UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT

Multi-Process Structuring and the Thoth Operating System

David R. Cheriton

May 1979
CS-79-19

UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT

Multi-Process Structuring and the Thoth Operating System

David R. Cheriton

Department of Computer Science
University of Waterloo

May 1979

This research was suported by the National Science and Engineering Research Council of Canada

Multi-Process Structuring and the Thoth Operating System

David R. Cheriton

This thesis explores the idea of structuring programs as many concurrent processes. It is based on work done in designing, implementing, and using the Thoth operating system. Thoth implements an abstraction that provides facilities to make this type of structuring attractive, namely inexpensive processes, efficient interprocess communication, dynamic process creation and destruction, and groups of processes sharing a common address space. The abstraction is first described, including measurements of its performance and comments on its portability. Next, the abstraction is motivated by considering various design and implementation trade-offs. Then, the design of multi-process programs is discussed, both in terms of general principles, and by giving specific uses and examples. The examples are drawn from the operating system and the Thoth text editor. Finally, the feasibility of verifying the system is considered. This is motivated by the desire to exploit the multi-process structure of the system to aid in verification. We conclude that structuring programs as multiple processes can have significant benefits, especially for programs that respond to asynchronous events.

1 Introduction

Multi-process structuring is the use of several processes to structure programs. The term *process* is used to mean a sequence of actions that can logically execute concurrently with those of other processes. A program can be structured by using several processes and their pattern of interaction to form the frame or skeleton on which the rest of the program is built. The term *program* includes large programs such as operating systems, compilers or data base management systems, medium-sized programs such as a text editor, and small programs such as file utilities. It is the contention of this thesis that:

(a) Processes can be used with considerable benefit to structure programs provided these programs can be designed for and executed in an environment suitable for multi-process structuring.
(b) An environment suitable for multi-process structuring can be provided at a reasonable cost.

These claims are supported by consideration of what constitutes a suitable environment for multi-process structuring, consideration of the design and implementation of such an environment (as the abstraction implemented by an operating system), discussion how to structure programs using processes in this abstraction, and consideration of the benefits of this type of structuring.

The research reported here was performed in the course of developing and experimenting with an actual operating system which uses these ideas, namely, the Thoth portable real-time operating system. Considerable experience was gained in the use of

multi-process structuring while implementing this system. Further experience came from developing application programs such as a text editor and an inter-machine communication program. Measurements of system performance and extensive use of the system have provided empirical evidence to support the practicality of the implementation. Finally, the many iterations of the design and implementation process led to the discovery of several problems, the consideration of many alternative designs, and the development of new designs. What we learned and abstracted from this experience is used to support the concepts of multi-process structuring.

1.1 Motivations

The exploration of multi-process structuring was motivated by problems encountered in the design of programs that must respond to asynchronous events. For example, an operating system must respond to many different asynchronous events signalled by interrupts. Multi-process structures can be used to achieve the same level of asynchroneity as exhibited by the set of events to which the program must respond. That is, events are mapped onto processes so that each event is synchronous with respect to the process that responds to it. This approach suggests a discipline of programming in which a process only responds to events synchronized with its execution and the global asynchroneity of the total program is realized by the asynchronous execution of its multi-process structure. The synchroneity of the individual processes then makes the program more understandable.

Further motivation comes from the observation that some programs comprise several somewhat independent sequences of action even though they are not required to respond to asynchronous events. In such programs, it is attractive to make each such sequence of actions a separate process, rather than using conventional control structures to implement the sequences within one process. This avoids complicated control flow making the program simpler, easier to understand, and easier to modify. For example, a program requiring garbage collection might be structured so that the garbage collection is the task of a separate process (Wadler (1976)). This motivates the use of processes for structuring programs at a level comparable to the use of control structures and data structures in programming languages. A process in a program can also be viewed as analogous to a person in an organization. With these people-like processes, we can exploit our understanding of and familiarity with human organizations to structure programs. This anthropomorphic view of processes has been used in the work reported here but is not considered explicitly in this thesis.

Multiple processes can be used to make a program execute with real or logical parallelism. The decomposition of a program into processes also suggests the possibility of distributing these smaller units over several machines. This more conventional motivation for multiple processes is of interest; however, it is not the central motivation for multi-process structuring.

Finally, curiosity prompts an exploration of multi-process structuring to gain more understanding of its problems, implications and applications.

1.2 Brief Survey of Related Work

The concept of process evolved from the development of multi-programming; different programs running concurrently were considered to be executed by different processes. That is, one way to run K programs concurrently on a single processor machine is to simulate a K processor machine. Thus the process evolved as what might be called a "virtual processor". Each virtual processor was implemented as a set of time slices of the real processor and was thus similar to the real processor. The UNIX operating system (Ritchie and Thompson (1974)) is an example of a system providing many processes in this manner.

Brinch Hansen (1969) is credited with introducing the idea of message-passing for interprocess communication with the RC 4000 system. Dijkstra has been most prominent in articulating problems arising with concurrent or non-deterministic programs and also in providing insight into their solution. The semaphores and secretaries of Dijkstra (1972), the critical regions of Brinch Hansen (1970), and the monitors of Brinch Hansen and Hoare (1974) are examples of work on process synchronization. Jammel and Steiger (1977) discuss their manager concept in comparison to monitors. Their manager is related to our use of a process stereotype, called a proprietor, for process synchronization.

Few operating systems are structured using processes beyond the implementation of processes to execute user programs. The system described by Jammel and Steiger was structured with processes but we understand that it has not and will not be completed. The DEMOS system being developed for the Cray-1 (Baskett et al. (1977) and Powell (1977)) is being implemented using processes and message-passing.

Multi-process structures for various operating system mechanisms have been proposed in the literature although few have been implemented. Hoare (1973) gives a design of a paging mechanism using processes; however, the design is unrealistic because it requires two processes per page of information. A design by Huber (1976) for a demand paging system uses one process per memory level in the memory hierarchy. The thesis by Janson (1976) explores the use of type extension for structuring virtual memory mechanisms. The extended types are implemented by so-called type managers, which Janson points out (but does not pursue) could be implemented as individual processes. Janson's thesis deals extensively with a redesign of the Multics virtual memory mechanisms as an example of the application of type extension.

This thesis includes a study of the feasibility of verifying the Thoth system. Most previous work on verification is devoted to formally verifying rather small example programs to demonstrate a technique. The approach we have taken differs from this because the program is given. Our objective is to consider how to verify it using any techniques available. In this sense, it can be regarded as applying existing verification ideas to an operating system, rather than exploring verification in itself. Our discussion draws on basic aspects of sequential program verification as developed by Floyd and Hoare, in the sense that one strategy is to reduce the verification of the system to units or modules that are manageable under sequential program verification. This discussion does not exploit the full power and generality of mechanisms presented in the parallel program verification literature (Ashcroft (1973), Owicki (1975), Lipton (1975)) because the restricted type of process interaction in the Thoth system considerably simplifies its parallel behavior.

1.3 Thesis Overview

Our objective has been to explore the idea of multi-process structuring; the method used was to design and implement an abstraction for multi-process programs, and then experiment with this abstraction. This thesis describes, in part, what we did, why we did it that way, and what we learned.

Chapter 2 describes the design and implementation of the Thoth operating system. Its history, development, and current state is discussed, followed by a description of the abstraction implemented by the current system. We regard this abstraction as a significant contribution in itself, being a simple yet adequate, efficient-to-implement, and machine-independent abstraction that has evolved through considerable experience with earlier versions of this and other operating systems. It provides facilities that have been found necessary or useful for multi-process structuring. This description captures the system at its present stage of development; critical comments throughout the thesis show that it is subject to improvement. The temptation of confusing what should have been implemented with what has been implemented was resisted. The chapter includes a discussion of the portability of Thoth, describing the main portability problems addressed during the development of Thoth. For a more complete discussion, the reader is referred to Cheriton et al. (1977). Chapter 2 ends with a section on measurements of time, space and real-time performance characteristics of the system. These measurements are machine-dependent but give the reader a cost basis on which to judge the practicality of the system, and thus the ideas presented.

Chapter 3 presents design and implementation issues associated with implementing an abstraction for multi-process structuring with particular emphasis on motivating the abstraction described in Chapter 2. Issues are considered under the headings of processes, process identification, interprocess communication and dynamic configurability.

Chapter 4 discusses the design of multi-process structures and the applicability of these structures to problems of resource management, synchronization, exception handling and distributed computing. Examples are drawn from the operating system and various programs that run under the Thoth system.

Chapter 5 examines the internal structure of the operating system. It constitutes our largest and most interesting example of multi-process structuring. Several advantages of this structuring are presented and current inadequacies are exposed. The latter prompt including a number of remedies which might be implemented in the future.

Chapter 6 presents a study of the feasibility of verifying the operating system. This is structured by outlining a scheme of verifying the system, and pointing out difficulties and suggesting some remedies. The outline is an application of techniques we have developed for reasoning about multi-process programs, a demonstration of how the multi-process structure of the system makes it amenable to verification, and a treatment of many problems arising in the verification of a large, concurrent program.

The final chapter draws together conclusions from preceding chapters and suggests directions for further research based on multi-process structuring.

It is conventional for an overview to indicate the original results or contributions of a thesis. Unfortunately, what are viewed as contributions cannot be concisely stated. Sacrificing precision for brevity, the main contributions are considered to be:

1. the design, implementation, and motivation of the Thoth abstraction.
2. recognition of the value of discarding concurrency as the main motivation for multi-process structuring.
3. recognition and clarification of problems with message-passing, process identification, synchronization, and process destruction.
4. a design approach in which the process structure forms the skeleton of a program from which a structure or an organization on functions and resources is derived.
5. the particular multi-process structures of Chapters 4 and 5 as programming techniques.
6. the technique used to decompose the verification of Thoth and the treatment of the basic abstraction, in particular the message-passing.

We also consider the implementation and testing of many of the ideas presented here as contributing to the evaluation of the practicality of these ideas.

The presentation that follows warrants a few general remarks. It is advantageous to remain faithful to the current design and implementation of Thoth in our discussion so that remarks are supported by implementation, testing, and experience. It is also advantageous to include how we now believe the system should have been done, drawing on the benefit of hindsight and experience. It is equally advantageous to abstract the discussion with a particular system to provide wider applicability of our conclusions. All three of these competing goals govern this thesis; we trust that the reader will recognize the different tacks in the course of the discussion.

2 Description of Thoth

A description of the operating system is provided for several reasons. First, examples used in the thesis are drawn from the operating system and various application programs running under Thoth so the reader should be familiar with the system. Second, because the system has been used as a research vehicle for testing ideas, a description of the functionality and performance of the system provides the reader with a basis on which to judge the practicality of these ideas. Third, design of the abstraction for an operating system is an important and a difficult aspect of developing the system. The abstraction described here has evolved from ideas in previous operating systems, and from versions and revisions of our own ideas; it is thus interesting in its own right.

What follows is a synopsis of the history and current state of Thoth, a description of the abstraction implemented by the system, comments on the portability of the system, and the results of measurements of system performance.

2.1 History and Current State

Thoth has evolved through several generations. It was first implemented by Lawrie Melen and Michael Malcolm as a small run-time executive in May 1976, running a single user or stand-alone system on a Data General NOVA; it was ported to a Texas instruments 990 by Melen in August 1976. System development was done using a compiler and other support software running on a Honeywell 6050, much of which was developed by Michael Malcolm, Reinaldo Braga, Gary Stafford, Fred Young, Morven Gentleman and Gary Sager. Melen (1976) reported some of this work in his master's thesis.

The author started working on the system in August 1976. This work included rewriting the run-time executive to form the kernel of the present system, designing and implementing a disc-based file system (operational in January 1977), and extending the system to use memory mapping hardware and swapping (August 1977). This work was done jointly with Michael Malcolm, and benefited greatly from input from other members of the Portability Group. The file system was rewritten by Alfredo Piquer in the spring of 1977. Bert Bonkowski has been a major worker on the compilers for the base language since September 1976. The Thoth text editor was originally written by Patricio Poblete in the summer of 1977. The author rewrote it early in 1978 to improve the multi-process structure, improve its performance, and to allow its use as an example in this thesis.

Currently, Thoth supports multiple processes, dynamic memory allocation, device-independent input/output, a tree-structured file system, multiple terminals, and swapping. All of the system software and documentation is now maintained on a TI 990/10 Thoth system. This system is used to support porting of the system to new machines and ongoing development.

As mentionned above, the system is running on the Texas Instruments 990 and the Data General NOVA, two minicomputers with quite different architectures. Thoth is currently being ported to the Honeywell Level 6 minicomputer.

A range of system configurations is possible. The facilities described above are those of a larger configuration; a smaller system can be configured with only multiple processes, dynamic memory allocation and interprocess communication.

2.2 The Abstraction

Thoth implements a program environment referred to as the *Thoth abstraction.* This abstraction is implemented as a base language and a set of system functions implemented in this language.

The base language, described by Braga (1976), is a stack-oriented language derived from B (Johnson and Kernighan (1973)), which is itself a descendant of BCPL (Richards (1969)). Programs in our base language, as in BCPL, are written as a set of functions and data modules which have global scope. Variables local to a particular function, including its parameters, are dynamically allocated on a stack so that functions can be reentrant. A function is invoked as either a subroutine or as a

separate process. When invoked as a subroutine, the function uses the caller's stack; when invoked as a process, a new stack is allocated separate from that of the invoking process.

The language includes statements for enabling and disabling hardware interrupts to provide indivisible execution of sections of code. There is also a *twit* statement which provides a way of inserting assembly language directly into the code. This makes it possible to use special I/O instructions, and makes the presence of such machine-dependent code obvious. Twit statements are used almost exclusively for implementing primitive functions in the operating system, and are seldom necessary (or desirable) in user programs.

A program comprises a tree of processes which interface with the Thoth abstraction via the base language and the invocation of the system functions described below. This tree of "user processes" is a subtree of the tree of system processes.

The system functions forming the rest of the abstraction are invoked as subroutines in the base language. The system can be configured as a run-time executive so that the system functions are linked and loaded with the user program into one executable core image. The system can also be configured so that system function calls in a user program are traps which are dynamically linked to system functions. We have used the convention of beginning all global system names with a "." so the user can avoid inadvertently replacing a system function or data module.

The remainder of this section describes the part of the abstraction available as system function calls.

Memory allocation: A contiguous array or vector of memory is allocated by the function call

```
vec = .Alloc_vec( size )
```

This returns a pointer to a vector of size+1 words, which can be indexed as vec[0] through vec[size]. The vector is returned to the free list by

```
.Free( vec )
```

Process creation: A process is created with a specified stack size by

```
id = .Create( funct, stack_size )
```

where funct is a pointer to the function to be invoked as a process. A unique nonzero process id is returned which is used in future references to the new process. The created process becomes a direct descendant of its creator in the tree of processes. An optional third argument can be passed to .Create to specify the priority of the new process; the default priority level is 0. The process is created in the *embryonic* state and cannot execute until it is readied:

```
.Ready( id, arg1, arg2, ... )
```

.Ready passes its (optional) arguments to the new process and makes it eligible for execution.

CPU allocation: A process eligible for execution is said to be *ready*. A process that

is not ready is said to be *blocked*. Of the highest priority ready processes, the process that has been ready for the longest time is allocated the CPU, and is said to be *active*.

The active process relinquishes the CPU either by blocking or by being preempted when a higher priority process becomes ready. The latter is caused either by a hardware interrupt or by an action of the active process. The active process may block by attempting to communicate with another process or by waiting for an interrupt to occur. Certain system functions, such as input and output primitives, use interprocess communication and may block the active process.

It follows from the CPU allocation rule that on a single processor machine, a process executes indivisibly with respect to processes of the same or lower priority until it blocks. This *relative indivisibility* is a useful property of Thoth processes.

The *highest* user priority is 0; lower priorities are greater than 0. The root of a subtree of user processes normally has priority 0. Thoth system processes have higher priorities than user processes.

Interprocess communication: There are four primitives for passing messages between processes. All messages are 8 words in length.

A process sends a message to another process by

```
id = .Send( msg, id )
```

The contents of the 8-word msg vector is sent to the process specified by id. The sending process blocks until the receiving process has both received the message and sent back an 8-word reply using .Reply. The reply message overwrites the original msg vector.

If the receiving process does not exist, .Send returns 0 and the msg vector remains unchanged. Otherwise .Send returns the id of the process that sent the reply.

The receiving process uses

```
id = .Receive( msg )
```

or

```
id1 = .Receive( msg, id2 )
```

The receiving process blocks, if necessary, to receive an 8-word message in its msg vector. When the optional id2 parameter is present, the message must come from the specified process. When the id2 parameter is not present, the first process sending to the receiving process will satisfy the receive. In the following, we refer to the former case as "receive-specific" and the latter case as "receive-any". The id of the sending process is returned for later use in

```
.Reply( msg, id )
```

The 8-word reply contained in the msg vector is sent to the specified process awaiting a reply from the receiving process. The sending process is readied upon receiving the reply, and the replying process does not block.

An attempt to receive from a non-existent process results in an undefined message and a 0 being returned by .Receive. A .Reply to a non-existent process is a null operation.

Instead of replying to a sender, the receiving process can forward the message, possibly changing its contents, to another process (or even itself):

.Forward(msg, from_id, to_id)

The process specified by from_id must be blocked awaiting a reply from the forwarding process. The effect of .Forward is the same as if the from_id process had performed a .Send to the process specified by to_id of the 8-word message in the forwarding process's msg vector. The forwarding process does not block. A process receiving a message can determine the forwarder of the message by

id2 = .Forwarder(id1)

where id1 specifies the sending process; id2 is equal to id1 if the process was not forwarded.

These primitives provide a facility for short messages. A process can transfer larger amounts of data between two processes using

.Transfer(dest_id, src_id, dest, src, size)

which transfers size+1 bytes from src in the address space of src_id to dest in the address space on dest_id. Both src_id and dest_id must specify a process that is awaiting reply from the invoking process or else is the invoking process.

Interrupts: Interrupts are serviced by system processes. These processes invoke

.Await_interrupt(device_id)

to block until an interrupt occurs for the specified device. Such an interrupt can only occur when no processes of the same or higher priority are active. Hence, an interrupt causes its associated process to become both ready and active.

Process destruction: Any process can destroy a process (possibly itself) by

.Destroy(id)

The destroyed process ceases to exist in the sense that its id becomes invalid, it can no longer execute, and its stack and all memory it has allocated are returned to the free list. When a process is destroyed, all of its descendants are also destroyed.

For any process blocked doing a .Send to or .Receive from a process that is destroyed, the result is the same as if the .Send or .Receive had been executed for a non-existent process. In particular, when a process is destroyed, all blocked processes attempting to send to or receive from the deceased process become ready.

Teams: Each process belongs to a *team* which is a set of processes sharing a common address space and a common free list of memory resources. There are two types of teams: resident and transient. Processes on *resident* teams remain in memory and are higher priority than those on transient teams; *transient* teams may be swapped to secondary storage. Transient teams can be created dynamically while resident teams

are created only when the system is initialized. The priority of a process on a resident team is fixed throughout its lifetime; the priority of a process on a transient team is fixed relative to other processes on that team but changes with respect to processes on other teams. Hence, the relative indivisibility property of a process on a transient team applies only with respect to other processes on the same team.

Teams allow the use of physical memories larger than the logical address space on machines with memory management hardware, and greater concurrency via swapping on machines with a secondary storage device suitable for swapping.

Clock: The system can be configured to support a clock abstraction on machines with a source of periodic interrupts. The clock abstraction supports

.Get_time(date_and_time)

which returns the current date and time,

.Set_time(date_and_time)

which sets the current date and time plus

.Sleep(date_and_time)

and

.Delay(seconds)

A process invoking .Sleep will block until the current time and date becomes equal to or later than that specified by the date_and_time vector. A process invoking .Delay will block until the specified number of seconds has elapsed. Using an optional second argument to .Delay, a process can block for as little as one clock interrupt (which is machine dependent). In both cases, the process then becomes ready.

.Wakeup(id)

unblocks the process specified by id if it is sleeping or delaying.

Input/output: The input/output system provides a reasonably uniform interface with peripheral devices and files so they can be used interchangeably by most programs. Each device is assigned a unique name. e.g. terminals are named "$tty0", "$tty1", ..., the discs are named "$disc0", ..., and so on. The random-access memory can be treated as an input/output device, called "$mem". This allows the use of input/output editing functions on strings of characters stored in memory.

A file or device is accessed by

fcb = .Open(pathname, mode)

where pathname is a string specifying either the name of a device or the pathname of a file (which will be defined later) and mode is a string specifying the mode of access (read, write, append or read/write). Append mode is equivalent to write mode on devices and is defined further for files in the next section. In the remainder of this section, we will use the word *file* to mean "file or device".

The function .Open returns a pointer to a *file control block* which contains a description of the accessed file and any necessary buffer(s); this pointer serves as an

identifier for the accessed file. The process is said to *own* the fcb and no other process can use it although other processes can still access the file using separate fcbs.

Each open file has a *current byte position.* When a file is opened, this current byte position is set to the beginning of the file except in append mode where it is set to the end of the file.

An fcb can be used to transfer data to or from a file after it is *selected.* An fcb is *selected for input* by

.Select_input(fcb)

An fcb is *selected for output* by:

.Select_output(fcb)

These two functions verify the fcb ownership and access mode. Only one fcb can be selected for input at a time, similarly for output.

Data is transferred one byte at a time from a file selected for input by

data = .Get()

.Get returns the current byte right-adjusted and zero-padded on the left. Similarly, data is transferred to a file selected for output by

.Put(data)

.Put writes the right-most byte of the data word to the current byte in the file. Both .Put and .Get have the effect of incrementing the current byte position.

These data transfers are implemented with device-dependent buffering schemes. To guarantee that all bytes transferred by .Put have actually been output to the file

.Flush()

ensures that data output to the file selected for output has been output to the actual file device.

.Seek(fcb, where, how)

changes the current byte position. The "how" parameter specifies the interpretation of the "where" parameter. The three possibilities are: *absolute byte* which sets the current byte position to the where-th byte in the file; *relative byte* which increments the current byte position by where, which may be negative; *absolute block* which sets the current byte position to the first byte in the where-th block of the file. Absolute block seeking applies only to devices and files on devices for which the concept of a block is appropriate.

.Close(fcb)

flushes output (if necessary) and removes access to the file. When a process is destroyed, all of its accessed files are automatically closed.

File system: The system can be configured to support a file system on target machines with one or more direct access secondary storage devices. The file system is structured as a tree in which each node is a file. In addition, each node may have substructure consisting of one or more descendant nodes.

Each node has a name consisting of up to 32 characters and all the immediate descendants of a node have unique names. The root node of the tree has the unique name *. A node is specified by a *pathname* which is a sequence of names separated by the / character. A pathname describes a path through the tree. For example, the pathname

```
*
```

refers to the root node, and the pathname

```
*/src/io/.Seek
```

refers to the node named .Seek which is an immediate descendant of io, which is an immediate descendant of src, and src is immediately under the root.

The nodes which are direct descendants, or *sons*, of a given node, their *father*, are ordered. The file system provides functions which return the pathname of the father, the next brother or the first son of the node specified by a given pathname. This enables a program to traverse a subtree of the file system.

Each process has an associated *current node* which is inherited from its parent; it may be changed by

```
.Set_current_node( pathname )
```

The concept of current node allows abbreviated references to files in the subtree rooted at the current node. A pathname starting with "/" describes a path starting at the current node. The current node is referred to by

```
@
```

For example, if the current node is */src/io then the file */src/io/.Seek can also be referred to as either

```
/.Seek
```

or

```
@/.Seek
```

Two functions are used to modify the file system tree structure. A new node is created by

```
.Make_node( pathname )
```

The space occupied by a file is reclaimed and, if it is a leaf, the node is deleted by

```
.Remove_node( pathname )
```

A file system structure can be *grafted* as a subtree to the file system by

```
.Graft( pathname, device_name )
```

The root node of the file system structure on the device specified by device_name can subsequently be referred to as pathname. Nodes in the grafted file system structure can be referred to as described above using pathname as the name of the root node.

