IMPLEMENTING MULTI-PROCESS PRIMITIVES IN A MULTIPLEXED COMPUTER SYSTEM

by

ROBERT LEE RAPPAPORT

S.B., Massachusetts Institute of Technology
(1964)

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August, 1968

Signature of Author ______________________
Department of Electrical Engineering, August 30, 1968

Certified by ______________________

Accepted by ______________________
Chairman, Departmental Committee on Graduate Students

IMPLEMENTING MULTI-PROCESS PRIMITIVES IN A MULTIPLEXED COMPUTER SYSTEM

by

ROBERT LEE RAPPAPORT

Submitted to the Department of Electrical Engineering on August 30, 1968, in partial fulfillment of the requirements for the Degree of Master of Science.

ABSTRACT

In any computer system primitive functions are needed to control the actions of processes in the system. This thesis discusses a set of six such process control primitives which are sufficient to solve many of the problems involved in parallel processing as well as in the efficient multiplexing of system resources among the many processes in a system. In particular, the thesis documents the work performed in implementing these primitives in a computer system, the Multics system, which is being developed at Project MAC of M.I.T. During the course of work that went into the implementation of these primitives, design problems were encountered which caused the overall design of the programs involved to go through two iterations before the performance of these programs was deemed acceptable. The thesis discusses the way the design of these programs evolved over the course of the work.

THESIS SUPERVISOR: Jerome Howard Saltzer
TITLE: Assistant Professor of Electrical Engineering

ACKNOWLEDGEMENT

During the course of the work documented in this thesis several people contributed ideas which were incorporated into the work. In particular, I would like to thank Michael Spier, Peter Schicker, and Robert Daley all of whom have been contributors of ideas as well as critics of my ideas. Also, I would especially like to thank my thesis supervisor, Prof. J.H. Saltzer, on whose earlier work this thesis is based, and to whom I am deeply indebted. The time and effort he has expended discussing numerous subjects with me, over the last three years, have been greatly appreciated.

Lastly, and most importantly, I would like to thank my wife Paula and my little Shelley who make it all worthwhile.

Robert Rappaport
August, 1968

CONTENTS

ILLUSTRATIONS

Implementing Multi-Process Primitives in a Multiplexed Computer System

CHAPTER ONE

Introduction

A computer system is a vehicle in which various tasks or processes are executed. In all computer systems, at least two primitive process control functions exist. These are:

1. The ability to create or introduce new processes. We will call this the process creation primitive.

2. The ability to forceably halt the execution of a process. This ability rests in some force or power outside the process (possibly in another process). We will call this the process destruction primitive.

In a sense these two functions are implemented implicitly in the hardware of the computer system. The first is implemented in the hardware that allows information to be input into a "cold" machine. The second is implemented by the existence of a switch to turn off the electric power. Clearly most computer systems provide more elaborate implementations of these primitives however sight should not be lost of the fact that their implementation is inherent in the nature of a computer system.

In addition to the above two primitives most systems for reasons of efficiency and convenience provide a third primitive function:

3. The ability for a process to declare that it has finished and ought to be terminated. We will call this the suicide primitive.

Many traditional computer operating systems are based on just these three primitive functions. Processes are begun using the process creation function and run until they either declare themselves finished (suicide function) or until someone or something decides that they should be terminated (process destruction). These three primitives in a system allow for the specification and implementation of serial (non parallel) computations in that system.

J.H. Saltzer in his thesis, "Traffic Control in a Multiplexed Computer System", (reference number 1 in the bibliography) proposed four additional primitives, one of which includes the suicide primitive. They are:

1. The block primitive which includes suicide.

2. The wakeup primitive.

3. The reschedule primitive (originally named restart by Saltzer).

4. The stop primitive (originally named quit by Saltzer).

These four primitives make up what Saltzer calls the Process Exchange.

The first of these, block, provides the ability for a process to declare itself unable to continue execution until some event has taken place. The wakeup primitive allows one process to signal another process about the occurrence of an event and thereby cause the return to execution of a blocked process. In a system without the wakeup primitive, block is synonomous with suicide. As Saltzer states, the block and wakeup functions are all that are necessary to solve the "intrinsic" problems associated with simple parallel processing.

The third of these primitives, reschedule, is intended to help solve the problem of processor multiplexing among the many processes in a system. This primitive allows a process to schedule future execution after it has had its execution forceably pre-empted. This primitive coupled with the ability to force pre-emption of executing processes allows a system to share a limited number of hardware processors among many processes.

The fourth of the Process Exchange primitives, stop, allows one process to forceably halt the execution of another process while leaving the halted process in such a state that continued execution is possible, at the discretion of the process that ordered the halt. Stop is basically a refined or narrowed form of the process destruction primitive. One can think of using this primitive as part of the implementation of the process destruction primitive, however stop has additional uses. For example, one could imagine stopping a process in order to examine various data items to determine whether everything was in order. For this reason stop should be kept distinct from process destruction.

The four Process Exchange primitives form a logical group distinct from the creation and destruction primitives in that the former functions deal only with actual processes (i.e., already created and not yet destroyed) and they therefore have no knowledge of how or why processes are created or destroyed. This allows us to discuss them more or less independently.

Scope of Thesis

The existence of these six primitives in a system provide the capability to solve many of the problems involved with parallel processing and the efficient multiplexing of system resources among many processes. For the past several years work has been progressing, at Project MAC of M.I.T., on the design and implementation of a computer system in which these six process control primitives are available. The system is known as the "Multics" system (the name Multics being an acronym for Multiplexed Information and Computing Service). In such a system the efficiency of the modules which implement the six primitives have a profound effect on the overall performance of the entire system. In a system such as the Multics system, in which segmentation and paging have been implemented, and which is designed to support many processes concurrently, primitives such as block and wakeup can be expected to be invoked many times per second. Also in a system supporting many processes, new processes can be expected to be created frequently while older ones are being constantly destroyed. Clearly the efficiency of the modules which implement these functions are of utmost importance to system performance.

The thesis presented here covers the work involved in designing an effective implementation for the six primitives in Multics and in actually implementing them in the system. During the course of the work the implementation went through two iterations. The first implementation provided working versions of these primitives however performance tests performed on these modules showed them to be unacceptably slow in execution. With insight and lessons learned from working on this original implementation, ways were found to redesign the modules so that the second iteration of the implementation brought the performance of these modules into acceptable limits. The thesis report discusses the problems encountered in implementing the modules and shows how the implemented modules evolved over the course of the work. The discussion is divided into five chapters (chapters 2 through 6).

The first of these chapters (chapter 2) is an introduction and review of the concepts involved in the ideas of a two dimensional addressing scheme (segmentation) and a large hierarchical file system. The background for the material presented in this chapter may be found in several published papers listed in the bibliography (references 2, 3, and 4) and cited in the text.

The second such chapter serves as a review and updating of the ideas involved in parallel processing and the multiplexing of hardware processors and primary addressable memory among many users. In major part this chapter is a review of the basic ideas presented in Saltzer's thesis, however several new concepts are introduced as well.

The next chapter (chapter 4) is a discussion of how the current design and implementation of the Process Exchange primitives in Multics evolved. For reasons of efficiency and performance several changes have been made in the internal structure of the Process Exchange although the user interface to the facility remains as originally designed. The changes include changes to the dynamic "loading" scheme, whereby information necessary for a running process is brought into primary memory, and also the addition of specially tailored versions of the block and wakeup functions (named the wait and notify functions respectively) to be used exclusively by the file system of Multics. The original block and wakeup remain as the user interface to the Process Exchange. The wait and notify functions were originally file system functions which served as the file system interface to the Process Exchange. They were absorbed into the Process Exchange because of the high volume of traffic from the file system and the need for fast efficient processing of this traffic.

Chapter five is a discussion of the problems involved in process creation and destruction. It includes a definition of a process (an address space and a processor stateword) and a discussion of how one goes about defining an address space that contains a minimum set of primitive functions needed by a new process so that it can continue to expand its address space by itself. Among this minimum set are the Process Exchange primitives, a "map" primitive that allows a process to add things to its address space and a "linker" primitive that allows a process to refer by name to addresses in its space or to names not currently in the address space. Included in this discussion is the presentation of how the process creation mechanism evolved during the implementation work.

The sixth and final chapter of the thesis is a summary which includes the results of performance analysis tests performed on the implemented programs and a discussion of the importance of these figures to overall Multics performance. Specifically, minimum standards of acceptability for the primitives are put forward and the implemented primitives are compared to these standards to ascertain whether their performance is acceptable. Also included in this chapter is a discussion of what was learned in the course of implementing these programs. In particular, the hypothesis is put forward that without a well structured theory of computer systems which allows one to design optimum systems from scratch, such design problems as were encountered here are more or less inevitable.

CHAPTER TWO

Segmentation, Paging, and File Systems

Segmentation and Paging

Segmentation is a device to allow hardware processors, in a computer system, to look at addressable memory as if it were a matrix. Each row of such a matrix is known as a segment. The four primary motivations for segmentation are:

1. Segmentation along with paging (as we shall see below) provide a simple way for a system to simulate the presence of a very large amount of primary addressable storage (i.e., in terms of current hardware, core storage) to user programs.

2. Segmentation, by segregating address spaces into distinct units, offers a handy way to organize a hardware protection scheme so as to permit differing access attributes to different parts of an address space. For example, some segments in an address

CHAPTER TWO

Segmentation, Paging, and File Systems

Segmentation and Paging

Segmentation is a device to allow hardware processors, in a computer system, to look at addressable memory as if it were a matrix. Each row of such a matrix is known as a segment. The four primary motivations for segmentation are:

1. Segmentation along with paging (as we shall see below) provide a simple way for a system to simulate the presence of a very large amount of primary addressable storage (i.e., in terms of current hardware, core storage) to user programs.

2. Segmentation, by segregating address spaces into distinct units, offers a handy way to organize a hardware protection scheme so as to permit differing access attributes to different parts of an address space. For example, some segments in an address space can be marked as "read only" and in this way the segments can be protected from errors and/or maliciousness.

3. Segmentation also allows for the sharing of procedure and data segments between address spaces.

4. Each segment of an address space may be individually expanded in length as storage needs change dynamically. This allows a user to expand data areas without relocating or overflowing other data areas.

Addresses in a segmented environment are ordered pairs of integers. The first element of such an ordered pair refers to the segment (i.e., which row of the matrix) and the second element refers to the relative word number in that segment (i.e., which column in the row selected). For example, a valid address in such an environment would be the ordered pair (16,29) which would be interpreted as word number 29 of segment number 16.

Segmentation can be implemented in several ways. On the General Electric 645 computer, on which the Multics system is being implemented, segmentation is implemented by special purpose hardware in each processor. This hardware allows a processor to look at a standard linear array of physical memory as if it were a matrix. The scheme makes use of an array of "descriptors", one per segment, which point to and "describe" the segment. This array of descriptors is known as a descriptor segment. All addresses (i.e., ordered pairs) generated by a processor are handled in the following way. The segment number of the address is used as an index into the descriptor segment (array of descriptors) to find the descriptor for the referenced segment. Part of this descriptor is a pointer to the base of the segment. In order to obtain the desired word of the segment, the word number part of the address is added to this base address. The processor is able to find the base of the descriptor segment because of a special purpose register in the processor which points to the base of the descriptor segment. This register is known as the Descriptor segment Base Register (DBR for short).

The way in which the descriptors describe their segments is by containing information as to the length of the given segment, and its access (e.g., readable, or writeable, or executable, etc.). The descriptor may also indicate that the given segment is not currently in core memory. It is through this last trait that the system can simulate a large addressable memory. Only a few segments need be kept in core memory at any time. The descriptors of segments not in core indicate that they are absent from core and references to these missing segments cause the processor to recognize a fault condition upon reference to the corresponding descriptor word. Upon recognition of the fault condition the processor automatically branches to a "missing segment fault" handler subroutine which retrieves the missing segment, updates the segment's descriptor word and restarts the processor operation at the point of the fault. In this way, the user of the system, who doesn't see this faulting mechanism, is allowed to see a huge addressable memory. Figure 1 shows a descriptor segment, pointed to by a DBR register and itself containing descriptors for several segments, some of which indicate the absence of the segments. Note that the descriptor segment may contain a descriptor for the descriptor segment itself, meaning that this segment may appear in the address space defined by the descriptor segment.

The above implementation implies that a segment be assigned to a contiguous block in core memory. If the segments themselves are potentially large, this would mean having to allocate large contiguous blocks of memory to individual segments. Such a requirement might necessitate needless movement of segments in core memory whenever new segments are brought into memory. A solution to this problem would be to break segments into smaller uniform sized blocks, called pages, and to force references to segments to be indirected through a second level "array of descriptors", called a page table. In this scheme, the descriptors for segments point to page tables for the segments rather than to the bases of the segments. The page tables then point at the individual pages of the segments. Figure 2 illustrates these ideas. Notice that here the descriptor segment is itself divided into pages and that the DBR register that was pointing to it in the previous figure now points to its page table. The above scheme requires only page tables to be stored in contiguous locations in

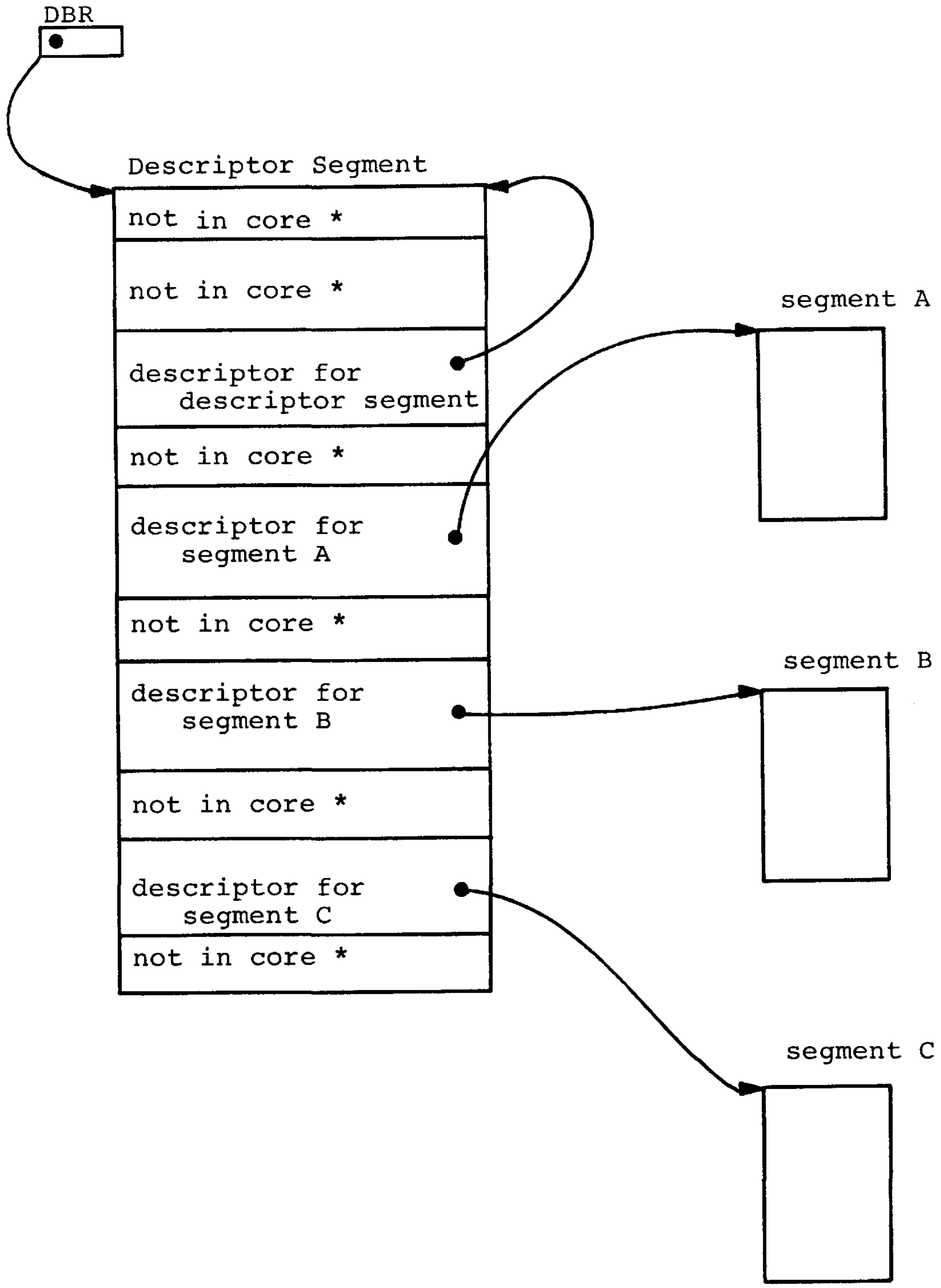

Figure 1. Implementation of segmentation

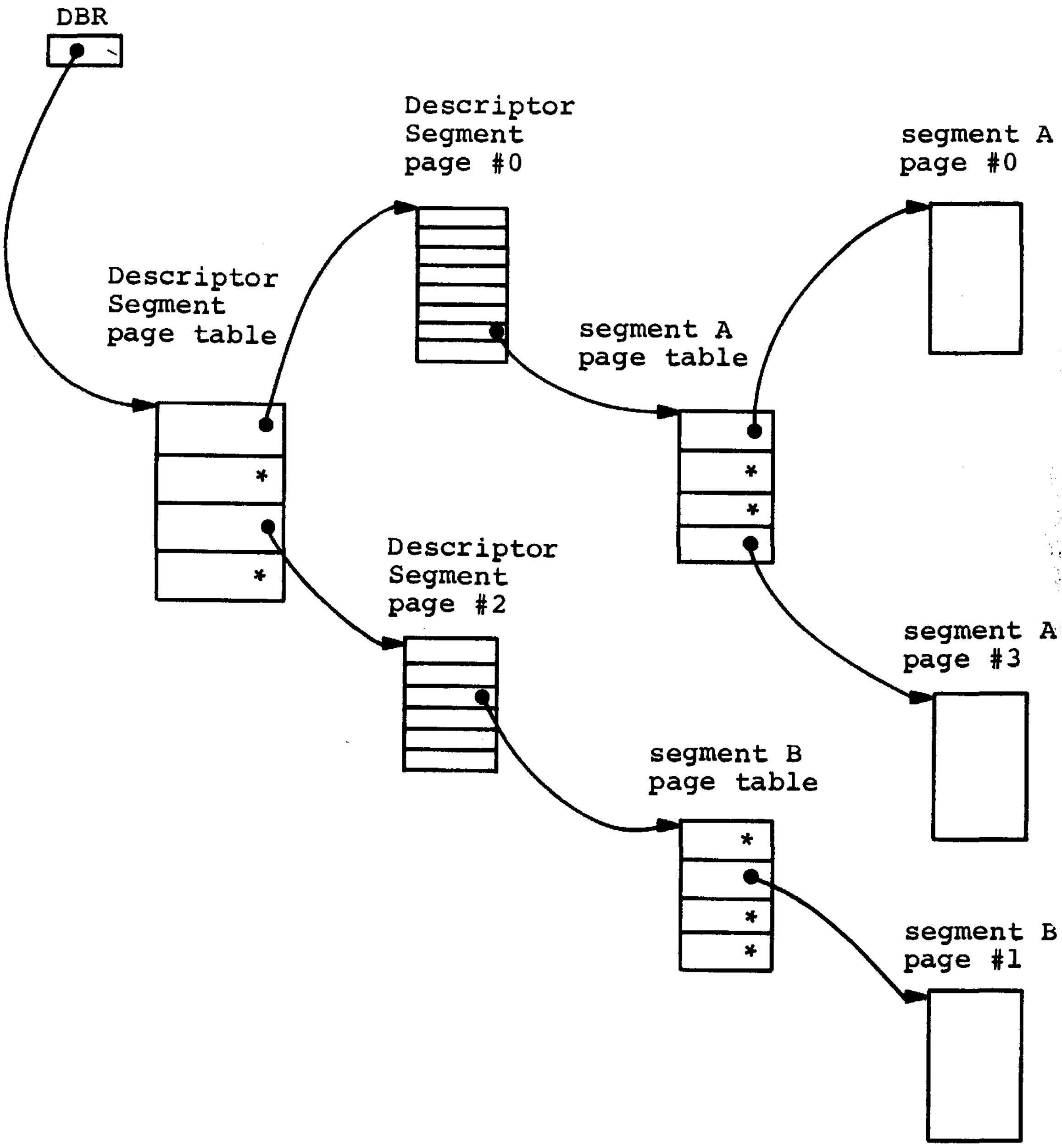

Figure 2. Segmentation and Paging

core memory, however since page tables are orders of magnitude smaller than segments this need not be a problem. Note also that page tables themselves, although instrumental in implementing the virtual memory, are not "in" the virtual memory and cannot be addressed directly, since they do not reside in any segment of the space. Just as descriptor words may indicate that a particular segment is not in core memory, page table words may indicate that a particular page is not in core. The absence of several pages is indicated in figure 2.

