrupt Functions

The interrupt functions include: an assembler module (.Interrupt_routines) that specifies the interrupt vectors; the routines that are invoked on each interrupt; and support routines that are specific to interrupt handling. The .Interrupt_routines module turns off the interrupts completely, and transfers control to the function for the specific interrupt. Whenever a new interrupt is added, the vector must be changed. Because the .Interrupt_routines module turns off interrupts, the other functions can be written in Zed, and some are, but often assembler is more reasonable because of the need to use special machine instructions. The function for each interrupt depends greatly on the device that causes the interrupt. That is, these functions are very device- (and interface-) dependent. The clock interrupt function (.RTC_interrupt) keeps a count of the interrupts in the kernel's three-word time vector (.Time_vec), maintains the timer (.Timer) and dispatches the process awaiting the timer when it expires. The character device interrupt functions are by far the most complicated. They have to determine the nature of the interrupt (read request, write request, or other), move the character

between the interface and the kernel's device buffers, and dispatch awaiting processes under the correct conditions. The functions that respond to internal interrupts, which occur as a result of errors or power failures, are also highly machine dependent.

When a new device is added to the system the interrupt vector has to be changed. If a new device requires a special interrupt function, it has to be written and added to the kernel. To date, special interrupt functions have been written for: the real time clock, and two types of terminal interfaces (EIA and CIM).

The Support Functions

The support functions include common kernel data structure operations such as: manipulating doubly linked lists, changing the state of a process, dispatching a process, etc. These are often used to make the calling functions clearer. The efficiency of the kernel could be increased by incorporating some of these functions directly into the callers.

The Initialization Functions

The initialization functions are used to setup the kernel data structures, and create the first process. They assume that compiler initializations are correct. This is reasonable for the current situation where the kernel is reloaded from disk before it is started. If however, the kernel was resident in ROM and did not have to be reloaded, the compiler initialization of changeable variables could not be trusted. Thus, if the kernel was put into ROM, the initialization functions would have to be

modified. This would not require major restructuring, however.

The Debugger Functions

There are two different sets of debugger functions. The first is the ROM debugger/loader, which is used to load the kernel and provides absolute debugging facilities similar to those provided by most minicomputer front panels. These are considered part of the kernel because they execute in privileged mode, use map0, and the rest of the kernel depends on them, at least to get going. They are really quite separate as they are not linked or loaded with the rest of the kernel. The second type of debugger functions constitute a symbolic debugger that may optionally be loaded with the kernel. This debugger is useful for kernel development.

4.2.3 Synchronization States

The synchronization state of a process indicates if it is ready to execute, or blocked waiting to synchronize with another process or a device. Every process is in one of the five states:

1. .READY - ready for execution, i.e. not blocked;
2. .SENDING - blocked awaiting its message to be received;
3. .AWAIT_REPLY - blocked awaiting a reply;
4. .AWAIT_MSG - blocked awaiting a message to be sent to it;
5. .AWAIT_EVENT - blocked awaiting a device to signal an event.

Process synchronization is based upon these states. That is, the communication operations are implemented by causing changes in the synchronization states of the processes involved. Figure 3 shows the state transitions and indicates which operations cause

them.