.Ungraft(pathname)

ungrafts the file system structure rooted at pathname making the root of the previously grafted substructure inaccessible using pathname. Grafts allow the use of multiple and removable secondary storage devices.

The contents of a file are regarded as an infinite sequence of bytes. Initially all of the bytes are null. The function .Put, described above, is used to modify individual bytes in a file when it is open with write or append access. A file is physically represented by one or more *blocks*, where the size of a block depends on what is convenient or efficient for the particular device. All bytes after the last physical block are null by definition. Files open with write or append access are automatically grown to contain the data written. A file open with write access can be explicitly changed to a specified size by

.Change_file(fcb, size_in_blocks)

after which it will only become smaller by another call to .Change_file or by removing the file, although the file will still be grown as required to contain data written.

The current byte position is determined by

position = .Where(fcb)

The file *mark* is a byte position in the file which is remembered over accesses to the file. The mark is initialized to 0 when the file is created. The mark of the file selected for output is set to the current byte position by

.Mark()

When a file open with only write or append access is closed, the mark is set to the current byte position. The current byte position is set to the mark by

.Seek_mark(fcb)

or when a file is opened for append access.

.At_mark(fcb)

returns 1 if the current byte position is equal to the mark, and 0 otherwise. The concept of mark generalizes that of "end-of-file" in being a position in the file that can be retained over several accesses to the file.

2.3 Portability

Thoth was developed as part of a project to produce a portable programming system for a large class of minicomputers. The interest in portability was motivated by problems in interfacing with the hardware and system software of target machines when porting application programs. Problems of interfacing with the target machine system software are generally more difficult than those of interfacing with the target machine hardware because of the wide variety of abstract machines presented by the compilers, assemblers, loaders, file systems and operating systems of the various target machines. Our goal was to investigate the feasibility of developing portable system software and porting it to "bare" hardware. Because the *same* system software would be used on different hardware, application programs can be designed for and executed in the same abstraction on different hardware. Consequently, most application programs which use Thoth can be portable if not machine independent.

Most previous work on software portability has focused on problems of porting programs over different operating systems as well as over different hardware. To our knowledge, this is the first time an entire system has been designed for portability. Our experience indicates that this approach is practical both in the cost of porting the system and in the time and space performance of the system on the target machine.

An earlier experiment in operating system portability has been reported by Cox (1975). More recently, the UNIX operating system (Ritchie and Thompson (1974)) has been moved from a PDP-11/45 to an INTERDATA 7/32; this port was done first by Miller (1978) at the University of Wollongong, and also Johnson and Ritchie (1977) at Bell Telephone Laboratories.

Three main portability problems were addressed during the development of Thoth. The first was to design an abstraction of a minicomputer that could be efficiently realized on a large number of machines. The dual problem is choosing the domain of target machines so that a reasonable (and efficient) abstraction is possible. The second problem was to represent the abstraction in such a form as to minimize the effort required to implement it on target machines. The third problem was to design and implement software tools to automate as much of the implementation as possible.

A high-level language has been used to represent most of the system so that most of the translation into machine code can be done by a compiler. The language has been designed to encourage the use of machine-independent constructs; however machine-specific code can (and sometimes must) be used. To document machine dependencies, the software is divided into components which are stored in separate files, each containing either a single function, a set of related global data modules, or a set of related manifest definitions. Each file is classified as either machine-invariant or machine-specific; this classification is implied by the subtree of the file system in which the file is stored. The machine-specific components which need to be modified during a port are thus isolated from machine-invariant code.

The machine-specific components of the system which must be changed during a port may be viewed as *interfaces* between system abstractions and machine hardware. The main interface is provided by the compiler which maps the machine-independent high-level language constructs into machine instructions. The code generation phase of the compiler is thus an important interface which must be changed during a port. Other interfaces are represented by assembly code and *twit* statements in the high-level language. Besides the compiler, there are three main interfaces.

The first interface is in the primitive operations of readying, blocking and preempting processes. The form of these primitives is machine-invariant, but a small amount of interface code must be written to load and store the volatile environments of processes. The abstraction of interrupts is a more complicated aspect of process preemption and activation; this abstraction is implemented by an assembly coded module called the *interrupt handler.*

A second interface is between the input/output system and the hardware interfaces. For character-oriented devices, such as teletypes, simple functions which "output a character and wait for an interrupt", or "wait for an interrupt and then input a character" are implemented using *twit* statements. A direct access secondary storage device is treated as an indexed sequence of fixed-size blocks which may be read or written randomly by referring to the index of the desired block. The block size may vary from one device to another and there may be several logical devices per physical

device as in UNIX (Ritchie and Thompson (1974)). This abstraction is easily implemented in the device handlers. For purposes of program loading and swapping, the random access device handlers also support reading and writing of multiple contiguous blocks, which is supported directly by the hardware on many machines.

A third interface can by used when appropriate memory mapping hardware is available. This consists of a small number of functions for changing memory maps.

Implementing these interfaces is relatively straight-forward because, except for the memory mapping interface, no design decisions need to be made. The interface functions have simple well-defined semantics. In most cases, the interface functions from a previous implementation for another machine can be used as *prototypes* in which only the machine-specific parts must be changed. That is, they serve as a model for the new implementation.

Interface code for the NOVA implementation contains 216 assembly language instructions and 201 *twit* instructions. The TI 990 implementation contains 638 assembly language instructions and 334 *twit* instructions. The Honeywell Level 6 implementation contains 251 *twit* statements and 457 assembly language instructions.

2.4 Performance and Real-time Properties

Measurements of operating system performance are important to the design of application programs, particularly real-time applications. Such measurements are also useful for evaluating the efficiency of the system design and implementation.

Thoth has been designed to be configurable to a variety of real-time applications. The basic Thoth configuration includes support for the following:

dynamic memory allocation
process creation
interprocess communication
input/output
$tty0

Optional facilities that can be configured include:

clock
process destruction
additional devices
file system
multiple teams and swapping

Thoth may be used in two different modes: multiple team and single team. In a multiple team system, the system code is loaded into primary memory before (i.e., separately from) any of the user teams. In a single team system, all system code is linked with the user code as a single core image. A configuration that allows multiple teams also provides the ability to load executable teams from files, swap teams when the primary memory resource becomes scarce, enforce an equitable sharing of the CPU resource among the teams, and reclaim resources from teams that terminate either normally or abnormally. For these reasons, any configuration that includes multiple teams must also include the file system, clock and process destruction options. If the system is configured to run a single user team "stand alone," there is no interdependence of the options; thus, the user's requirements alone dictate the options to

be included.

The space requirements for a null user team are given in Table 1; the figures are separated into requirements for code and data. Code includes instructions and external variables, while data includes team descriptors, process descriptors, stacks, and buffers allocated during system initialization.

A substantial decrease in size can be realized by eliminating the input/output primitives and support for $tty0. The code size for a stripped version of Thoth can be reduced to approximately 2000 words for the NOVA/2 and 2700 words for the TI 990. The stripped versions are indicative of the sizes possible for specialized control applications.

	NOVA		TI 990		Level 6	
	code	data	code	data	code	data
basic system	5166	257	5588	307	5499	243
clock	767	166	930	180	956	175
destruction	715	283	1024	464	972	229
disc i/o	668	121	1359	303	963	224
file system	4319	415	4730	697	4935	425
additional tty	0	247	0	305	0	232
multiple teams	+	*	4396	*	+	*

Table 1: Sizes of Thoth configurations. All sizes are given in 16-bit words. The total space required by a given configuration can be computed (approximately) by adding the requirements of the options to those of the basic system. Note that since the second tty shares code with the first, it requires only the space for an entry in the configuration table and data for additional driver processes.

\+ These measurements are not currently available.

* The data size for multiple teams is decided when the system is generated; the data space required depends on the maximum number of user teams and processes and the number of devices supported by the system. Systems currently running use from 3600 to 6300 words.

The system running on a TI 990 that is used for all software development and maintenance and which supports 5 terminals, 2 disc drives, a printer and a synchronous communication line uses approximately 27.6 K for the memory resident part of the system and uses the remaining 44.4 K words for swapping of user and system teams.

The speed of various Thoth primitives was measured by counting the number of times a primitive could be executed within a given period of time and calculating the unit costs from these figures. This measurement technique introduces some error due to overhead for dispatching, handling clock interrupts, loop control and counting. The results of these measurements are presented in Table 2.

	NOVA/2	990/10	Level 6
.Send/.Receive/.Reply	656	1862	1320
.Forward	323	1025	630
.Create/.Ready/.Destroy	5000	16160	11023
.Put	88	176	164
.Get	69	131	128

Table 2: Times in microseconds for primitives.

Input and output that is block rather than byte oriented can be done in a portable manner using facilities not described in this thesis.

A further design goal has been that the system meet the demands of real-time applications. To this end, the system guarantees that the worst-case time for response to certain external events (interrupt requests) is bounded by a small constant. To achieve this goal, we have required that the system have *bounded disable time*. That is, the maximum time that interrupts are disabled in the system is bounded by a small constant. It was not clear initially that this goal was possible or practical to achieve. We contend that the multi-process structuring of the system has aided in achieving this goal. We discuss below some measurements of the real-time properties of the system.

One measure of the real-time response of a system is the maximum amount of time required to respond to an interrupt. In lieu of a hardware monitor to measure these times directly, we have used a simulator (Sager(1976)) which provides measurement of worst-case disable times and response times to interrupts. The disable times are given in Table 3.

Table 3: Worst case disable times in microseconds for Thoth on the NOVA/2. The associated function name indicates where the disabled code begins.

disable time	function
297	.Send
266	.Free
254	.Receive
187	clock processes
162	.Ready
111	.Alloc_vec
107	.Await_interrupt

Response time for a specific device is defined to be the elapsed real time from "device completion" until the CPU "services" the resulting interrupt. For the NOVA computer, Data General (1974), "device completion" occurs when the device sets its DONE flip-flop to 1, and the CPU services the interrupt by setting the device's DONE flip-flop to 0. The NOVA is capable of selectively disabling devices for interrupts; this allows the use of multiple priority levels for interrupt handler processes. When a device's handler process is active, interrupts are selectively disabled for dev-

ices with handler processes of the same or lower priority; those with handler processes of higher priority may still interrupt and thereby preempt the lower priority process.

The best response time for the $tty0 output is 68 microseconds. The best response time for the real-time clock is also 68 microseconds. The worst cases occur when both devices complete just as a lower priority process enters the worst-case disabled section of code (see Table 3). Because the real-time clock handler process has a higher priority than the $tty0 output handler, the worst case time for the clock is 68 + 297 = 365 microseconds, assuming all other device handler processes are of lower priority. With this assumption the worst case time for the $tty0 output is 68 + 68 + 297 + x > 433 microseconds, where x is the time spent in the real-time clock handler from the occurence of one clock interrupt to waiting for the next clock interrupt. Unfortunately, we have no convenient way of measuring the worst case value for x; this value would occur at the beginning of a year. This is consistent with the observed worst-case response times of 316 microseconds for the real-time clock and 594 microseconds for $tty0 output.

3 Design of a Multi-Process Abstraction

This chapter describes some of the problems encountered when designing and implementing Thoth and discusses the rationale behind design decisions that lead to the abstraction described in the previous chapter.

The design of the abstraction was based on two general requirements: (1) The abstraction must support the multi-process structuring of programs; (2) The abstraction must be realizable by an efficient implementation. The second requirement made it necessary to consider the implementation as the system was designed. Because of this, the two issues are discussed together in this chapter.

The following is a list of facilities that the abstraction should provide to make multi-process structuring attractive:

1. many inexpensive processes - A system that provides few or relatively expensive processes discourages or forbids their use for structuring programs. Fast process switching is required because many processes executing concurrently generally requires more process switching. Also, sets of processes interact, requiring synchronization, and consequently process switching.
2. efficient process identification - Processes must be identified or named to specify process interactions, such as which process is to receive a message.
3. inexpensive interprocess communication - Processes must communicate to interact and cooperate productively. This requirement includes the need to communicate events such as breaks and faults to a process, events that occur asynchronously with respect to the process's execution. To make this communication practical, it must be inexpensive.
4. Dynamic configurability - It must be possible to dynamically allocate

resources as required by the program and release them if and when they are no longer required. Dynamic process creation and destruction are required because processes are resources.

Problems and solutions associated with satisfying these requirements are considered in the following sections.

3.1 Process Implementation

The abstraction of a process is implemented by a process descriptor, a process data area and functions that switch the processor among these processes. The process descriptor contains system information such as process state, priority, creator and the volatile environment. The *volatile environment* is a set of values for the set of machine registers that are physically but not logically shared by all processes. Typically this includes the instruction counter, condition or status words and other processor registers. In Thoth, the volatile environment also includes a global variable, *active*, containing a pointer to the process descriptor of the active process, and a variable containing a pointer to the top of the stack for the active process (used in stack overflow checking). A process is the active process at a given time if its volatile environment is defined by the respective machine registers at that time. All other processes have their volatile environments specified by their process descriptors.

Providing many processes requires keeping the cost per process small. This cost has two components: time and space.

The time cost is mainly that of switching processes. This cost is the cost of loading and storing the volatile environment plus the cost of searching for the next process to activate. The first cost depends on the size of the volatile environment and the hardware support for loading and storing it. The second cost is determined by the CPU allocation rule and its implementation. In the following, we consider the Thoth CPU allocation rule, how it is implemented, and its advantages and disadvantages.

The next process to activate must be a ready process so it is more efficient in time to only search over the set of ready processes (rather than over all processes). Thoth employs a data structure that allows fast identification of the ready processes. For each priority level, there is a queue of ready processes. The processes in each queue are ordered by the time at which each process became ready, the most recently readied process last in the queue. There is also a queue with two entries per priority level, ordered by priority. The first entry for a priority level points to the first ready process (if any) at that priority level. The second entry points to the last process in this queue, used for adding processes to the end of the queue. Only this last queue of ready heads needs to be searched to find the process to activate.

A process switch is caused by either a higher priority process becoming ready or the active process blocking during the execution of a message-passing primitive or awaiting an interrupt. A higher priority process can be readied either by the active process or by an interrupt. The two cases to consider are: (1) a higher priority process becoming ready; (2) the active process blocking. In the first case, the function readying the higher priority process also activates it, so only the cost of readying the process (storing the current volatile environment, and loading the other environment) is incurred. Thus, a search is required only when the active process blocks. In the second case, there can be no ready processes of higher priority than the active pro-

cess, so the blocking function need only search the same and lower priority ready queue heads, thus reducing the scope and hence the cost of the search. There is always a ready process because the *idle process*, a system process of lowest priority, is always ready. Thus, the search is always successful, making it simpler than if it had to anticipate not finding a ready process, and more efficient if it usually finds a ready process other than the idle process.

This simple scheme is adequate for a system configured as a single team with the user program linked and loaded directly with system functions. The use of multiple teams suggests two options in extending the CPU allocation scheme: the priorities of processes on a team can be absolute system priorities so processes on different teams created with the same priority execute with relative indivisibility, or the priority of a process can be relative to its team. The latter option was chosen for Thoth for several reasons. The system team needs to be of higher priority than other teams so at least two levels of team priorities are needed: system and user. Real-time applications require that it be possible to make some teams higher priority than others, hence resident teams. Dynamically changing the team priority of transient teams is used to multiplex (time-slice) the CPU among these teams and to exclude teams from CPU allocation when they are swapped out to effect a time-sharing mode of operation. Finally, although choosing this option abandons the relative indivisibility property between processes created at some priority on different teams, this property seems impossible to retain along with a time-sharing mode for transient teams. That is, with relative indivisibility, a process executes until it blocks. If it never blocks, no process of the same or lower priority in the system could ever execute, which is unacceptable in a timesharing mode. Thus, the main advantage of the first option is incompatible with timesharing.

In Thoth, the absolute (or system) priority of a process is its team priority plus its relative process priority. The relative priority of a process remains fixed but the system priority of a process on a transient team is changed by changing its team priority. The system increases a team's priority to give the processes on the team CPU preference and reduces the priority to bias the CPU allocation to other teams. When a transient team is swapped out, it is assigned a very low swapped priority and the heads of its ready queues are removed from the list of ready queue heads so swapped processes are never considered for activation.

This CPU allocation scheme has several advantages:

1. Finding the next process to activate is always a simple successful linear search of a short queue, the length being the number of different priorities levels of processes in memory.
2. The process priorities can be used to guarantee real-time response properties of the system and resident teams. They can be used in general within a team to give preference in CPU allocation and to achieve relative indivisibility.
3. Process switching is oblivious to swapping yet the scheme can be used to allocate time slices of CPU preference to transient teams and to make swapped processes ineligible for activation.

Therefore, it is possible to realize both real-time constraints and a time-sharing mode with this CPU allocation scheme. In contrast, a strict time-slice CPU allocation scheme would not allow sufficiently fast activation of processes to allow processes to be used to service interrupts, as is done in Thoth. A fixed priority scheme would not provide a time-sharing mode of operation. The time cost of the process switching is

small; moreover, machines are beginning to appear that provide hardware support for process switching, an example being the Honeywell Level 6 minicomputer.

The Thoth CPU allocation scheme has the property that two processes on the same team can effectively "deadlock on the CPU" because the CPU allocation rule does not guarantee progress to all processes. Two processes on different transient teams cannot deadlock on the CPU because the relative priorities between processes on different transient teams change periodically. It is not clear how to remove the possibility of "CPU deadlocking" without discarding the concept of process priorities because high priority processes must be guaranteed CPU preference. The existence of several levels of priorities has proven to be a useful tool, and the absence of the guaranteed progress property has not been a problem. Because deadlocking on the CPU is local to a transient team and because all deadlock situations involve "busy waiting" on some event, it has been easy to avoid.

The space cost of process implementation is that of the process descriptor and the process data area. The space cost of the code for process switching is not included because the cost is small and independent of the number of processes. In Thoth, the size of the process data area (the process stack) is determined by the base language, the machine architecture, and the call tree of functions for the particular process. The size of the process descriptor depends partly on the machine architecture because it must be large enough to store the volatile environment of the process, which includes the processor state. Thus, machine architectures with large processor states like the CRAY-1 with its many vector registers would require a large process descriptor. The size of the process descriptor is larger than the size of the volatile environment because of additional fields providing other per process information such as process priority, process identification and process hierarchy information.

There are a number of trade-offs in the implementation of processes. Features can be added to the basic notion of process presumably making it more powerful but also raising the cost per process. An example of such features would be additional scheduling information. We have instead followed the design philosophy of keeping the notion of process simple and using many of these inexpensive processes to achieve the required power. There is also a trade-off between the space and time costs per process. For example, the ready queue structure described above increases the space cost but decreases the time cost of process switching. Also, Thoth process descriptors contain redundant information to make certain operations fast at the expense of additional space in the process descriptor. For example, the process descriptor contains pointers to the process's ready queue to make process switching fast. Some of this redundant information is required for the bounded disable time property. That is, the extra information allows an indivisible operation to take only a bounded number of instructions. The system might benefit from a reexamination of the value of some process descriptor fields now that the trade-offs are more evident.

3.2 Process Identification

A mechanism of naming processes is required when sending and receiving messages and when readying or destroying a process; this mechanism is called process identification. A process identifier, called an id, is *valid* if it refers to an existing process. Process identification should be efficient in both time and space. The process id should require little storage because it is used and stored in many places. Similarly, the following operations should be efficient:

1. Testing the validity of a given id.
2. Generating new *unique* id's.
3. Mapping a valid id to the corresponding process descriptor.
4. Invalidating a given id.

The validity test should be efficient because of its use in the message-passing primitives to check the existence of the specified processes, thus contributing to the time required for message-passing. More importantly, this test may affect the maximum disable times. That is, testing an id's validity and using it must be done indivisibly with respect to process destruction to guarantee its validity when it is used. This indivisibility is most easily achieved by disabling interrupts.

A further requirement arises from the need to recycle id's. There is only a finite number of unique id's available. Consequently, after a sufficient number of process creations and destructions, an id must be reused. Because an id may be known to other processes and is the sole means of identifying a process, reusing an id can cause problems analogous to that of the telephone company reusing telephone numbers. That is, an id may be valid but identify a different process than the user of the id expected. To reduce the likelihood of this confusion, a final requirement of the scheme is that it guarantee a (long) minimum recycle time for process id's. The recycle time for an id is the number of process creations between the destruction of a process with that id and creation of another process with the same id.

In Thoth, an id may be any non-zero value that can be stored in a machine word. This provides a large space of possible id's allowing a long recycle time. The system reserves space for some maximum number of process descriptors. (For the purposes of this section, process descriptor is used to mean one of the blocks reserved for a process descriptor and not the contents of the block.) The number of id's is larger than the number of process descriptors, so a many-to-one mapping is used to map id's onto process descriptors. Let N be the number of process descriptors and therefore the maximum number of processes. The id is mapped into a process descriptor as follows. There is a vector with N entries, .Pd__bundle, each entry of which points to a process descriptor. Given an id, the (id modulo N)-th entry of .Pd__bundle points to the process descriptor corresponding to the id. Therefore, for each k between 0 and N-1, every value i such that k = i mod N maps to the same process descriptor.

Each process descriptor contains a field called ID. A process id, k, is *valid* if the ID field of the process descriptor to which it maps contains the value k. Therefore, it is simple to check in a bounded amount of time whether a 1-word value is a valid process id.

In practice, we restrict the number of entries in .Pd__bundle to be a power of 2 so that the modulo operation may be performed by a bit-wise logical AND. To avoid forcing N to be much larger than the maximum number of processes expected or required, the extra entries of .Pd__bundle are mapped to a "pseudo process descriptor" which is marked as in use but not valid. All id's that map to these entries are not valid. For example, if 258 process descriptors were required, the .Pd__bundle would have 512 entries but only the first 258 entries would point to distinct process descriptors. The remaining entries would point to the same pseudo-process descriptor which is marked as in use but not valid. The modification to the basic scheme to have the size of the .Pd__bundle a power of 2 increases the speed of the test while the "pseudo process descriptor" trick avoids allocating space for unneeded process descriptors.

The test is thus

```
if( ID[.Pd__bundle[id & .Id__mask]] == id ) then valid
```

where (id & .Id__mask) is equivalent to (id modulo N) because N is a power of 2 and .Id__mask has the value N-1. The ID field of the pseudo process descriptor mentioned above is set to one so its value does not map to this process descriptor so it is not valid.

Unused process descriptors are kept in a singly-linked list with a pointer to the head of the list and a pointer to the last process descriptor in the list, thus allowing fast deletion from the head of the list and fast insertion at the end. A process id is only invalidated when a process is destroyed. When a process is destroyed, the id is saved in the OLD__ID field, the ID field is set to zero and the process descriptor is added to the end of the list of free process descriptors. A new id is allocated only when a process is created, and thus only when a process descriptor is allocated. Process descriptors are allocated from the beginning of this free list. Each free process descriptor contains a field called OLD__ID which contains the last id used with the process descriptor. The new id is

```
new__id = OLD__ID[pd] + .Id__mask + 1;
```

where pd is a pointer to the allocated process descriptor. Thus, the new id is the next larger value (in modulo machine arithmetic) after OLD__ID[pd] which maps to this process descriptor. The worst case cycle time for the new id is now the least k greater than zero such that

```
id = (id + k*(.Id__mask + 1)) mod (2 ** m)
```

where m is the number of bits in the machine word, e.g. on a 16-bit machine with 64 processes, k = 1024. Therefore, this scheme guarantees a reasonable though non-optimal recycle worst case at an acceptable cost. The worst case occurs when the same process descriptor is used for each of these process creations so generally, the recycle time for id's is much better than this worst case.

Instead of the above scheme, one can make the id space sufficiently large, possibly by storing id's as multi-word values, so that the id's will never be recycled in practice. On a machine with a large word size (say 32 bits or more), one might reasonably expect that id's would never recycle just using single word id's. This maximizes the recycle time because no id is ever recycled. However, on machines with small word size, multi-word id's would be required, making process identification more awkward and less efficient. Also, on a machine with a large word size, it may be attractive to use part of the id to specify attributes of the process, effectively reducing the cycling part of the id and thus requiring some guarantees on the recycling of id's.