For the reader interested in more background material on segmentation and paging, he is referred to two papers. The first is by J.B. Dennis of M.I.T. and is entitled "Segmentation and the Design of Multiprogrammed Computer Systems". It was published in the Journal of the ACM in October 1965. The second paper is co-authored by E.L. Glaser, formerly of M.I.T., and J.F. Couleur and G.A. Oliver of the General Electric Company. The paper is entitled "System Design of a Computer for Time Sharing Applications", and it was presented at the Fall Joint Computer Conference of 1965. These two papers are references 2 and 3 respectively, in the bibliography that follows this thesis.

Hierarchical File System

By a file system we mean a mechanism by which we systematically keep track of blocks of data. In a typical file system, we might find entries for each logically distinct block of information (file). The respective entries, and therefore, the files, are accessed by name. In such an entry we might record the physical location of the file, its length, a list of authorized users, etc.

The simple file system provided a computation by a descriptor segment, which maintains an array of core memory addresses, does not suffice when it proves impossible to maintain all information in core. The file system which we briefly describe below is organized around a mechanism, known as a directory, which in function, is very similar to a descriptor segment. A directory, which is itself a file in the system, maintains an array of entries which point to other files in the system. Some, all or none of the files pointed to by a directory may themselves be directories.

Figure 3 illustrates the hierarchical structure of our file system. In the diagram, circles represent files which happen to be directories while rectangles represent non-directory files. The lines drawn between directories and the files represent the entries in the directories for the files. As can be seen, the file system has as its base, the directory at the top of the picture. This directory is known as the Root directory.

Within a single directory, all the entries (and therefore the associated files) have different names and the name of a file in the system is a combination of the name of the directory in which the file lives (the parent directory) and the file's entry name within this directory. The

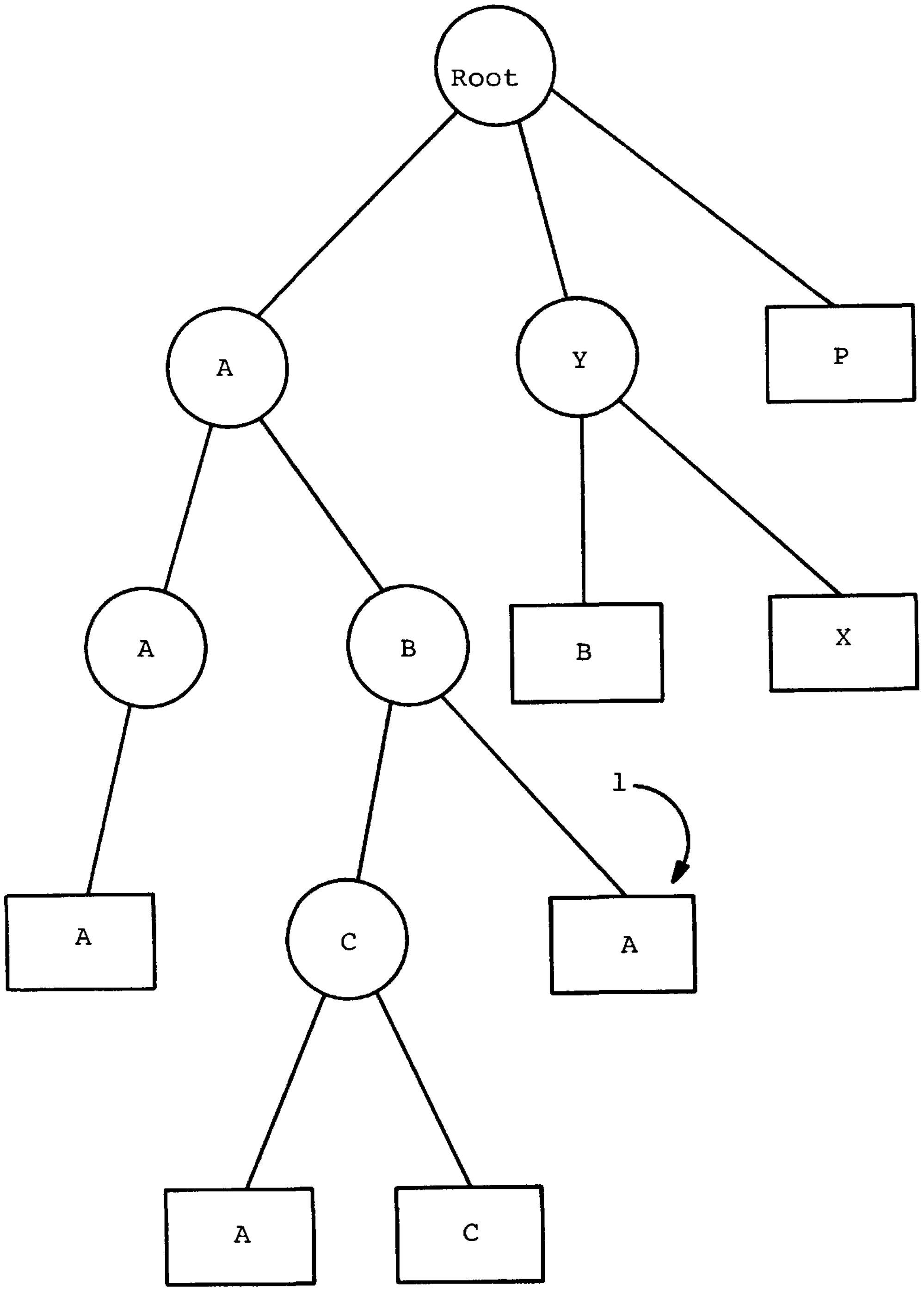

Figure 3. Hierarchical File System

syntactic rule we will use to form names of the files will be to concatenate the name of the parent directory and that of the file, separating them by the special character " ". Therefore, the name of file "A" listed in directory "B" will be "B A". Similarly, the name of the parent directory is formed from the name of its parent and its entry name in its parent directory. Since all names within a directory are different, we then see each file in the entire system has a unique name made by combining the entry names of all the directories on the unique path from the root of the tree structured file system to the respective file. Such a name will be known as a "tree name". In the figure, the tree name of the file labeled "1" is "A B A".

To the above file system we will add one additional feature. This feature is the ability to create a special entry in a directory (known as a link) which describes a file already contained in some directory (possibly the same one) in the system. A link serves as a shortcut to a file located somewhere else in the hierarchy of files and serves to give the illusion that the file pointed at by the link actually resides in the directory containg the link. Links can be implemented simply by having the directory entry corresponding to the link contain the complete name (e.g., tree name) of the file.

The addition of links to the file system adds no new capabilities to the system, however it facilitates the ease with which the file system may be used. For example, it may be desired to maintain the illusion that a particular file exists in a number of different directories. Links make this possible without the cost of maintaining duplicate copies of the file. With links, we can see that there may be several "paths" from the root directory to a particular file. It is desirable to have a name analogous to the tree name which includes links. Such a name will be known as a "pathname".

Files and Segments

The memory space defined by the hierarchical file system is the global memory space of the entire computer system and this space is always maintained by the system. The memory space defined by a descriptor segment in the computer system, is a subspace of the global space which is of interest to one or more computations (i.e., processes). It is a subspace in the sense that all of the data contained in it (that is addressable in it) can also be addressed directly in the global space, providing suitable pathnames are used. That is to say that the address space defined by a descriptor segment is formed by mapping several files into the segments of the space. For several reasons we will choose to map each distinct file into a distinct segment. These reasons include the fact that this simple mapping means that procedure files (segments) need not be "relocated" or "loaded" when mapped into segments.

To accomplish this mapping between files and segments, a data base is maintained for each descriptor segment defined address space in the system. This data base, known as the Known Segment Table (KST) of the address space serves to record the correspondences made between files and segments in the address space. That is, the KST maintains information such as: file "X Y Z" is mapped into segment #q. With such a data base available, it is clear that the KST serves as the principal piece of data at the time of a missing segment fault.

Let us now look at the mechanisms by which files are mapped into segments and the mechanisms by which the mappings are kept track of. Let us assume that we wish to map the file with pathname "X Y Z" into our address space. To do this we need merely locate the file in secondary storage and assign an unused segment number to it. Locating the file in secondary storage amounts to searching its parent directory for the appropriate entry. However, to search this directory, the directory (itself a file in the hierarchy) must be mapped into a segment, if it had not been previously. Similarly the parent's parent must also be mapped into a segment. If the root directory is guaranteed to be a segment in the address space then the procedure has a finite end. Implicit in the above steps is that a record is being maintained in the KST that file "X" has been mapped into segment #i, file "X Y" has been mapped into segment #j, and file "X Y Z" has been mapped into segment #k. In this way we will remember where to find segments (files) "X" and "X Y", when we want to map "X Y W" into our address space.

Several things should be noticed about the information contained in the KST. The KST contains the names of each of the segments in the address space and the segment numbers by which they are addressed. In this way we can see that the KST effectively defines the address space. Also, the KST maintaining a correspondence between pathnames and segment numbers, defines exactly the same space that would be defined by a directory appended to the File System Hierarchy which contained a link for each segment in the space. Effectively the KST serves as such a directory and the analogue of a link name for a particular file is the associated segment number assigned when a file is mapped into the space.

As was mentioned at the beginning of this section, one of the prime motivations for segmentation is the ease with which segments (files) can be shared between computations. This can be seen by the fact that the same file may be mapped into any number of address spaces in just the same way that many directories can have links to the same file.

Segment and Page Faults

In a manner analogous to the way an address space keeps track of its known segments, the system must keep track of which segments are currently active (i.e., have page tables in core memory). These records are kept in the Active

Segment Table (AST) which is a system wide table (i.e., it appears in each address space). Each active segment has an AST entry and this entry effectively serves as the current up-to-date copy of this segment's directory entry. At the time of a missing page fault on a page of an active segment, its AST entry serves as the principal piece of data since this entry contains the information as to where the missing page is located in secondary storage.

Now we can conceptually trace the sequence of events that occurs at the time of missing segment and pages faults. At the time of the missing segment fault the KST is scanned to determine which segment (file) was accessed. An AST entry is then established for the segment (if not already existent) and a page table consisting entirely of missing page faults is constructed. The descriptor word for this segment is then made to point to this page table. On any subsequent reference to the segment a missing page fault will be incurred which will make use of this AST entry to locate and bring into core the appropriate page.

For the reader interested in reading further about the concepts involved in maintaining an hierarchical file system, a paper by R.C. Daley of M.I.T. and P.G. Neumann of the Bell Telephone Laboratories is recommended. The paper, "A General-Purpose File System for Secondary Storage", was presented at the Fall Joint Computer conference of 1965 and this paper is reference number 4 in the bibliography.

CHAPTER THREE

Processes and Traffic Control

A process can be defined as an address space and a set of machine conditions (processor state). Intuitively we can think of a process as simply a program in execution (i.e., a program being executed by a processor). In our discussion the address space of a process is defined by a descriptor segment (which is in turn defined by a KST) and the processor state is the state of the processor executing the process. The machine conditions include, by definition, at least the value of a descriptor segment base register, an instruction pointer register and a current stack frame pointer (see below). They might also include the contents of accumulators, index registers, etc. There is no inherent reason to limit the number of processes executing in a single address space however throughout this thesis, in the interest of simplicity we will impose the limitation of one process to an address space. Of course many of the segments in any particular address space will be shared segments which appear in many or all other spaces.

Along with shared segments each address space contains several per-process (or at least per address space) data segments. Among these are the previously mentioned KST and one or more so-called "stack" segments. Stacks (call stacks, pushdown stacks, etc.) are areas in which procedures allocate blocks of space (frames) for the storage of temporary variables, contents of registers, and return points. Stacks are composed of frames which are threaded together. Each procedure of a program that has been entered (called) but has not yet returned has an associated stack frame which was allocated upon entry to the procedure. A pointer to the current stack frame is part of the processor state and the above mentioned threading mechanism allows one to locate "older" or previously allocated frames. The current stack frame and all previously allocated and still active stack frames comprise the relevant history of the process. In our discussion we will assume the existence of a system-wide table (i.e., a shared segment which appears in each and every address space) known as the Process Table which has an entry for each process. The contents of an entry in the Process Table include the identification of the process and various other data items, the need for which will be developed as we go along.

Block and Wakeup

As was mentioned above, a process is a program in execution. In most computing systems, in an attempt to achieve more efficient usage of system hardware, it is generally profitable to maintain more processes than processors. In particular most processes reach points in their execution when they have no real need for a processor. For example, a process may have no need for a processor until an I/O operation is complete or a page is brought back to core, or perhaps until another process has completed a computation. In order to maintain the fiction of a process as always being in execution, Saltzer introduced the concept of a pseudo-processor which is the repositor of the processor state associated with a process. The pseudo-processor assigned to a process (there is a one to one relationship between pseudo-processors and processes) is "always executing" its process. When a process is actually in execution by a processor, its pseudo-processor state is contained in the state of the hardware processor. When a process stops running for a period of time its pseudo-processor state must be maintained until the next time the process executes.

Granted that it is desirable to maintain more processes than processors a mechanism must be found for handling the orderly switching of processors between the processes of the system. The Traffic Controller (of which the Process Exchange is an important part) introduced by Saltzer, will serve as our model. In the following refer to figure 4 (Program Structure of Simple Traffic Controller).

As mentioned above, from time to time a running process (i.e., one currently in execution) decides that it has no further need of a processor for the time being. The process declares this by issuing a call to the system primitive

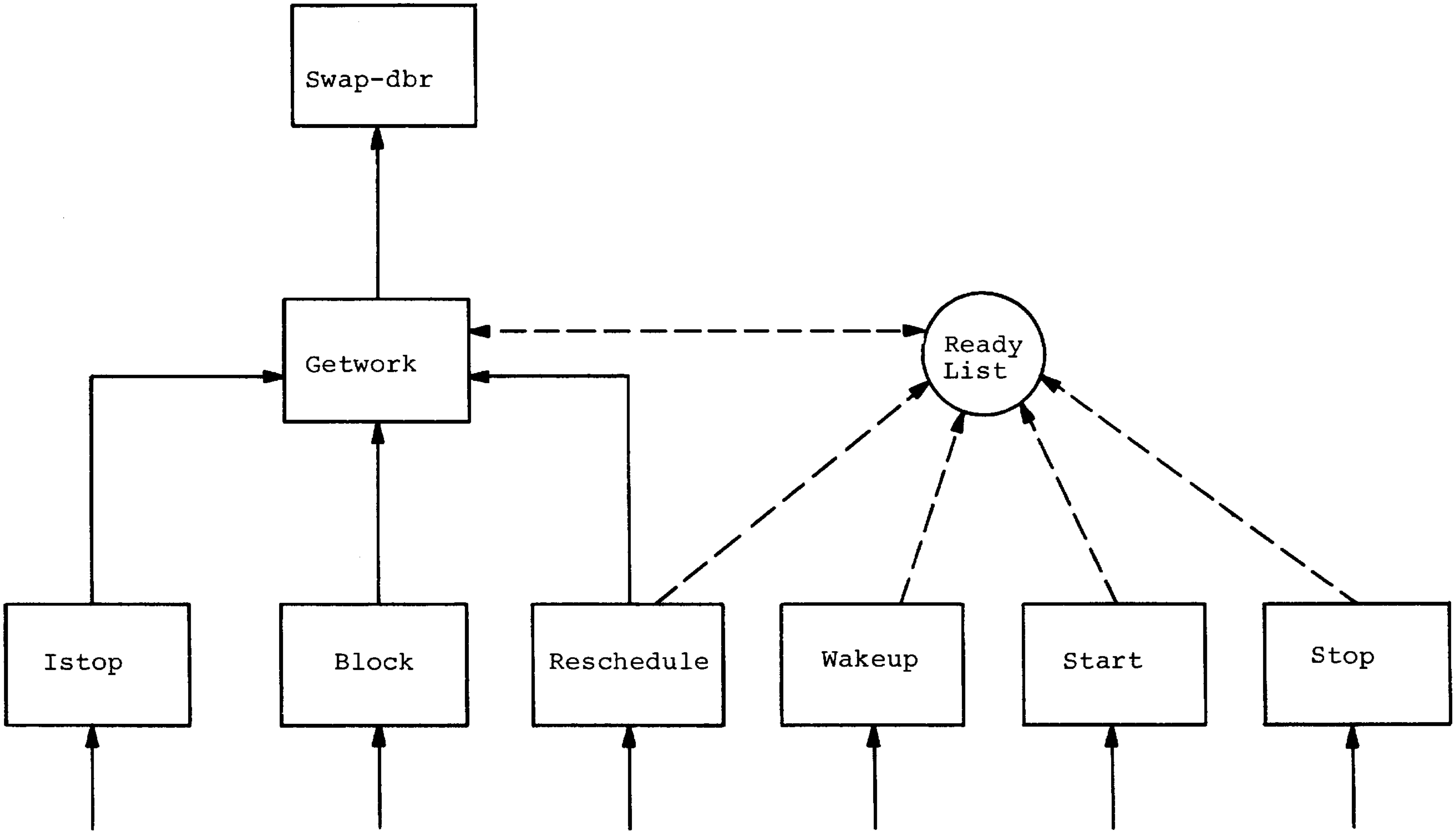

Figure 4. Block Diagram of Simple Traffic Controller. Solid Lines Indicate Closed Subroutine Calls Whereas Dashed Lines Indicate Data Flow.

function known as block. The purpose of the call to block is to give the processor away to some other process which can make effective use of it. In order to be able to identify processes which can make use of a processor, a shared data base, known as the Ready List, is maintained which lists all the processes which currently desire the use of a processor. (The Ready List will be implemented by a thread running through the Process Table linking the entries of processes desiring a processor.) Subroutine getwork, called by block, makes the selection of a process from the Ready List and then itself calls subroutine swap-dbr. The purpose of swap-dbr is to record the state of the current pseudo-processor (i.e., the current state of the processor itself) and load into the processor the state of the pseudo-processor associated with the process chosen from the Ready List. (Note, that having saved the pseudo-processor state of the process which called block, we will be able to resume its execution at a later time.) Now with the processor executing the newly chosen process, we return to the point at which this process stopped executing some ago. A process that has given up a processor by calling block is said to be in the blocked state, while a process which is executing is said to be in the running state, and a process on the Ready List is said to be in the ready state.