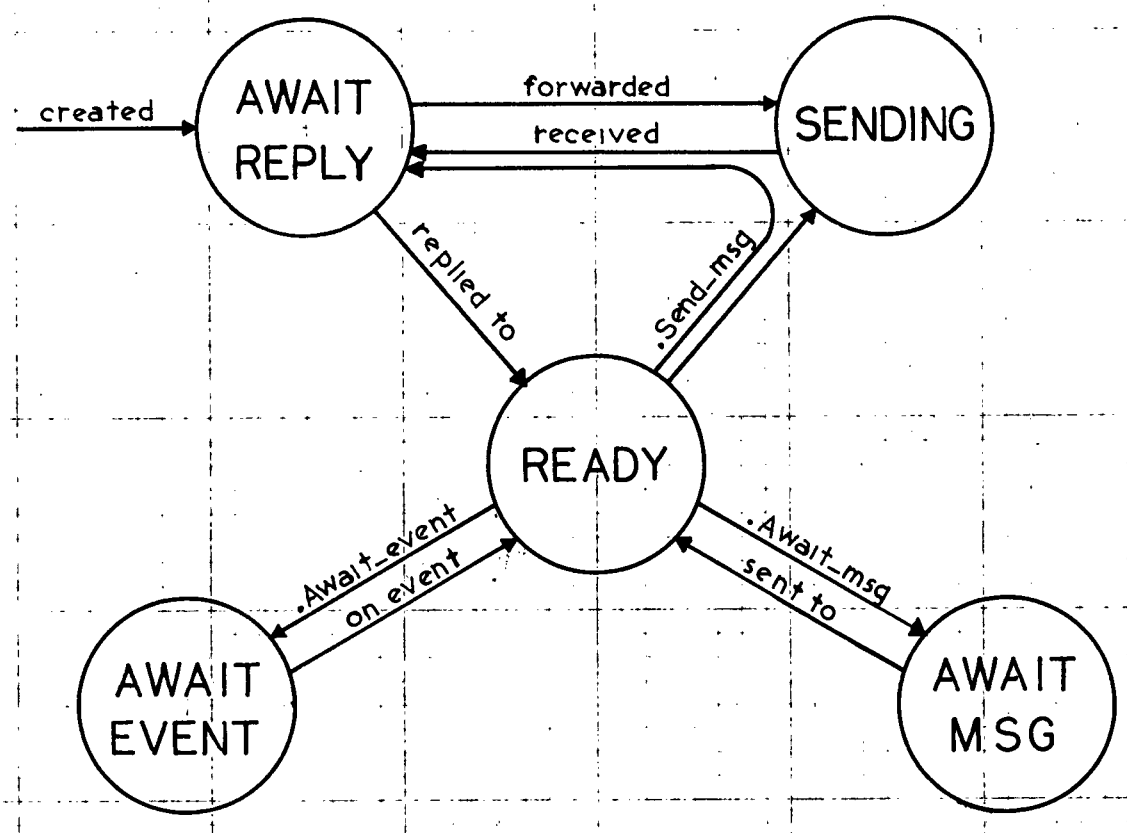


FIGURE 3: SYNCHRONIZATION STATE DIAGRAM

Each process can be in one list of processes, as dictated by its synchronization state. These lists, and the conditions for a process to be in the list are:

1. .Ready_queue - corresponding to the process's .PRIORITY, if in the .READY state.
2. .SENDERS - list associated with the process being sent to, if in the .SENDING state.
3. .AWAITERS - of the process being awaited, if in the .AWAIT_REPLY or .AWAIT_MSG states. (A process awaiting any

message is in the .AWAITERS list of the first pd.)

4. No list - if in the .AWAIT_EVENT state.
5. .Free_pd_list - if the pd is not being used.
6. .To_be_arrested_list - if the process it was blocked on has been destroyed.

The state of the process is irrelevant to membership in either of these last two lists. The support function, .Set_state, not only changes a process's state, but also its list membership.

4.3 Test System

Several simple systems have been written to test the kernel. The most comprehensive includes four processes residing in three distinct address spaces.

The root process, created by the kernel, creates and initializes the other processes. It is driven by the .Resident_process_table in its address space. All of the processes it creates are assumed to be in memory when it gets control.

The clock proprietor and its event notifier share an address space. These two processes provide the complete timing facilities for a general purpose system such as Thoth.

The user_agent is a dummy process, in that it only gives the "user" the ability to send messages. It does not provide a nice user interface. This facility is useful to test the message-passing operations and to send requests to proprietor processes.

The test system is deficient in a number of areas. The most prominent of these is that there are no facilities for using the disk drives. In fact, the kernel operations for disk I/O have not even been written. The other device operations have undergone insufficient testing. The test system does not include any facilities for creating processes, or memory management, although these operations are used by the root process. Process destruction has also not been tested.