The optimal scheme for maximizing the recycle time in general requires maintaining a queue of unused id's ordered by their last use. Process creation would then select the least recently used id that maps to a free process descriptor. However, it would be too expensive to store the queue of unused id's because this must be large to allow a long recycle time.

The scheme presented above guarantees a reasonably high minimum recycle time process id's at a small cost. Note that this scheme of identification could be applied to objects other than processes that are created and destroyed and pose similar prob-

lems of identification.

3.3 Interprocess Communication

Interprocess communication includes message-passing, synchronization, interrupt handling and data sharing. Interrupt handling is included by viewing it as communication with a hardware process.

Processes can communicate via shared data or by system primitives. The wide spectrum of possible semantics for the primitives can be rendered more manageable by recognizing the following classification:

1. data transmission capability - Data transmission capabilities range from none to arbitrary length messages of unrestricted content. Pure synchronizing primitives are characterized by their lack of data transmission ability.
2. synchronous - A primitive is synchronous if all processes it directly affects must be in some appropriate state for the action of the primitive to complete. Otherwise the primitive is asynchronous. Thoth message-passing primitives are synchronous.
3. explicit mention of processes affected - A primitive can be explicit as to which process(es) it affects; e.g. the .Send primitive of Thoth is explicit because the process affected, the receiver, is explicitly named. A semaphore is not explicit.
4. number of processes affected - A primitive can affect just the process executing the primitive, one other process or several others. .Send, .Receive and .Reply just affect one process other than the invoking process while .Forward affects two other processes. A primitive can affect some fixed maximum number of processes or a potentially unbounded set as with a broadcast-type primitive where the effect is broadcast to a large set of processes.

Designing interprocess communication primitives is a non-trivial task because their semantics and efficiency can have a significant impact on the system. We propose basing the design of these primitives on the following criteria:

1. The primitives should have simple, well-defined semantics.
2. The semantics should allow an efficient implementation.
3. The semantics and the implementation should be such that a set of processes can only be deadlocked using the primitives when they are logically deadlocked in the abstraction, i.e. deadlock is always understandable within the abstraction.
4. The primitives should be *safe* in the sense that the semantics are well-defined and harmless to the invoking process independent of other processes being destroyed. By harmless, we mean that the invoker of the primitive is not destroyed or left blocked indefinitely.
5. The primitives should be both implementable and useful for a distributed architecture.

We now consider the design of interprocess communication via data sharing and various primitives, keeping these criteria in mind.

3.3.1 Data Sharing

Data sharing is the sharing of access to a segment of memory by two or more processes. This provides an implicit mechanism for communicating between processes. Data sharing is attractive for programming problems that decompose as several processes operating on shared data structures, where each process does relatively little computing in comparison to the amount of data accessed. Examples include sorting, searching, text editing and formatting, and various graph algorithms. In these cases, the data structures are more naturally shared than duplicated and the cost of moving data is avoided.

Several alternatives are possible for the design of a mechanism to provide data sharing. The virtual memory mechanism of Multics (Bensoussan et al. (1972)) allows shared segments between processes; however, the mechanism used is expensive because of the elaborate address translation on every memory reference, not portable because of special hardware requirements, and under utilized (Montgomery (1977)). Alternatively, data can be shared by storing it in a file if the constraints of secondary storage are acceptable.

The design chosen for Thoth derives from the observation that most data sharing occurs between closely interacting processes. It is necessary that they interact closely if the data is changed because sharing is only safe providing the processes accessing the data are synchronized. Thus, the scheme chosen was to implement an address space unit, called a team, can contain sevearl processes. It is created with one process in it and that process (or subsequently created processes on the team) can create more processes on the same team, and therefore in the same address space. A *team* is a set of processes that share the same address space, memory free list, code segments and data segments. The cost of interprocess communication between processes on the same team is small because data segments are shared. Also, because a team is swapped as a unit, message-passing between processes on the same team does not interact with the swapping (such as a process blocking because of a process that is swapped out). Therefore, the team provides a means of associating a set of closely interacting processes in terms of data and code sharing, and in terms of swapping.

The team implementation extends the conventional virtual address space implementation of one process per address space by adding the facility for additional processes in the same address space, not by extending the address space implementation to shared data areas. Thus, the implementation of this address space is portable.

The team concept is not a completely general scheme for data sharing because of the (necessarily) close interrelation between processes on the same team. For example, all processes on a team other than the initial process on the team are descendants of this initial process, so two processes that are created independently cannot share data via the team concept. There are also several costs associated allowing an arbitrary number of processes per team as opposed to having only one process. The operating system cannot safely use the team space for handling a system call because there may be other processes on the team active in this same address space that could interfer with the system call handling. It is also much harder to have a protected system area per process as part of the swapping unit (as many systems do) because the number of processes per team can be arbitrary. Therefore, more of the description of each process may have to be memory-resident or else a sophisticated scheme of swap-

ping these protected areas is required. The provision of dynamic memory allocation is more expensive because of the need for synchronizssarily imply that this is a good time to swap the entire team. On systems with one process as a swapping unit, a process can be swapped to secondary storage when it blocks, thus freeing up primary memory immediately. Finally, with several processes on a team, there are several separate stacks, making dynamically growing of a process stack more difficult. (Currently, the size of a process's stack in Thoth is fixed when the process is created).

3.3.2 Data Transmission

Data transmission is the movement of data between memory owned by different processes. Transmitting data reduces the need to share data, reducing the need for synchronization and for shared memory. Data transmission is required for communicating processes that are in disjoint address spaces, such as when two processes are on separate machines. Also, transmitting the data is often more natural than sharing it. For example, sending a message to another process logically involves moving the data rather than putting the data in a shared location. In the following, the data transmitted is called a message and the primitives are called message-passing primitives.

The basic actions are sending and receiving messages. The specific semantics of the primitives affect their efficiency. We first consider the sending of messages. A primitive that allows a process to send a message and continue before the message is received by the recipient process requires a buffering scheme for storing in-transit messages. Message buffering costs in both time and space. The cost depends on the buffering scheme used and the size of the buffers. First, buffers can be either statically or dynamically allocated. In the *static* scheme, each process has some number of associated buffers which determines the maximum number of in-transit messages that it can have at any given time. In the *dynamic* scheme, a message buffer is allocated (or requested) at the time the message is sent. The advantage of dynamic allocation is that a process not sending messages does not consume message buffers, thus providing more buffers for processes actively sending messages. Buffers can also be allocated for the particular size of the message rather than requiring them to all be the same (maximum) size. Finally, memory can be dynamically allocated for message buffers and released when not in use, reducing the amount of memory dedicated to message-passing. However, if processes are almost always blocked sending messages and there are more senders than receivers, this advantage will not be realized.

A variation on dynamic buffer allocation is to use a segment of the sender's address space for the message. The message is sent by transferring the segment to the address space of the receiver to avoid copying the message, a noticeable saving for large messages. However, this scheme has a number of disadvantages: it is expensive to allocate, transfer, release, and swap segments; the sender loses the message from its address space so it may have to make a copy of it; and the scheme depends on special memory management hardware being available, so it has limited portability.

Dynamic buffer allocation has the disadvantage of being more expensive that static allocation. However, the major disadvantage arises when there are no unused message buffers available (we assume that the space for buffers is finite). If a process attempts to send a message and no buffer is available, there appear to be two basic options: The process can block pending the availability of a message buffer, or the send primitive can return an indication of failure and let the process decide what to do. We reject the second option as unusable because, in general, the process cannot

do anything to make a message buffer available. With the first option, exhaustion of message buffers can lead to deadlock from circular dependencies in the conditions for message buffers becoming available. This type of deadlock violates Criterion 3 because it results from blocking on a resource internal to the implementation of the abstraction. That is, processes that are not logically deadlocked in the abstraction can become deadlocked because of the finite number of message buffers available. The finiteness of the message buffer pool seems difficult to reflect in the abstraction because of the dynamic allocation. In the static allocation scheme, the process can know how many messages it can send before exhausting its buffer pool so the process need not deadlock on the availability of message buffers. In the dynamic scheme, there may be no message buffers available the first time a process attempts to send a message.

Two solutions to the deadlocking are the following. A supervising mechanism can either prevent deadlock, or detect and somehow remove deadlocked processes. The cost of deadlock detection adds considerable overhead by known deadlock detection algorithms (Shaw (1974)). Moreover, it is unclear what the semantics should be at the abstraction level of resolving a deadlocked situation. A second solution is to limit the number of messages a process can have outstanding at any time. This limit must guarantee that each process can send some minimum number of messages before blocking from the unavailability of message buffers, and no process should have more than that limit in outstanding messages. Therefore, there must be that many buffers per process, and no process will ever use more than its limit. This reduces the scheme to essentially that of static buffer allocation (even if the limit for messages is different among the processes).

It is necessary to limit the maximum number of in-transit messages per process to prevent a process from consuming all the message buffers, by sending output via messages to a slow or non-functioning device for example. There is a trade-off between allowing many in-transit messages per process with the resulting cost per process (and per message) versus allowing few in-transit messages per process and having less expensive message-passing. Providing many in-transit messages allows a high degree of concurrency in the message-passing. For example, a process may send several requests for service and then wait for completion or responses from them rather than issuing these requests sequentially. However, achieving concurrency through the message-passing seems to lead to more complicated programming because the process is no longer functioning purely sequentially. Besides raising the cost of message-passing, the use of in-transit messages requires added complexity and administration to keep track of outstanding messages and to handle errors. For example, using a message-passing mechanism providing many in-transit messages, a file system could be implemented as a single process that could process many file system requests concurrently by using in-transit messages to concurrently request the required disc I/O. However, the process would have to maintain a record of the state of each request, the messages outstanding, plus checking that the requests being processed concurrently did not conflict (both in the sense of logical conflicts and in the sense of head contention of two requests on the same drive). In essence, the process is simulating several processes with the state information and the concurrent messages. In general, our experience leaves us unconvinced of the utility of allowing many in-transit messages even though a number of designers stress the importance of this facility (Jammel and Steiger (1977), Brinch Hansen (1969)).

The design philosophy of this thesis is to achieve concurrency with many processes rather than with the message-passing so it was choosen not to provide in-transit messages. That is, a message is not effectively sent until it is received by the

recipient process. This is called fully synchronized message-passing. With this constraint, the message can reside in the environment of the sender until received so message buffering is not required. Only fully synchronized message-passing is considered in the following. This is consistent with Criteria 1, 2 and 3 in having simple, well-defined semantics, being efficient, and not allowing internal deadlocking.

The cost of transferring a message depends on three factors: the number of times it is moved; the size of the message; and how difficult it is to move. A message has to be moved more than once if it is not possible to move the message directly from the environment of the sender to the environment of the receiver. For example, if the two processes are on two different transient teams that cannot fit into memory together, the message must be moved through a memory resident buffer so it has to be moved at least twice. A message also has to be moved more than once if the message is to be broadcast to several receivers. Our experience to date has not motivated the use of a broadcasting-type primitive so this alternative is not considered further. The difficulty of moving the message is affected by the need for special memory mapping instructions, interaction with swapping or paging, and communication between machines. This is usually a consequence of not being able to move the message directly.

In a system configured with transient teams, either messages sent by processes on transient teams need to be stored in a memory-resident buffer, or the recipient processes must be prepared to fetch the message directly or indirectly from the swapping device (in the case of the transient team being swapped). The first option effectively requires one buffer per process; if buffers are statically allocated to processes, they need to be big enough to hold the maximum sized message. Buffers incur a space cost multiplicative in the number of processes so this scheme has the disadvantage of increasing the space cost per process and limiting the maximum size of message. It has the advantage of making message-passing independent of swapping. The second option provides a powerful message-passing mechanism but requires a more complicated implementation. It is also less efficient if small messages are common.

The first option is the scheme used in Thoth. The criteria of simple, efficient semantics without internal deadlock (Criteria 1, 2 and 3 above) have lead to fully synchronized primitives, passing small, fixed length messages. The synchroneity avoids the inefficiency and deadlock of dynamic buffer allocation and the memory cost of several buffers per process. One statically allocated message buffer per process avoids interaction with swapping, which otherwise complicates receiving messages. Small, fixed length messages reduce the overhead of buffers and moving data.

Sending and receiving messages are the basic actions of message-passing. In that sense, .Send and .Forward are compound because both could be implemented in terms of several basic actions. Providing these compound actions as primitives has advantages over just providing the basic actions.

The combined "send and receive a reply" function of .Send has the following advantages. .Send can have the simple semantics of a subroutine call in the sense that the send action supplies arguments to the recipient process and the reply returns the result. If the recipient only acts on a message between the time that it receives the message and the time that it replies to the sender, the process interaction is entirely sequential from the point of view of the sender, just as if this action had been accomplished by invoking a subroutine. This sequentiality is exploited in considering the

verification of the system in Chapter 6.

There are typically more .Send's than .Receive's in programs so it is of benefit to minimize the code required to perform a .Send. The combined action of .Send avoids extra code that would be required for receiving a reply message by a separate primitive. Also, if receiving the reply was a separate operation, the original sender might not be ready to receive the message so the sending of the reply would cause the receiver to block. In the case of a user process sending to a system process, the system process would block sending the reply and the user process would have to be activated to unblock the system process. This would be particularly inefficient if the user process was swapped out at the time. In general, under this scheme a process sending to a higher priority process would require two extra process switches to receive the reply. With the current primitives, only two process switches are required for a message to be sent, received and replied to; one switch from the sender to the receiver and a switch back to the sender after the reply is sent.

The main disadvantage of the reply in .Send is the additional overhead when a reply is not needed. In our experience, the reply is required most of the time, so this is not a practical disadvantage.

The semantics of .Forward were motivated by the need to forward a message to another process without blocking and without requiring the sender to resend to the other process. This allows a process to act as an intermediary in message-passing between several processes.

In general, the primitives have proved adequate for the synchronized communication of short messages prevalent in the system; however, they have shortcomings. If the logical message being sent is larger than the 8 words that will fit into a message, the message must either be broken into packets of 8-words and sent as several messages or transferred by some other mechanism. Sending a logical message as several physical messages means that the message is not transmitted indivisibly. That is, an 8-word message is either fully received or not received at all, which is not true of a (say) 16-word message sent as two 8-word messages. In general, sending a message as packets raises some of the same issues found in a packet-switched network, (Davies and Barber(1973)), such as not all the packets of a message arriving.

This shortcoming could be remedied by allowing longer or arbitrary size messages as was mentioned as an option earlier. An alternative is to provide a second data transmission facility, a virtual direct memory access mechanism for moving data. The .Transfer primitive in Thoth provides the facility as described in Chapter 2. This was chosen over extending the message-passing for several reasons. The .Transfer mechanism is more general and more powerful than a mechanism providing arbitrary size messages because the data transmission is separate from the blocking and queuing of the message-passing. For example, the Thoth debugger inserts break points that result in the process encountering the breakpoint sending to the debugger process. The .Transfer mechanism can be used by the debugger to fetch and store words from that process's address space. As another example, a large amount of data can be transferred between processes by the sender specifying the location of the buffer containing the data and its size in a message. The receiver can then allocate a buffer of that size and use .Transfer to move the data into the buffer. That is, the message can provide information about the data to be transferred, making it possible for the receiver to prepare to receive it and to specify where the data is to be placed. The .Transfer primitive in combination with the message-passing primitives also provides a means of sending data that is not naturally a linear array. For example, a request to

the file system might contain various control information, and strings specifying pathnames. Using only arbitrary length messages, this would require formatting these into a single array to send in the message and require the file system to decompose the message once received. With .Transfer, just a pointer to the pathname strings and their size need be sent in the message. The file system can then allocate space for these strings and transfer them into the allocated space. The .Transfer mechanism is also easier to implement than arbitrary length messages in a system structured as many processes interacting via messages (as Thoth is) because some system processes implement the disc I/O and the swapping mechanism. These processes communicate via message-passing yet they cannot directly use the mechanism that they implement without sending to (and thus deadlocking) on themselves. That is, a primitive for receiving a long message would use the services of the swapping device to access the message if the sender is currently swapped out. However, the swapper process can receive messages from a currently swapped process so it may be hard for it to use the same primitive that it implements. These processes can use the short messages currently implemented without circularity because they are independent of the swapping. Finally, our experience to date suggests that communication is of the nature of many short control messages plus fewer large data transmissions. The two levels provided by the message-passing and the .Transfer mechanism recognize and exploit this characteristic of communication.

3.3.3 Synchronization

By synchronization, we mean the forcing of a partial ordering on certain program events. Many systems provide explicit primitives for synchronization. We claim that specific synchronization primitives are not needed in Thoth because a Dijkstra binary semaphore can be simulated by a process and message-passing. This is shown as follows:

The P and V operations are Send's to a process, called the semaphore process, with the process id semaphore_id.

```
P( semaphore_id )
 {
  auto message[MESSAGE_SIZE];

  .Send( message, semaphore_id );
 }

V( semaphore_id )
 {
  auto message[MESSAGE_SIZE];

  .Send( message, semaphore_id );
 }
```

The semaphore process executes the following code.

```
Semaphore_process()

 {
  auto requestor, request[MESSAGE_SIZE];

  repeat
   {
    requestor = .Receive( request );  Receive a P
    .Reply( request, requestor );     and allow to proceed
    .Receive( request, requestor );   Wait for V
    .Reply( request, requestor );
   }
 }
```

Notice that the content of messages here is irrelevant because only the queuing and synchronizing of the message-passing is being used. Also, the error-checking behavior of this construction is rather poor.

This construction does not fully simulate a binary semaphore because the same process doing the P must also do the V operation. For example, the usual semaphore solution to the producer-consumer problem (Dijkstra (1972)) needs to violate this restriction. A binary semaphore can be fully simulated using two processes and the message-passing as follows. The semaphore is created by

```
Create_binary_semaphore()
 {
  extrn P_process, V_process;
  auto s;

  s = .Alloc_vec( 1 );  2 word semaphore descriptor
  s[0] = .Create( &P_process, P_STACK );
  s[1] = .Create( &V_process, V_STACK );
  .Ready( s[0], s[1] );
  .Ready( s[1], s[0] );
  return( s );
 }
```

which creates the two processes, passing each the other's process id. The operation of the two processes can be interpreted as passing a token. The V process

```
V_process( p_process )
 {
  auto msg[MESSAGE_SIZE];

  repeat
   {
    .Send( msg, p_process );  Send token to p_process
    .Reply( msg, .Receive(msg) );  Get a token
   }
 }
```

starts out with a token and passes it to the P process. It then waits to receive another

token. The P process

```
P_process( v_process )
 {
  auto msg[MESSAGE_SIZE];

  repeat
   {
    .Receive( msg, v_process ); get a token
    .Reply( msg, .Receive(msg) ); give out the token
    .Reply( msg, v_process );
   }
 }
```

receives a token from the V process and then hands it out to a process executing a P operation. The P operation

```
P( s )
 {
  auto msg[MESSAGE_SIZE];

  .Send( msg, s[0] ); send to P process
 }
```

and the V operation

```
V( s )
 {
  auto msg[MESSAGE_SIZE];

  .Send( msg, s[1] ); send to V process
 }
```

are simply .Send's to the appropriate process. This same process structure can be used to simulate a general semaphore: the V process maintains the semaphore count and the V operation sends to it to increment the count; the P process sends to the V process to decrement the count before allowing its own sender (executing a P) to continue.

These constructions demonstrate that no explicit synchronization primitives are necessary in addition to the message-passing primitives. However, no form of semaphore is used in the system for reasons described below in Section 3.4.2. Instead, Thoth employs processes and message-passing directly as described in Chapters 4 and 5 to implement synchronization.

The message-passing primitives allow a process to synchronize with other processes. There is also a need for processes to synchronize with hardware processes via interrupts. Interrupts are part of the communication between the device requesting the interrupt and the processor. Regarding the device as a hardware process, it is attractive to communicate with the device using .Send so this communication can be identical to other interprocess communication. That is, a process sends a message requesting service from the device and receives a reply when the service has been performed or an error is encountered. This is the approach taken in Thoth.

This abstraction is implemented as a software process acting as the representative of the hardware device process. This device handling process receives messages, interacts with the actual hardware interface to accomplish the requested service and then replies with possibly some completion status information.

The use of device handlers raises the issue of how to synchronize them with the devices, in particular interrupts. For this purpose, the .Await_interrupt primitive of Thoth allows a process to block until an interrupt from some specified device has occurred. It seems easier to implement and is more portable than directly simulating message-passing with hardware processes. For example, not all device interaction maps easily onto receiving one fixed length message. This interrupt abstraction is easy to implement because the blocking mechanism is already available. The fact that a process unblocked by an interrupt is immediately activated means that this activation is more efficient than if the process was readied and then later activated after a search of the ready queues. Also, the use of this interrupt abstraction allows many of the device handlers to be machine-independent up to their interface with the hardware. Further advantages of device handling processes are described in Chapter 5. Note that this structure does not preclude message-passing with the device more directly on some machines, dispensing with the device handler process.

A special case of synchronization is indivisibility. An action is indivisible with respect to a class of actions if none of these actions can proceed concurrently with the indivisible action. Actions can be made absolutely indivisible by disabling interrupts while the code causing the action is executed. This is true providing that the code does not involve blocking or readying another process of higher priority (unless indivisibility is not required beyond that point). Blocking or readying a higher priority process can only occur in the Thoth abstraction by awaiting an interrupt, invoking a message-passing primitive, or invoking .Ready on a newly created higher priority process. The execution time for the disabled code must be bounded by a small constant if the system is to have good real-time properties. Ideally, disabled code should be conceived as an extension of the hardware instruction set and used only in the kernel of the system. User processes cannot, in general, disable interrupts so this is not a widely applicable means of achieving indivisibility.

Frequently, indivisibility is only required with respect to the execution of other processes. This relative indivisibility can be achieved using process priority levels. Note that this type of indivisibility is essentially of no cost because it is derived from the CPU allocation rule; the CPU allocation rule is inexpensive and motivated by the considerations of Section 3.1.

3.3.4 Asynchronous Communication

The communication described to this point is strictly synchronous yet there is also a need for some type of asynchronous communication. For example, it is necessary to communicate the occurrence of a break to a program. A process cannot predict if and when such a break will occur and it would be absurd for it to block waiting for the break to occur. Therefore, the process needs to either incur the cost of polling for breaks or have the occurrence of a break communicated to it asynchronously. We reject the polling option because of inefficiency and its inadequacy for communicating with erring processes. The need for asynchronous communication is analogous to that required between the processor and various peripheral devices. One possible approach (not taken in Thoth) is to implement the concept of software inter-

rupt or signal (Ritchie and Thompson(1974)).

A *signal* is a software process interrupt; a process that has been sent a signal will start to execute code corresponding to the signal the next time it is activated. This code is analogous to the interrupt routine invoked in response to a hardware interrupt.

Signals provide a general mechanism for asynchronous communication, e.g. certain error conditions such as an arithmetic exception can be signalled to a process. However, signals have a number of disadvantages. Signals can require sophisticated programming like that needed to deal with hardware interrupts. That is, the signal mechanism should provide for disabling signals, receiving and acting upon a signal and specifying which routine to invoke in response to a signal. These complications make processes harder to program; they also make processes more expensive. Second, signals are asynchronous events so the execution of a process is non-deterministic in the same sense that the execution of the operating system is made non-deterministic by hardware interrupts. Non-deterministic processes are harder to verify because of the large number of control flows that usually result from non-determinism. Third, a signal is logically only acted on after the process unblocks, making a signal ineffective for dealing with deadlocked processes. Thus, a mechanism such as process destruction is required for removing deadlocked processes. One could implement signals so that a blocked process receiving a signal is unblocked in order to respond to the signal. This complicates the semantics of message-passing and awaiting an interrupt. For example, does a process unblocked by a signal return to its previous blocked state after responding to the signal? Do all signals unblock blocked processes or just some?