The only needed piece now, is a mechanism whereby a blocked process may become ready. This facility is provided by the system primitive known as wakeup. When a process

(process A for example) discovers some event that has occurred which is of interest to some other process (say process B) process A calls wakeup passing as an argument the process identification of process B. The function of wakeup is to place process B on the Ready List and redefine its execution state to the ready state, if process B is currently blocked. The current state of B (running, blocked, or ready) is maintained in the Process Table entry for B.

The dynamic operation of a simple system containing the block and wakeup primitives can be visualized. Running processes would execute until they reached points from which they could not immediately continue. They would call block to give away their processors to processes which were listed for them on the Ready List. Wakeup, when called on behalf of a blocked process would change the state of this process to the ready state and would append the process to the Ready List to await the appearance of a processor. Wakeup, when called on behalf of a running or a ready process would have no discernible effect.

A potential race condition would exist if block and wakeup were implemented exactly as described above. Consider the case where process A decides to call block if a particular condition is not met. Process A tests the condition, finds it is not met and proceeds to call block. Before A reaches block, process B reverses the state of the condition that A was waiting for and calls wakeup for A.

Because A is running (on the way to calling block) the call to wakeup has no effect. A meanwhile reaches block gives away its processor possibly forever.

This race condition can be resolved by having wakeup set on a switch or flag belonging to the process on whose behalf wakeup was called, no matter what state the process was in at the time of the call. If this had been the case in our example above, process A would have discovered the switch set on by process B's call to awaken A and A could have returned from block immediately instead of giving away its processor. This per-process switch will be known as the "wakeup-waiting" switch and will be maintained in the Process Table entry of each process.

Given this discussion we can present flow diagrams (see figure 5) for each of the subroutines discussed and understand their use and relationship.

Time Sharing, Pre-emption and Reschedule

In attempting to satisfy the processor requirements of a large number of processes in a system, it is necessary to have the ability to temporarily halt or interrupt the execution of a running process in order to give its processor to a more "worhty" process. This ability can be realized by using a hardware mechanism, an interrupt, and a software mechanism, an interrupt handling subroutine. The meaning of this interrupt to a running process will be that it has used up its current allotment of processor time and

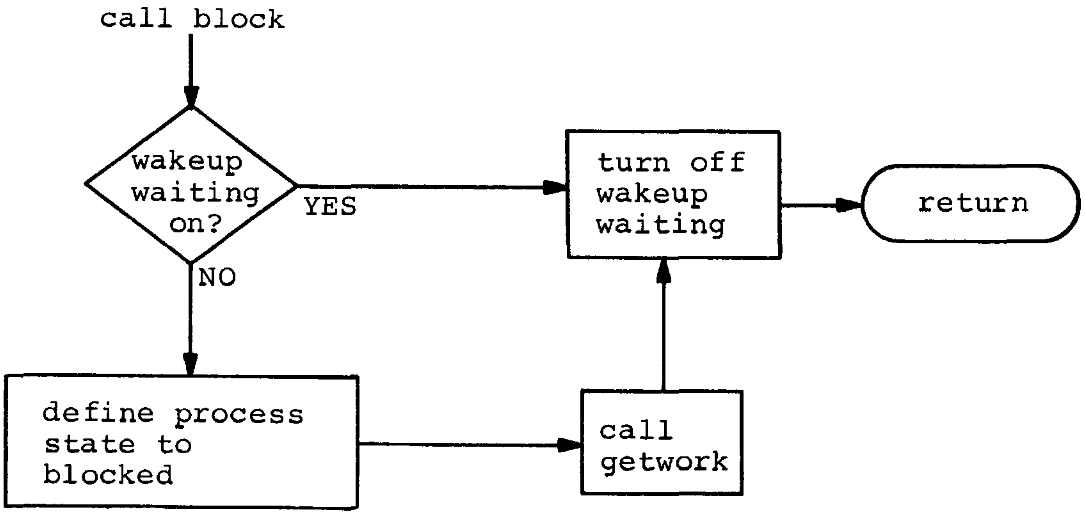

Figure 5a. Flow diagram of block.

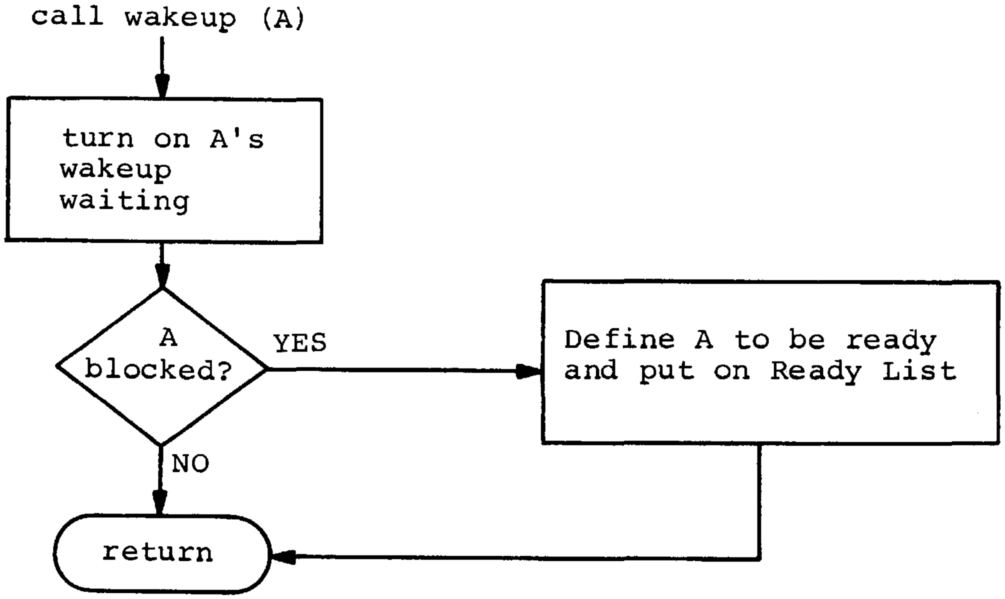
call wakeup (A)
turn on A's wakeup waiting
A blocked?
YES
NO
Define A to be ready and put on Ready List
return

Figure 5b. Flow diagram of wakeup.

that the process should prepare to give up control of the processor. This interrupt will be known as the timer-runout interrupt.

The interrupt handling subroutine, known as <u>reschedule</u>, is similar in function tn block, except that reschedule schedules future processor use for the process and places the process onto the Ready List directly, in addition to calling getwork to give away the processor.

One can imagine the changes in execution state that a process might go through. A running process may go blocked waiting for I/O or it may go into the ready state because it ran out of time. Similarly, a blocked process may enter the ready state as a result of a call to wakeup and a ready process may enter the running state as a result of being chosen to run by subroutine getwork.

Stop, Start and Istop

The ability to forceably halt the execution of a possibly run away process is essential. This ability is provided by primitive <u>stop</u>. Subroutine stop, called with one argument, effectively forces the process identified by the argument to call (or appear to call) subroutine <u>istop</u> and thereby enter the <u>stopped</u> state. Once in this state the process is dormant. It does not execute and calls to wakeup for it have no effect. The only way to bring the process back to "life" is for come other process to call procedure <u>start</u>. Istop and start are very similar to block and wakeup in function, differing only in the execution states with which they deal. Figure 6 gives a picture of process execution state transitions including the stopped state. The method by which subroutine stop actually brings a process to a halt, depends on the execution state of the process to be stopped, at the time of the call to stop. Stop operates by first turning on a flag in the Process Table entry of the process to be stopped (stop-pending flag) and then testing to see whether the process is currently running, ready, blocked or stopped. If the process is running, an interrupt is sent to the processor on which the process is executing. The interrupt will cause the process to call istop. If the process is ready or blocked, stop redefines the state to stopped and (if necessary) removes it from the Ready List. If the process is stopped at the time of the call, clearly nothing need be done. The stop-pending flag is of use as a running process to be stopped may be masked against stop interrupts. Figure 7 contains flow diagrams of stop, start, and istop.

Conditions and Events

As has been mentioned, the Traffic Controller primitives, block and wakeup, allow a process to relinquish a processor when the process cannot proceed with its execution. The reason that a process cannot proceed is that one or more "conditions" are not satisfied. An understanding of the concept of a "condition" and the

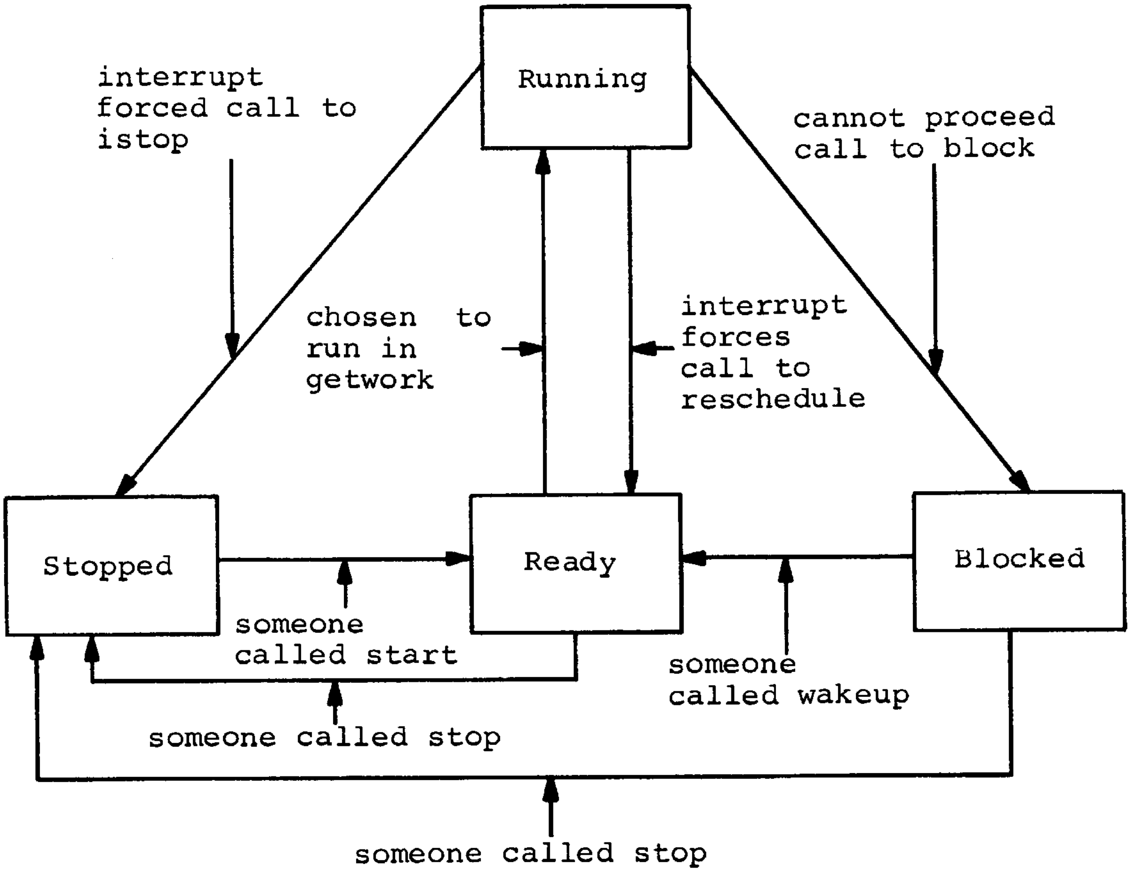

Figure 6. Execution state transitions

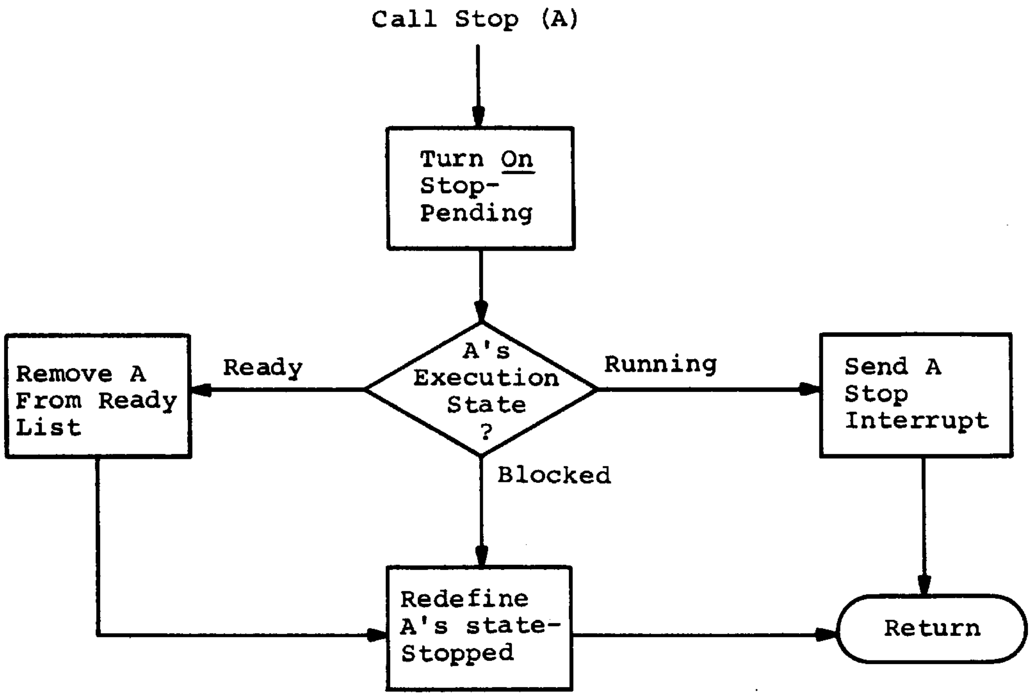

Figure 7A. Stop.

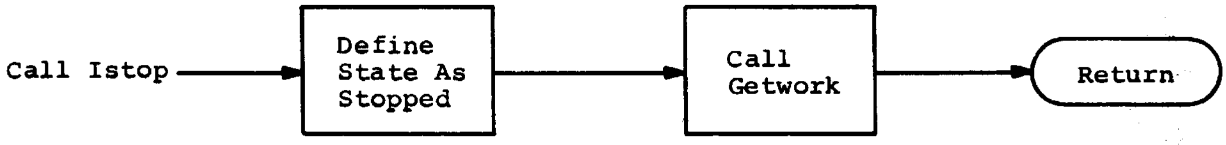

Figure 7B. Istop.

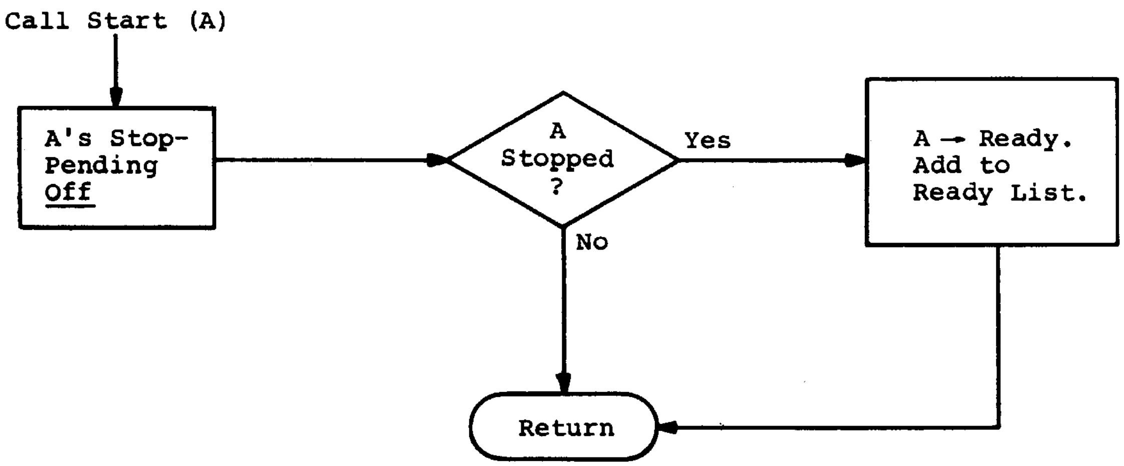

Figure 7C. Start

closely related idea of an "event" are useful when trying to understand the problems involved in implementing the Traffic Controller primitives.

A *condition* is a discrete valued function of one or more variables. The variables which make up the domain of the condition are functions of time which makes the condition also a function of time. The value of the condition at a point in time is known as the *state* of the condition. The state of a condition can only change at discrete instances of time and the state of the condition remains constant in the intervals between these discrete instants. Finally, an *event* is said to occur whenever a condition changes state.

An example of a condition is the function defined on a particular word of memory which is equal to zero whenever the word is equal to zero and equal to one otherwise. That is, if we denote the word by W and the condition by C, then:

$$C(W) = \begin{matrix} 0 \text{ if } W = 0 \\ 1 \text{ otherwise} \end{matrix}$$

Whenever the value of C goes from one to zero or from zero to one, an event occurs. If the value of W changes without changing the value of C, (e.g., W goes from 3 to 4) no event for condition C occurs. However this change in the value of W might denote an event for some other condition defined on W.

Another example of a condition is the condition defined to be equal to one whenever a particular page of a particular segment is in core memory and zero otherwise. To be more precise, this condition is a function of the page table word for the page and is equal to one if the page table word indicates the page is in core and is equal to zero if the page table word contains a fault or if the page table word itself does not exist (segment not active).

In general computations and processes are interested in the states of conditions and they are interested in events only in so far as the events indicate the current states of conditions. For example, given a condition that changes state (from zero to one) once and only once for all time, the fact that an event of the condition has occurred, tells us implicitly that the state of the condition for all time after the event has occurred is equal to one. Whereas, given a condition capable of changing its state after each unit of time, the occurrence of an event does not insure the future state of the condition for anything but one unit of time. Using the above examples we can divide the set of conditions into two classes: simple conditions -- those whose current states can be implied by a given observer from knowledge of a single event that has occurred in the past (not necessarily the last) and complex conditions -- those whose current states can only be ascertained by a given observer by explicit testing. A particular condition may be a simple one, to one observer, and very complex to another.

It is possible to analyze conditions to see where this inherent property of simplicity or complexity comes from.

Briefly, a condition is a simple condition to a particular observer if the observer is the only one who can control the state of the condition. If others can independently cause the condition to change state then the condition is complex to the observer.