4.4 Summary

The implementation of the kernel consists of a set of function and data modules, most of which were written in the intermediate-level language Zed, which is similar to the C language. Only structured control constructs were used, despite the low-level nature of the kernel. Zed's "escape to assembler language" feature significantly reduced the number of assembler modules required. A macro facility, similar to C's, would have improved efficiency without loss of perspicuity.

A Texas Instruments TI990/10 16-bit minicomputer was used for the test implementation. Its memory mapping hardware and privileged mode (including trap mechanism) were essential for the kernel to guarantee its own integrity. The processor's workspace register architecture, with memory resident general registers, allowed rapid context switching which, although not essential, did improve the efficiency of the design. Single instructions for loading address maps also contributed to the rapid context switching.

The major kernel data structures include: process descriptors, which represent the attributes of (and other information about) processes; device descriptors, which contain the information about the configuration of devices necessary for the kernel to access the devices; and, the ready queues, which are lists of ready processes (one list per priority level).

The kernel functions classify into the following six groups: the operation functions, which correspond on a one-to-one basis with the kernel operations; the interface functions, which pass control from processes to the operation functions; the interrupt functions, which are invoked in response to hardware interrupts; the support functions, which provide low-level operations (such as list manipulation) to the other kernel functions; the initialization functions, which initialize the kernel data structures and the first process; and the debugger functions, which include a low-level ROM-resident debugger/loader and a symbolic debugger which may be optionally linked into the kernel.

The implementation of interprocess communication is based upon the synchronization states of processes. Each process is in one of the following five states:

.READY, if it is ready for execution, i.e. not blocked;

.SENDING, if it is blocked waiting for its message to be received;

.AWAIT_REPLY, if it is blocked awaiting a reply;

.AWAIT_MSG, if it is awaiting a message to be sent to it; or,

.AWAIT_EVENT, if it is blocked awaiting a device event.

The possible transitions, and their causes, are shown in the synchronization state diagram, figure 3. These states also indicate which process list a process is in. For example, processes in the ready state are in one of the ready queues.

A complete operating system has not yet been implemented, but a test system has been developed for experimentation. This system consists of four processes residing in three different address spaces. It has been used to test the interprocess communication and most of the process management operations. Process destruction has not been tested. Some of the device operations have been tested, although only inadequately.

The kernel has been implemented to a degree sufficient to indicate the viability of many of the techniques, however it requires more work before it will be useable.

Chapter 5 Conclusions

This thesis has explored techniques for designing verifiable operating systems. It is based on the experience of designing and implementing a multiprogramming kernel intended to provide a foundation for message-based, multiprocess-structured operating systems. Particular emphasis has been placed on reducing the interdependencies of system components, in an effort to improve verifiability. To a significant degree, this effort has been successful.

The following section expands upon this conclusion and draws attention to significant aspects of this research. The chapter ends with a discussion of a number of areas that could benefit from further exploration.

5.1 Conclusions

Three unusual features of the kernel improve the verifiability of both the kernel and the systems it supports.

1. The system processes can be designed to exhibit acyclic dependencies, and hence be individually verifiable.
2. The kernel does not depend upon any processes for its correct operation, hence it can also be verified independently.
3. The kernel operations are indivisible, thus the kernel executes sequentially and deterministically, allowing the use of standard program proving techniques for its verification.

Two features of the environment were instrumental in allowing

systems to exhibit acyclic dependency graphs. Senders await replies, thus proprietor processes need not depend upon their requestors. Very few interprocess communication mechanisms have this property. Secondly, the capability/susceptibility protection scheme eliminates system process interdependencies which a simple "privileged-mode" scheme cannot. Individual capabilities for process management and device manipulation operations must be provided on a per-process basis to eliminate indirect dependencies. A new mechanism, which we call susceptibilities, was necessary to make it possible to prevent powerful system processes from disrupting each other.

Direct memory access (DMA) by devices is another potential source of interprocess dependencies. Because of this, the kernel monitors all DMA activity and locks the involved process into memory. This kind of dependency is not very well understood, and the kernel may have to be modified slightly to reduce its effect. This should be delayed until the memory management component of the system is designed, and the effects are more fully understood.