We propose using process destruction to handle all asynchronous communication in the system. Thus, the destruction of a process is the only event that can happen to a process asynchronously with respect to its execution. This avoids the extra cost of more sophisticated forms of asynchronous communication. The handling of this asynchronous event is simpler than handling signals because the resources need not satisfy the process's local consistency constraints after the event since the process does not exist after the destruction. It is only necessary to guarantee the integrity of resources that are relinquished by the destruction. Aspects of the process destruction mechanism are described more fully in Section 3.4 and its use is illustrated in Chapter 4.

3.3.5 Safety of Communication

Safety of communication is the security and reliability properties of the interprocess communication provided by the data sharing and the various primitives.

The data sharing is limited to processes on the same team which are expected to trust one another and synchronize on shared data using the message-passing primitives. That is, because a team is an execution of a program or part of the execution of a program, this unprotected interaction is local to closely interacting processes.

The message-passing primitives need to be safe in the presence of process destruction because a process cannot know that a specific process will exist when it invokes a primitive on this process. Therefore, a primitive invoked on a non-existent process or on a process that is subsequently destroyed behaves as a null operation

other than returning an indicative status. Thus, it is safe to send to, receive from, reply to, forward, or transfer data to or from a process even though it may be destroyed at any time. This makes it safe for system processes to interact with user processes.

The primitives are also safe in the sense that all blocking behavior is explainable in the abstraction. For example, there is no possibility of deadlocking internal to the implementation of the abstraction.

The semantics of the primitives make receiving messages safe provided the receiver is prepared to receive messages from any process when invoking the receive-any form of .Receive. The receiver can safely reply to the sender because .Reply does not block. If the reply was received by a separate receive operation, then with synchronous message-passing, the original sender would have to be trusted to attempt to receive the reply. This problem is illustrated by the interaction between operating system and user processes; a system process must be able to safely reply to a user process. It can also transfer data in and out of the sender's address space using .Transfer, which is safe even if the source or destination arguments are illegal. That is, a receiver can receive a message specifying a buffer location and the size of the buffer that it is to transfer into its own space. If the buffer location is outside the address space of the sender, then .Transfer will just return an error status and not memory fault. The receiving process has control over the sending process in several ways. It can keep the sender blocked, allowing the receiver to synchronize the sender with the action requested rather than trusting the sender to synchronize properly. The receiver also has control over the sender by the ability to forward its message to another process with a possibly changed contents. Thus, the receiver has full control over the sender except that the sender may be destroyed independent of the receiver.

As a consequence of the control of the sender by the receiver, the sender must trust the process it sends to. That is, user processes must trust system processes and system processes cannot be allowed to send to user processes. This simple protection policy has been adequate for the system to date. Its simple, efficient implementation prompts exploring its applicability and limitations, rather than extending it to a more sophisticated scheme.

An unsafe aspect of the primitives lies in their blocking behavior. A process executing .Receive, .Send or .Await_interrupt can block indefinitely if the required unblocking event does not occur. However, the usual solution of time-outs on the blocking can effectively be implemented using these primitives, the clock abstraction and possibly extra processes. This is discussed further in Chapter 4.

In conclusion, we claim that the interprocess communication mechanisms in Thoth have followed the criteria listed at the beginning of this section. The adequacy of the primitives provided is best evaluated from experience with their use. Comparing the Thoth interprocess communication to that provided by most systems, we would expect that the most contentious issue in the design would be the synchronous regime of communication imposed in Thoth. We claim that the design is consistent with the philosophy of avoiding the expense of providing parallelism in the communication mechanisms and achieving the required parallelism by using many processes instead. Finally, problems of distributed systems have been considered in designing these primitives but there is no experience to date to support their utility in this environment.

3.4 Dynamic Configurability

Dynamic configurability is the ability to dynamically allocate and release program resources. The resources in the abstraction include memory, files and processes. Note that the CPU is not an explicit resource within the abstraction.

Dynamic configurability has several advantages. Resources need be allocated only when first needed and can be released when no longer required, thus avoiding allocating the maximum ever anticipated or allocating in advance of need. This leads to better resource utilization. It also allows a more elegant manipulation of resources because resources can be released when not needed and then reallocated for another purpose, rather than having to awkwardly transfer the resources from one activity to another. Process creation and destruction allow a program to dynamically and asynchronously reconfigure its multi-process structure. Finally, the facilities for dynamic allocation of files and memory form the basis for a mechanism to recover resources as part of process destruction.

Dynamic resource allocation must handle the case of not being able to immediately satisfy an allocation request. There are three options for this situation: block until the request can be satisfied; preempt the resource to satisfy the request; or return an indication that the allocation cannot be satisfied currently. The first option can lead to deadlock on the availability of resources. The second option requires a mechanism for deciding which process to preempt and for returning the resource at some later time. Usually, it is simpler to just destroy the current owner of the resource. Thoth implements resource allocation corresponding to the third option. Therefore, processes in Thoth cannot block on the availability of resources so they cannot deadlock on system resources. This does not preclude user programs implementing their own local resource allocation (using message-passing) that causes blocking. To date, a more sophisticated scheme has not been required.

3.4.1 Dynamic Memory Allocation

Dynamic memory allocation is provided through the primitives of .Alloc_vec and .Free. They provide a mechanism by which processes on a team can share a common pool of memory without interference. The memory allocation is structured as two levels. On the first level, the system allocates memory to the team, part of which is used to contain the team memory free list. On the second level, processes on the team allocate from the free list. The two level allocation allows a team to be created with a large enough free list that it does not require more memory and reduces the overhead that would be incurred by growing the data segment of the team on every allocation request. The former is necessary for resident teams that cannot be increased in size because their location in memory is fixed at the time they are created. The allocation of memory to teams is discussed as part of the discussion of the swapping mechanism in Chapter 5.

In a conventional system with one process per address space, the memory free list is only accessed by that one process so no synchronization is required and the allocation can be done by a set of run-time library functions. With multiple processes on a team, operations on the free list need to be synchronized to avoid uncontrolled

concurrent operations on it. There are several alternatives for synchronizing .Alloc_vec and .Free. First, both operations could be executed with interrupts disabled making them absolutely indivisible and therefore, properly synchronized. However, the time for allocation is not bounded by a small constant because it involves a search for a suitably sized free block of memory plus possibly growing the data segment of the team. Similarly, freeing a vector can involve inserting the vector to be freed in order in the free list in some memory allocation schemes, also taking a potentially unbounded length of time.

Another alternative is to impose a higher level synchronization through say the message-passing primitives. For example, the memory allocation could be performed by a system process. This has the disadvantage that this process must be able to do this safely independently of the behavior of the processes on that team. Many memory allocation schemes maintain the data structures of free blocks right in the free blocks to avoid additional storage. Because this is addressable by all processes on the team, such a system process can make no assumptions about the correctness of the data structure. In particular, it must check the validity of every pointer to avoid incurring a memory address violation. Even if the system was designed so that the memory allocator process could incur such a fault and continue, one would need to guarantee that it could not get into an infinite loop. Another difficulty is that memory allocation is useful at low levels in the implementation of the system so implementing it to rely on higher levels of the system such as processes leads to circularities in the implementation. Finally, using higher level synchronization primitives makes memory allocation slower in comparison to the unsynchronized functions that are adequate for the single process situation. The concern with speed of memory allocation is prompted by the desire to use memory allocation in time-critical applications often found in real-time systems rather than by any measurements of high utilization of the CPU by memory allocation.

The solution used in Thoth is to decompose the allocation and freeing of memory into steps, each of which is executed with interrupts disabled. Each step is a fixed, acceptably small number of instructions that leaves the free list data structure in a consistent state. This scheme just adds the overhead of function calls to the step functions plus the cost of disabling and enabling interrupts.

The particular scheme currently used is a first-fit algorithm in which the free blocks are maintained in a doubly-linked list sorted by address to allow coalescing. There is two words of overhead per allocated vector. One word for the size of the vector and one word to link the vector into a per-process singly linked list of allocated vectors. To clarify the description of the technique used, the code for .Alloc_vec and .Free is included.

```
.Alloc__vec( size; abort, zero )

{
  auto nargs, start, vec, words;

  nargs = .Nargs();
  if( nargs < 3 ) zero = 1;  default to zeroed vector
  if( nargs < 2 ) abort = 1;  default to abort
  start = 1;
  words = (size < 1) ? 4 : size + 3;

  repeat
   {
     vec = .Alloc__one__step( words, start );
     start = 0;

     if( vec == 1 )
      { Try to grow the memory
        if( .Grow__memory(MEM__INCREMENT) != OK ) break;
        start = 1;
      }
     else if( vec )
      { Allocated vector
        if( zero ) .Zero( vec, size );
        return( vec );
      }
   }
  if( abort ) .Abort( "out of memory" );

  return( 0 );  out of memory
}
```

The function .Alloc__vec repeatedly calls .Alloc__one__step which is executed totally disabled. If .Alloc__one__step is called with the argument start non-zero, it sets a free block pointer associated with the free list to the first free block and determines if this block satisfies the allocation request. If not, the free block is advanced to the next free block and it returns zero. If the request is satisfied, the free block pointer is set to the first free block, the memory allocated is removed from the free list and a pointer to the allocated block is returned. If the variable start is zero then .Alloc__one__step examines the free block after the free block pointer. The function returns one if the end of the free list is reached without satisfying the request, whereupon it attempts to grow the free list and then restarts the search at the beginning of the free list.

Similarly, .Free

```
.Free( vec )
 {
  extrn .Death__proprietor__id;
  auto  prev, t;

  t = MEMORY__RESOURCE[prev = active];
  while( t && t != vec ) t = MEMORY__LINK[prev = t];

  if( t == 0 || *(t-2) < 4 )
     if( ID[active] == .Death__proprietor__id )
      {
       TEAM__STATE[TEAM[active]] |= BAD__FREE__LIST;
       return( 0 );
      }
     else .Abort( "bad vector" );

  if( prev == active )
     prev = active + MEMORY__RESOURCE - MEMORY__LINK;
  while( .Free__one__step(vec, prev) ) ;
  return( 1 );
 }
```

calls .Free__one__step which is executed totally disabled. .Free__one__step adds the vector passed as an argument to the free list if the free block pointer points to the block after which this vector is to be added and returns 1. Otherwise, it moves the free block pointer to the next block in the free list in the direction of where the vector is to be freed.

One can argue from the CPU allocation rism is necessary because the free block pointer must be reset to the beginning of the free list when a process on that team is destroyed in case this process has advanced the pointer after interrupting a lower priority process that was allocating a vector.

This scheme provides synchronized memory allocation for multiple processes requiring only a bounded amount of disable time and with only a slight increase in overhead over the unsynchronized allocation functions that suffice for the single process case. The main disadvantage seems to be its more complex behavior. The memory allocation deserves further investigation.

3.4.2 Dynamic Process and Team Creation

Process creation adds another concurrent entity to the system with resources and identification. It involves allocation and initialization of a process descriptor, process id, and stack for the new process, and the linking of this process descriptor into the tree of processes. There are two types of process creation in Thoth.

In the first type, the new process is added to the team of its creator. This type of process creation is fast because it does not involve creating a new address space, allocating swap space, or locating the file containing code and initialized data. These are all available from the team of the creator of the process. Also, creating the process in the same address space and swapping unit allows inexpensive interaction with

other processes on the team. This type of process creation forms the basis for the data sharing discussed earlier. It has the disadvantage of requiring that the code to be executed be already available in the team; it also requires part of the team's address space for data.

One is motivated to make process creation a two stage operation as is done in Thoth using .Create and .Ready because the process id returned by .Create may be needed as an argument for .Ready. This situation is illustrated in Section 3.3.3 by the Create_binary_semaphore function used in the semaphore simulation.

The second type of process creation, called team creation, creates the new process as the first process on a new team. Therefore, it is created in a separate address space with code and initialized data from a specified file.

In conclusion, dynamic process creation allows a program to dynamically configure its process structure in responding to the different demands. The full utility of this configurability is only realized with the provision of dynamic process destruction.

3.4.3 Dynamic Process Destruction

Process destruction is halting a process, removing it from the system and releasing all its resources. Process destruction is useful for disposing of processes, reclaiming resources, and handling faults, breaks and exceptions. For this utility, process destruction must have the properties of being complete, immediate and indivisible with respect to the process requesting the destruction.

Process destruction is *complete* if all resources allocated to a process are relinquished in a consistent state when the process is destroyed. Complete process destruction is required because a process is often destroyed in order to reclaim resources; it is unacceptable for resources to be lost. The concept of a team complicates complete process destruction because some resources are returned to the system and some are returned to the team. For example, the memory owned by a process is allocated from its team free list and is thus returned to this free list when it is destroyed. Resources such as the process descriptor and open files are returned to the system. All the team's resources are returned to the system if there are no remaining processes on the team of the destroyed process. That is, a team ceases to exist when there are no processes left on the team so there is no separate team destruction.

Process destruction is *immediate* if the process is destroyed without requiring it to reach some particular state. Immediacy guarantees that the process can always be destroyed (even if it is deadlocked) and thus the process's resources are always reclaimable. A system prooviding message-passing must either ignore the possibility of processes deadlocking via the message-passing, or provide facilities for preventing this deadlock or for removing deadlocked processes. Ignoring the possibility of deadlock is not reasonable. Preventing or arbitrating deadlock imposes an additional overhead on the message-passing mechanism.

Process destruction is indivisible with respect to the process requesting the destruction if the process is destroyed and the resources released before the requesting process can continue. This assures the requestor that the resources belonging to the destroyed process are available for allocation immediately after the destruction. Re-

claiming some resources takes time. For example, a file may require disc I/O before the buffers can be released. This limits the speed of destroying processes and delays the process requesting the destruction as well.

The semantics of the Thoth process destruction primitive .Destroy are complete, immediate and indivisible to the requestor. The use of process destruction is considered more fully in Chapter 4. The implementation is discussed in Section 5.2.5.

There is a conflict between the need for mutual exclusion of an operation and the need for immediate process destruction of a process performing that operation. In general, an operation that must be executed mutually exclusively operates on shared data and it must leave this data consistent. Therefore, it needs to be completed once started so the process executing the operation must not be destroyed until the operation is completed.

However, immediacy makes process destruction an asynchronous event with respect to the execution of the process to be destroyed. This problem is illustrated by considering the use of semaphores for mutual exclusion in the presence of immediate process destruction.

Suppose the system made use of Dijkstra's semaphores with the P and V operations (Dijkstra (1972)) for imposing mutual exclusion on access to shared resources. Let s be a semaphore on some resource. Suppose a process has executed a P(s) for this resource and has proceeded into a critical section and then this process is requested to be destroyed. Three options are available in the design at this point.

1. The destruction could be postponed until the process has done the corresponding V, or more correctly, until it has done no P's without the corresponding V's. This trusts that the process will eventually reach such a state and violates the requirement that process destruction be immediate. Note that it is necessary to provide synchronizing primitives for the user because Thoth provides the user with a multi-process facility, and ideally, these should be the same as the system primitives. Therefore, it is necessary to make no assumptions in the process destruction mechanism about the correct use of semaphores. For example, there might be no V corresponding to the P. Also, using semaphores for producer-consumer type problems, there are usually semaphores that one process executes P's on and the other executes V's on. Sometimes the initial state is as if a P had been done, but not a V. It is unclear how to identify these cases when destroying a process.
2. The process could be destroyed without regard for semaphores. In this case, resources owned by the process through semaphores are lost because the corresponding V's will never be done. This contravenes process destruction being complete, reduces the usefulness of process destruction for the user, and endangers the correct operation of the system.
3. The process descriptor could be used to record the semaphores that the process currently owns via P's and the destruction of the process could include doing V's on these semaphores to release the resources. This violates the requirement that process destruction be complete because there is no guarantee that the resources are left in a consistent state. Also, as mentioned above, the process doing the V is not always the process that did the P.

Thus semaphores do not suit an environment including complete and immediate process destruction. The same argument holds for the monitors proposed by Hoare (1974) and the critical regions proposed by Brinch Hansen (1970) if we suppose a pro-

cess is destroyed while executing a monitor (or critical region) because there is no synchronization with respect to process destruction in these schemes.

Several solutions can be used for synchronization in the presence of immediate process destruction. Certain operations can be made indivisible so process destruction cannot occur while the operation is performed. Disabling of interrupts is used in Thoth for indivisibility in several places including message-passing primitives, process creation and memory allocation. For code that involves blocking on another process, this is not sufficient. For example, the message-passing primitives are indivisible with respect to process destruction until they block. Making these primitives indivisible with respect to process destruction violates process destruction being immediate. We instead define the semantics of sending to or receiving from a destroyed (non-existent) process and the semantics of message-passing with a process that is subsequently destroyed (see Chapter 2). These semantics are implemented with the aid of the process destruction mechanism.

A form of synchronization with process destruction available only to operating system processes is the use of relative indivisibility with respect to process destruction. System processes of the same priority or higher as the process responsible for process destruction can make use of the fact that no processes can be destroyed until they block.

Finally, a solution is required for operations that block, operations for which the indivisibility cannot be achieved by disabling interrupts because of long execution time, and user processes for which disabling interrupts and relative indivisibility with respect to death are not available. The solution proposed is a concept called a proprietor, a stereotype process that does not need to be destroyed. This is described in Section 4.3.

3.5 Summary

This chapter has described the considerations and decisions behind the design of the Thoth abstraction for multi-process structuring. The simple notions of process and CPU allocation make processes relatively inexpensive. The process identification scheme provides a way of specifying processes with little danger of confusion, even though process id's are recycled. Teams provide a means of grouping processes so that they can interact inexpensively because, as a team, they share code and data segments and swap as a unit. The fully synchronized message-passing provides data transmission and synchronization between processes without the danger of deadlock internal to the implementation of the abstraction or the cost of dynamic buffer allocation. The Transfer primitive provides large data transfer as a supplement to the message-passing. It was shown that the message-passing could be used for synchronization between processes and that .Awaitinterrupt provided synchronization with interrupts. Process destruction was proposed as a means of asynchronous communication. The safety of the communication was defined by a simple policy based on trust between processes on the same team, sending processes trusting their receiving processes, and the interprocess communication primitives having well-defined semantics with respect to process destruction. The provision of process creation and destruction allows programs to configure their multi-process structures dynamically. Process destruction provides a simple mechanism for exception handling and resource reclamation.

These facilities form the raw materials for developing multi-process programs. The remainder of the thesis is concerned with understanding how to use these facilities to structure programs. In Chapters 4 and 5, we support the sufficiency and utility of the abstraction for multi-process structuring by giving examples of multi-process structures and by describing the structure of the Thoth operating system which both provides and uses these facilities. Chapter 6 explores the value of multi-process structuring in facilitating verification of the system.

The costs of some of the primitives are given in Section 2.4 for the TI 990, Honeywell Level 6 and the NOVA 2 and are indicative of the efficiency of the implementation.

4 Multi-Process Structuring

The multi-process structure of a program is defined by its processes and their interaction, particularly interaction via message-passing. This structure imposes a structure on the functions and resources of a program by associating each function with the processes that execute it and each resource with the processes that allocate (and thus own) the resource at some time. The structure can change dynamically by the use of .Create and .Destroy to create and destroy processes. This chapter discusses designing these structures.

The discussion has several purposes. It serves to document insights into the use of multi-process structures gleaned from the development of Thoth. It explores the adequacy of the abstraction described in Chapter 2 for multi-process structuring by demonstrating how to solve various problems in this abstraction, and by showing how this abstraction is used by various programs. Finally, it supports the contention that the parallelism of multi-process structures can be used to make a program more understandable than if it was written as a sequential program. This is not to imply that developing good multi-process structures is easy; on the contrary, perceiving a good process structure for a program can be a major obstacle in its development.

4.1 Multi-Process Structuring Principles

A program comprises a tree (or a forest) of processes, possibly on different teams. Several principles for structuring programs as multiple processes can be abstracted from the experience gained in developing and using Thoth programs.

If a program as part of its specification must respond to events that occur asynchronously with respect to its execution, several processes can be used to achieve this asynchronous response. For each asynchronous event, there is a process that blocks waiting for that event to occur, and responds to or handles the event. The event is synchronous with respect to this process although it is asynchronous with respect to the rest of the program. This approach suggests a discipline of programming in which each process only responds to events synchronized with its execution. The re-

quired asynchroneity of the program is realized by its multi-process structure. The synchroneity of the individual processes then makes the program more understandable. In a real-time environment, this use of processes can provide fast response to events because the process waiting for the event can be readied immediately after the event occurs and can execute logically concurrently with respect to other processes. On a single processor machine, the processes do not really execute in parallel, but the preemption of the CPU by high priority processes and the resulting interleaving of the executions of processes provides the logical concurrency required. Events requiring particularly fast response can be handled by high priority processes. For example, the operating system must respond to hardware interrupts from peripheral devices. Interrupts are handled by device handling processes, one per device with the higher priority processes servicing the devices requiring faster service.

Concurrent response to events can allow efficient resource utilization in general. For example, a program could be structured into compute-bound and I/O-bound processes so that the compute-bound processes execute while the I/O bound processes block on I/O, overlapping the use of the CPU with the I/O, thus decreasing the real time for execution and increasing throughput.

Invoking a function as a separate process means that the activity of this process can be halted by destroying the process independently of the activity of other processes. (Using process destruction here has the additional benefit of reclaiming all resources allocated by that process.) Conversely, the activity of this process can continue independently of other processes being destroyed. This is used in implementing break and exception handling, as described in Section 4.5. It is also important in the use of proprietors, as described in Section 4.2. Finally, it can be used by the program to dynamically reconfigure itself to perform a different activity by destroying some processes and creating others. For example, this is used in an inter-machine communication program to change from the terminal communication mode to the file transfer mode and vice versa.

A process can be created on a separate team so that it executes in a separate address space (or even a separate machine although this is not currently supported in Thoth). Designing a program as several teams reduces the memory requirements of the program to that of the largest team. It also reduces the run-time cost if not all teams need to run concurrently. For example, the Thoth base language compiler is structured as several teams which communicate via files. The initial team creates the phases of the compiler as separate teams and coordinates the overall execution. This means that at any time, the address space required for running the compiler is much less than the total space required for all of the compiler, allowing the compiler to execute on small machines. Also, the compiler can dynamically configure which phases to run for different types of compilations and for different target machines.

Real-time programs can use separate teams as well. A program can be divided into resident teams which guarantee a certain level of real-time response plus transient teams which can handle less time-critical functions, thus avoiding the memory commitment of having the entire program memory-resident.

Finally, there are examples in the operating system offl operations that a process cannot logically perform on itself directly. These include a process destroying itself, going to sleep, or being swapped out. For example, a process destroying itself must be completely destroyed when the destruction is finished but in existence to the end to complete the destruction. Implementing the operation as a separate process invoked by sending it a message resolves this difficulty because this separate process can act to

release all the resources of the process being destroyed.

A process can be used as a control structure to implement an independent sequence of actions. This is used in programs that comprise several relatively independent sequences of actions even though they may not be required to respond to asynchronous events. It is attractive to make each sequence of actions a separate process, rather than using conventional control structures to implement the sequences as one process. This use of processes is analogous to the use of control structures and data structures in conventional programming languages. For example, in the implementation of a file copy program, one is motivated by efficiency to buffer the data between the input and output files. The moving of data into a buffer from the input file and the moving of data from the buffer to the output file are two sequences of actions coordinated by the amount of data in the buffer. In many systems, these two activities must be simulated by imposing two modes of action on one process, namely a mode of reading and a mode of writing, with the one process switching between these two modes depending on the amount of data in the buffer. Multi-process structuring allows each sequence of actions to be implemented by a separate process. This process structure can also allow concurrent reading and writing of the files.

Multi-process programs can be difficult to develop and understand because of two basic problems not found in sequential programs. There is the possibility of deadlock because processes block when message-passing. In a given set of processes in some state, a process is *deadlocked* if it is blocked and the condition required for it to be unblocked is dependent upon it not being blocked. The important aspect is that of circularity in the dependencies of the unblocking conditions.