The Traffic Controller primitives block and wakeup, provide processes with the ability to wait until events occur and to communicate the occurrence of events. Processes wait for events to occur by calling block and they signal other processes that events have occurred by calling wakeup. A process calls block to wait for an event of a condition whenever the state of the condition is such that the process cannot proceed. The rationale behind this is that after an event has occurred, the condition is more likely to satisfied. Of course if the condition is a simple one, it will be satisfied after the event with certainty.

As an example of the above ideas let us consider an algorithm that one might use in implementing a procedure that is used to interlock data. In a system making use of shared data bases, it is sometimes necessary to limit access to some of the data to one process at a time. This is done using so-called "interlocks". An interlock is a word (or a group of words) associated with a particular data base that serves as a flag to warn processes whether the shared data base is currently being accessed. By convention when the value of the interlock is non-zero, the data is being manipulated whereas when the value of the interlock is zero, the data is not being accessed. A process locks the data base by storing a non-zero value into the lock, at a time when the lockword is equal to zero. This operation can be done if hardware is available which allows a processor to read the lockword and store into it if and only if the word is zero. By convention the non-zero value that a process stores in a lockword is its own process identification. After a process is finished with a data base, it stores a zero into the lockword, thereby opening the data base up to be used by others. The locker procedure will be called with one argument: a pointer to the word serving as the lockword and the locker does not return until the lockword contains the value of the caller's process identification. From the point of view of the caller to the locker, the caller is not willing to proceed with the computation until the value of the lockword is equal to the process identification of the caller. We can define a condition that describes this situation. This condition is a function of the value of the lockword (denoted by L in the following). We will name the condition F and define the function as follows:

$$F(L) = \begin{array}{l} 1 \text{ if } L = \text{process-id of this process} \\ 0 \text{ otherwise} \end{array}$$

This condition is a simple one since only this process will ever place this process identification into the lockword (i.e., set F(L) = 1) and then subsequently place a zero in the lockword (i.e., set F(L) = 0) after the data base is no longer needed. Before the call to the locker, the process knows that the state of F is zero since the process executing in a serial fashion knows that it has not locked the data. Upon return from the locker, the process knows that the lockword has been locked and that F(L) = 1. Return from the locker can be considered the signalling of an event in the condition F to the caller of the locker. This condition is a simple to the caller, therefore it can determine the state of F.

Inside the locker procedure condition F is of little interest. The locker is interested in storing the value of the caller's identification in the lockword if the current value of the lockword is zero. In other words, the locker is interested in the condition, G, defined on the lockword, L, in the following manner:

$$G(L) = \begin{cases} 1 \text{ if } L = 0 \\ 0 \text{ otherwise} \end{cases}$$

Condition G is a complex condition, to the caller, since other processes are capable of changing the state of G (by storing their process identifications in L, for example). Therefore the only way that the locker can be positive that the condition G prevails is to test it explicitly (by means of the special hardware described which will store into the lockword only if the word is zero before the store). Basically, the locker is programmed as a loop waiting for the lockword to go to zero whereupon the caller' identification will be stored and a return will be executed.

Instead of looping, more efficient use can be made of the processor by making use of the Traffic Controller primitives and giving the processor away whenever the locker finds the lockword already locked. That is, if a process executing in the locker finds the particular lockword locked to another process (state of condition G = 0) the former process should somehow pass its name (identification) to the latter process and then call block. After the latter process unlocks the data base (i.e., causes state of G = 1), it can pick up the list of all the process identifications passed to it (including that of the former process above) and call wakeup for each and every such process to signal them of the occurrence of the event in condition G. Upon return from block our original process will again test condition G and attempt to lock the lockword if it equals zero. If the lockword is again locked to someone else, we again pass the process identification to the current process which has the lock set and call block again.

Another example of how conditions are defined and used is seen in the way I/O operations are usually handled. In the case of input, a buffer is assigned into which data will be read. Associated with the buffer is a flag that is set to zero before the input operation is begun and then set to one after the input operation is ended. The simple condition:

$$I(F) = F$$

where F denotes the flag, is the condition that the process waiting for the input is interested in. The process could loop waiting waiting for condition I to change state or it could block to wait for the event of I's changing state to occur.

A third and final example of an interesting condition is the one associated with the residence in core of a page. Each time a processes accesses a page, it must explicitly test the condition by referring to the page indirectly through the page table word. The process (through its processor) may never take for granted the fact that the page is located at a particular place in memory. When a needed page is not in core, the process waits for an event to occur associated with the particular condition defined on the page table word, knowing that after at least one such event has occurred it is more likely that the page will be in core.

CHAPTER FOUR

Evolution of the Traffic Controller

Hindsight shows that in designing the original implementation of the Traffic Controller several assumptions and policy decisions were made. These include:

1. It is always "desirable" to allow processes to do as much as possible by themselves. That is, among other things, processes should "load" (the idea of a loaded process will be defined later) themselves and schedule themselves.

2. The interlock strategy to be followed in the Traffic Controller would be to use numerous interlocks, rather than a single global interlock that limits access to the Traffic Controller to one process at a time. Such a strategy would permit numerous processes to execute in the Traffic Controller simultaneously and prevent the Traffic Controller from becoming a bottleneck.

3. The user interface to the Traffic Controller (i.e., the block-wakeup primitives) should remain ignorant of the various "customers" of these primitives. This was done in an effort to remain modular.

Performance analysis tests of the Multics system containing the original Traffic Controller showed that each of the above decisions was very costly for various reasons. As a result, a reworked implementation of the Traffic Controller was prepared and this second implementation was shown to be a marked improvement over the earlier version. Appendix I contains flow charts of the subroutines of the original implementation. It is not necessary for the reader to understand each step of each program. Rather these flow charts are meant only as a basis of comparison for the flow charts of Appendix II wherein the final implementation is illustrated. The remainder of this chapter describes why the Traffic Controller evolved in this way.

Original Traffic Controller Implementation

The traffic Controller originally implemented was basically the one described in Saltzer's thesis. There was one major point of difference between them and the difference was motivated by a logical problem that is not mentioned in the thesis presentation. This problem is known as the "lost wakeup" problem and it arises from the fact that the block primitive has two distinct customers in Multics (the user program and the basic file system) and that the memory provided by the wakeup-waiting switch is not sufficient to keep these customers from interfering with each other. The wakeup-waiting switch, as defined by Saltzer, is capable of remembering whether any wakeups have been received since the last (chronologically) call to block by a process. Since at any time while executing in user programs a process may experience missing page or segment faults which result in "concealed" calls to block, the last logical call to block by a process (i.e., the last time a user program called block) is not always equivalent to the last chronological call to block. An example will clarify this point.

Suppose process "A" cannot proceed with its execution until condition X is met. A takes the following steps:

1. Test if condition X is met. If yes go on to step 4.
2. Call block since condition X has not been met.
3. Go to step 1 to test again.
4. Continue..

Suppose further that process A completes the test in step 1, finds that condition X is not met and proceeds to step 2. While executing step 2, (i.e., calling block) but before step 2 is complete, process A experiences a missing page fault that results in a call to block from within the missing page fault handling subroutine. Having called block, process A gives up its processor until some time after the next call to wakeup. However, in this case, suppose that two wakeups are received before process A resumes its computation: the wakeup signaling an event in condition X and the wakeup signaling the arrival of the missing page. These two wakeups leave the wakeup-waiting switch *on* however upon return to the caller of block (i.e., the page fault handler) the switch is turned off. Since the page has been retrieved, process A resumes its execution of step 2 above and completes the call to block. In this case block finds the wakeup-waiting switch *off* even though an event in condition X has been signaled. The reason is that the last logical call to block, which occurred before the execution of step 1, was not the last chronological call to block.

The way that this problem was solved in the original implementation of the Traffic Controller involved two steps. First a wakeup-waiting count was substituted for the wakeup-waiting switch. Second, in addition to wakeup incrementing this count on each call, block was also given the ability to selectively increment the count in special cases. The change to wakeup was trivial, the count was incremented instead of the switch being turned on. The change to block was more fundamental. Block was provided with an argument that was used in the following way:

1. Arg = 0.

2. Test is condition is met. If yes go to step 5.

3. Call block (Arg).

4. Go to step 2.

5. Continue...

Whenever block returned, the value of Arg was set equal to the value of the wakeup-waiting count prior to its being decremented by block. If a process returned from block and upon testing the condition in step 2 found it had still not been met the recall to block would transmit a non-zero value of Arg to block. Block would notice this and interpret this to mean that a caller to block received a return when in fact its condition had not been met. Therefore the wakeup-waiting count should be incremented by 1, in order not to lose track of the wakeup that was decremented by the incorrect return from block. After having incremented the count the value of the count is compared to the value of Arg (which was the value of the count at the time of the last logical call to block). If the count is greater than Arg, wakeups have been received since the last logical call to block and return should be immediate whereas if the count is equal to Arg, no wakeups have been received since the last logical call and the process should give the processor away.

Other known solutions to the lost wakeup problem exist. One such solution would, in fact, allow the block primitive to be implemented in exactly the way described by Saltzer. This solution, however, would force some users of the block-wakeup facility to exercise the facility in a slightly different way than has been previously stated. Specifically all those procedures, using block which might be invoked because of a fault condition (e.g., the missing page fault handler) would be required to turn on the wakeup-waiting switch, explicitly, after each successful return from block. That is, such procedures would be required to use the following algorithm to go blocked.

1. Test if condition is met. If yes go to step 4.

2. Call block.

3. Go to step 1.

4. Turn on wakeup-waiting switch and continue.

Use of this algorithm would solve the lost wakeup problem because concealed calls to block would always result in the wakeup-waiting switch being left on. This algorithm would also mean that many returns from block would be in cases where wakeups had not been received, but rather that page faults occurred between two logical uses of block.

One other difference between the Traffic Controller described by Saltzer and the original implementation is that the implemented one had an elaborate interlocking mechanism which allowed any number of processes to be executing in the Traffic Controller simultaneously. This meant that several intermediate execution states (i.e., in addition to the ready, running, blocked, and stopped states) had to be defined and detectable. An example of such a state is the state that occurs after a process defines itself to be ready (in reschedule) but before it can give away the processor in swap-dbr. Another such state occurs after a ready process has been chosen to run (by some other process executing in getwork) but before it is actually given control of a processor. In the flowcharts presented in Appendix I the reader sees all the logical complication introduced by this strategy. (It is not necessary to understand the interlocking mechanism for one to continue reading and therefore the reader is advised not to laboriously pore over these flow charts to gain such an understanding.)

A further point should be mentioned before proceeding. A loaded process is defined as one that has enough information in core memory so that the process can be given a processor. Operationally in Multics, this means the process has, in core storage, a descriptor segment and also another segment known as the Process Data Segment (PDS) which contains a special stack (known as the concealed stack). This stack is only used, by a process, when executing procedures which cannot tolerate page faults. An example of such a procedure is the page fault handler procedure. The strategy, with regard to unloaded processes, adopted in the original Traffic Controller was that processes were to "load themselves". That is, when called upon to switch to an unloaded process, an interim descriptor segment and an interim PDS were created for the unloaded process and this process was switched to and made to use these interim segments. The process was forced to branch to subroutines (process bootstrap module) which recreated the original descriptor segment and retrieved the original PDS to core storage. Having thereby loaded itself the process then switched from its interim segments to its actual segments and discarded the interim versions.

This approach to the problem of loading processes actually solves the problem not by allowing the process to load itself, but rather by effectively creating a new process manifested by the interim segments, which accomplish the loading. This new process however executes in the address space of the unloaded process.

Finally, one may notice references in the flow charts of Appendix I to programs not mentioned previously in the text. All of these programs are ones which disappear from the second implementation. They include ready-him (see figure I.10) which implemented the ability to switch to a process' address space in order that the process schedule itself at the time the process was awakened; setup-proc (and unsetup-proc) used to create (and destroy) an interim process; and ready-ds (and unready-ds) used to create (and destroy) an interim descriptor segment for a blocked, unloaded process that was awakened.

Reworking the Original Implementation

The performance of the original implementation of the Traffic Controller in action was analyzed and found to be deficient in several ways. In the first case it was too slow in execution. In the second case, the size of the programs (which were required to be permanently resident in core storage) was large. Several logically independent improvements were incorporated in the second iteration of the Traffic Controller which improved performance substantially. However since these improvements are logically independent they are presented as if they were implemented in a serial manner, one by one, instead of all at once. This allows us to discuss them independently and to understand the motivations behind each of the changes.

Interlocking Changes

From just a casual perusal of the flow charts in Appendix I, one can see that a large portion of each flow chart is devoted to implementing the complicated interlock strategy. In point of fact, more than 30 per cent of the machine instructions in the original Traffic Controller were there for this purpose alone. For this reason and for several others, a simple strategy was incorporated into the second implementation. The simplified scheme works as follows. Only one processor is ever allowed to execute in the Traffic Controller at any one time. This is accomplished by the use of a single global interlock. Since a processor can be forced to attempt to call subroutine wakeup by the receipt of an external interrupt to the processor, all interrupts must be masked by the processor executing inside the Traffic Controller to prevent a processor from looping forever on a lock that it had set itself. The scheme is implemented by providing code at each external entry into the Traffic Controller, whereby the processor masks itself and then attempts to lock the global interlock. If the interlock is already locked, the processor loops, waiting for it to become available. Use of this strategy does away with the need to worry about intermediate execution states of processes, since these states are now invisible behind the global interlock.

As was mentioned above, there are several valid reasons for abandoning the complex interlock strategy. The first one is that the complex strategy costs alot and buys little. The only gain experienced from this strategy is that the Traffic Controller is prevented from being a bottleneck that processors continually bang into. A paper presented to the national conference of the Association for Computing Machinery in August of 1968, by Stuart Madnick (reference number 5), discusses this problem of processor time wasted in looping on global interlocks. Madnick's conclusions show that in a system with a small number of processors (e.g., five or less) the probability of encountering a locked interlock is roughly proportional to the number of processors in the system and to the fraction of time that a processor spends executing such interlocked code, compared to total execution time. That is, if processors on average spend two per cent of their time executing such interlocked code and if the system contains three processors, then approximately six per cent of the time a processor will encounter such an interlock locked. Since the Multics system currently contains only two processors (and can never have more than seven due to hardware limitations) it seems as though the Traffic Controller even with a global interlock, is in no danger of becoming a bottleneck. Secondly, use of the complex scheme means that extra code is executed on each pass through the Traffic Controller. As stated above, approximately 30 per cent of the original code was devoted to implementing the complex strategy. Since there are no loops to speak of, in these programs, we can assume that the average running time of Traffic Controller programs was increased by approximately 30 per cent as a result of the extra code. The simple strategy tends to increase the average running time of the Traffic Controller by locking out processors for periods of time, however as stated above, until we introduce a large number of processors into the system, this increase in average running time does not approach this 30 per cent figure. Therefore the only possible motivation for the complex scheme is not justified in our case.

The second major problem with the complex scheme is in maintaining it. Even minor changes to Traffic Controller modules mean that the interlock strategy would have to be scrutinized to assure that no bugs had been introduced. Because of the complexity of the scheme, such scrutinization is difficult and would effectively rule out changes to the Traffic Controller once it was "completely debugged".

Summarizing these reasons, we see that the original scheme suffered from:

1. Wasted space devoted to instructions realizing the complex interlock strategy.

2. Wasted time spent executing the above instructions.

3. Difficulty in maintaining and debugging this scheme.

4. No real need for such a scheme.

Changes in Wakeup Scheme

In the flow charts of Appendix I presented for subroutine wakeup and ready-him, we see that calls to wakeup for blocked processes result in a processor switch to the address space of the blocked process in order to call the scheduler in that address space. The original motivation for such a strategy was that each process could potentially have its own scheduler procedure and that such a scheduler could most sensibly make any decisions necessary in the address space of the process. For example, private data bases of that address space could be used in arriving at decisions. However, the decisions that must be made in awakening a blocked process (such as where in the Ready List the process should be inserted, what priority, etc.) need not be made at the time of the call to wakeup. They can be made at the time that the process puts itself into the blocked state and the data can be left in a shared data base. Then at the time of the wakeup no decisions need be made. The process need merely be placed on the Ready List at the point indicated by the data left behind. In this way the work (and the expense) associated with switching address spaces at the time of wakeups can be dispensed with.

This revised strategy brings savings in two ways. First two subroutines disappear: ready-him and ready-ds. Second, the time spent in awakening blocked processes (particularly unloaded ones) is decreased. This revision meant minor changes to block and wakeup. In block this revision amounts to computing the priority to be given the process when it is awakened. The criteria for making this decision will be ignored for the present. In wakeup the change amounts to having wakeup directly place the awakened process on the Ready List at the appropriate place.

Changes to Loading Scheme

The next major logical change to the Traffic Controller comes in the area of loading processes. in the original implementation, unloaded processes loaded themselves when they were chosen to run by subroutine getwork. The process choosing the unloaded process to run had to provide the unloaded process with an interim descriptor segment and an interim PDS in order for the unloaded process to be able to load itself. That is, the choosing process had to effectively create a new process that would perform the work of loading the unloaded process. In addition, the unloaded process, once switched to, had to choose a candidate for unloading (i.e., a loaded process to be unloaded) if by loading itself, the number of loaded processes would cross a threshold. This scheme had several drawbacks.

1. Some of the procedures involved in process loading were required to be always resident in core because these procedures were called directly by the Traffic Controller at a time when page faults could not be tolerated. These procedures included the file system procedures which prepared the interim segments and discarded them after use (setup-proc and unsetup-proc).

2. During the execution of process loading, the loading process ties up twice as much core storage as does an ordinary process. It has two descriptor segments (interim and real) and it has two PDS segments (interim and real).

3. The above solution for the loading problem means that loading decisions be made during execution of the Traffic Controller, in getwork and swap-dbr specifically. In a Traffic Controller using a global interlock strategy, serious bottleneck situations could develop if such decisions involved much computation.

Once it is observed that the above loading scheme effectively creates (and later destroys) a new process for each loading operation, a more efficient solution to the problem immediately comes to mind. Why not provide a system process dedicated to loading and unloading other processes? The revised Traffic Controller makes use of such a special system process devoted to loading and unloading processes. This special process, referred to as the "loader-daemon" process, spends its life executing in a loop. Whenever the loader-daemon is awakened, this process peruses the Traffic Controller data bases, makes decisions as to whom to load and unload and actually performs the loading and unloading. The loader-daemon then calls block before perusing the data bases again. The mechanism whereby the loader-daemon is periodically awakened and the actual algorithms whereby the loading decisions are made will be ignored for now and will be presented in a later section. The consequences of the revised loading scheme to the modules of the Traffic Controller are manifested in the immense simplification to subroutine swap-dbr.

Block-Wakeup Facility Environment

The block-wakeup primitives provide the capability whereby one process can signal a second process about the occurrence of an event of a condition that this second process is interested in. The extra mechanism that must be provided before such a facility becomes useful is the mechanism whereby this second process can inform the first process that it is interested in the condition. for example, suppose process A goes blocked waiting for a missing page to be returned to core storage. When the I/O request for the page has been completed, some other process will notice this event and must be able to ascertain that process A must be awakened.

To handle the communication problem between processes executing in the file system, a module, known as the Process Wait and Notify (PWN) module, was designed. This facility allowed processes to post messages to all other processes. Sample messages might say in effect: "Process A is waiting for page X". The facility was organized around a data base known as the Process Wait Table (PWT). The PWT was required to remain in core at all times.