The process management operations are the key to the independence of the kernel. They allow processes to perform most of the process management functions, without the kernel losing its independence. Device manipulation operations were also necessary to ensure that devices cannot affect the kernel (e.g. via DMA).

Two features of the hardware necessary to achieve the independence of the kernel include: memory mapping hardware, to

protect address spaces; and privileged mode, including a trap mechanism, to permit controlled use of devices and the mapping hardware. These facilities are, however, available on any machine that supports a protected operating system. If privileged mode is the only way to restrict access to devices, the kernel must abstract all devices, because it is the only software that executes in privileged mode.

Careful consideration of the semantics of the kernel operations was required to ensure that they could be implemented to be indivisible with execution times bounded by a constant. Process destruction was the most difficult operation to do this for, as it involves an arbitrary number of processes on each destruction. Because of this, the kernel maintains lists of the processes blocked on each process. Although somewhat expensive, this allows process destruction to be indivisible. The separation of data movement and synchronization by the process communication operations is also motivated by the indivisibility criterion.

Some of the important conclusions from the justification in chapter 3 warrant repeating here.

The interprocess communication primitives were selected for a number of reasons. They are sufficient to solve the "mutual exclusion" problem. They are easy to understand and convenient to use. They allow the system to be structured with an acyclic dependency graph. And, they are efficient.

The restriction of the process creation and destruction

operations eliminates the necessity for the kernel to maintain a record of the creator and descendants of each process. Most similar kernels have maintained such a process hierarchy.

It is not necessary for the kernel to provide a high-level abstraction of memory. A memory manager process can abstract memory with the kernel operation for setting process address maps. Thus memory management by the kernel reduces to the relatively simple task of checking process maps as they are set, to ensure they do not overlap with the kernel.

The kernel's two processor allocation mechanisms provide both fast and flexible scheduling. The micro dispatching strategy, implemented entirely by the kernel, allows the time for determination of the process to dispatch to be bounded by a constant. The .Set_priority operation allows a scheduler process to implement any macro allocation strategy desired.

The arrest mechanism used by the kernel to handle exceptions allows the system considerable flexibility. Despite this, however, exception handling is a weak area of the kernel design. It does not seem to fit nicely with the rest of the design, nor are all exceptions handled consistently. This area deserves further investigation.

Device abstraction is a difficult and poorly understood area of machine and system design. We found it necessary to explore two approaches to it. The first is to provide general-purpose device operations in order to reduce the kernel's dependence upon device interfaces and configuration. These general-purpose

operations specify the effect upon the device's control registers, not the action of the device. The second approach is to provide special operations for each of three generic classes of devices (timing, character, and block). This is a much higher level of abstraction.

The kernel extends the hardware timing facilities from a simple periodic interrupt, to a time vector representing the current time and a settable timer. This allows a more efficient and elegant implementation of the timing facilities which the system provides. This is an example of a minor increase in kernel complexity significantly improving the efficiency and elegance of the systems it supports.

Character devices cause most of the device dependency problems the system designer faces. Interface details vary widely, even among devices from a single manufacturer. Systems tend to contain a number of these devices and they are reconfigured often. A signficant proportion of the processor time is spent servicing them, because they are heavily used. These aspects put conflicting pressures on the system designer. The variation in interface details (and even functionality) coupled with the desire for a stable kernel with simple semantics indicates that these details should be handled by processes. But, this is less efficient, which is significant because of heavy use. We have compromised. The kernel provides efficient, device-independent operations for reading and writing characters, but only general purpose operations for manipulating the more esoteric features of the devices. Like all compromises,

this one is not particularly satisfying.

The semantics of the block device operations have been specified, bearing in mind the dependency problems associated with direct memory access (DMA). To date they have been neither implemented nor tested, but there does not appear to be any problem doing so.

Much work remains to achieve the ultimate goal of a verified full-function operating system. However, the independent kernel/system-process approach seems sufficiently promising to explore further.

5.2 Suggestions for Future Research

Three immediate extensions to this research include: verification of the kernel; construction of a set of system processes which implement a complete operating system; and, verification of these system processes. These extensions would substantiate our claims that the kernel is verifiable and suitable for supporting verifiable operating systems. They would also be interesting experiments in their own right, as they would result in a complete verified operating system. In addition to these extensions, three areas of the kernel design warrant further investigation. These are exception handling, device abstraction, and memory management.