A process can only be deadlocked if it is blocked doing a .Receive in the receive-specific form or if it is blocked doing a .Send. A process blocked waiting for an interrupt is not deadlocked because the unblocking of the process is dependent only on the occurrence of the interrupt and the interrupt occurrence is independent of the process being blocked waiting for the interrupt, even though it can block indefinitely until the interrupt occurs. Assuming the correct functioning of the hardware and normal use of the system, all requested interrupts occur within a finite amount of time, thus unblocking processes awaiting interrupts. A process cannot block on executing .Reply or .Forward because these primitives do not block. A process cannot deadlock by executing the receive-any form of .Receive because any other process can do a .Send to this process and unblock it. (A program can be in a state such that there will never be a send to the blocked process so it is effectively deadlocked. This is discussed later.) Thus, a process can only be deadlocked if it is blocked doing a .Send or a receive-specific. It can deadlock with these primitives trivially by sending to or receiving from itself. The possibility of a set of processes deadlocking using .Send or receive-specific is examined by considering its blocking graph. The *blocking graph* of a set of processes is the labelled directed graph defined such that:

1. There is a node in the graph corresponding to each process.
2. There is a directed edge from node A to node B labelled S if and only if process A sends to the process B.
3. There is a directed edge from node A to node B labelled R if and only if process A does a receive-specific on process B.

Therefore, an edge from A to B means that process A can block on process B.

If the blocking graph of a set of processes contains no cycles, the set of processes cannot deadlock because no circularity in the blocking can occur. The presence of cycles in the resulting blocking graph does not necessarily imply that the system can

deadlock. However, showing the absence of deadlock requires a more detailed argument that the cyclical blocking represented by the cycles cannot actually occur. For example, if there is an S edge from A to B and an R edge from B to A, this cycle of blocking cannot occur because either the .Send by A unblocks B or else the .Receive by B does not block because the .Send has already been executed.

A process can be effectively deadlocked even though its blocking graph contains no cycles. A process blocked executing a receive-any may, by intention, only receive messages from a certain set of processes so it effectively blocks on that set of processes when executing a receive-any. If, for example, all processes in the set are also effectively receive-blocked on this process, these processes are then effectively deadlocked. The blocking graph can be extended to handle this case by adding an edge labelled RA from the receiver to each process it implicitly blocks on with the receive-any. One can then argue that the receiver can deadlock executing the receive-any if and only if there is a cycle through each RA edge in the graph and that all the cyclical blocking indicated by these cycles can occur simultaneously.

Another way for processes to effectively deadlock is for a process to effectively block on another process by looping waiting for, say, a variable to be changed by the other process. That is, the process is not actually blocked but is effectively blocked in the loop. This case can be handled by adding appropriately labelled edges to the blocking graph corresponding to this blocking and treating them as analogous to the receive-any as described above.

The blocking graph of the operating system is shown in Figure 1 of Chapter 5. This graph is used in Chapter 6 to argue that the system cannot deadlock. The blocking graph can be used in general to argue the absence of deadlock in multi-process programs. In our experience, the blocking graph usually contains no cycles or only cycles of the receive-specific form described above, so arguments to prove absence of deadlock tend to be straight forward if not immediate given the blocking graph. In general, the blocking graph is a useful tool for understanding the process structure of a program.

The second major problem introduced by multi-process structures is the need for synchronization. Synchronization involves imposing some partial ordering on program events to ensure correct behavior. This is required because concurrency removes the total ordering of events imposed by the sequentiality of a single process program. There are two aspects to synchronization: mutual exclusion and sequencing. Mutual exclusion imposes an ordering on events because if A and B are mutually exclusive events, either A must happen strictly before B or vice versa; they cannot happen concurrently. Sequencing requires a set of events to happen in some specified order, namely A must precede B. We consider event sequencing in Section 4.4.

Mutual exclusion is required because of sharing. In the conventional approach to multi-process programs, program resources are pooled and shared among the processes. The consistency of the resource and avoidance of timing-dependent problems usually requires that only one process use the resource at a time, or mutual exclusion.

An alternate view is based on resource allocation and ownership. A process *owns* a resource if it has allocated the resource (or has been allocated the resource) and has not relinquished it. Ownership of a resource implies exclusive access to the resource. Therefore at any time, a resource is owned by at most one process so mutual exclusion is implemented by resource allocation. This view suggests two options

for a process requiring a service that uses a resource it does not currently own:

1. The process can allocate the resource and perform the required service. This is the more traditional approach, typified by semaphores and monitors.
2. It can send a message to the process owning the resource requesting that this process act on its behalf to perform the service. This assumes that the process owning the resource is prepared to offer this service to other processes.

Option 1 associates allocation (and ownership) of a resource with the function that performs the service. Option 2 means that the ownership of a resource can be associated with the process that performs the service. This leads to a static ownership of resources because a process offering one of these services can retain ownership of resources for its entire lifetime. Static ownership of resources means that program resources can be associated with the process in the multi-process structure that owns the resource, thus inducing the same structure on the resources.

This view suggests that Option 2 be used for resources that require mutually exclusive access, especially if the resource cannot be subdivided. It also suggests that activities be allocated to processes so that the resource allocation among the processes is understandable and sharing is minimized. That is, each resource requiring mutually exclusive or sequential access is owned by one process. If a resource is needed by several processes, it is allocated to a process that makes the required services of this resource available to other processes. We refer to this type of process as a proprietor.

4.2 Proprietors

A *proprietor* is a process owning a resource that it uses to provide services to other processes. Other processes can use these services by sending the proprietor a message, called a request; the requesting process blocks until receiving a reply. The proprietor acts on behalf of the requesting process as specified by the message. Thus, although the proprietor makes the services based on the resource available to other processes, there is no concurrent access to the resource because only one process, the proprietor, acts directly on the resource. The general form of a Thoth proprietor is

```
Proprietor()

 {
  auto requestor, request[MESSAGE_SIZE];

  initialize
  repeat
   {
    prepare for request
    requestor = .Receive( request );
    act on request and compose a reply
    .Reply( request, requestor );
    complete request processing
   }
 }
```

The general form of sending a request to a proprietor is

```
.Send( request, proprietor_id );
```

The proprietor structure is used to eliminate the need for specific mutual exclusion

mechanisms as illustrated in the following example with semaphores.

Consider a simple file system in which the binary semaphore File_sem imposes mutual exclusion on the operations in order to protect the integrity of file system data structures. For example, two files must not be created with the same name. File_sem is used to impose mutual exclusion on making files as follows.

```
Make_file( pathname )
 {
  extrn File_sem;
  auto reply;

  P( File_sem );
  reply = Make_f( pathname );
  V( File_sem );
  return( reply );
 }
```

Using a file system proprietor to replace the semaphore, the proprietor performs all file system operations on behalf of other processes as follows.

```
File_system_proprietor()

 {
  auto reply, requestor, request[MESSAGE_SIZE];

  initialize the file system

  repeat
   {
    requestor = .Receive( request );
    select( TYPE[request] )
     {
      case MAKE_FILE:
            reply = Make_f( PATHNAME[request] );

      case REMOVE_FILE:
            reply = Remove_f( PATHNAME[request] );

      case OPEN_FILE:
            reply = Open_f( PATHNAME[request] );

       . . .

     }
    request[0] = reply;
    .Reply( request, requestor );
   }
 }
```

To make a new file, other processes call

```
Make_file( pathname )
 {
  extrn File_sys_id;
  auto request[MESSAGE_SIZE];

  TYPE[request] = MAKE_FILE;
  PATHNAME[request] = pathname;

  .Send( request, File_sys_id );
  return( request[0] );
 }
```

The file system proprietor shown here is similar to that used to implement the Thoth file system.

The implementation of proprietors only requires the existing mechanisms of the abstraction, namely processes and message-passing, so the implementation is portable and requires no special features in the base language, in contrast to monitors which Hoare(1972) describes as a "high-level language construction". Proprietors can be used instead of disabling interrupts to achieve mutual exclusion on a resource or operation, reducing the need for disabling interrupts. Thus, their use can aid in keeping disable times bounded. Finally, because processes and message-passing are available at the user level, user programs can use proprietors. The text editor presented in Section 4.5 is an example of such a program.

A proprietor localizes the ownership of and access to a set of resources to one process so operations on the resources are independent of other processes. In particular, operations on the resource complete independently of whether the requesting process is destroyed while the request is being processed, avoiding the problem with semaphores discussed in Section 3.4.3. This locality also means that verification of properties about the resource and resource operations are independent of the other processes of the program. As we argue in Chapter 6, the individual system proprietors can be verified as though they were sequential programs, allowing the use of standard sequential verification techniques.

The proprietor structure implements the policy of servicing requests in a first-come-first-served order. Therefore, requests that cannot be handled immediately are not queued but simply replied to with an indication of the reason. For example, a request to remove a file that is currently busy is rejected and a code returned in the reply indicating this condition. Implementation of more sophisticated scheduling policies is discussed in the next two sections.

Proprietors can be compared to other schemes for achieving mutual exclusion such as semaphores, monitors, critical regions, managers and secretaries.

The concept of proprietor is close to concepts that have been called managers but we have avoided the term "manager" because it has been used in many different ways, many of which not referring to a process or concerned with mutual exclusion. For instance, the operating system of Jammel and Steiger (1977) used a concept of manager similar to that of a proprietor; however, their concept of process and message-passing is more restrictive. Conversely, the virtual type manager of Janson (1976) is simply a set of functions for implementing a data type with no concept of process, message-passing or mutual exclusion inherent in the definition. Certain

proprietors, such as the file system, implement a data type whose operations are the available requests; however, regarding a proprietor as a type manager is not always natural because, by the definition of type manager, each proprietor would implement a different data type, so each file system proprietor implements a different data type.

Dijkstra (1972) proposes the concept of a secretary which appears similar to that of proprietor. It is, to our knowledge, unimplemented and lacking in detailed description. He remarks that this approach is aesthetically pleasing but that the practicality of this approach remains to be seen.

All of the above schemes, except secretaries and some managers, are passive structures in the sense that the process requiring the service must execute them. This requires that the process have addressability to the structure, limiting the value of these schemes on some architectures, particularly distributed machines. In contrast, a proprietor can be independent of the requesting process, with the only interaction via message-passing. This seems more promising for use in a distributed system.

The concept of proprietor centralizes in one process the access to a resource. In comparison, semaphores can distribute the access and synchronization throughout the code executed by many processes. Monitors centralize the access of the resource to the functions of the monitor but distribute the use of these functions over all processes. Centralization can have the implication of imposing excessive sequentiality of access to resources owned by proprietors. However, reducing concurrency can have the benefit of reducing the cost of process switching and contention for resources. Also, ownership of resources can be subdivided to provide more concurrency in total if desired. For example, a system could be designed with a proprietor of disc storage, or a proprietor for each disc channel, or a proprietor for each disc drive.

The most significant argument in favor of proprietors is that they have been, in our experience, adequate for mutual exclusion in the system and user programs, whereas semaphores, monitors and critical regions have been shown to be inadequate in the presence of process destruction (see Section 3.4.3). Proprietors seem incomparable with these schemes in terms of simulating each other. That is, it is not clear what it means to simulate a proprietor with semaphores and a single proprietor cannot simulate a semaphore without using an internal queuing mechanism. Because semaphores can be simulated with processes and message-passing, and appealing to various comparisons in the literature between semaphores and other schemes, like Hoare(1974) and Brinch Hansen(1970), there is no loss in power by not including these schemes as primitives in the abstraction.

The cost of the proprietor structure is the space cost of a process, the time cost of message-passing plus some overhead for producing and interpreting messages. Thus, its efficiency is dependent on the availability of inexpensive processes and efficient message-passing, which do not exist in many operating systems. Therefore, a major disadvantage of proprietors is their lack of applicability to existing systems other than Thoth. Comparing the costs of message-passing to semaphores as a means of synchronization, the execution of a P does not always result in a process switch while execution of .Send does. However, with heavily utilized resources, one would expect execution of a P to cause blocking most of the time. Thus, if process switching constitutes most of the cost of message-passing on a machine, the cost of the two schemes would be comparable under heavy utilization.

4.3 General Queuing and Serving

A proprietor is, in queuing theory terminology, a server serving a single queue of customers, its send queue, in first-come-first-served order (FCFS). In this section, we consider how to implement more general queuing and serving using the existing primitives of Thoth.

First, a queue need not be served in purely FCFS order. A proprietor can impose a simple round-robin service on its send queue as follows. The proprietor serves the first request in its queue up to some maximum amount of service. If that satisfies the request, the service would complete normally and the requestor would be replied to. Otherwise the proprietor would forward the requestor's message to itself (the proprietor) with a request for the remaining service. This has the effect of placing this process at the end of the proprietor's send queue. For example, a disc proprietor that received requests to read or write large amounts of data could read or write at most some maximum at a time; requests for larger amounts would be handled by a round-robin scheme of several smaller amounts.

There is sometimes a need to discriminate more than one class of customer. For example, processes accessing a data base can be classed by whether they are requesting queries or updates. These classes can be realized as several queues, one for each class, which can be provided by having a process for each queue; the send queue of the process serves as the queue for the associated class. In some applications, each sending process will know which queue to enter, namely which process to send to; otherwise one process could receive all messages and forward the messages to the appropriate process, thus queuing the sending process in the appropriate class. Each queue process acts as a server for its queue. This service can be FCFS or round-robin as above, or made more sophisticated by servers forwarding messages to other server processes. The servers send to the proprietor of the resource to perform the service. Using these techniques, it is possible to implement many different queuing disciplines.

A proprietor owns its resources to guarantee the required synchronization, thus imposing sequential service on the resources. However, the sequentiality can be excessive in terms of that required to ensure integrity of the resource. For example, the proprietor of a data base owns all the records of the data base because updates can affect any part of the data base. However, it is common to have two requests that are independent of each other in the sense that the two operations can be performed concurrently with no ill effects because they affect disjoint sets of records. A proprietor can effectively service requestors concurrently by employing a set of subservient *server* processes that perform the requested service. It assumes the role of a resource allocator to these servers. For example, a database proprietor would allocate to the server process, the records required to handle a request rather than the server locking the records (as in the conventional design). If several classes of requests were recognized using the multi-queuing mechanism described above, the servers for the queues could also act as subservient servers for the proprietor. The proprietor could also enqueue requesting processes by forwarding requestors to the appropriate queue. This scheme uses the dynamic allocation of resources typified by the semaphore but restricts the allocation to a special set of processes, the servers. Note that if servers could be arbitrarily destroyed, there would be the same problem of resource inconsistency observed in Section 3.4.3.

The dynamic allocation of parts of a proprietor's resource introduces the possibility of the servers deadlocking. The proprietor can avoid deadlock by requiring that all the allocation for a request be done at one time, or by analyzing each allocation to detect deadlock.

4.4 Event Sequencing

The previous sections discussed the mutual exclusion aspect of synchronization; we now consider event sequencing. *Event sequencing* is the imposition of a partial ordering on the set of program events and external events. This is implemented as follows: If event A must precede event B, the process causing B waits for event A to happen. A process can wait for an event by polling repeatedly to see if the event has occurred. This is inefficient and does not work for a process waiting on an event caused by a process of the same or lower priority (because of the CPU allocation rule). A process can also wait by blocking on certain events, called *simple events*. A *simple event* is an event that a process can block on by using a primitive in the abstraction. Thus, the occurrence of an interrupt, completion of a time period, arrival of a particular time, and the sending or receiving of messages are all simple events. The primitive .Receive used in its receive-specific form allows a process to block until the specified process is destroyed (providing it does not send to the receiver). Similarly .Send could be used if the process to be destroyed does not do a receive-any or a receive-specific from the waiting process. Therefore, it is also straight forward to implement a process that waits for a specific process to be destroyed.

We now consider the need for blocking on complex events specified as arbitrary Boolean functions of simple events, called *multi-events*. An example of a commonly used multi-event is an interrupt time-out. The process blocks waiting for an interrupt or else a time-out if the interrupt does not occur within some predetermined time period.

Blocking on a multi-event can be implemented by the process blocking on a receive-any form of .Receive, assuming that all events defining the multi-events are communicated to it by messages. This would mean that for an interrupt, a separate process would be required to block on the interrupt and then send a message to the process doing the receive-any. Also, it would be difficult for a proprietor to use a receive-any in this way because it would then also receive any outstanding requests to it.

A more general structure consists of a "coordinator" process that receives requests from processes to block on a specified multi-event. The coordinator only replies to a requesting process after its respective multi-event has occurred. It also receives *event messages* indicating the occurence of events. After receiving an event message, the coordinator determines whether this event has satisfied the Boolean expression of one or more multi-events for which processes are waiting and replies to each such process. Therefore, the coordinator must maintain information specifying multi-events and processes blocked on them. It must also arrange to receive these event messages. This requires having one process per simple event that blocks on this event and then sends to the coordinator. For example, such a process acting an as alarm clock would delay for a specified time and then send to the coordinator, thus providing a message indicating that the time period had expired. If one of these processes is blocked on a simple event that is no longer of interest to the coordinator,

the coordinator can ignore the process, destroy the process, or unblock it using .Wakeup if the process is sleeping or delaying. The first option is usually too expensive because of resources consumed by such blocked processes.

A coordinator process can also be used for debouncing an event. That is, if several events of the same type occurring within some small time period are considered to be one event, the event must be debounced. The term "debounce" comes from the problem of a mechanical switch that generates several interrupts for each change in the switch position because of mechanical bounce in the contacts. Debouncing the switch is ignoring the interrupts due to the bounce. After the first event occurs, the coordinator can arrange for an alarm clock process to send to it some specified time period. It then ignores event message about this same event until the alarm clock process sends to indicate that the time period has expired.

This structure could also be used for resource allocation in which the events are requested resources becoming available. A process would send a message requesting the allocation of some set of resources. It would be left blocked by the coordinator until the resources are available. In some cases, the coordinator would have to detect and resolve deadlock.

There is a trade-off between the number of coordinators and their complexity. One coordinator can manage all multi-events or a simple coordinator can manage a single multi-event. A coordinator only blocks when it is blocked waiting to receive a message so it cannot deadlock and using one coordinator does not produce a bottleneck (unless multiple processors are available and the computation load of the coordinator is significant).

The main disadvantage of the coordinator process is the cost, especially if extra processes are required solely to block on simple events and send event messages to the coordinator. Evaluating arbitrarily complex Boolean functions specifying multi-events can be costly, but the functions that occur in practice are, in our experience, simple. Moreover, more efficient structures can be used for specific multi-events. For example, a time-out on sending to a process could be implemented with one extra process as follows. The extra process sends to the receiver while the original process delays for the time-out period. If the extra process receives the reply before the time-out, it uses .Wakeup to ready the original process. This structure works as well for a time-out on an interrupt; the extra process would block on the interrupt. The communication between the two processes and the action of the delaying process on a time-out would depend on the application.

Finally, the above scheme suggests how the clock proprietor can be used as a simple form of coordinator. A process can delay and wait to be unblocked by any number of events communicated to it via .Wakeup. If a time-out was not desired, the delay time could be made long enough so that the time period would never expire.

4.5 Fault, Break and Exception Handling

Faults, breaks and exceptions require a change in the normal hierarchic control flow of a process. A *fault* is an abnormal condition that is detected externally to a process and asynchronously to its execution, yet caused by that process. Faults include illegal operations, memory faults and priviledged operations. A *break* is the asynchronous halting of a program's activity. An *exception* is an abnormal condition detected by the process.

On some systems, faults and breaks are handled by a software interrupt or signal as mentioned in Section 3.3.4. On one of these conditions, the process is sent a signal to which it can respond as it chooses. The process is destroyed if it has not provided for handling the signal. For exception handling, one technique is for the function detecting the exception to return an error status indicating the error condition. This can require passing the error condition back through several levels of function calls before the error condition is acted upon. Another technique is to provide a means of breaking out of the hierarchic control flow specified by the process's call graph of functions. For example, the setexit and reset functions of UNIX provide a non-local goto mechanism. The execution of reset causes a simulated return from the last call to setexit, assuming there was a call to setexit executed in a function that is an ancestor to the reset in the process call graph.

Faults, breaks and exceptions can be handled by multi-process structures with process destruction using the following principles. First, each process that encounters a fault or exception, or that is the subject of a break, is destroyed. In the case of an exception, the process first leaves an indication of which exceptional condition was encountered. (It seems desirable to extend this to fault handling in the future.) Second, if the program is to continue running after one of these conditions, the part to remain is designed as separate processes from the processes to be destroyed.

These principles are applied to implement break and exception handling in Thoth programs. Faults are handled by destroying the team (i.e. all the processes on the team) of the process incurring the fault. Process destruction is used to handle breaks as follows. Each team has a associated teletype. Each process is either breakable or unbreakable. The function .Breakable makes a process breakable or unbreakable depending on its argument. If a process is breakable, it is destroyed when a break is generated by its associated teletype. Generally, all processes on a team are breakable so a break on the team's teletype destroys the entire team; however, an interactive program may need to halt the execution of the current command and return to command level or translate the break into some action without the whole program terminating, so not all its processes would be breakable.

The Thoth text editor is an example of the use of multi-process structures for break and exception handling. It is a simple, line-oriented text editor with a user interface similar to Edit of *Software Tools* (Kernighan and Plauger (1975)). The text is stored in a buffer which is maintained in memory. As an interactive program, the text editor must handle exceptions caused by incorrect command lines, running out of memory, unsuccessful pattern searches, and the user pressing break. After any of these conditions, the text editor must continue running with the text buffer in a consistent state.

The text editor is structured as two processes on one team. The first process is the proprietor of the text buffer and performs all text buffer operations. (This process is referred to as the Main process because it is created executing a function called Main.) The Main process guarantees the integrity of the text buffer; it is not destroyed by an exception or a break. The operations on the text buffer are performed at the request of a second process (referred to as the Editor process because it is created executing a function called Editor). The Editor process reads command lines and text from the input, parses command lines, and performs the commands,

The Main process is unbreakable while the Editor process is breakable so only the Editor process is destroyed on a break. Because the Editor process executes the commands, a break causes the execution of the current command line to stop. The text buffer remains consistent because it is maintained by the Main process. The Main process detects when the Editor process has been destroyed and creates another Editor process which proceeds to read a command line from the input. Thus, the handling of breaks appears to the user as that of the Editor immediately returning to command level.

This structure is also exploited in handling exceptions. On detecting an exception, the Editor process destroys itself after leaving a pointer to an error message in a global variable indicating the exception. The user is notified of the exception by the next incarnation of the Editor process, which prints the error message.

This discussion is clarified by considering the source code for the text editor. When the text editor is invoked, the system creates a new team from the code and the initialized data stored in a file associated with the text editor. The new team is initially one process which exeecutes a standard function which opens the default input and output and calls the function

```
Main( optc, argc, cmd_vec )

    e - The Thoth Text Editor
{
  extrn Editor_id;
  auto  requestor, request[MESSAGE_SIZE];

  if( optc > 0  ||  argc > 1 )
    {
      .Err_msg( "use: e [ <pathname> ]" );
      .Exit( 1 );
    }
  Initialize( optc, argc, cmd_vec );
  .Breakable( FALSE );

  repeat
    {
      requestor = .Receive( request, Editor_id);  (*)
       Editor does not exist first time
      if( requestor == 0 )
        {
          Recreate_editor(); next;
        }
      select( OPERATION[request] )
        {
```

```
            case APPEND_LINE: Append_line();

            case DELETE_LINES: Delete_lines( request );

            case COPY_LINES: Copy_lines( request );

            case NORMALIZE_LINES: Normalize_lines();

            case SUBSTITUTE_LINE: Substitute_line( request );

            case SET_CURRENT_LINE: Set_current_line( request );
           }
          .Reply( request, requestor );
        }
    }
```

The first time .Receive is executed at (*), the Editor process does not exist so requestor is set to zero and Main invokes the Recreate_editor function, which creates a second process executing the function

```
Editor()
 {
  extrn Message, Index, Command, Error_messages, Reserved_core;
  extrn Input, Output, Output_mode, Flush;

  .Select_input( .Open(Input, "r") );
  .Select_output( .Open(Output, Output_mode) );
  .Put_str( Message ); ++Flush;
  Message = Error_messages[BREAK];

  .Breakable( TRUE );
  Reserved_core = .Alloc_vec( RESERVE_WSIZE );
  .Reset_option( .Select_input(), ECHO_NULL_LINES );

  repeat
   {
    Execute_command_line( NON_GLOBAL );

     Get a new command line
    if( Flush ) {.Flush(); Flush = 0;}
    Index = 0;
    if( .Get_str(Command, MAX_LINE) == '*0' )
       .Exit( 0 );

    if( Command{0} == '!' ) Error( ESCAPE );

    if( Skip_blanks() == '*0' )
     {  Print the next line on null line
       .Concat( Command, ".+1p" );
       Index = 0;
     }
   }
 }
```

Notice that the Editor process prints the message from the previous Editor process and then sets it to the break message. The Editor process responds to exceptions and sets the error message with the function

```
Error( msg_number )
 {
  extrn  Message, Error_messages, Editor_id;

  Message = Error_messages[msg_number];
  .Destroy( Editor_id );
 }
```

The Main process also invokes Error to halt the Editor process when there is no memory available to grow the buffer. This is the only exception detected by Main.