Basically the PWN facility maintained a finite number of lists in the PWT. The number of lists was a system constant. The lists were referred to by the numbers 1, 2, ..., N where N is the number of lists. Each element on a list was conceptually a process identification. The facility operated in the following way. Each possible condition of interest in the file system was associated by convention with one list. Several different conditions might all be associated with the same list but one condition could not be associated with several lists. When a process decided to go blocked to wait for a condition to be met, the process merely allocated an entry in the PWT, wrote its process identification into this entry and threaded the entry onto the list associated with the condition. The process then called block to give away the processor. When a process noticed the occurrence of an event of a condition it would pick up the entire list associated with the condition and call wakeup for each process on the list and deal locate all the entries on the list.

The PWN facility offered its users four entry points: <u>addevent</u>, <u>delevent</u>, <u>wait</u>, and <u>notify</u>. Addevent allowed a process to allocate an entry and to thread the entry onto a particular list. Delevent allowed a process to unthread itself from a list and deallocate its entry. Wait allowed a process to check that it was still on a given list. If the process was still on the list wait called block. If not, wait returned. Notify allowed a process to pick up an entire list, call wakeup for each process on the list and unthread the entries from the list.

These entries were exercised in the following way. A process desiring to wait for condition X would:

1. Test to see if X was met. If yes go to step 7.

2. If X was not met, call addevent(X) to thread the process onto the list associated with condition X.

3. Test X again. (The reason for the retest is that an event of X may have occurred after step 1 but before step 2.) If yes, go to step 6.

4. If retest fails, call wait(X). This may result in a call to block.

5. Upon return from wait go to step 1.

6. Control comes here only from step 3. In this case call delevent(X) to unthread process from list.

7. Control comes here from steps 1 or 6. In either case, X has been met so the process merely continues.

The above described PWN facility was implemented in Multics and was found to be exceedingly inefficient in operation. In studying ways to improve the performance of this critical module, a redesign of the inner workings of the PWN module was put forward which essentially incorporated the PWN facility into the Traffic Controller. The implementation of this new package has improved the PWN performance by about a factor of ten.

The decisions that led to the redesign of the PWN module were made after several facts became clear.

1. If the PWN entry wait is only executed while the Traffic Controller global interlock is <u>on</u>, then wait need not call subroutine block to give away its processor, it can call getwork directly.

2. Similarly if entry point notify is only executed while the global lock is on then it need not call

wakeup for each process on a list, it need merely put the waiting processes on the Ready List directly.

3. The time spent allocating and deallocating entries in the old PWT was almost completely unnecessary due to a combination of two factors. First, processes in the file system never wait for more than one thing at a time. Therefore, a process never needs more then one entry in the PWT at any one time. Second, a table already exists (permanently in core) which has one entry per process: the Process Table. If the lists previously in the PWT, were instead implemented as threads running through the Process Table then a considerable savings could be affected. Space would be saved as the PWT would become obsolete and time would be saved as the need for allocation and deallocation of entries would no longer be necessary.

4. If wait is altered so that it no longer calls block, then block can be simplified. Recall that it was only because of file system use of the primitive that the "lost wakeup" problem arose. Therefore, block can now return to the simple subroutine proposed by Saltzer.

Given these insights into how to improve PWN performance, the PWN facility was incorporated into the Traffic Controller.

Design of New PWN Facility

The intent in redesigning the PWN facility was to essentially preserve the original PWN interface but to provide a more efficient implementation for it.

The redesigned implementation is still organized around the idea of a finite number (N) of lists or threads. The method adopted for implementing these lists is the creation of a table known as the Active Thread Table (ATT). Each such thread has an entry in the ATT. An ATT entry consists of two pieces of data: a single bit wnich indicates whether the associated thread is active (bit = 1) or inactive (bit = 0) and a pointer which is only valid if the thread is active and which then points to the Process Table entry of the first process threaded on the list. If a thread is active and no processes are as yet on the thread, the value of the pointer is that of a null pointer. The ATT entries each occupy a single word of core storage so that the entire storage requirements for the ATT is N words.

The four PWN subroutines in redesign, turn out to be very simple affairs. In fact one of them, delevent, simply disappears and becomes an alternate entry for notify. All of the subroutines are called with one argument: the "name" of the condition. A name is assigned to a particular condition by conventions honored by the procedures interested in the particular condition. Condition names need not be unique (i.e., two distinct conditions could have the same name) however, all references to any particular condition must use the same name. Each condition name is automatically associated with a particular thread by a simple mapping scheme used by each of the PWN subroutines.

Subroutine addevent is the simplest of all. The function of this subroutine is simply to "activate" the thread associated with the condition named by the argument to addevent. That is, addevent associates the condition with a thread by mapping the condition name passed to it into one of the integers between 1 and N. Then addevent simply turns on the bit item in the ATT entry associated with the chosen thread. If this thread was active before the call to addevent, then addevent has no discernible effect. Subroutine addevent is used in exactly the same way in which the old subroutine named addevent was used. That is, it is called in step 2 of the table presented earlier.

Subroutine wait is called when a process desires to give up the processor because a particular condition has not yet been satisfied. The actions of wait are also simple. If the thread associated with the condition passed to wait is active, then wait threads the calling process onto this list and calls getwork to give away the processor. If, on the other hand, the associated thread is inactive, then wait immediately returns to its caller. Since a process calls wait only after explicitly calling addevent, this thread can only be inactive if an event of the condition has been notified after the call to addevent but before the call to wait.

Subroutine notify is called when a process notices the occurrence of an event of a condition. The results of a call to notify are that the associated thread is left inactive and each process previously listed on that thread are put onto the Ready List. Notify accomplishes this in a straight forward way. Upon entry, the ATT entry associated with the thread is picked up and its value is saved, while the ATT entry itself is reset to the inactive state. That is, the bit is set to zero and the pointer is set to the null value. Then the saved contents of the ATT entry are examined. If previously the thread was active then the pointer points to the head of a list of processes. The last process on this list has a null pointer to indicate the end of the list. Therefore, notify simply goes down the list putting each successive process on the Ready list until a null pointer is encountered. If the thread was inactive before the call to notify then notify has no tangible effect. If the thread was active but its list was empty, notify will leave the thread inactive.

Subroutine delevent, in the old scheme, was the inverse of the old addevent. A strict inverse of the new addevent would deactivate an ATT entry if its associated thread were empty and have no effect otherwise. From the above we can see that notify performs as an inverse to addevent if the list is empty, but if it is not empty then notify does more.

Because of a three way race condition that exists, we can show that the extra computation performed by notify is necessary to prevent a process from possibly placing itself on a list for an indefinite length of time. An example that illustrates this race condition makes use of two distinct conditions, X and Y, which both are associated with the same list, L, and four processes, A, B, C, and D. Suppose process A is interested in awaiting the occurrence of an event of condition X, process B is interested in the occurrence of an event of condition Y; process C notices the occurrence of an event of condition X; and process D notices the occurrence of an event of condition Y. Suppose further that the sequence of actions occur in the following order:

1. A calls addevent(X) prior to waiting for X to be satisfied. This call results in thread L being activated. A then tests and finds X has not yet been satisfied.

2. C calls notify(X) to signal the occurrence of an event of X. This call results in deactivation of thread L.

3. D calls notify(Y) to signal the occurrence of an event of Y. This call has no effect.

4. B calls addevent(Y) prior to waiting for Y to be satisfied. This call results in thread L being activated. B then tests and finds condition Y is

satisfied. Therefore, B will not call to wait for Y.

5. A calls wait(X). Since thread L is not active (from step 4) this call results in A threading itself onto thread L.

6. B calls delevent(Y) to undo what it did in step 4. If delevent were a simple inverse of addevent, it would discover thread L active and with a non-empty list. But consider process A on this list waiting for an event of X which has already been signaled. If B leaves this list as is, then A will stay on this list until some process calls notify to signal an event of a condition which accidentally happens to be associated with list L. Therefore, B must place A on the Ready List and deactivate the thread. In other words we see that delevent and notify have the same function and should be merged. The reason that this race condition did not come up in the old PWN was that in the environment a process could have several entries in the PWT and be on several lists concurrently. In that scheme the addevent of step 1 above would have created a new entry for process A on thread L which would have been explicitly deleted by the notify of step 2.

As was stated above, this new implementation of the PWN facility solves the lost wakeup problem. However, a skeptical reader might object that this is not clear. At first glance the lost wakeup problem might appear to have been replaced by a "lost notify" problem. That is a process can now get a fault on the way to calling subroutine wait and the fault handler may itself call wait. However, such is not the case as can be seen after careful analysis. As was stated previously, the old lost wakeup problem could have been solved by having all file system users of block, explicitly turn on the wakeup-waiting switch after a successful interaction with block. This is in effect the way the "lost notify" problem is solved. We can show that in effect, the single bit data items in each ATT entry serve as wakeup (or notify) waiting switches. However, they differ from the actual wakeup-waiting switches in two ways. First, they do not belong to individual processes but rather they are shared among all processes. Second, we will say that such a switch is *on* when its contents are zero and that it is *off* when its contents are one. Then we can reinterpret the actions taken by a process about to call wait for condition X in the following manner.

1. The process calls addevent(X). This results in the associated thread being activated. That is the bit associated with the thread is turned *on*. In our re-interpretation we say that the system wide notify-waiting switch associated with this condition is explicitly turned *off* by this action.

2. The process calls wait(X). If the thread is still active the process puts itself on the list and calls getwork. Re-interpreting, if the system wide notify-waiting switch associated with this condition is still off then give the processor away. On the other hand if the thread is inactive wait immediately returns which can be re-interpreted to mean that the system wide notify-waiting switch is on and therefore wait returns. Notice that wait leaves this notify-waiting switch on (bit = 0) when it returns, therefore, we get the effect of leaving such a switch on after successful interaction by default rather than by explicit action.

3. Calls to notify deactivate threads which can be seen as turning on notify-waiting switches.

Since these subroutine effectively implement this solution to the lost wakeup problem, we see that no such problem exists.

Loading and Unloading of Processes

Until this point, the mechanism whereby processes are loaded and unloaded has only been alluded to. With the description of this last piece, the entire logical structure of the Traffic Controller will have been presented.

In many phases of human endeavor one seems to function at levels substantially below one's capacity when one attempts to function above this capacity. Similar statements (although not proved theorems) can be made about computing systems. That is, if you attempt to do more work then the system is intrinsically capable of performing, you will wind up accomplishing less than the system is capable of. A qualitative example drawn from the Multics system should clarify this. Suppose that in a typical system we have 100K of core storage available for user processes. Further suppose that on average, a process needs about 25K of core storage for its own needs. This storage is used for procedures, private data segments, etc. If the system attempts to multiplex the hardware resources between eight processes at a time, then it is clear that a considerable amount of system capability will be expended by processes trying to keep their own pages in core by removing pages belonging to other processes. It is important that the loading-unloading scheme we use be capable of preventing such wasteful thrashing.

With this brief introduction we are able to present the three independent groups of processes on which the Multics loading scheme is based. The first is the group of processes listed on the Ready List which is a list of those processes which <u>want</u> to execute. It is not necessarily the list of processes that the system is willing to run, but merely those wishing to run. This distinction is important. The second group is the group of loaded processes. The processes in this group are capable of running if given a processor, in that they have enough information in core storage to be able to handle all fault conditions (e.g., missing page faults, etc.). Although all loaded processes are capable of running, only those concurrently on the Ready List actually wish to run. The third group is called the eligible group and consists of those processes that the system is actually willing to run at any one time. That is, a system wide limit is set on the number of processes that the system is willing to multiplex between at any one time. The factors entering into the setting of this parameter include the amount of core storage available to the system and the average amount needed by a typical process. The algorithm used in subroutine getwork whereby it chooses processes to run can be seen to be simply: choose the first process on the Ready List which is also loaded and eligible. There is always at least one such process on the Ready List due to the existence of so called "idle" processes, one per processor in the system, which serve to soak up excess processor capacity when no other customers are available. The task of the loader-daemon process can now also be stated briefly. Its task is simply to keep all eligible processes in the loaded state. This of course may involve unloading previously eligible processes which have lost their eligibility.

The scheme is simple and works in the following way. A running process upon giving away its processor decides whether or not to give away its eligibility. If it does so decide, then this retiring process bestows its forsaken eligibility upon the most "worthy" process in the system. By definition, the most worthy process is the first process on the Ready List which is not eligible. If by chance, this newly eligible process is not currently loaded, then the retiring process informs the loader-daemon process of this by waking it up. It is the task of the loader-daemon, upon awakening, to determine that an eligible process needs loading and then to load it.

Given this definition of most worthy, one small problem crops up. What happens if no such worthy process exists at the time eligibility is forsaken because all ready processes are already eligible? This is a problem since, a previously blocked process which has lost its eligibility may at any moment appear on the Ready List and be entitled to eligibility which is available. The problem is solved by having all subroutines which add processes to the Ready List, such as wakeup, compare the number of currently eligible processes to the maximum permitted number of eligible processes whenever a process is added to the Ready List. If the current number is less than the maximum number (this implies all ready processes are eligible) and if the process being added to the Ready List is ineligible, then eligibility is automatically conferred upon this process, since by definition it is the most worthy (in fact the only process on the Ready list not eligible). If this process is unloaded, then a wakeup is sent to the loader-daemon process and it will load this newly eligible process.

The only piece not yet presented is the algorithm whereby processes give up their eligibility. If the call to give away the processor originated because the process has used up its allotted time quantum or if the process called to block itself, then the process gives away its eligibility. Calls to subroutine wait are always performed by a process executing in the file system. The process is executing in the file system because of a deliberate call to a file system primitive, or because of a page or segment fault. In the first case it is not wise to take the process' eligibility away in that the process may have one or more system wide interlocks locked. In the second case it is not wise to remove its eligibility because then the process, whose page will be in core in a very short time, would not be able to execute when its page or segment is brought back to core storage. The alternative to this policy would lead to a thrashing situation. Stated simply then, calls to reschedule and block result in loss of eligibility while calls to wait do not. Of course, calls to reschedule and block by the loader-daemon and the idle processes do not result in loss of eligibility since these special processes are guaranteed to be eligible always. Operationally these processes can be identified by a flag in their respective Process Data Segments. The flag, known as the special-process flag is examined on each trip through reschedule and block. If its value is zero this indicates

The caller is an ordinary process which should relinquish its eligibility. If the flag's value is one however then this indicates the caller is a special process which should retain its eligibility.

Some people might protest that the above criteria for giving up eligibility might lead to needless thrashing in certain instances. For example, suppose we have a process which is reading (or writing) records from (onto) a tape. This process blocks itself and is awakened very frequently due to the high volume of I/O being transmitted. One might think that with the above criteria, this process is liable to be unloaded after each call to block. However, if the maximum number of loaded processes exceeds the maximum number of eligible processes, such need not be the case, since in this case, a process will probably not be unloaded unless it were blocked for some length of time. As soon as the process was awakened it would be likely to regain its eligibility immediately.

In the implementation of this scheme the state of a process as to whether it is loaded, unloaded, eligible, or ineligible is defined by the value of a variable maintained in the Process Table of the process. The variable, known as the runability-state variable, can take on values between one and four. They are interpreted as follows:

1. Variable equals one means that the process is unloaded and ineligible.

2. Variable equals two means that the process is unloaded but eligible. This state prevails immediately after an unloaded process has been granted eligibility but before it has been loaded.

3. Variable equals three means that the process is loaded but ineligible. This state occurs after a process gives up its eligibility.

4. Variable equals four means that the process is both loaded and eligible.

In subroutine getwork the choice of a process is made by looking for the first process on the Ready list whose runability state is equal to four. The length of such a search tends to be short since processes in this state are usually near the top of the Ready List. Candidates for unloading can be found through use of the Blocked List, which lists all blocked processes in the order in which they went blocked (oldest blocked process last), and the Ready List. One merely scans the Blocked List backwards to find the oldest blocked loaded process. If no such process is found, then one scans the Ready List backwards to find the lowest priority loaded, ineligible process. Candidates for unloading are needed whenever the total number of loaded processes is equal to the maximum number and eligibility has just been granted to an unloaded process. At such a time at least one loaded process is ineligible since clearly the maximum number of loaded processes is at least as great as the maximum number of eligible processes and the only reason that eligibility has been conferred on an unloaded process is because some previously eligible process has just given it up.

How one chooses the maximum numbers of loaded and eligible processes permitted is an interesting question. At present these are constants set at the time the system is initialized and little thought has been given as to algorithms whereby they might vary dynamically during system operation.

Block and Scheduling

As was mentioned previously, the decision to forego scheduling at the time of a call to wakeup meant that some form of scheduling occurs in block. To understand the following discussion it is important to know the structure of the Ready List.

The Ready List is actually an ordered collection of lists, known as queues, strung together. The queues are numbered from 1 to N, where N is the number of queues. Each process maintains two data items in its Process Data Segment: the number of the highest queue which it is allowed to be in and the number of the lowest queue which it is allowed to be in. For example, an idle process has both of these data items set to N, the lowest queue, since this process is supposed to permanently occupy the bottom of the Ready List. On the hand, the loader-daemon has both of these items set to 1 to indicate that when ready, this process always has high priority. User processes might be expected to have these set to 2 and N-1. Associated with each queue is a time limit, which is the amount of time a process is allowed to remain at that level. At the end of a time limit a process goes down one queue level until it reaches its lowest level whère it remains. If something known as an "interaction" occurs however, a process is moved back to its highest level from wherever it was. A process maintains its current queue level in its Process Table entry. At the time of a call to wakeup for a blocked process, it is this queue in the Ready List, onto which the process is entered. At the time of a call to block, it is possible to tell whether an interaction has occurred and therefore, whether the process queue level should be set to the highest level. This information is passed to block via an argument, known as the interaction switch, from the caller to block. It should be mentioned that all calls to block are filtered through a supervisor program known as the "wait coordinator". It is this supervisor program, not the user program, which makes the determination as to whether an interaction has occurred.

This then concludes our discussion of the implementation of the Traffic Controller. As mentioned previously, the flow diagrams of each of the Traffic Controller subroutines as well as the flow diagram of the driver program executed by the loader-daemon are presented in Appendix II. These flow charts represent the completely evolved Traffic Controller as implemented and working.

CHAPTER FIVE

Process Creation and Destruction

Introduction

As has been stated previously, a process may be defined as the combination of an address space and a set of processor conditions. Creating a process (assuming one process per address) means creating both these pieces, of which the creation of the address space is by far the more formidable task. Destroying a process means destroying these pieces plus all the no longer needed segments (files) that the process created in its lifetime. As was the case with the Traffic Controller, the process creation implementation has gone through a major revision to improve its performance. An understanding of the problems involved in process creation in Multics demands at least some familiarity with several parts of the Multics system. In particular, the reader should have some knowledge of how the "dynamic linking" mechanism of Multics works. For a detailed description of this mechanism, the reader is directed to a paper by R.C. Daley and J.B. Dennis of M.I.T. entitled, "Virtual Memory, Processes, and Sharing in Multics", which was presented to the Symposium on Operating System Principles held in Gatlinburg, Tennessee in October of 1967. The symposium was sponsored by the Association for Computing Machinery. This paper is reference 6 in the bibliography. We present here a brief review of the subject.