The issues of exception handling need to be considered from a more global point of view than we have adopted. What kinds of exceptions occur? What are reasonable actions for the system to

take when they do occur? Once these questions are answered, it will be more obvious what exception handling support the kernel should provide.

The kernel's abstraction of devices is ad hoc. The problem is to determine what level of device abstraction the kernel should provide, given verifiability, configurability, portability, and efficiency considerations. Improved hardware facilities that can be abstracted simply and easily would be interesting to explore. For example, mapping DMA and enforcing standard mechanisms for interrupt handling would help considerably.

The kernel's abstraction of memory is sufficient, but needs to be reworked in the context of a higher-level memory abstraction. This would lead to kernel facilities more suitable for the desired abstraction. In this way, it might even be possible for the kernel to provide a machine-independent abstraction of memory. This approach could also point to better designs for memory management hardware.

It would be interesting to explore to what extent the kernel could be micro-coded, and what efficiency gains could be made. Also of interest would be investigation of machine architectures that had built-in knowledge of multiple processes. That is, the machine could include special registers indicating the active process and/or important attributes of the active process.

A potentially attractive research direction, is the exploration of the kernel as a mechanism for coordinating multiple processors. The basic idea here is for multiple

"closely coupled" (same memory) processors to share the kernel with a hardware lock. That is, there would be only one copy of the kernel code and data structures which would be shared by all of the processors, only one of which could be in kernel mode at a time. In this way it might be possible for the kernel to be the only software aware of how many processors were in the system.

In conclusion, it seems that there are many interesting issues to pursue in the area of verifiable operating system design. The kernel/system-process approach still appears fruitful. There is much work to be done before we have systems that are proven to function correctly, never fail, and always do what is expected of them.

References

Baskett, F., Howard, J.H. and Montague, J.T. Task communication in DEMOS. Proc. 6th Symp. on Operating System Principles, A.C.M., New York, 1977.

Brinch Hansen, P. RC4000 Software Multiprogramming System. A/S Regnecentralen, Copenhagen, April 1969.

_____. The nucleus of a multiprogramming system. Comm. A.C.M. 13, 4 (April 1970).

Cheriton, D.R. Multi-process structuring and the Thoth operating system. Technical Report 79-5, University of British Columbia, Vancouver, 1979a. (based on Ph.D. thesis University of Waterloo, 1978).

_____. Designing an operating system to be verifiable. Technical Report 79-9, University of British Columbia, 1979b.

Cheriton, D.R., Malcolm, M.A., Melen, L.S. and Sager, G.R. Thoth, a portable real-time operating system. Comm. A.C.M. 22, 1 (Jan. 1979c).

Cheriton, D.R., and Steeves, P.J. The Zed reference manual. Technical Manual 79-2, University of British Columbia, 1979d.

Clifford, J., and Montague, J. DEMOS kernel functional description. Los Alamos Scientific Laboratory, 1978.

Dijkstra, E.W. Co-operating sequential processes. In Programming Languages, F. Genuys, Ed., Academic Press, New York, 1968a.

_____. The structure of the "THE" multiprogramming system. Comm. A.C.M. 11, 5 (May 1968b).

Lycklama, H., and Bayer, D.L. The MERT operating system. The Bell System Technical Journal, 57, 6 (July-August 1978).

Popek, G.J., and Kline, C.S. A verifiable protection system. Proc. 6th Symp. on Operating System Principles, A.C.M., New York, 1977.

Ritchie, D.M. and Thompson, K. The UNIX timesharing system. Comm. A.C.M. 17, 7 (July 1974).

Schroeder, M.D., Clark, D.D., and Saltzer, J.H. The Multics kernel design project. Proc. 6th Symp. on Operating System Principles, A.C.M., New York, 1977.

Texas Instruments Incorporated. 990 Computer Family Systems Handbook. No. 945250-9701, Austin, Texas 1976.