This structure has several advantages. The error handling is elegant in the sense that it is local to where the error was detected except for recreating the Editor process and printing the error message. This contrasts with the use of error codes that are typically handled and returned through several levels of function invocations. The break handling is elegant because there is no need for any synchronizing code with respect to breaks and there are no critical sections. The Main process simply makes itself unbreakable and the Editor process makes itself breakable. This structure is portable because it only relies on message-passing, multiple processes and process destruction which are largely machine-independent in implementation in Thoth and which could be implemented in other systems as well. In contrast to the reset scheme of UNIX, we claim this structure is logically clean because the hierarchic flow of control is not violated, just halted. The reset scheme of UNIX is also less portable because it relies on being able to reset the stack to the setexit call, which is difficult on some machines (Johnson (1978)).

The structure can be applied to many interactive programs for exception and break handling. The functionality of the program is implemented by a process corresponding to the Editor process and the data structures that must remain consistent over breaks and exceptions are maintained by a process corresponding to the Main process. Conceivably, the process corresponding to the Main process could function solely to recreate the first process after breaks and exceptions.

The handling of faults in Thoth is so far very simple, namely destroying the team of the process incurring the fault. For more sophisticated machines with floating point arithmetic faults, it seems desirable to only destroy the process incurring the fault and provide some means for other processes to determine why that process was destroyed.

4.6 Agents

A *distributed system* is two or more machines connected so that they can communicate and utilize each other's resources. An instance of such resource sharing is a distributed file system can access files of the others.

Connecting multiple machines introduces the non-local use of system resources. This raises questions of ownership, reclaimability and cost, both in size of code and reliability. Thoth currently has no concept of an external system, or any mechanism

for handling this. Moreover, there are many places where advantage is taken of a process being local. For example, a process's process descriptor is often used in the system to directly access information about the process. One approach is to extend the operating system to handle external use of resources in a general fashion. This increases the size and complexity of the system and introduces several problems regarding the external process's interface to the system. An alternate approach is to restrict the knowledge of the external use of resources to one local process, called an agent. An *agent* is a process whose function is to act as a representative of other machines, and to use the resources of its machine on their behalf.

An agent, as a local process, uses the standard mechanisms for interacting with the rest of the local system, including allocating resources. Similarly, the local machine views the agent as a local process so it need not know about the external use of the resources and it retains full control of these resources via the normal mechanisms. Therefore, no special mechanisms are needed in the rest of the local system.

The use of an agent also simplifies the interface between machines because each machine only deals with one process on the other machine. That is, the agent for machine A that resides in machine B is, from the point of view of machine A, the proprietor of machine B because all requests to access any of machine B's resources from machine A must be sent to this process. For example, in a distributed file system, to access a file on another machine, a request is sent to the agent in that machine who attempts to access that file and if successful, returns the necessary file information and data.

The notion of agent is applicable to programs extending over separate address spaces. A program can be structured as several teams with teams having agents on other teams.

The concept of agent has not been implemented for communication between machines yet; however, the structure of the operating system can be viewed as using agents. That is, the system processes act as agents for the user processes. We explore the structure of the system further in the next chapter.

5 Multi-Process Structures in Operating Systems

This chapter considers the application of multi-process structuring to operating systems by describing the multi-process structure of the Thoth operating system. Multi-process structuring is well-suited for use in operating systems because of the asynchronous and parallel operation of peripherals and the logical concurrency of programs that results from multi-programming. This type of structuring also provides further techniques for dealing with the problems of synchronization, deadlock avoidance, resource allocation and reclamation, protection and system interfaces. In the course of the discussion, we abstract several principles from the design; we also note problems with the current structure and propose solutions. The overall structure of the system is considered first.

5.1 Overall Structure

The operating system is structured as two distinct levels of implementation. The kernel, the lowest level, implements a basic abstraction. It consists of a set of data structures and functions that operate on these data structures. The multi-process structure, the second level, consists of system processes which execute within the basic abstraction and which implement several internal levels of abstraction that together implement the external abstraction. The system processes create various system data abstractions and functional abstractions from hardware resources and abstract resources made available by other system processes.

The services provided by system processes are made available to user processes (as well as other system processes) via the message-passing. Rather than being aware of sending a message to a system process, the user invokes a function supplied by the system library that prepares a message and sends it to the appropriate system proprietor. Thus, even though communication with the system proprietors is implemented with message-passing, the presence of system processes and message-passing communication can be transparent to user processes because of these library functions and the "subroutine" semantics of .Send.

The use of message-passing as a means of communication between address spaces eliminates the need for a special mechanism to implement "system calls" other than the message-passing primitives. Restricting all interaction with the system to the message-passing can lead to a more secure system because the interface is more specific. That is, assuming the message-passing primitives are secure, a user process can only access system resources by sending to a system process. Then, one need only show that no system process can be "out-smarted" by a strange message to show that the system is secure.

System processes act as agents for the user processes in several ways. They give user processes (indirect) access to the system space and resources. In theory, this means that user processes do not even have to be on the same machine. System processes also execute at system priority and are synchronized with other system events by the relative indivisibility property. The sender need not be in memory for the proprietor to act on its behalf for some operations, thus reducing interaction with the swapping mechanism. Finally, system processes execute in privileged mode making certain instructions available that are not available to user processes.

System calls in Thoth are implemented as a combination of the traditional function calls (that trap on a multi-team system) and via the message-passing. We hope to evolve the system to use message-passing for most of the system calls. For example, currently most of the I/O-related system calls are implemented as traditional trap-type functions calls. They would be much better implemented through the message-passing directly. However, it is not clear that it is possible or entirely desirable to restrict all interaction with the system to message-passing. For relatively expensive system facilities like accessing a file, the cost of the message-passing is small in relation to the total cost plus the message-passing allows the use of the proprietor structure with advantages as described in Section 4.2. Also, accessing a file is performed with system data structures only so the error checking is simple because the file system can trust system data structures. In contrast, implementing .Alloc_vec by message-passing means that the system process doing the memory allocation must do

extensive error checking because it is operating on a data structure in the user's address space. Moreover, the cost of the message-passing becomes a significant proportion of the cost of allocating a vector. Finally, because .Alloc_vec is a simpler operation, the advantages of doing it by proprietor are less significant.

Several types of processes are used in the system. At the lowest levels are the device proprietors. A *device proprietor* is a proprietor of a device. The device proprietor owns the device interface in the sense that it is the only process that accesses the device interface and waits for interrupts from the device. The device proprietor implements an abstract device based on the hardware it owns and makes the services of this device available to other processes via the message-passing. The abstract device is designed to be simpler and more portable than the real hardware. The device proprietor can exploit the facilities of the Thoth abstraction on behalf of the device. For example, several device proprietors send to the team proprietor to request that the team of their current requestor be swapped into memory to give them access to that team's data segment. However, as discussed in Section 3.1, using processes to handle interrupts requires fast activation of these processes.

The conventional design of device handling routines uses interrupt routines invoked directly by the hardware mechanism for handling the interrupts and managing the device activity. One set of routines initiates device activities and the other set performs the necessary actions to continue the device operation after an interrupt signalling the completion of an operation. This scheme has the disadvantage of being less uniform than using device proprietors because communicating with the device is usually unlike the message-passing. Also, a device proprietor has the facilities of the process abstraction available to it for performing its functions while interrupt routines would not. Finally, our experience has been that device proprietors are easier to understand and to program.

Above the device proprietors are processes that use the services of these device proprietors. For example, the file system uses the services of the two disc drive proprietors to implement the file system abstraction. Similarly, the team invoker uses the file system when invoking teams.

The structure of the system in terms of processes is given in Figure 1. Each block is a process and a "*" or "+" indicates a process that receives messages and is thus a proprietor. Only the RTC process and the swapping alarm clock are not proprietors. The system given is configured with two disc drives and one teletype although Thoth will support more disc drives and teletypes, as well as other devices. User processes send messages to any of the processes marked with a "*".

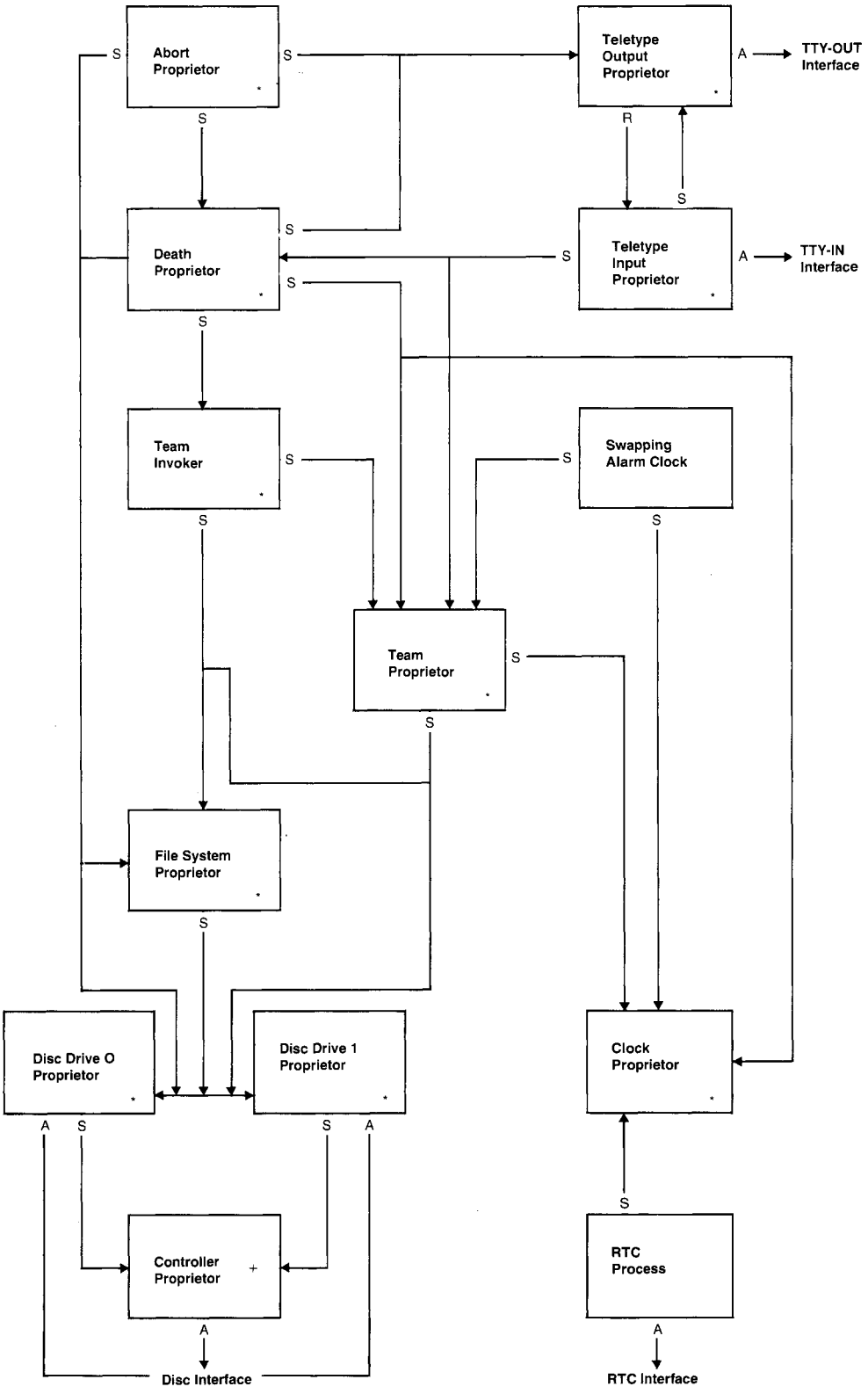

Figure 1: Blocking Graph of Thoth

Legend: Each block is a system process.

* – receives messages from user processes
\+ – receives messages from priviledged processes only
A – blocks on interrupt (.Await_interrupt)
S – blocks on process by .Send
R – blocks on process by .Receive – specific

In the following sections, we discuss the structure of the kernel and the particular process structures used in Thoth.

5.2 The Kernel

The basic abstraction implemented by the kernel includes:

1. processes and process identification
2. message-passing
3. interrupt abstraction
4. dynamic memory allocation
5. dynamic process creation

The implementation of these facilities was discussed in Chapter 3 and is further developed in Chapter 6 in the discussion of verification.

The structure of this level of the software is conventional in the sense that the kernel consists of a set of functions acting on shared data structures and interrupts are disabled to achieve the required mutual exclusion. The structure is necessarily primitive because the facilities of the full abstraction are not available at this level. Therefore, it is appealing to implement the minimal facilities required at this level and use the multi-process structure to implement the rest. These minimal facilities must therefore include processes, process identification and message-passing. The interrupt abstraction is included because it involves process switching. The memory allocation is included for reasons described in Section 3.4.1, primarily for simplicity and efficiency. Process creation is part of the Thoth kernel largely for historical reasons although having it there has efficiency advantages.

The memory allocation and process creation could be removed from the kernel by having these services provided by proprietors. This would add to the cost of these operations and to the code required to implement them as well as potentially introducing logical problems. For example, the memory proprietor could not handle growing the team's data segment to satisfy an allocation request without depending on the swapping mechanism which, as it is currently implemented, uses the memory allocation itself. Process creation seems like a better candidate for implementing via a proprietor although the operation is actually sufficiently short that disabling interrupts is adequate for the required indivisibility. However, because process destruction is implemented by a system process, it would seem more consistent to have this same process perform process creation and make it the process proprietor.

In general, the kernel should present an abstraction that processes use without violating (by directly accessing kernel data structures) and the facilities of the kernel should be as efficient as possible. One might also like all operations of the kernel to be executed with interrupts disabled to simplify their behavior and verification.

While the Thoth kernel is fairly efficient, the basic abstraction of the kernel is violated by both the swapping and the process destruction mechanisms. Also, not all instructions have interrupts disabled for the full duration of their execution. This is done in order to achieve a small, bounded maximum disable time. We have been evolving the kernel more in the direction of removing abstraction violations and undisabled operations; however, it is not clear how to best resolve the abstraction violations. For instance, because the process descriptor is a data structure internal to the kernel implementation, process destruction should be a kernel operation, yet this is much easier to implement as a process. Moreover, to reduce the size of the kernel, it

was suggested earlier that process creation could also be done by a proprietor, thus adding another abstraction violation. These issues deserve considerably more investigation.

5.3 Real-time Clock

The real-time clock abstraction is implemented by two processes. One process, the *clock proprietor* receives messages to:

1. wake up processes whose time has expired.
2. set the time to a specified time.
3. return the current time in the reply.
4. cause the requesting process to sleep (or block) until the specified time.
5. cause the requesting process to delay for the requested time period.
6. wake up a specific process even though its time to delay or sleep has not expired.

Thus, .Sleep, .Delay, .Get_time, .Set_time and .Wakeup are implemented as sending messages to the clock proprietor. The clock proprietor maintains a record for each process that is sleeping or delaying that indicates the time to wake up this process. The clock proprietor leaves a process requesting to sleep or delay blocked for the required time interval simply by not replying to its message until the time has expired.

A process delaying must delay for the specified time period independent of the absolute time being changed by .Set_time. Thus, when the time is changed by .Set_time, the clock proprietor recalculates the remaining delay time for each delaying process and forwards the message of each delaying process to itself (the clock proprietor) with a message to delay for the remaining period of time. This allows the clock proprietor to use its normal delay mechanism for adjusting the delay of these processes.

The clock proprietor relies on two services. It relies on a vector being maintained with the current date and time and it relies on receiving a message to wake up processes when the wakeup time of the next process to wake up is equal to or earlier than the current time. These services are provided by the RTC process.

The RTC process is a process that awaits the periodic interrupts generated by the real-time clock (RTC) hardware. On each interrupt, it updates the current time and effectively sends to the clock proprietor if the current time is now equal or later than the wakeup time for the first process to be awoken. It is necessary to limit the disable time and restrict the amount of activity at the RTC process's priority or higher to guarantee that the proper frequency of clock interrupts is achieved so the correct time can be maintained. For the same reason, it is necessary to guarantee that the RTC process only blocks on interrupts. Thus, the "effective" sending to the clock proprietor is accomplished by violating the abstraction and not by the normal .Send mechanism. If the normal .Send mechanism was used, the RTC process could remain blocked on the clock proprietor for an unbounded period of time while the clock proprietor served various requests and not be ready for the next clock interrupt.

This violation could be avoided by updating the current time in the interrupt handler and only having an interrupt occur in the abstraction when the current time is equal to or past the next wakeup time. This simulates a hardware clock with two registers: one is maintained as the current date and time; the other is the wakeup regis-

ter containing a later time. This hardware generates an interrupt when the contents of the current time register are greater than or equal to the contents of the wakeup register. Thus, the RTC process could be replaced by a process that awaits this "wakeup" interrupt and sends to the clock proprietor by the normal mechanism.

All the services of the clock are implemented by messages so these services would be immediately available to another machine if the message-passing was implemented between the two machines.

5.4 Block I/O

Each block device has an associated device proprietor that receives messages for device service. The device proprietor is deemed to own the interface to the device but not the device itself. For example, the controller proprietor does not own the blocks on the disc or the heads on the drives.

The process structure can be made more sophisticated if the controller controls more than one device. The system in Figure 1 is configured for a disc controller that controls two disc drives. There is a controller proprietor process that owns the disc controller plus two disc proprietors that each own their respective drives which they access by sending to the controller proprietor process. This structure allows a seek on one drive concurrent with disc operations on other drives providing the disc controller hardware allows this. That is, a disc proprietor can request a seek on its drive and then await the completion of the seek by waiting for the seek complete interrupt itself. Meanwhile, the controller proprietor is free to accept requests for services on other drives.

This structure can also be used for several tape drives attached to one tape controller, allowing a tape drive to rewind concurrently with tape operations on other drives. The drive proprietors plus the controller proprietor form an interesting multiprocess structure but this structure has provided little gain in performance. That is, overlapping of seeks has not brought a noticeable gain in performance.

5.5 Teletype I/O

A teletype is effectively two devices, one the input and the other the output. Therefore, it is handled as two processes. The teletype output proprietor receives messages containing characters to output to the teletype. The message also contains a specification of options to be used on output such as whether tab characters are to be expanded. That is, blanks are inserted to achieve the indicated tabulation. If the teletype is paged, it outputs at most a screen or page of characters between receiving characters from the teletype input proprietor. This is implemented by the output proprietor doing a receive-specific from the input proprietor when it reaches this limit.

The teletype input proprietor awaits interrupts from the keyboard and builds a line of characters. Other processes send to the input proprietor to receive a line of input. The teletype input proprietor sends to the output proprietor to have characters echoed, if required. Because a line of input does not in general fit into a message, the teletype input handler uses .Transfer to transfer the input line into the requesting process's buffer.

The teletype input proprietor also handles breaks. A break is implemented by the teletype input proprietor destroying every process that is marked as breakable for that teletype. To accomplish this, the teletype input proprietor sends to the death proprietor. Thus, the input proprietor uses a number of system services on behalf of the teletype input hardware.

5.6 File System

The file system is implemented as a proprietor that owns the disc blocks and the file system data structures on disc and in memory. (It owns the disc blocks as containers while not necessarily owning their contents.) It receives messages to perform file system operations such as open file, close file, create file and so on. This structure was used as an example in Section 4.2. File operations involve disc I/O which the proprietor performs by sending messages to the disc proprietors. Thus, it implements an abstract resource consisting of a set of files using the abstraction of the discs provided by the disc proprietors.

The file system has several problems that we hope to fix in the future. The file system proprietor restricts concurrency of file system operations when the file system is spread over several independent drives because only one operation is performed at a time (because there is only one file system proprietor). Most of the time to perform an operation is spent waiting for disc I/O so operations on separate drives could be performed concurrently with considerable benefit. This is partially alleviated by the fact that the disc I/O for reading and writing a file is not done by the file system proprietor, but is done directly by the appropriate disc proprietor. This is reasonable because file I/O involves changing the contents of files and not the file system structure so it does not conflict with the file system proprietor's ownership. However, to protect the file system and files against unauthorized disc I/O, it is necessary to check that the requested disc I/O is permitted for the requestor by the file system. Unfortunately, this protection requires that the disc proprietor access file system data structures thus violating the exclusive ownership of these by the file system proprietor. Consequently, this protection is not currently implemented. The file system also needs to transmit larger amounts of data between teams than allowed by the message-passing primitives, and it has not been changed since before the introduction of .Transfer so its current state still reflects some early design flaws.

Several improvements are planned for the file system. We plan to allow several file system proprietors, each owning some disjoint subset of the disc drives. This would allow concurrent file system operations and make the reading and writing of files going through the associated file system proprietor feasible thus solving the protection problem. We plan to change the file system and swapping so requestors can be swapped out of main memory while most file system operations take place. The file system proprietor will then act as a high priority memory resident agent for user processes. We also hope to make part of the file system processes swappable thus reducing the amount of memory dedicated to the file system.

In summary, the file system is currently implemented as a simple proprietor and this process structure works well in small systems. It is not adequate as it stands for larger configurations, but this will be remedied.

5.7 Process Destruction

The death proprietor receives requests to destroy processes and sends messages to other system proprietors in the course of reclaiming the resources of the destroyed process. Thus, .Destroy is implemented as sending a message to the death proprietor.

The death proprietor resolves concurrency problems arising from two or more processes attempting to destroy the same process. It also acts as an external agent for a process wanting to destroy itself. It would be awkward for a process to directly destroy itself because a process cannot exist without resources so it cannot logically release all its own resources.

The death proprietor uses the standard relinquish functions to release the resources of the process it is destroying. This causes it to send to various system proprietors. For example, it sends to the file system proprietor to close files. The relinquishing of resources can be slow if it requires closing files and so the death proprietor can be a bottleneck in a large system. Worse still, the death proprietor can block indefinitely on closing the printer file when the printer is out of paper. A possible solution is for the destruction proprietor to modify each process it is destroying to close its own files and free its own memory. This process would then send to the death proprietor which finishes destroying the process. This would aid in solving another problem. Currently, the death proprietor must free memory into the team free list of the process being destroyed. If the free list is inconsistent, it is very hard to guarantee that the death proprietor cannot memory fault. The process being destroyed could memory fault without affecting the death proprietor. We plan to further investigate this design once the file system is changed.

5.8 Team Invoker and Team Proprietor

Teams are implemented by the team invoker and the team proprietor. The team proprietor owns the data structures involved in implementing teams, namely those describing the memory allocated to teams, the swap file, and the team descriptors. The team invoker acts as an intermediary between the user and the team proprietor when invoking a team. That is, the team invoker receives requests to invoke teams. It locates the file containing the code for the team plus initialized data, requests the creation of a team address space from the team proprietor, loads the file into this team space and creates the first process on the new team. The team proprietor receives messages requesting:

1. creation of a new team address space.
2. reclamation of team resources allocated to a team whose processes have all been destroyed.
3. swapping into memory of a team currently swapped out.
4. reordering of priorities of teams in the ready queues and in the list of swapped teams (which can result in swapping).
5. transfer of data between two team spaces. That is, .Transfer is implemented as sending an appropriate message to the team proprietor.
6. modification of team attributes.
7. growing of the team's data segment.