Intersegment Linkage in Multics

In Multics each program is divided into two distinct segments: a "pure" segment and an "impure" segment. The pure segment, known as the "text" segment, is that part of the program which never needs to be modified. That is, the text segment once produced (by compiler or assembler) need never be modified during execution. On the other hand, the impure segment, known as the "linkage" segment, contains all those pieces of the program which may have to modified during execution of the program. For example all external references to points in other programs are made indirectly through pointer variables maintained in the linkage segment. Since the addresses of external points cannot be determined at compilation or assembly, the values of these pointers must be determined at execution. The values stored in these pointers at compilation time contain bit patterns such that, a processor attempting to use them as indirect addresses will experience a fault condition. Such a fault will cause a processor trap to a subroutine designed to deal with the situation. During execution of the program one processor register (known as register Lp, linkage pointer) is assumed to be pointing at the linkage segment. In this way the text segment can reference the linkage segment directly without use of modifiable storage. Generally, the text segment is appreciably larger than the linkage segment and the former segment contains all the logic of the program.

Operationally, this mechanism can be (and is, in Multics) used in two ways. The set of pointers in a linkage section can be left as at compile time, to be modified dynamically as the program is executed. That is, a particular pointer variable's value can be allowed to remain the bit pattern that causes a processor fault, until the time of the first attempted access of the pointer during execution of the program. At that time a specially designed linkage fault handler can determine the address to which to direct the pointer variable and resume the faulted computation at the point at which the fault occurred. On the other hand, these potential linkage faults in linkage segments could be turned into valid pointers prior to execution time in an operation analogous to "loading" in a conventional computer system. This first method of satisfying intersegment references will be referred to as "dynamic linking" while the latter will be known as "pre-linking".

Generally, shared segments appearing in several address spaces need not occupy the same relative locations in the various address spaces. That is, they need not have the same segment number in each space. For this reason the set of addresses maintained in a linkage segment are valid as addresses only for one address space. Therefore, when a text segment is shared between two or more address spaces, it (usually) implies that the associated linkage segment is not shared between these spaces but rather each space has its own modifiable copy of this segment. In this way each address space maintains its own set of external addresses needed by the text segment and valid for that address space.

Address Spaces in Multics

As mentioned in chapter 2, an address space is defined by the Known Segment Table contained in that address space. However, since a considerable portion of the Multics supervisor (known collectively as the hardcore supervisor to distinguish it from the rest of the supervisor) is required to appear in each address space, it would be very wasteful to require each KST to have separate identical copies of KST entries for each hardcore supervisor segment. For this reason another table has been prepared. This table, known as the Hardcore Segment Table (HST), defines the space occupied by the segments which comprise the hardcore supervisor. The HST, which is a shared segment, effectively serves as the "front end" of each KST and the union of the two serve to define an address space. The HST is constructed at the time the Multics system is initialized, and in many ways the function of the entire system initialization mechanism is simply the construction of this table. Constructed at the same time as the HST, is the "template descriptor segment" which describes that portion of the space defined by the HST, that always remains in core storage. Since the segments described by the template descriptor segment always remain in core, the addresses appearing in this descriptor segment (which were computed at system initialization time) remain valid all the time the system is on the air.

During the initialization of the Multics system, several primitive functions are needed including some sort of "linker" primitive. Since it is not desirable to design the dynamic linkage handling module as a single monolithic program without any external references, it is necessary to have a pre-linking ability available to the system initialization procedures. To satisfy this need, a simple table driven pre-linker has been designed which makes no external references. This pre-linker is driven by a table that describes a group of segments. The pre-linker is capable of finding and satisfying each external reference from one segment in the group to another in the group. It does this by scanning each linkage segment in the group, searching for potential linkage faults. With this primitive it is possible to pre-link all intersegment references in the hardcore supervisor of Multics at system initialization and since this portion of the supervisor as a whole is especially designed to have no references to any segments outside this hardcore group, all external references in each linkage segment of the hardcore supervisor are satisfied at system initialization time. Since all intersegment references between hardcore supervisor segments are pre-linked at system initialization time, the linkage segments included in the hardcore supervisor are not modified during actual system operation. Further, since the space defined by the HST is a proper subset of each address space in the system the relative addresses of points in the hardcore supervisor are identical in each address space. This means that the linkage segments contained in the hardcore supervisor can be shared among all processes. Given such an environment, no dynamic linking capability is needed at system initialization time. For this reason and for others that will be discussed below, the dynamic linking module was not originally included in the hardcore supervisor of Multics. This decision meant that each newly created process in the system, which desired to execute outside the hardcore supervisor, had to initialize the dynamic linking mechanism in its own address space. This initialization amounted to the pre-linking of the dynamic linker in the new address spaces.

Basic Process Creation

Because a fledgling address space is defined by the HST, the creation of an address space for a process is a relatively simple matter. The only thing that needs be done is to create three private data bases which the process will need before it will be able to execute. These data bases are:

1. An empty KST which the process will use to expand its address space.

2. A Process Data Segment (PDS) that is guaranteed to be completely resident in core storage whenever the process is running. This segment contains a data area which holds information such as the process identification, etc., and it also contains the process concealed stack, which is used whenever supervisor programs which cannot afford to take missing page faults are executed by the process. The data items in the data area of the PDS are ones that are occasionally accessed by procedures incapable of accepting missing page faults.

3. A Process Definitions Segment (PDF) which is similar in structure to the PDS in that it contains a data area and a stack. The difference between these segments is that the PDF is not required to remain in core storage at all times that the process is running. What is required is that this segment remain active, which, as was mentioned in chapter 2, effectively means the page table for this segment remains in core at all times the process is running. The stack maintained in the PDF is used to process faults, such as the missing segment fault, which can tolerate missing page faults in their handling.

Having created these three segments and a directory segment in which they reside, the process directory, one has provided all the ingredients of a minimal address space. If the process were loaded, the process could now operate in the limited space defined by the HST. The descriptor segment that would be provided this new process would simply be a copy of the previously mentioned template descriptor segment. The other part of process creation is to create a set of processor conditions. Recall that in our earlier discussion of processor multiplexing, we found that the task of subroutine swap-dbr was to save the processor state of one process and to restore that of the one being switched to. What is required is to produce a set of conditions that make it appear to swap-dbr, that a newly created process has issued a call to swap-dbr in the past and saved the appropriate data. This can be done by concocting a stack history of a call to swap-dbr and putting this history into the concealed stack of the newly created process.

The major work involved in accomplishing this task is done at system initialization time when a "template-PDS" segment is synthesized. This segment contains a stack history that purports to show of a call to swap-dbr from a procedure known as init-proc. At the creation of a new process, all that needs be done is to copy the contents of this template-PDS into the PDS of the new process. In this way the processor state of a new process is synthesized.

Given that we can now create processes at will, a mechanism is needed so that newly created processes can be directed to the task for which they were created. This is accomplished in Multics, by creating an additional segment for the new process known as the Process Initiation Table (PIT). The PIT contains the name of a procedure that the new process should call, after it begins its execution, plus any data which this procedure will need. Through the use of this mechanism, a new process can be started off in any desired direction.

One last issue will conclude our discussion of basic process creation. That is, the new process needs an ordinary, paged, call stack on which it will execute ordinary procedures. The creation of this stack segment can be deferred until the time of process initialization (i.e., until the process reaches init-proc).

We can now briefly outline the steps that are taken in creating a process and in initializing it. First to create it:

1. Create a process directory.

2. Create a PIT in this directory, which contains the name of a procedure that the new process should call upon finishing its initialization.

3. Create an empty KST in the new process directory.

4. Create a PDS with a stack history purporting to show of a call to swap-dbr from init-proc. Also initialize data items in this segment such as the process identification.

5. Create a PDF segment with initialized data items.

6. Add an entry for the new process to the Process Table and call wakeup for the new process.

At some point in the future, the new process will be loaded, by the loader-daemon, and will begin execution. In order for it to initialize itself it must:

1. Return from swap-dbr to init-proc.

2. Create an ordinary stack segment in its own process directory, and begin to use it.

3. Map the created PIT into the new address space.

4. Decode the data in the PIT and call to the procedure named.

Process Destruction

The aim of process destruction is the elimination of all traces of a process from the system. This task is made simple in Multics by the existence of the special directory maintained for the process, the process directory, into which all per process segments are placed. During the lifetime of a process all the per process data segments such as copies of linkage segments, stack segments, etc., are created in this directory. At the time of process destruction all the pieces can be found in this one place.

In order to destroy the process it must be in a dormant state. Therefore, the first thing to do is to call subroutine stop, specifying the appropriate process. Since the effect of a call to stop does not necessarily take place instantaneously, the caller to stop must be able to determine when the process to be destroyed has actually been stopped. This ability is provided by a status subroutine in the Traffic Controller. This subroutine returns the execution state and the runability state as arguments. By defining a condition that a process stopping itself notifies, a process trying to stop another can call subroutine wait to wait for its target to stop itself.

Process destruction is then seen to be simple. We call stop to bring the process to rest and then we wait until it is stopped. Then we remove the process from the Process Table and delete all the segments listed in its process directory. Finally we delete the directory itself.

Dynamic Linking in New Processes

Any process in the system desiring to execute outside the bounds of the hardcore supervisor must possess a dynamic linking ability in its address space. Since all external references within the hardcore supervisor are pre-linked no such capability is needed if a process never leaves this region. This capability can be provided in one of two distinct ways.

The first way in which this can be accomplished would be to take advantage of the pre-linking capability available in the hardcore supervisor. That is, a new process initializing itself could make use of the pre-linker residing in the hardcore supervisor to pre-link the dynamic linking modules within the address space of the process.

The second way in which this problem can be solved would be to move the dynamic linking modules into the hardcore supervisor, although they are not required to be there. Such a move would mean that a pre-linked dynamic linking module would be contained in the space defined by the HST and it therefore would need no special treatment at the time of process initialization. A process merely by being handed a minimal address space would be capable of dynamically linking itself.

Both of these mutually exclusive approaches have advantages as well as disadvantages. The first approach has the advantage that since each process pre-links its own dynamic linker module, it is possible to provide separate processes with different versions of the dynamic linking module. Such a policy would allow one to "checkout" a new linker module in one process while not affecting the other users of the system. However, this approach has one distinct disadvantage in that the cost of creating processes (more properly initializing processes) is made that much higher by the expense of repetitively pre-linking a dynamic linker in each new address space. The second approach, obviously reverses these two conditions. That is, process creation with the second scheme is fairly cheap in that not much need be done. However, use of this scheme means that all processes are forced to use the same dynamic linker module and that routine changes to it become that much more difficult to debug.

One of the major policy decisions made during the planning and design of the Multics system was to try to limit the number of programs required to reside in the hardcore supervisor. Two informal criteria were operationally adopted in order to judge where a particular module belonged. These two criteria were:

1. A module belonged in the hardcore supervisor if the effect of the module executing in one address space could possibly be felt in all other address spaces. That is a module (if operating incorrectly) capable of "clobbering" the system belonged in the hardcore supervisor. An ordinary supervisor module only capable of clobbering the process in which it was executing was not to be included.

2. Since the hardcore supervisor was not to have any external references to points outside its domain, any primitive functions needed by the other hardcore modules were to be available in the hardcore.

Since the dynamic linker module satisfied neither of these criteria, the original implementation of the process creation module was organized around the concept of per process pre-linking of the dynamic linker. Such a decision meant that additional logic beyond that presented in the section on Basic Process Creation, is needed in both the creation and initialization of processes. On the creation side the logical addition amounts to the preparation of a table, the pre-linker driving table, that will be used to drive the pre-linker in the newly created process. Such a table contains a list of all segments involved in dynamic linking in order that all intersegment references between them may be pre-linked. In order to provide different linkers to distinct processes, distinct driving tables would be provided. On the initialization side, the logical addition amounts to the execution of the pre-linker. The pre-linker, in its execution, makes two passes over the driving table; the first to map each named segment into the space and the second to locate and scan each linkage segment listed in the table, in order to find external references.

The above described process creation and initialization package was implemented in Multics and its performance was analyzed. From data gathered in experiments, it was determined that more than 75 per cent of the execution time spent in this module was directly attributable to the approach taken to the dynamic linking problem. With this result in hand, a decision was made to re-implement the module organizing it around the idea of the dynamic linker residing in the hardcore supervisor. This was done and very favorable results were realized from this move.

The above few paragraphs tend to leave the reader with a poor idea of how the process creation module evolved in the Multics system. The ideas presented here have been organized in the order of their logical importance, not necessarily in the order in which they were actually developed. For example, one might feel after reading the section on Basic Process Creation and the subsequent paragraphs outlining the problems of dynamic linking, that the designer and implementer of the process creation module had all these facts and observations at hand. In point of fact the designer of the module was faced with a fairly rigid environment, before he began his task, which included the fact that the dynamic linker module was outside the hardcore supervisor. His task at that point was to design and implement a module that would conform to this environment. It was only after the original package was implemented and analyzed that the basic structure of the problem was understood and the basic inefficiency of the original approach was fully appreciated. Armed with the above observations, gleaned from the experience gained in working on the original design and implementation, the second approach seemed to be the logical way to implement the facility.

CHAPTER SIX

Analysis, Summary and Conclusions

During the course of work documented in this thesis, several performance analysis tests were made on the Multics system to determine how well various parts of the system performed. Tests were made on the Traffic Controller as well as on the Process Creation Mechanism.

Traffic Controller - Before

A simple test was devised to meter the amount of time spent in executing the Traffic Controller primitives. A few extra instructions were added at the entry and exit points and the amount of time spent between these points was read from a clock. At the time of the first implementation of the Traffic Controller, tests were made on the block, wakeup and reschedule subroutines and they indicated that it took approximately:

1. Six milliseconds for a processor to enter block in one process and return from the Traffic Controller in the process which was switched to.

2. Six milliseconds for a process to enter reschedule and return from the Traffic Controller in a new process.

3. Four milliseconds to wakeup a blocked process that was loaded and approximately 315 milliseconds to wakeup an unloaded, blocked process.

Since the Process Wait and Notify subroutines (addevent, wait and notify) were not included in the original Traffic Controller, they were not tested in this way. However, other metering tests performed on the system as a whole, give an indication of how well the old PWN facility operated. The specific test involved here was to allow the system to operate in a normal way, but to interrupt its execution every ten milliseconds to determine exactly which subroutine was then being executed. In this way one could determine where the system was spending its time. This test was designed so that one could selectively meter certain parts of the system. For example, it was possible to meter, in this way, only during the time the system spent in handling missing page faults. As a result of just such a test it was determined that approximately fifteen milliseconds were spent, per page fault, executing in the old PWN facility. Figures are not available that would make it possible to determine how this fifteen milliseconds was distributed over the separate PWN subroutines, however, in general, each page fault generates one call to addevent, one call to notify and one call to wait. Since calls to wait usually result in calls to block and calls to notify usually result in calls to wakeup, one should add the time spent in these two subroutines to find the total cost of the old PWN facility. This result is about 25 milliseconds.

Traffic Controller - After

With the new implementation came a retesting of the subroutines. This time the PWN subroutines were tested by making specific clock readings. The results follow:

1. Time to block (i.e., to block self, choose a successor, switch to him and return to where he blocked himself): approximately 1.4 milliseconds.

2. Time spent in a complete trip through reschedule until the successor process returns from the Traffic Controller: approximately 1.5 milliseconds.

3. Time to wakeup a blocked process (whether loaded or not): approximately 0.9 milliseconds.

4. Time to wait for something (i.e., put self in waiting state on a thread, choose a successor process, switch to him and return to where he called the Traffic Controller): approximately 1.3 milliseconds.

5. Time to wait for nothing (i.e., wait for a condition that has already been notified): approximately 0.23 milliseconds.

6. Time to notify a condition for which one process is waiting: approximately 0.34 milliseconds.

7. Time to notify a condition for which no one is waiting: approximately 0.23 milliseconds.

8. Time to activate a thread (i.e., call addevent): approximately 0.23 milliseconds.

The above results (before and after) are summarized in the following table.

Nature of test	Before	After
Blocking time	6.0 ms	1.4 ms
Reschedule time	6.0 ms	1.5 ms
Wakeup time (loaded)	4.0 ms	0.9 ms
Wakeup time (unloaded)	315.0 ms	0.9 ms
Block-wakeup time	10.0 ms	2.3 ms
Wait (for something)	not available	1.3 ms
Wait (for nothing)	" "	0.23 ms
Notify (something)	" "	0.34 ms
Notify (nothing)	" "	0.23 ms
Addevent	" "	0.23 ms
Addevent-wait-notify	25.0 ms (with old block-wakeup)	1.87 ms

Process Creation - Before and After

A simple test was performed on the system in which the original process creation module was implemented. A clock was turned on just before a new process was created and not turned off until after the initialization of this process was completed. The test revealed several interesting things:

1. The total time to create, load and initialize the new process was more than 3 minutes and 50 seconds.

2. During the course of this test, the system serviced 1251 missing page faults and 489 missing segment faults. At that time in system development, primitive page and segment fault handlers were each taking on the order of about 100 milliseconds to service their respective faults. Therefore, about 2 minutes and 50 seconds was spent in these activities.

The above observations indicated that process creation, loading and initailization took about one minute. This was confirmed when more effective page and segment fault handlers were installed in the system. With no change in the process creation strategy, the above tasks were roughly observed to take on the order of about two minutes; a 100 per cent improvement.

The new process creation module which implements the linker module in the hardcore supervisor has been installed and very preliminary observations indicate that process creation and initialization now take on the order of a very few seconds. Further tests will be performed in the future, but it appears that the new implementation has substantially improved the performance of this task.

Targets

At this point in the discussion it is appropriate to explore the meaning of the above results. For example, how important to system performance is it that the wait-notify time is reduced to about 1.8 milliseconds from about 25 milliseconds? Or how important is the efficiency of the process creation module to the user of the system?

The Multics system is dedicated to multiplexing system resources among many user processes in an efficient way so that these resources are used effectively to the maximum possible degree. For this reason, system philosophy dictates that a process should give away its processor when waiting for a page to be retrieved from secondary storage. On the current hardware on which the system is implemented, the drum (i.e., one of the devices from which pages come into core), for example, has a rotation speed which is about 34 milliseconds. Therefore the average time to wait for a page from this device is about 17 milliseconds. In the original implementation of the Traffic Controller, the wait-notify-block-wakeup time was about 25 milliseconds. That is, it took about 25 milliseconds for a process, to give its processor away and to be subsequently restored to the running state, after waiting for a page. In other words, in such an environment it would have been more economical to simply loop waiting for a page, rather than to be fancy and attempt to maximize processor usage. Given these figures, it is clear that the wait-notify time must be a small fraction of the drum rotation time before an attempt to multiprogram while awaiting pages becomes justified. The current figure of approximately 1.8 milliseconds clearly makes such multiprogramming profitable. The processor can be used on average for more than 15 milliseconds of the expected drum wait, presuming eligible processes are available and willing to run.

In regards to targets for the performance of other primitives of the Traffic Controller, the arguments are not as imperative. Since wait and notify are the most heavily exercised, their performance is of utmost importance. But the fact that all the other Traffic Controller primitives have comparable performance figures means that clearly their performance is in the acceptable range.

A target for the process creation operation is harder to arrive at. This operation, although exercised less frequently than the above primitives, is directly initiated in most cases, by human users of the system, who demand fairly quick responses to such requests. Although the new mechanism has just been integrated into the system and not much experience has been gained with it, it is believed that the actual computation needs of the new process creation package do not exceed one second of processor time, and further that the response time to such operations (which is greater than computation time due to page waits, etc.) can be brought down to the range of several seconds. These predictions are made from analysis and knowledge of the code contained in this package, and the steps needed to perform the tasks, rather than from actual analysis of performance figures which are not yet available.

Conclusions

After having found acceptable solutions for the problems at hand, one asks oneself why it took so long to arrive at these solutions and was there anyway to have done it more quickly? One might further ask if the arrived at solutions are in any sense optimum?

After being involved in designing a large system involving the work of many people, one gets the feeling that such problems as were encountered here are bound to crop up. The development of any large system can only remain manageable if distinct parts of the system remain modular and independent. It was in such a modular, independent atmosphere that the Traffic Controller and the PWN facility were originally designed, by separate groups of people. As a result, independent, but parallel structures were established in each domain. It was only after one stepped back and looked at the two pieces with understanding, that the duplication of function between the old Process Wait Table and the Process Table became clear. Similarly, the decisions which forced the original structure of the process creation module were made at a time when their full consequences were unknown. As a result much work proceeded from those decision points which later had to be redone. This is not to say that this original work was wasted effort, for it was only from the insights gained in performing this work that it proved possible to improve upon it.

Without a theory of computing systems to fall back on, designing of such complex systems becomes an art, rather than a science, in which it is impossible to prove the degree to which working solutions to problems are in any sense optimum solutions. In much the same way as authors write books, large computer systems go through several drafts before they begin to take shape. In the absence of a theory one can only cope with the complexity of the situation by proceeding in an orderly fashion to first produce an initial working model of the desired system. This part of the work represents the major effort of the design and implementation project. Once having arrived at this benchmark, many of the problems may then be seen in a clearer light and revisions to the working model are implemented much more quickly than were the original modules. As to the development of a theory, one gets the impression that it will be a long time in coming.

APPENDIX I

This appendix contains flow diagrams of the subroutines of the Traffic Controller as originally implemented.

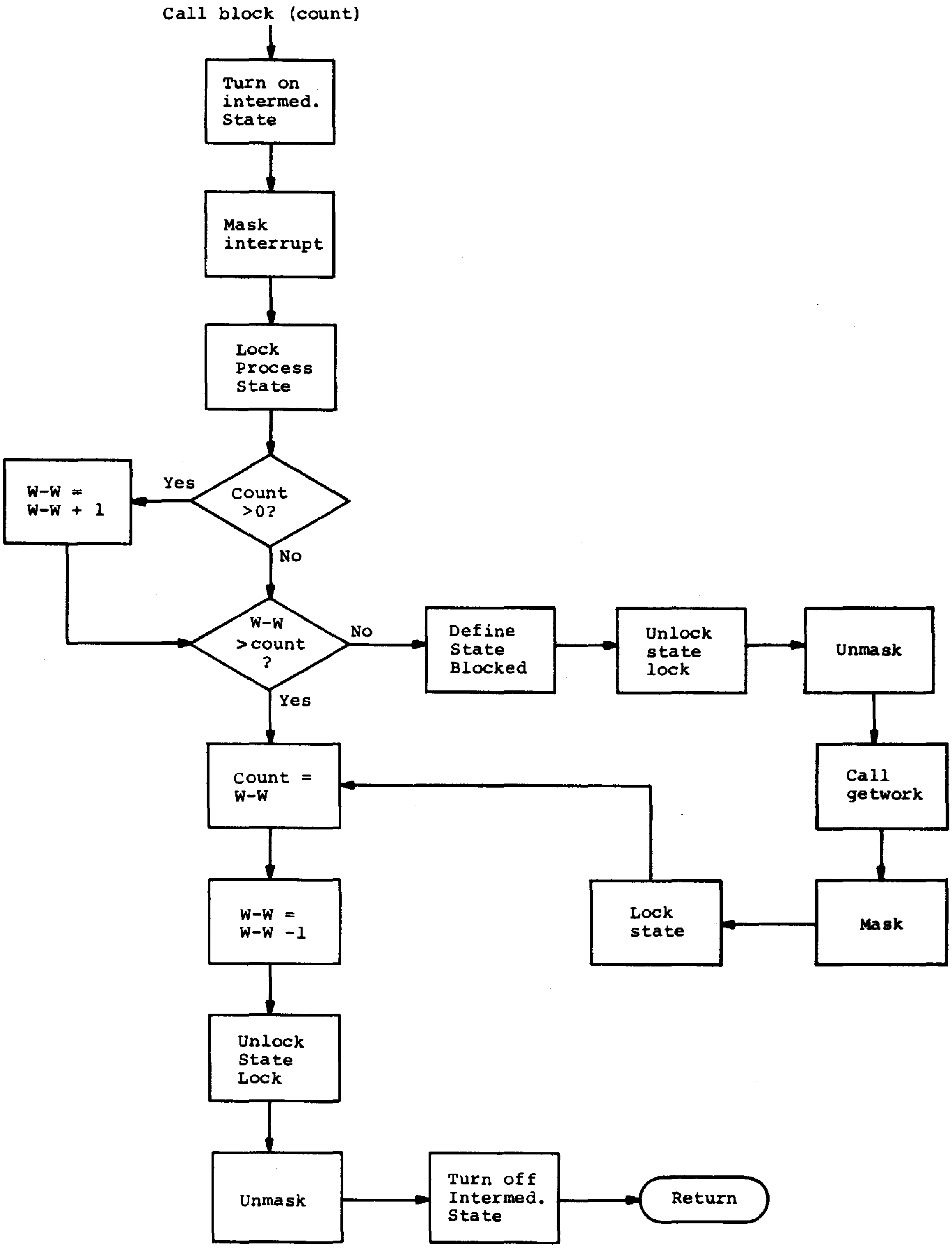

Figure I.1. Block with comples interlocking. W-W stands for wakeup-waiting count.

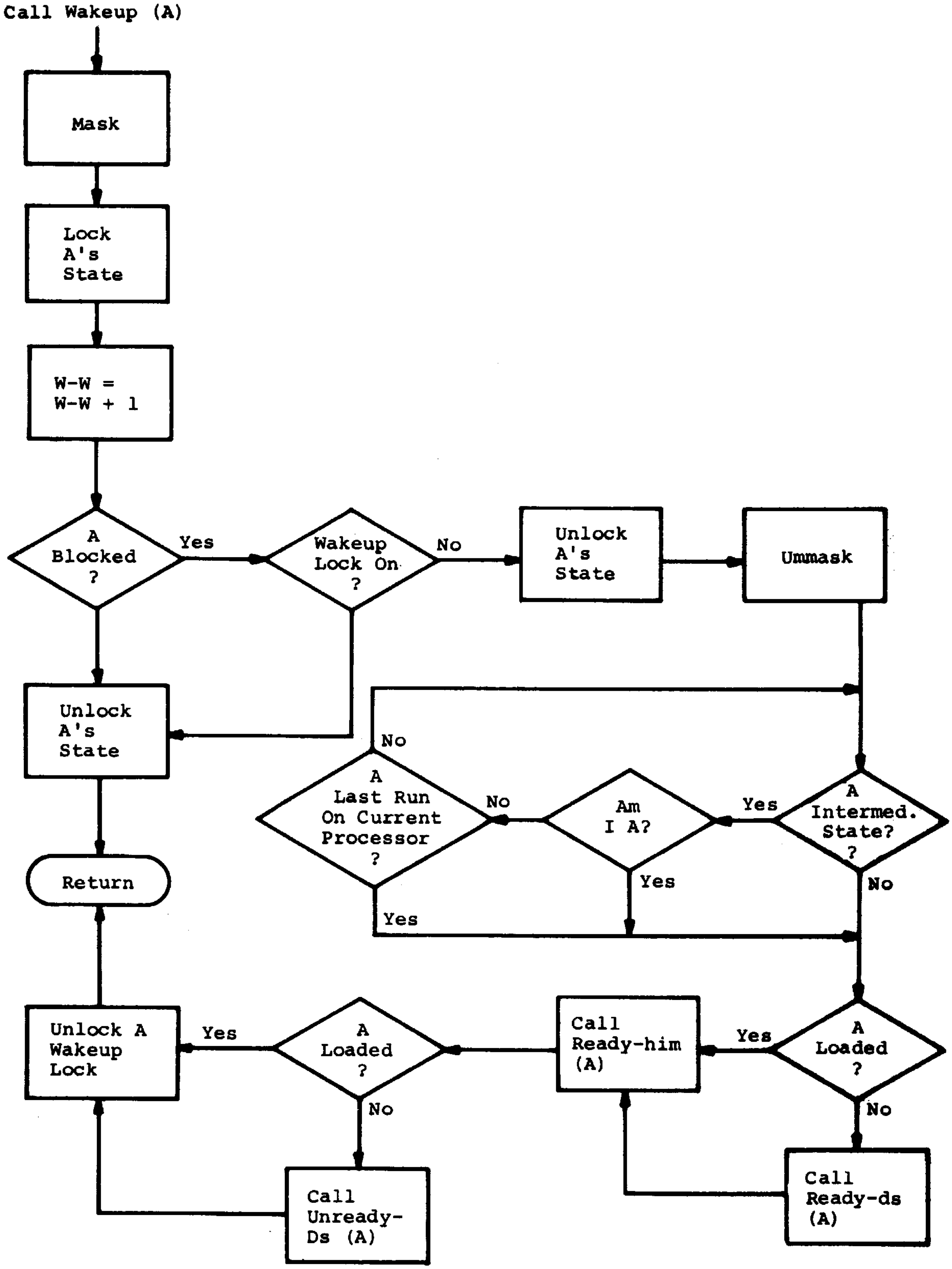

Figure I.2. Original Wakeup.

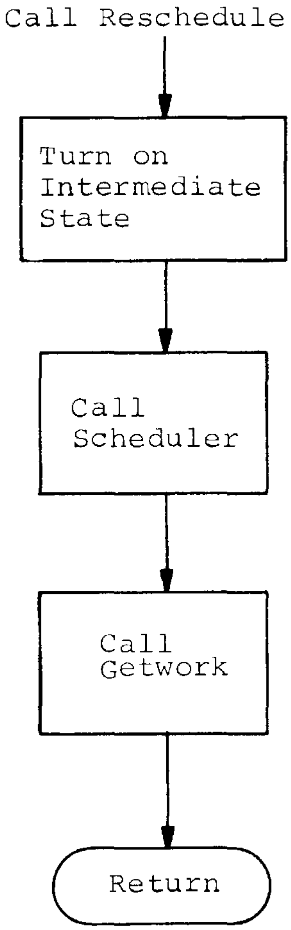

Figure I.3. Reschedule

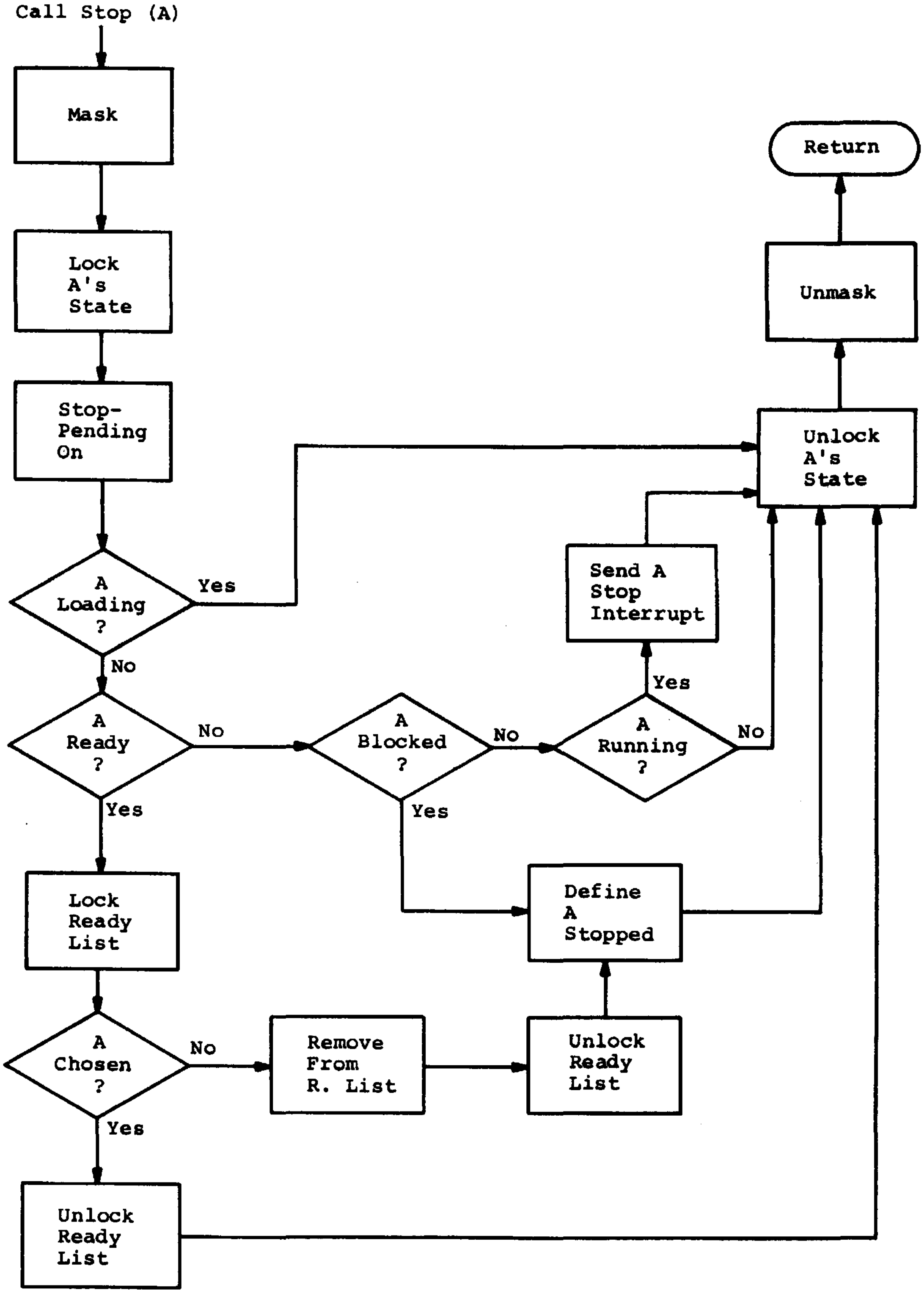

Figure I.4. Stop.