The team proprietor requires real-time input to drive the swapping and CPU allocation. This is provided by the swapping alarm clock which repeatedly delays for a fixed time period and sends to the team proprietor requesting a reordering of the teams. The reordering causes teams that are swapped out to be considered for being brought into memory and it changes the priorities of teams in memory. This causes a cycling of teams through memory and through the various levels of CPU preference. The cycling is disrupted by the team proprietor receiving requests to bring in a particular swapped team. For example, the disc proprietor needs to ensure that the data segment of the requestor's team is in memory to do direct memory access between the disc drive and a buffer in the team's data segment. In this sense, the swapping and CPU allocation are driven both by real-time input and various system events.

The swapping is a level of memory management and is best implemented with as little dependence on other facilities as possible. This motivated the separation of the team implementation into the two processes. The team proprietor allows more system processes to send to it without the danger of deadlock because it only depends on the swapping-device proprietor and the clock proprietor. Thus all other processes can safely send to the team proprietor without the possibility of deadlock. Also, the services of the team proprietor are performed independently of the file accessing done by the team invoker. For example, the swapping is done independently of the file system.

As a result of several design errors as well as historical considerations, the file system needs to access its requestor's data segment. Therefore, the file system needs to get the requestor's team in memory plus lock the team in memory while the file system operation is performed.

It is hoped that the system could be ported to a machine with paging hardware and that a page fault process would handle page faults in this configuration. That is, the routine invoked by a page fault trap would simulate a send by the process incurring the page fault to the page fault process, requesting that the page be swapped into memory. We then must require that the page fault process and all the facilities it uses cannot page fault.

5.9 Summary

The multi-process structuring of Thoth has been successful in making the system relatively easy to implement and to understand. For example, timing-dependent errors have not been a problem. These claims are supported in the next chapter in considering the feasibility of verifying the system.

There are a number of problems with the current implementation of Thoth but nothing that is intrinsic to the techniques used. The I/O and file system are currently the most inadequately designed parts of the system. However, a new file system has been designed which we hope to implement to replace the current rather archaic system.

Structuring the system as processes that only communicate via messages suggests the possibility of distributing the system over several machines. Currently, Thoth is not sufficiently pure for this because communication takes place outside the message-passing. Also, assumptions are used in the system such as relative indivisibility that assume execution on one processor. However, the system is evolving to remove some

of these problems.

We expect that an advantage of Thoth for a distributed environment is that the structure would facilitate connection of disjoint Thoth systems running on separate machines. In this regard, we mentioned previously the use of agents to implement a distributed file system.

6 Feasibility of System Verification

In this chapter, the feasibility of verifying the operating system is discussed. Verification is used to mean the development of formal or semi-formal arguments that properties of interest are true for the system.

Exploring the verifiability of the system has several motivations. First, verification involves developing a framework in which to reason about and to understand the operating system, leading to greater confidence in its behavior. Second, ease of verification is one measure of goodness of the design. The work reported here has lead to the recognition of difficulties in verifying the system caused by flaws and inconsistencies in the design. Finally, the verification attempt has lead to the discovery of several errors in the system and errors in our understanding of the system's behavior. This points out the value of verification in addition to testing; exhaustive testing is impractical and many problems are difficult to generate and detect with test data. Moreover, with a portable system, testing may establish a certain level of functionality on one machine, but the system may fail to run at all when ported to another machine through no fault of the porting.

The verifiability of the system is explored by developing an outline of a verification of the system. A complete verification seems to be impractically large, at least in this thesis. Rather than describe the outline in total, the discussion focuses on techniques of interest and particular problems encountered, avoiding the bulk of system details.

A strategy for the verification is needed to handle the large size and non-determinism of the system. Size is handled by verifying a small, simplified version of the system first, then adding further components and considerations, and arguing that the extension does not disturb the earlier verification arguments. This progressive extension of the verification is useful for handling several apparent circularities in the design. Non-determinism, which is due to interrupts, is handled by mapping it onto the process switching, thus appearing as the concurrent execution of many processes, each one deterministic.

Verification of parallel programs is difficult and not currently well understood so the outline tries to reduce as much of the verification as possible to units that can be verified using sequential program verification techniques. In general, the outline does not develop most topics past this point.

The discussion provides: a description of the verification as a series of smaller

tasks; some of the significant definitions and lemmas; and mention of certain sources of difficulty. Closely following the structure of the system, the verification consists of two main parts, the kernel and the multi-process structure.

6.1 Kernel

Verification of the kernel involves showing that the semantics of the kernel primitives are correctly implemented and verifying properties of objects in the basic abstraction implemented by the kernel. This abstraction was described in Section 2.2; its implementation was discussed in Chapter 3 and Section 5.2.

Ideally, processes should only use this basic abstraction and not access kernel data structures directly (violate the abstraction) so that the kernel can be treated as a "black box" once verified. The problem is that the basic abstraction is violated by several processes. For example, the death proprietor operates on process descriptors when destroying processes. This leads to a circularity in the implementation because processes are implemented by the kernel. If the process manipulates the kernel data structures, the process depends on the kernel for its correct behavior and the kernel similarly depends on the process. A similar circularity problem was observed in the Multics supervisor by Schroeder, Clark and Saltzer (1977). Their solution was to try to remove all such loops. As mentioned in Section 5.2, the desire to have the kernel as small and simple as possible plus the desire to use processes to implement most facilities leads to implementing facilities via processes that would otherwise be part of the kernel, thus introducing more circularities.

The approach taken here is to ignore these violations during the verification of the kernel and introduce them later, arguing that their introduction does not invalidate the earlier arguments. Thus, the verification of the kernel should be structured so that the arguments are not invalidated when the later complications are introduced.

The alternate solution of trying to eliminate the violations by providing the required operations as part of the kernel seems to lead to a much larger kernel as well as unbounded maximum disable time.

Verification of the kernel seems most easily handled by reducing it to a series of sequential verifications. We first assume that there are no interrupts so all operations are executed sequentially. Interrupts are then introduced with the assumption that the previously discussed operations are executed with interrupts disabled so the verification of these operations remains valid. We then discuss relaxing this assumption. This provides a scheme for verifying the semantics of the primitives. Verifying properties about objects in the basic abstraction involves proving properties about the kernel data structures. Some properties, such as the sequentiality of processes, follow from considering the operations that affect the relevant data structures. Because these operations are indivisible, they can be considered sequentially. Other properties, such as the minimum recycle time for process id's, cannot be shown just from the kernel, but require (in the case of process id's) also considering the death proprietor.

Three terms required in the following discussion are that of invariant, data structure consistency and machine instruction.
Definition: An *invariant* is an assertion about an object that is true for the object over its lifetime although it may be violated while the object is being changed.
Definition: A data structure is *consistent* at a given time if its invariants are satisfied

at that time.
Definition: A *machine instruction* is a basic machine state transition that is executed indivisibly with respect to interrupts when interrupts are enabled.
An interrupt can occur between instructions but not during an instruction. (Many machines have interruptable machine instructions but we take the liberty of assuming that these instructions would not be used in the kernel.) Error traps that are invoked by the execution of the instruction, such as illegal operation traps, are not included because the process that caused the error trap does not continue after the trap.

Until further notice, it is assumed that no interrupts occur.

6.1.1 Process Implementation

The main difficulty in verifying the process implementation is making the transition from viewing the processor as executing functions to viewing processes as executing functions. This is done by defining what it means for a process to have executed an instruction, and proving properties about the set of instructions executed by a process. In more detail, the strategy is as follows. As mentioned in Section 3.1, processes are implemented by process descriptors, a ready queue structure and process stacks, plus process switching operations on these data structures. Various properties can be proved at this point using sequential techniques because the processor executes these operations sequentially.

One approach to verifying the kernel is to extend this simulation view to the entire verification, that is, to verify the entire kernel as a simulation performed by the processor executing operations on data structures. However, some operations block and some operations are interruptable making them hard to treat in this sequential fashion. Therefore, the approach taken is to verify primitives as being executed by processes, starting at the earliest possible point in the verification.
Definition: The *state of a process* is its eligibility for execution.
A process is either eligible for CPU allocation, *ready*, or its state indicates the reason for ineligibility, namely why it is *blocked*. The state of a process is indicated by the value of the STATE field in its process descriptor. All the other states are blocked states and are discussed later. The process switching operations are:

1. .Block() - block the active process and activate one of the ready processes according to the CPU allocation rule. The STATE field of the active process is assumed to be set to indicate the new blocked state.
2. .Add_ready(pd) - make the state of the process described by pd ready and activate this process if required by the CPU allocation rule. That is, if this process is of higher priority than the current active process, it is activated.
3. .Block_and_add_ready(pd) - block the active process, make the state of the process described by pd ready and activate one of the ready processes according to the CPU allocation rule. The STATE field of the active process is assumed to be set to indicate the new blocked state of the active process.

Once the semantics of these operations are verified, it is useful to regard the processor as executing the process switching operations as single instructions. This allows the following definition to be unambiguous.
Definition: A process is said to have *executed* a given instruction if the processor executed the instruction when that process was the active process.
Condensing the process switching to a single instruction means that the transition in-

volved in a process switch occurs in the course of that one instruction. Thus, a process is defined to be active from the instruction after the process switching instruction that activated it to the process switching instruction that activates another process including this last process switching instruction.

Finally, it is necessary to show that the process executes sequentially and deterministically. There is a one-to-one correspondence between the instructions executed by the processor and the instructions executed by the process between process switches. Thus, a process is sequential and deterministic between process switches because the processor is sequential and deterministic. The argument is completed by showing that:
Lemma A: The next instruction executed by a process after executing a process switching instruction is the instruction following that process switching instruction.

At this point, primitives can be considered to be executed by a process rather than by the processor. By the above, we can continue to apply sequential verification techniques.

In a complete verification, it would be necessary to verify the implementation of the CPU allocation rule given in Chapter 2 and the process identification scheme (see Section 3.2). They have the property that they do not involve blocking, making them relatively easy to verify. The message-passing primitives, which do block, are considered in the next section.

6.1.2 Message-Passing

The next step in the verification is to argue that the semantics specified for the message-passing primitives are those actually implemented. It is attractive to consider the message-passing primitives as operations on process data structures and then treat them as single instructions executed by a process, instead of treating the lower level process switching operations in this fashion. The problem is that some of the message-passing primitives block and then continue executing, so they are not indivisible, while indivisibility is required by the definition of instruction. As a consequence of the blocking, the semantics of an individual primitive cannot be derived from just the code for that primitive; its interaction with other primitives must be considered.

To verify these primitives, we first consider the semantics of the different message-passing blocked states. Then, the blocking primitives of .Send and .Receive are verified from the semantics of their code, using the semantics of the states in which they block to complete the semantics of the primitive. The semantics of .Reply and .Forward are straight-forward because they do not block.

When a process blocks, it blocks until another process performs some action to unblock it. If this action must be performed by a particular process (as when sending to a process), the blocked process is said to be *blocked on* this process. The "blocked on" process is the only process that may act on the blocked process to unblock it or change its state. Two conditions are defined for each blocked state.
Definition: The *block condition* of a blocked process state is the condition or assertion that is true of a process blocked in this state.
Definition: The *unblock condition* of a blocked state is the condition or assertion that is true when a process in this state becomes ready (or unblocks).
The BLOCKED_ON field of the process descriptor indicates which process the

blocked process is blocked on. If this field is zero, any process may act on the blocked process. For example, this is used to discriminate between the "receive any" and the "receive specific."

The code for .Send, .Receive and .Reply is included at this point for reference in the following discussion of verification. Note that .Copy is a subroutine that takes its first argument as a pointer for the destination, the second argument as a pointer for its source and copies the number of words specified by its third argument plus one from the source to the destination.

```
.Send( message, receiver_id )
 {
  extrn .Pd_bundle, .Id_mask;
  auto  receiver, pd;

  receiver = .Pd_bundle[receiver_id&.Id_mask];
  disable;
  if( ID[receiver] != receiver_id )
   {
    enable;
    return( 0 );  receiver does not exist
   }

  if( STATE[receiver] == RECEIVE_BLOCKED &&
     (BLOCKED_ON[receiver] == ID[active] ||
      BLOCKED_ON[receiver] == 0) )
   {  Receiver receives the message and is unblocked
     STATE[active] = AWAITING_REPLY;
     BLOCKED_ON[active] = receiver_id;
     .Copy( &MESSAGE[receiver], message, MESSAGE_SIZE );
     BLOCKED_ON[receiver] = ID[active];
     .Block_and_add_ready( receiver );
   }
  else
/   {  Enter the send queue of the receiver
     BLOCKED_ON[active] = receiver_id;
     STATE[active] = SEND_BLOCKED|QUEUED;
     pd = SENDERQ_TAIL[receiver];
     BLOCK_PREV[active] = pd;
     BLOCK_NEXT[active] = BLOCK_NEXT[pd];
     SENDERQ_TAIL[receiver] = BLOCK_NEXT[pd] = active;

     .Copy( &MESSAGE[active], message, MESSAGE_SIZE );
     .Block();
   }
(*) enable;

  if( !.In_data_area(TEAM[active], message, MESSAGE_SIZE) )
     .Abort( "bad parm" );

  .Copy( message, &MESSAGE[active], MESSAGE_SIZE );

  return( BLOCKED_ON[active] );
 }
```

```
.Receive( message; sender__id )
 {
   extrn .Pd__bundle, .Id__mask;
   auto  sender;

   if( !.In__data__area(TEAM[active], message, 7) )
      .Abort( "bad parm" );

   if( .Nargs() == 2 )
    { receive specific
      sender = .Pd__bundle[sender__id & .Id__mask];
      disable;
      if( ID[sender] != sender__id )
       {
         enable;
         return( 0 );
       }
      if( !(STATE[sender]&SEND__BLOCKED &&
                    BLOCKED__ON[sender] == ID[active] )
         sender = 0;
    }
   else
    { receive any
      sender__id = 0;
      disable;
      if( (sender=SENDERQ__HEAD[active]) == &SENDQ__EMPTY[active] )
        sender = 0;
    }
   if( sender == 0 )
    { No sender
      STATE[active] = RECEIVE__BLOCKED;
      BLOCKED__ON[active] = sender__id;
(*)   .Block();
      enable;
      .Copy( message, &MESSAGE[active], 7 );
      return( BLOCKED__ON[a sender__id = ID[sender.];
   enable;
   return( sender__id );
 }
```

```
.Reply( reply, sender_id )
 {
  extrn .Pd_bundle, .Id_mask;
  auto  sender;

  disable;
  if( (ID[sender = .Pd_bundle[sender_id&.Id_mask]] != sender_id) )
   {
    enable;
    return;
   }
  if( STATE[sender] != AWAITING_REPLY ||
      BLOCKED_ON[sender] != ID[active] )
   {
    enable;
    return;
   }

  .Copy( &MESSAGE[sender], reply, 7 );
  .Add_ready( sender );
  enable;
  return;
 }
```

The source code for .Forward is not included but is similar to that of .Send and can be derived quite easily. The following discussion is also simplified by not considering .Forward.

The states associated with the message-passing are send-blocked, receive-blocked and awaiting-reply. These states are described as follows in terms of their block and unblock conditions:

1. receive-blocked

   block condition - the process has executed .Receive and either the process's send queue was empty or a sender was specified that was not blocked on this process. Therefore, the process is blocked at (*) in the code for .Receive and the next action performed by the process blocked in the receive-blocked state will be to copy the 8 words of its message field into its message vector.

   unblock condition - only a .Send to this process will change its state, so when it executes its next instruction, a process has executed .Send to this process so there are 8 words from the process identified by BLOCKED_ON in its message field and the sender is awaiting-reply.

2. awaiting-reply

   block condition - a process awaiting-reply is blocked executing .Send such that the next instruction it will execute is at (*) in the code for .Send. The process that it sent to has received the message, but has not yet replied.

   unblock condition - the process that this process, the sender, is blocked on must execute a .Reply to the sender, putting the reply message in the sender's message buffer.

3. send-blocked

block condition - the process has executed .Send and is blocked in the send queue of the process identified by its BLOCKED_ON field. The next instruction that it will execute when unblocked is at (*) in the .Send code.
unblock condition - the only way the process unblocks is by the process it is blocked on executing a receive-any when this process is at the head of the send queue, or a receive-specific on this process. This process is then awaiting-reply and is unblocked according to the unblock condition for awaiting-reply.

The block condition for a blocked state is verified by examining the code that can put a process in this state, similarly for the unblock condition. These conditions are used as follows to verify the primitives.

For .Receive, there are the two cases to consider, the receive-any and the receive-specific. It is only necessary to discriminate between these two cases to argue that the correct determination of the sender is done. That is, in the case of the receive-specific, we need to show that the specified process is used. In the case of the receive-any, the first process (if any) in the send queue is used. There are two main flows of control to consider. First, the appropriate sender exists and is send-blocked on this process. By the semantics of the block condition of send-blocked, the process is in the send queue of the receiving process and its message is in its message buffer. With this, we can complete the verification of this flow of control.

If the sender is not available in the sense that the send queue is empty or a sender is specified that is not send-blocked on this process, the process executing the .Receive blocks in the receive-blocked state. By the semantics of the unblock condition for receive-blocked, a message will be available in this process's message buffer and the sender will be identified by the process id in the receiver's BLOCKED_ON field when the receive unblocks. This is used to complete the verification.

For .Send, there are two flows of control to consider, one in which the receiver is blocked waiting for this message (or any message) and one in which it is not. Considering the latter case first, the sender executes until it becomes send-blocked. The semantics of the unblock condition for send-blocked are used to complete the verification of .Send. In the second flow of control, the process blocks awaiting-reply so the unblock condition for this state is used.

The semantics of .Reply can be verified when verifying the unblock condition for awaiting-reply. The inclusion of .Forward only complicates the specification of the block and unblock conditions.

6.1.3 Interrupts

The introduction of interrupts and the interrupt abstraction requires arguing that previous arguments remain valid and that the semantics of .Await_interrupt are implemented correctly.

Initially, we assume that all primitives verified previously are executed with interrupts disabled so the assumption of absence of interrupts remains valid. To show that processes still execute sequentially and deterministically, it is necessary to show that:
Lemma A: The next instruction executed by the active process is independent of an

interrupt occurring before that instruction.
From this we conclude that all processes still execute sequentially and deterministically.

The interrupt abstraction is implemented by the primitive .Await_interrupt and the interrupt handler, which is an assembly-coded module invoked directly by the hardware interrupt mechanism. The verification of .Await_interrupt is similar to that of the message-passing because it uses .Block to block in the state awaiting-interrupt. It is only necessary to show that a process awaiting-interrupt is unblocked by the interrupt handler when the interrupt occurs. This provides the unblock condition for awaiting-interrupt to complete the verification of .Await_interrupt.

When interrupts are enabled, an interrupt causes execution of the interrupt handler, which performs a process switch to the process awaiting the interrupt. We refine the definition of a process executing to
Definition: A process is said to have *executed* a given instruction if the processor executed the instruction while the process was the active process and the processor was not executing the interrupt handler.
This does not disturb earlier arguments because, assuming the absence of interrupts as we were earlier, the processor would never be executing the interrupt handler. Lemma A can be verified by arguing that the active process is not considered to be executing after an interrupt occurs and that the interrupt handler correctly saves the volatile environment of the active process so the process will continue independently of the interrupt when it is activated again. Therefore, the non-deterministic behavior of the processor appears only as the asynchronous activation of processes awaiting interrupt, which does not affect the execution of other processes.

It is desirable to relax the assumption that certain primitives are executed totally disabled because this leads to long or unbounded disable times. It can be argued on an individual basis that reducing the disabling (to what it actually is) does not cause problems. However, the arguments appear harder to formalize because they involve the possible concurrent interaction of processes. Three cases are considered.

First, if part of the operation is local to a process or accesses variables that never change, there can be no interference and therefore, no need for mutual exclusion. Second, if the operation involves a potentially unbounded search of a data structure, for example finding a free memory block of sufficient size, the search can be executed in disabled steps with the data structure consistent between steps (where interrupts are enabled). Third, the blocking of a process includes a potentially unbounded search of the ready queue heads after saving the environment of the blocking process. Because this action was considered above as a single instruction, it is necessary to separate the blocking action from the search and consider the search to be executed by the processor and not by any process, just like the interrupt handler.

This completes the discussion of the kernel except for initialization, which is considered in Section 6.3. The discussion provides a framework for a verification; however, it seems to be difficult to handle the relaxing of interrupt disabling. It seems better to change the system so that primitives of the kernel are completely disabled rather than pursuing the verification of the system further as it now stands. For example, process creation could be changed so that the stack is supplied. Then it is feasible for it to be executed with interrupts disabled. The kernel is small so a complete verification is not infeasible, but it is a much larger task than can be undertaken here.

6.2 Multi-Process Structure

The multi-process structure of the system extends the basic abstraction implemented by the kernel to the full Thoth abstraction. Verification of the multi-process structure includes verifying semantics of primitives, properties of objects such as files, and global properties such as absence of deadlock.

Some properties can be established by considering just one process, assuming the absence of interference (asynchronous interaction between processes). For example, .Get_time is implemented as sending a message to the clock proprietor, which returns the current time in the reply. Its semantics can be verified by just considering the clock proprietor. However, many properties, such as absence of deadlock, are global properties, being the result of the collective behavior of processes.

The verification of the multi-process structure is structured by first verifying each system process, assuming the absence of interference. A process is a small (relative to the system), somewhat independent, sequential unit. Then, the collective behavior of these processes is considered in verifying global properties, including discharging assumptions of no interference.

6.2.1 Process Verification

By the term verifying a process, we mean proving properties about that a process such as the semantics of sending a message to the process, the resources it accesses and the processes it sends to.

The advantage of considering one process at a time is that it is a natural sequential unit in the multi-process structure. Properties can be verified using sequential program proof techniques because a process is sequential and deterministic and interaction with other processes is sequential (assuming no interference). That is, because of the subroutine semantics of .Send, and because a system process only acts for a sender until it replies, its use can be interpreted as invoking a subroutine that accomplishes the effect of the .Send to the other process. The use of .Receive or .Await_interrupt can be considered as receiving input and use of .Reply can be thought of as generating output. Processes are partially ordered by the "sends to" relation within the levels of abstraction. If processes are verified starting at the lowest level (processes that do not send to other processes), each process can be verified after verifying each process it sends to (avoiding possible circularity). Thus, each process can be verified with known semantics attached to each of its calls to .Send.

Each system process executes some local initialization when it is first created and then proceeds into an infinite loop, performing its function. The local initialization sets up resources used by the process for the function it performs in the loop. The loop is governed or driven by external (to the process) events. For example, the loop of the RTC process is driven by clock interrupts; the loops of proprietors are driven by request messages from other processes.

One aspect of verifying a process is considering resources that the process acts on and establishing certain invariant assertions about these resources. These invari-

ants are defined to hold at certain points in the loop because they can be legitimately violated while the process is changing these resources. Invariants can be verified inductively. To establish that an invariant I holds immediately before the receipt of each request in a proprietor, the invariant is first shown to hold after initialization and just before receiving the first request. Then, assuming the I is true just before one request, I is shown to hold just before receiving the next request. By induction, the invariant holds just before the receipt of every request. In the course of proving these assertions, it can be established which resources the process accesses. This is used later to show the absence of interference.

The action of a loop is the change that occurs to the data structures as a result of executing the loop. For example, the RTC process updates the current time every iteration through the loop (on every clock interrupt).

With two exceptions, the function of every system process is specified by the semantics of sending to that process. The two exceptions are the RTC process and the swapping alarm clock. The RTC process maintains the current time and sends to the clock proprietor when it is time to wakeup a process. The swapping alarm clock

```
.Swapping_alarm_clock()#(THOTH_CODE, THOTH_DATA)

    This process delays for SWAP_CLICKS clock interrupts
    and then sends to the team proprietor to reorder
    the team priorities.
    The first team in the list of swapped teams
    that is ready being brought into memory at the head
    of the "user" ready queues if such a team exists.
    Otherwise, the last user team in memory is moved
    to the front of the user ready queues.
/   {
    extrn .Team_proprietor_id;
    auto request[MESSAGE_SIZE];

    repeat
  /   {
        request[0] = REORDER_TEAM_PRIORITIES;
        .Send( request, .Team_proprietor_id );

        .Delay( 0, SWAP_CLICKS );
  /   }
/   }
```

simply sends to the team proprietor periodically to provide real-time input to the multiplexing of the CPU and memory among transient teams. For these two cases, these actions are verified. Their impact on the global behavior is considered later.

Several system processes are simple proprietors in the sense that they receive a request, act on it, and then reply before going on to another request. Their only interaction with other processes when acting on the request is by sending messages. Thus, the semantics of sending to these processes follow simply from the sequence of actions taken by the process from the receipt of the request to the reply. The file system proprietor, the teletype input proprietor, the disc proprietor, the controller proprietor, the abort proprietor, and the death proprietor are all in this category.

The other proprietors do not necessarily reply to a request before receiving the next request. For example, the implementation of .Sleep is the sending of a request to sleep and a wakeup time to the clock proprietor; the clock proprietor only replies when the wakeup time arrives. This complicates the verification of the process in two ways. First, the receiving of requests and the sending of replies is intermixed. This can be handled by arguing the semantics of sending to such a process using induction on the number of intervening requests received between the receipt of a message and the corresponding reply. Second, these processes rely on receiving certain messages from other processes to complete the servicing of the ones not yet replied to. For example, the clock proprietor relies on the RTC process to send wakeup messages to cause it to reply to sleeping processes when the wakeup time arrives. Therefore, the semantics of sending to these processes includes the assumption that they receive certain messages at certain times. In considering the global behavior of the system, these assumptions can be removed. The clock proprietor, the team proprietor, and the teletype output process have this type of behavior.

In general, verifying a system process is simplified by treating it like a sequential program and by the simple process interaction. However, several problems arise. First, as mentioned earlier, some processes violate the abstraction by directly manipulating kernel data structures. For example, the death proprietor

```
.Death_proprietor()#(THOTH_CODE,THOTH_DATA)

    The .Death_proprietor destroys the process whose id
    it receives as well as all its descendants, reclaiming
    reclaiming resources and unblocking all senders and
    receivers blocked on these processes.
/   {
    auto msg[MESSAGE_SIZE], pd, current, prev, id;

    repeat
/   {
      id = .Receive( msg );

      pd = .Convert_to_pd( msg[0] );
      if( pd == 0 || !.Authorized(id, TEAM_USER[TEAM[pd]]) )
        msg[0] = 0;
      else
/     {
        prev = FATHER[pd] + (SONS - BROTHER);
        while( (current = BROTHER[prev]) != pd )
          prev = current;
        BROTHER[prev] = BROTHER[current];

        BROTHER[pd] = FATHER[pd] = 0;

        .Remove_processes( pd );

        .Settle_estates( pd );

        .Unblock_senders_and_receivers();
        msg[0] = 1;
/     }
      .Reply( msg, id );
```

```
    }
}
```

manipulates the process descriptors of the processes it destroys to remove them from the tree of processes and send queues, to invalidate process id's, and to reclaim resources. This requires arguing that the actions on the process descriptors has the desired effect in the abstraction and that the actions can not generate problems. That is, the data structures of the kernel are shared so it must be argued that these actions are properly synchronized on the process descriptors. Generally, the required mutual exclusion is achieved by disabling interrupts or by relative indivisibility if no processes of higher priority can affect this data.

Second, process interaction extends outside the message-passing. For example, the disc proprietor can directly transfer data from the disc to a buffer in a transient team. Thus, the disc proprietor must communicate to the team proprietor not to swap that team out to disc while this transfer is in progress.

Third, a process awaiting reply can be destroyed, so no proprietor can assume that its requestor is awaiting reply until the reply is returned.

Fourth, the sequential verification proposed for processes seems hard to apply to the use of .Forward. For instance, the clock proprietor uses .Forward as described in Section 5.3 to forward delaying processes to itself.

Finally, the verification of a system process is only a partial correctness because the semantics of sending to it are contingent on it replying. Two global properties affect whether or not the process receives and replies to a message. The process must be allocated enough CPU time to execute. This is impossible to establish unless underlying machine properties are considered. For instance, if the machine were so slow that it spent all its time running the highest priority process, such as say the RTC process, no other processes would be allocated the CPU. In practice, this is not a problem. One can show for a given machine, that the percentage of CPU cycles used by the system (that is, used by system processes and interrupt routines) decreases as the cycle time decreases, providing that the frequency of interrupts and system calls remains constant. That is, the CPU cycles used by the system is dependent on the frequency of interrupts and the frequency of system calls, not on the speed of the processor. Another reason why a process might not reply to a message is that it might become deadlocked and thus unable to continue. This is considered in the next section.

6.2.2 Global Properties

Global properties are properties of the multi-process structure that must be argued from the behavior of two or more processes. So far, the properties of individual processes have been considered. Three categories of global properties are considered here:

1. blocking behavior including absence of deadlock
2. absence of interference or timing-dependent behavior
3. collective functional behavior

These properties involve the logically concurrent behavior of several processes. To consider this behavior in a sequential argument, the following general approach is used. For a given property, certain system events are relevant. These events can be

sufficiently small that they can be considered to occur sequentially even though some are logically concurrent. Thus, the technique is to enumerate every possible sequence of relevant events and show that each sequence has the desired property. For instance, to show the absence of deadlock, the events are the blocking of processes. It is necessary to show that no sequence of blocking that can occur produces deadlock.

This technique reduces consideration of a parallel operation to a series of sequential arguments; however, the number of sequences of events can potentially be very large. The structure of the system limits the number of interacting events to reduce the number of sequences that must be considered. Various techniques can be used to further reduce the number of sequences and provide arguments that the ones considered are sufficient. We first consider deadlock.

The blocking graph of a set of processes, described in Section 4.1, superimposes the blocking behavior of each process, giving a worst case for the system (in terms of blocking behavior at any time). The only processes that can possibly deadlock are those on a cycle in the blocking graph, so only the different possible sequences of blocking for those processes need to be considered in the deadlock argument. In the blocking graph of the system in Figure 1 (page (fig1)), only the death proprietor, the teletype input proprietor and the teletype output proprietor are contained in a cycle, so only the difference sequences of these processes blocking need to be considered to show that this cycle of blocking cannot actually occur.

One other form of blocking needs to be considered to show that the system cannot deadlock. System processes can lock a transient team in memory to do direct memory access (DMA) between a device and the data segment of the team without the team being swapped out. For example, the disc proprietor "locks" a transient team in memory when doing direct transfers between disc and memory to or from the team data area because the team proprietor can otherwise swap the team out to a file asynchronously with respect to the disc data transfer. The team proprietor can effectively block waiting for a team to become unlocked in order to bring in a swapped transient team. It must be argued that no process can have a team locked in memory while it is blocked (directly or indirectly) on the team proprietor.

Absence of interference requires showing that there is no timing-dependent or asynchronous interaction between processes that would make the abstraction behave non-deterministically. This involves showing that reordering unsynchronized events in each sequence of events that can occur does not change the behavior in the abstraction. First, different concurrent sequences of events result from the execution of different processes, including user processes executing system primitives. Second, it is only necessary to consider events that access common resources. Thus, in the pure proprietor structure, there is no sharing and all interaction is via message-passing (and thus synchronized) so there can be no interference. That is, the effect of reordering the unsynchronized events of any sequence of events is the same as the original sequence. However, processes interact outside the message-passing in several ways. For example, in the above example of the disc proprietor locking a team in memory, it must be argued that the locking forces the team to remain in memory, that the disc proprietor ensures that the team is in memory, and that the team is only unlocked when the data transfer is complete. Similarly, a user process can be destroyed asynchronously while being served by a system process. However, the death proprietor is synchronized by executing at the lowest system priority, so process destruction can only happen when other system processes block. Also, the death proprietor synchronizes with other system processes by closing the files of the process it is destroying. Thus the sequences of events that need to be considered involve swapping, various

I/O operations and process destruction.

We now consider the collective functional behavior of the system. Each system primitive is implemented as a function which possibly sends to system processes. As mentioned previously, the .Send's have the semantics of subroutine calls, so the semantics of these primitives can be verified using sequential techniques. For several proprietors, verifying the semantics of sending messages to them requires assumptions about the messages they receive. These assumptions can be discharged by the previous verification of processes that send to these proprietors. For example, verification of the RTC process shows that it sends a wakeup message to the clock proprietor when the wakeup time becomes equal to the current time. The semantics of .Sleep follow from the semantics of sending a sleep request to the clock proprietor, the semantics of sending a wakeup message, and the knowledge that the clock proprietor receives a wakeup message after the wakeup time for the sleep request arrives.

Functional properties other than semantics of the primitives are of interest. For instance, the swapping alarm clock has the property of sending a particular message to the team proprietor periodically. The team proprietor has the semantics of invoking a certain function on receipt of this message. The function adjusts the priorities of transient teams by bringing in a swapped team and rearranging the CPU allocation to transient teams in memory. The global property is then a periodic adjustment of transient team priorities. It can be shown, for instance, that a transient team with ready processes does not remain swapped indefinitely. Another global property is the minimum recycle time for process id's.

This discussion of the multi-process structure shows that the simple semantics of the message-passing and its simple hierarchic use (in the sense of the partial ordering by .Send's) aid the verification of many properties. The verification of properties local to individual processes provided a foundation for establishing global properties. Treating arguments about concurrent structures as a series of arguments about the enumerated possible sequences of events is applicable when the number of sequences can be reduced to a manageable number.

6.3 Initialization

The initialization of the system is the set of actions that take place from the moment the system executes the first instruction (after being loaded into memory) until it arrives in its normal state of operation. This occurs in two stages. The first is the initialization of the kernel. The second is the concurrent execution of the initialization of each system process.

At the beginning of the first stage, a "handcrafted" process is created as the root process. It first sets up the data structures implementing the basic abstraction. These include the memory free list for the system team, process descriptors and ready queues. At this point, other processes can be created. The root process then sets up the system configuration as dictated by the *configuration table* (see Section 2.4). This includes creating other system processes and setting various hardware configured to an initial state. For example, the speed and mode of the teletypes are set. User processes are created either by the root process or by another system process, depending on the configuration. The root process changes itself to the lowest priority process to become the "idle" process, inserts itself into the slumber queue to effectively sleep forever, and switches to activate the highest priority ready process. Because this

first process violates the abstraction so much in order to set up the abstraction, it might be better to regard the initialization as executed by the processor rather than by a process executing in the abstraction.

Up to the point of this process switch, the initialization is executed sequentially and deterministically by the root process because it is the first process to execute, it is of the highest priority, interrupts are disabled, and it does not block. Thus, the initialization to this point forms a unit of sequential verification.

After the first process switch, system processes begin to execute in the order dictated by the CPU allocation rule. Each system process executes its initialization concurrently with the other processes. Because this initialization is local to each process, its verification is considered as part of the verification of the process.

It remains to verify the concurrent part of the initialization. This is argued as follows. First, initialization local to a process does not require synchronization because it does not affect variables shared with other processes and this is completed before receiving any requests. Similarly, because the initialization of the devices is done by the root process, it is completed before any process accesses a device. Thus, even though different processes can complete their initialization at different times, the required sequencing of events is accomplished by the root process executing some actions before any other process executes and the requirement that no process receive requests before completing its local initialization. Finally, the blocking graph of the system in Figure 1 is valid for the initialization as well.

Thus, the verification of the initialization involves the sequential verification of the initialization of the abstraction and the local initializations of each system process plus basically the argument given about the concurrent part of the initialization.

6.4 Summary

The system was not specifically designed to be verified, but verification does not appear to be totally infeasible. Much of the verification can be reduced to sequential verification tasks. The kernel is verified as a set of sequential operations on process data structures. A major step is making the transition to verifying operations as being executed by a processs from verifying operations as being executed by the processor. The interaction in the message-passing is handled by attaching semantics to the blocked states used by the message-passing. The subroutine semantics of .Send and the partial order it defines on the multi-process structure make it possible to verify each system process as if it were a sequential program. The technique of progressively extending the abstraction by the addition of new components renders the size of the system more manageable and aids in handling apparent circularities of dependencies. The technique of enumerating the different sequences of events relevant to a global property involving asynchronous events allows the use of sequential arguments to establish the property.

Several problems make the system difficult to verify. Several processes violate the basic abstraction by directly manipulating the kernel data structures. Some instances of this, such as process destruction and swapping, seem hard to avoid unless they are changed so they are not implemented by processes. Process interaction is more complicated than just message-passing so a number of complicated cases are en-

countered, particularly involving process destruction.

In conclusion, the system should be changed to minimize violations of the basic abstraction and to provide more disciplined process interaction. However, further development of an informal outline of verification would be of benefit in providing a framework in which to reason about the system, document assumptions, and specify the design. The work on this to date has given us increased confidence in the design as well as uncovering a number of errors.

7 Summary, Conclusions and Future Research

This thesis has explored structuring programs as many asynchronous processes. It records ideas arising from the engineering experience of building an operating system. Much of the claim of the practicality of the ideas is based on their implementation and use in the Thoth operating system.

We have attempted to draw pertinent conclusions in the course of this work but we will use this chapter as an opportunity to summarize the thesis and draw together conclusions from the preceding chapters. Also, time did not allow pursuing all ideas as far as we would have liked. We conclude with possibilities for further related research.

7.1 Summary

Chapter 1 was introductory in nature, describing the motivations for this research, previous related research, and providing an overview of the thesis.

Chapter 2 provided background to this research in the form of the history of Thoth, and described the abstraction implemented by the system, referred to as the Thoth abstraction. The chapter included a discussion of the portability of Thoth and gave results of measurements of system performance, including space, time and real-time response.

Chapter 3 described the facilities considered desirable for the multi-process structuring of programs. This served to motivate the design of Thoth presented in Chapter 2 and also to point out several problems that had to be solved to implement these facilities. The problems included the deadlocking and inefficiency normally associated with message-passing, the incompatibility between process synchronization and process destruction, and the need for complete and immediate process destruction. We discussed the need for efficient communication between interacting processes and thus motivated the concept of team described in Chapter 2.

Chapter 4 addressed the problem of how to use processes to structure programs. The concepts of process resource ownership and proprietors were presented as conceptual tools for structuring a program using processes and for solving concurrency prob-

lems. A set of general process structures were presented suitable for a number of situations including more generalized queuing and serving, multi-events, break and exception handling. Also, a process structure for connecting machines called an agent was discussed. This chapter documented the insights we have gained into various useful process structures.

Chapter 5 described the current structure of the Thoth operating system, pointing out some of the problems that have been recognized and suggesting possible solutions. It served to provide more information about the system as background for Chapter 6 which discussed the feasibility of verifying the operating system. It demonstrated how process structuring had been used to make the system more amenable to verification. The outline of verification provided a framework on which to understand, and thus gain more confidence in the design of the system.

7.2 Conclusions

The major conclusion we draw from this thesis is that the use of processes to structure programs is worthwhile and deserves further investigation. As a consequence, we conclude that the facilities for structuring programs as multiple processes should be made available at the user level in future systems. However, the proper use of processes seems non-trivial and more techniques are required for developing and understanding multi-process programs, particularly if the concurrency problems of deadlock and interference are to be avoided. A number of other conclusions can be drawn from the discussions in particular chapters.

In Chapter 3, we concluded that there was strong motivation for using synchronous message-passing to avoid the dynamic buffering and the resulting inefficiency and possible deadlock. We demonstrated the need for complete and immediate process destruction and showed its incompatibility with semaphore- or monitor-like synchronization schemes. This problem, plus the incompatibility with distributed systems, does not support the use of shared memory synchronization techniques. In Chapter 4, as a solution to synchronization, we showed that the proprietor concept could be used to synchronize in the presence of process destruction and that its implementation was only dependent on processes and the message-passing mechanism. From the examples and discussion of this chapter, we conclude that multi-process structuring provides elegant solutions to a number of programming problems and its use has many significant advantages. We contend that these examples support the adequacy of the abstraction described in Chapter 2 and developed in Chapter 3. Chapter 5 shows that the use of multi-process structuring in operating systems has many advantages and suggests some solutions to problems present in the current structure of the system.

In Chapter 6, we show that the process structuring of the Thoth system has provided a structure that is more amenable to verification than the more conventional monolithic operating systems structure. This demonstrates that a real operating system can be structured to facilitate verification without detriment to its utility. The outline of verification is also a useful form of documentation of the system.

As a final, overall conclusion, the work invested in this thesis has been of real benefit to the design and implementation of the system. The discussion of design issues for the implementation of the abstraction has clarified the trade-offs in the system and lead to improvements, for example in process identification. The process

structures presented in Chapter 4 have suggested new structures in the system and other programs, some of which are currently in use. The description of the system in Chapter 5 has pointed out problems that had not been previously recognized. Finally, the outline of verification lead to the discovery of several minor bugs in the system, and increased our awareness of inconsistencies in the design.

7.3 Future Research

Many ideas have not been fully explored, tested or perhaps even thought of with respect to multi-process structuring. We expect that further insights and understanding will result from the application of systems like Thoth to a variety of problems, particularly communications and distributed system development.

The further application of these ideas will reflect considerably on the message-passing mechanism and interprocess communication in general. First, it will be necessary to implement the message-passing between different machines, raising the need for a more sophisticated message-buffering scheme. Second, the more extensive use of messages raises issues of size of messages and the amount of data that needs to be moved. That is, is the message-passing mechanism suitable for moving large amounts of data between machines and if not, what mechanism would be suitable? Third, what conventions should be used for message format? Finally, the work to date has not addressed the issues of protection and security. For example, there is no restriction on which processes can send to a given process. This should be explored with considerable attention given to the ideas of the DEMOS link and DEMOS security and protection mechanisms described in Baskett et al. (1977).

The use of processes to structure operating systems, especially distributed systems, needs much more study. Even in the conventional single machine setting, there are many ideas to explore in using processes to implement virtual memory and I/O. It is hoped that this will be investigated as part of the future porting of Thoth to machines whose hardware supports demand paging and segmentation.

The work on verification in Chapter 6 should be developed further. The outline of verification should be made much more complete and a more rigorous. It would be nice to establish a level of rigor and a style of presentation comparable to that of mathematics to present a convincing yet readable verification of the system. Such a verification could act as a detailed level of documentation plus a measure of assurance to users of the system. It could also provide a wealth of problems to motivate work on new verification techniques and to motivate rethinking and redesigning parts of the system.

Much of the work done on Thoth to date can be viewed as a software implementation of a desirable machine. Hopefully, besides improving this abstraction, future work will improve the implementation of this abstraction both in software and in the exploration of its implementation in firmware and hardware. For example, the basic process switching operations could be implemented in firmware on some machines.

In conclusion, the use of many small processes in programs raises many design and implementation questions. While the literature contains theoretical work in this area, little seems to have been implemented and tested. This thesis is based on work aimed at exploring the practical value of multi-process structuring using conventional hardware. The implementation work involved has been considerable, much of it not

of research interest. However, in exploring engineering issues, implementation and testing is required. Hopefully, further use of Thoth and the ideas presented here will support the conclusions of this thesis and motivate further research in this area.

References § %

Ashcroft, E. A. (1974), Proving assertions about parallel programs, Research Report CS-73-01, University of Waterloo.

Baskett, F., J. H. Howard and J. T. Montague (1977), Task communication in DEMOS, *Proc. 6th Symp. on Operating System Principles*, A.C.M., New York, November, 23-32.

Bensoussan, A., C.T. Clingen and R.C. Daley(1972), The Multics virtual memory: concepts and design, *Comm. A. C. M.* vol. 4, no. 5, May, 308-318.

Braga, R. S. C. (1976), Eh reference manual. University of Waterloo, Research Report CS-76-45, November.

Brinch Hansen, P. (1969), *RC 4000 Software Multi-programming System*, A/S Regnecentralen, Copenhagen - April 1969.

Brinch Hansen, P. (1970), The nucleus of a multiprogramming system. *Comm. A. C. M.*, vol. 13, no. 4, April, 238-241, 250.

Brinch Hansen, P. (1973), *Operating Systems Priniciples*, Prentice-Hall.

Cheriton, D. R., Malcolm, M. A., Melen, L. S. and G. R. Sager (1978), Thoth, a Portable Real-time Operating System. to appear in CACM.

Cox, G. W. (1975), Portability and adaptability in operating system design, Ph.D. thesis, Purdue University, West Lafeyette, Indiana.

Data General Corporation (1974), How to use the NOVA computer, No. 015-000009 Southboro, Mass.

DeMillo, R. A., R. J. Lipton and A. J. Perlis (1977), Social processes and proofs of theorems and programs, *Fourth Annual ACM Conference on Programming Languages, 1-9.*

Dijkstra, E.W. (1968), Co-operating sequential processes, In F. Genuys(ed.), *Programming Languages*, Academic Press, New York, 43-112.

Dijkstra, E. W. (1972), Hierachical ordering of sequential processes. In *Operating Systems Techniques*, Academic Press, New York, 1972, 72-93.

Feiertag, R. J. and E. I. Organick (1971), The Multics input-output system. *Proc. 3rd Symp. on Operating System Principles*, A. C. M., New York, Oct., 35-41.

Hoare, C. A. (1972), discussion on monitors In *Operating Systems Techniques*, Academic Press, New York, 1972, p.109.

Hoare, C. A. (1973), A structured paging system, *Computer Journal* 16 3, 209-215 August.

Hoare, C. A. (1974), Monitors: an operating systems structuring concept. *Com. A. C. M.* vol. 17, no. 10, October, 549-557.

Huber, A. R. (1976), A multi-process design of a paging system, Masters Thesis, MIT LCS-TR-171.

Jammel, A. J. (1978), letter to the editor, *Oper. Syst. Review 12, No. 1, p. 16.*

Jammel, A. J. and H. G. Stiegler (1977), Managers versus monitors. *IFIP Congree Proc.* North-Holland, 827-830.

Janson, P. A. (1976), Using type extension to organize virtual memory mechanisms, Ph.D. Thesis, MIT/LCS TR-167.

Johnson, S. C. and B. W. Kernighan (1973), The programming language B, Bell Laboratories Computing Science Technical Report, No. 8.

Johnson, S. C. (1978), personal communication.

Johnson, S. C. and D. R. Ritchie (1977), personal communications.

Kernighan, B. W. and P. J. Plauger (1976), *Software Tools*, Addison-Wesley.

Knuth, D. E. (1973), *The Art of Computer Programming*, Vol. 1, Second Edition, Addision-Wesley, Reading, Mass.

Lipton, R. J. (1975), Reduction: a method of proving properties of parallel programs, *Comm. A.C.M.*, December, 717-721.

Malcolm, M. A. and P. V. Poblete (1977), Ted, the Thoth text editor. Department of Computer Science, University of Waterloo, Canada.

Melen, L.S. (1976), A portable real-time executive: Thoth, Master's thesis, University of Waterloo.

Miller, R. (1978), UNIX - a portable operating system, *Proc. of the Australian Universities Computer Science Seminar*, Feb., 23-25.

Montgomery, W. A.(1977), Measurements of Sharing in Multics, *Proc. 6th Symp. on Operating System Principles*, A.C.M., New York, November, 85-90.

Owicki, S. (1975), Axiomatic proof techniques for parallel programs, Ph.D. Thesis, Technical Report TR-75-251, Cornell University.

Powell, M. L. (1977), The DEMOS file system, *Proc. 6th Symp. on Operating System Principles*, A.C.M., New York, November, 33-42.

Richards, M. (1969), BCPL: a tool for compiler writing and system programming. Proc. Spring Joint Computer Conf., 557-566.

Ritchie, D. M. and K. Thompson (1974), The UNIX time sharing system. *Comm. A. C. M.* vol. 17, no. 7. July, 365-375.

Schroeder, M.D., D.D. Clark and J.H. Saltzer (1977), The Multics design project,

Proc. 6th Symp. on Operating System Principles, A.C.M., New York, November, 43-56.

Shaw, A.C. (1974), *The Logical Design of Operating Systems*, Prentice-Hall, Inc., Englewood Cliffs, N.J.

Wadler, P. L. (1976), Analysis of an algorithm for real-time garbage collection, *Comm. A.C.M.* vol. 19, no. 9, September, 491-501.