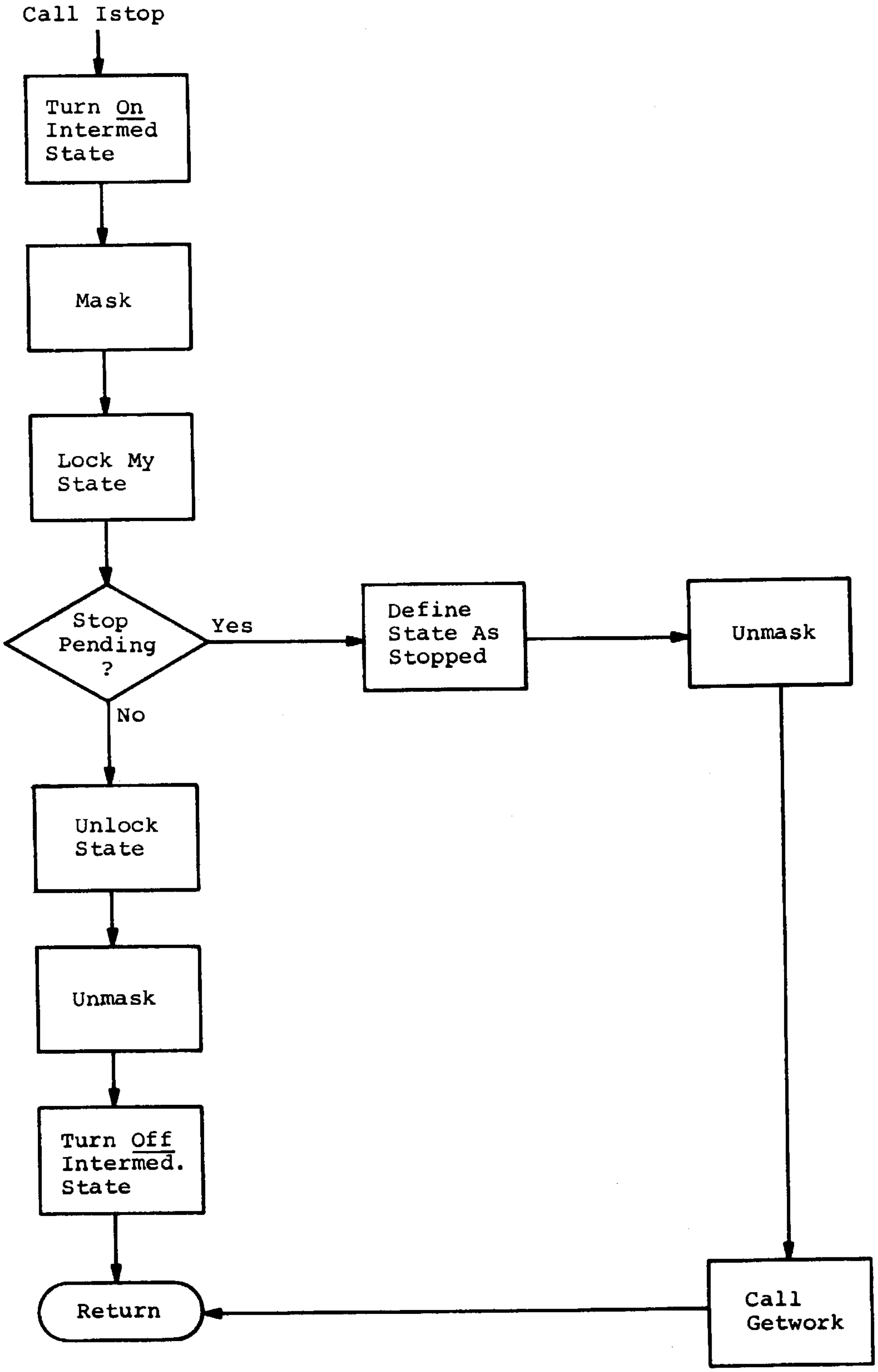

Figure I.5. Istop

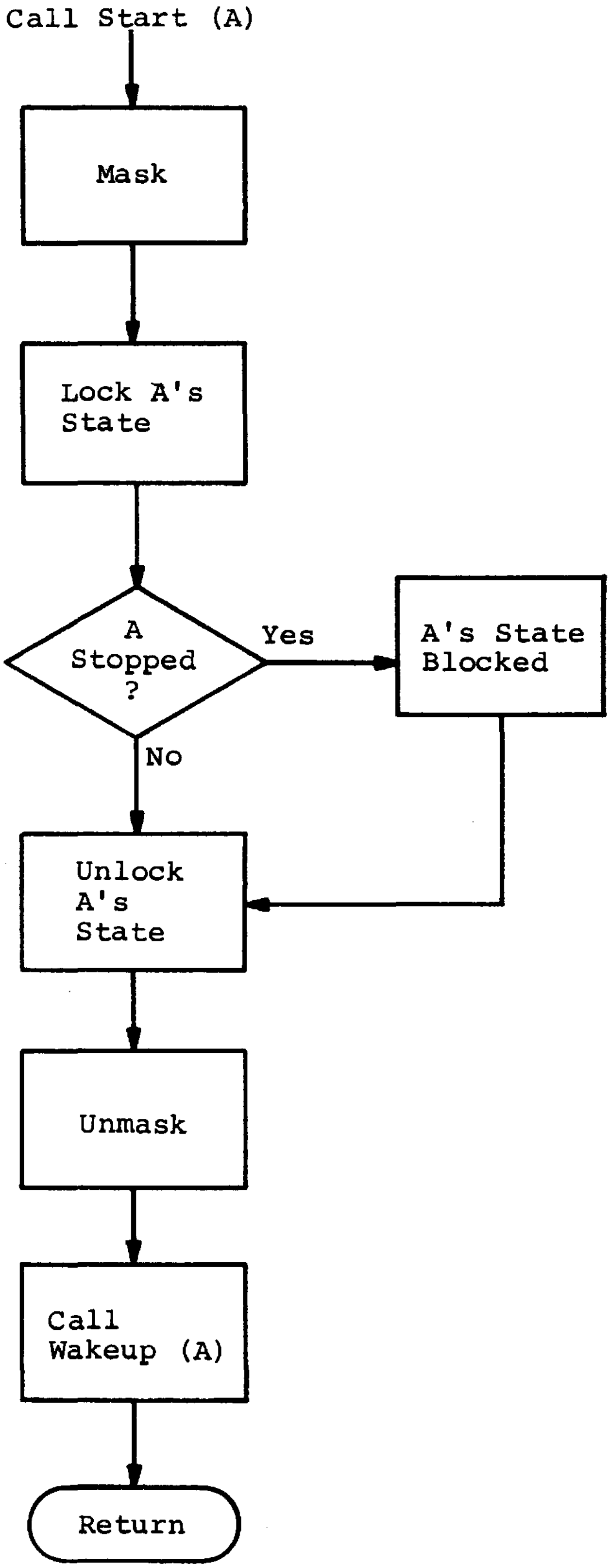

Figure I.6. Start

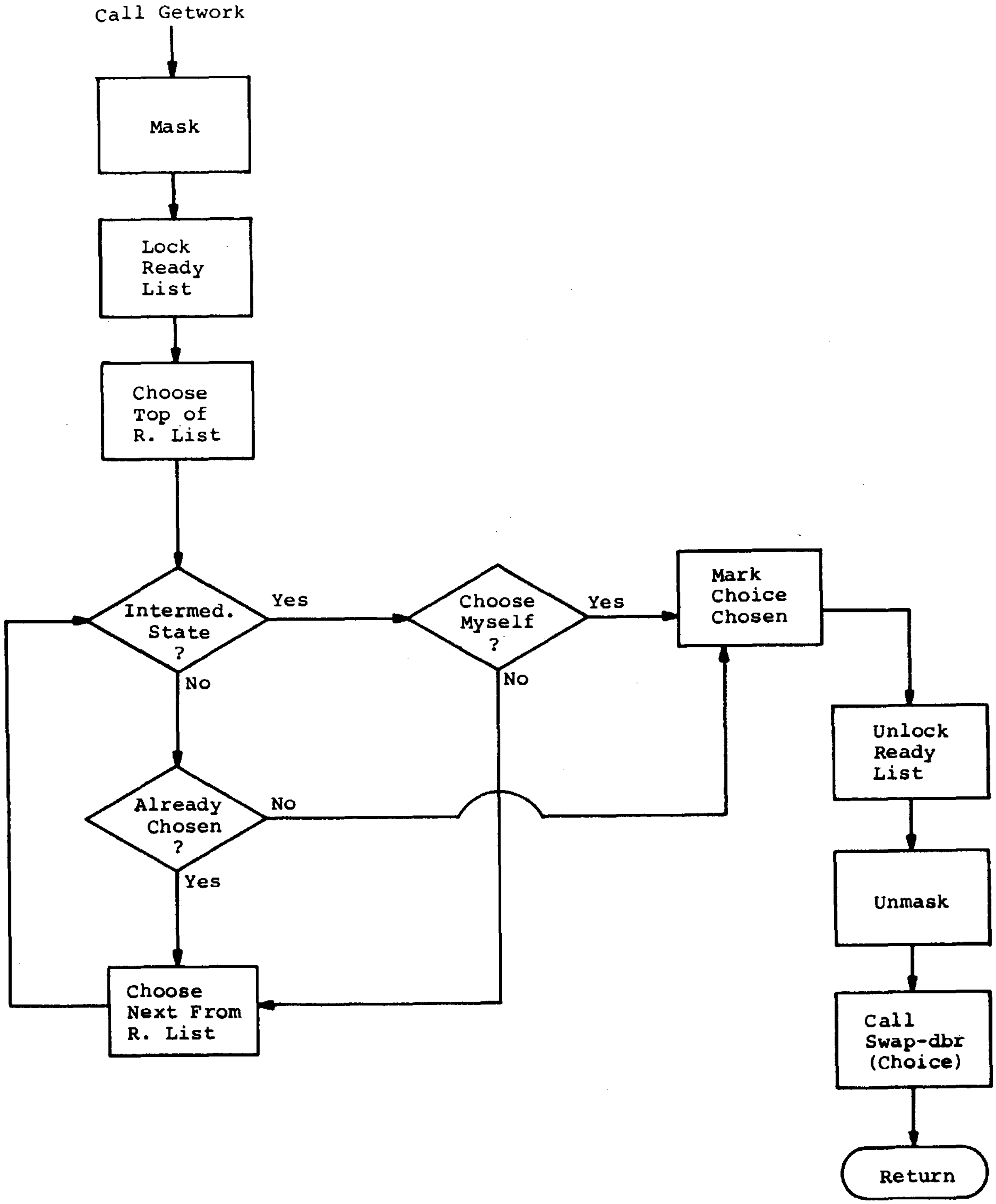

Figure I.7. Getwork

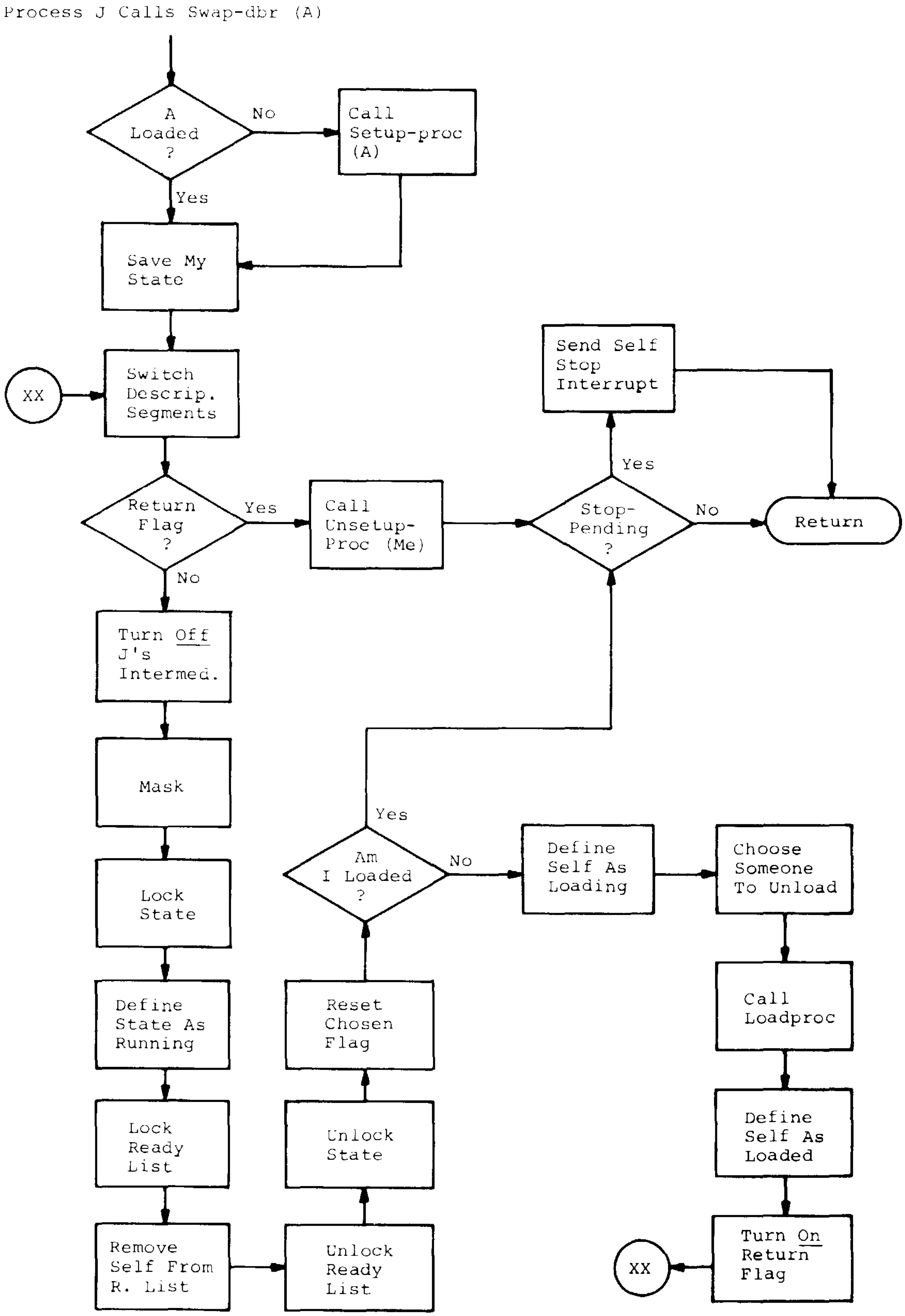

Figure I.8. Swap-dbr

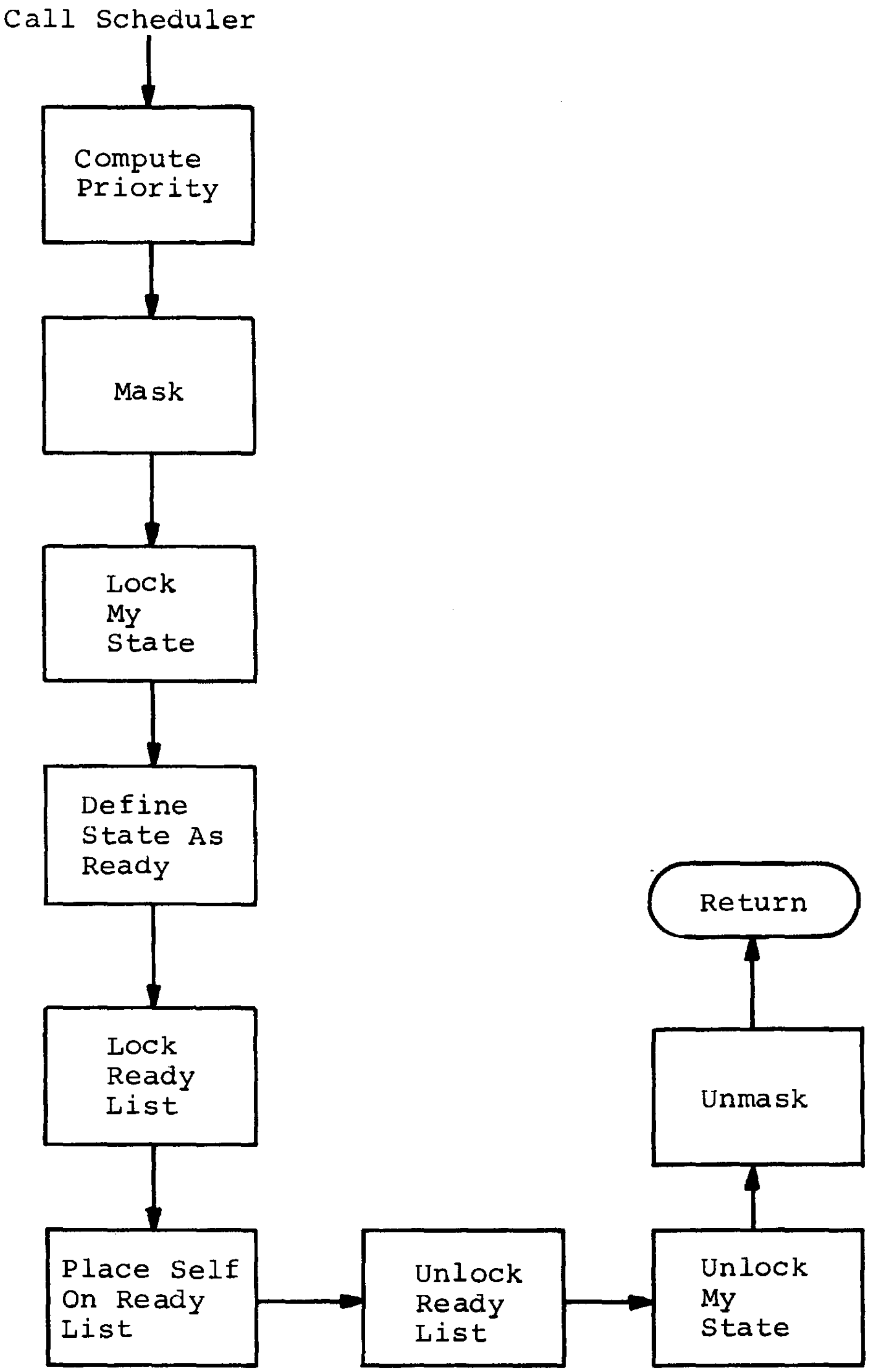

Figure I.9. Scheduler

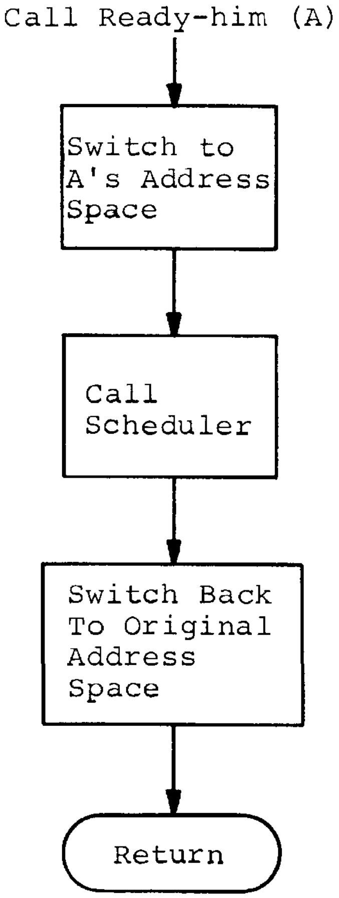
Call Ready-him (A)
Switch to A's Address Space
Call Scheduler
Switch Back To Original Address Space
Return

Figure I.10. Ready-him

APPENDIX II

This appendix contains flow diagrams of the subroutines of the second iteration of the Traffic Controller implementation.

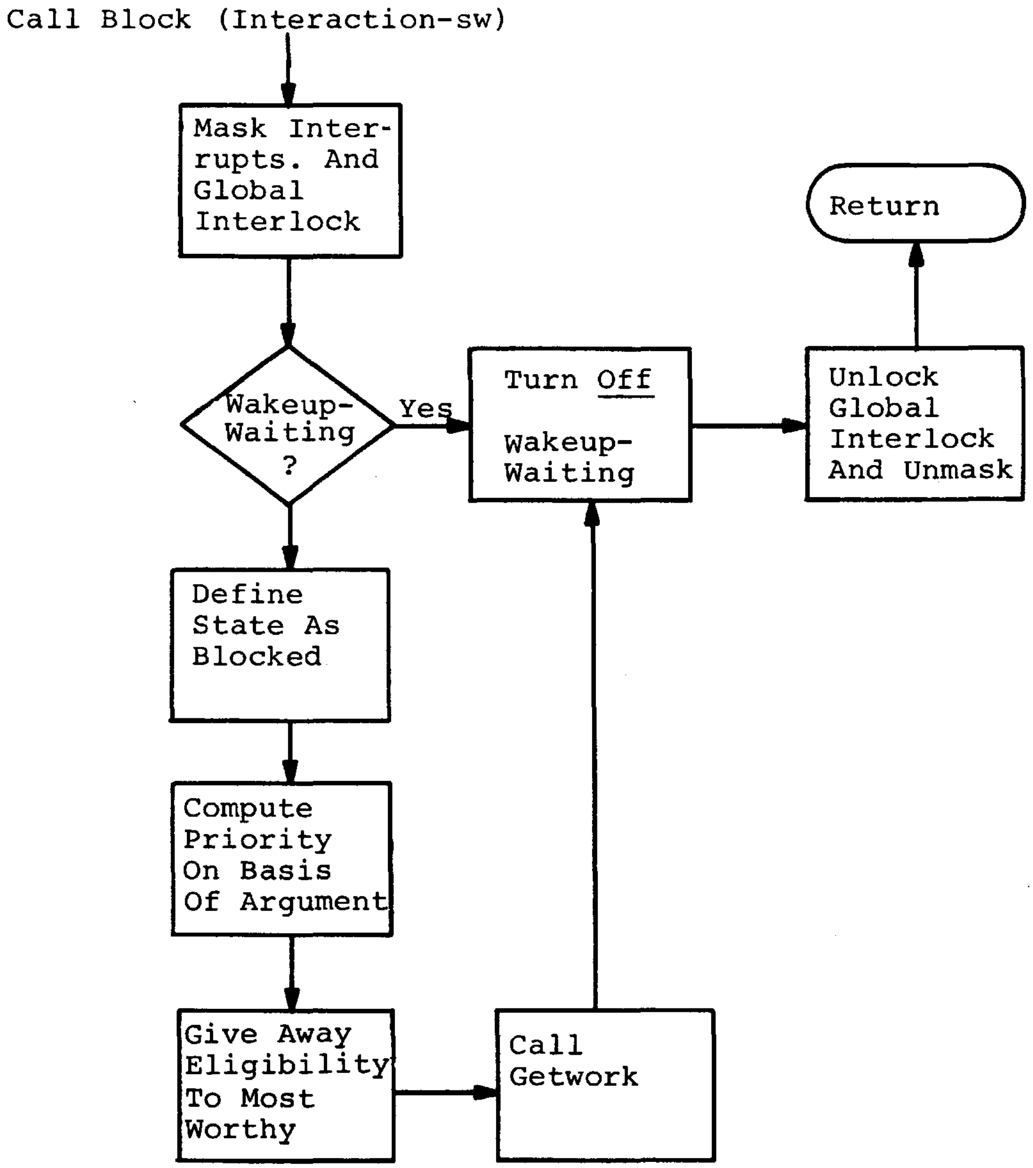
Call Block (Interaction-sw)
Mask Interrupts. And Global Interlock
Wakeup-Waiting ?
Yes
Turn Off Wakeup-Waiting
Unlock Global Interlock And Unmask
Return
Define State As Blocked
Compute Priority On Basis Of Argument
Give Away Eligibility To Most Worthy
Call Getwork

Figure II.1. Block

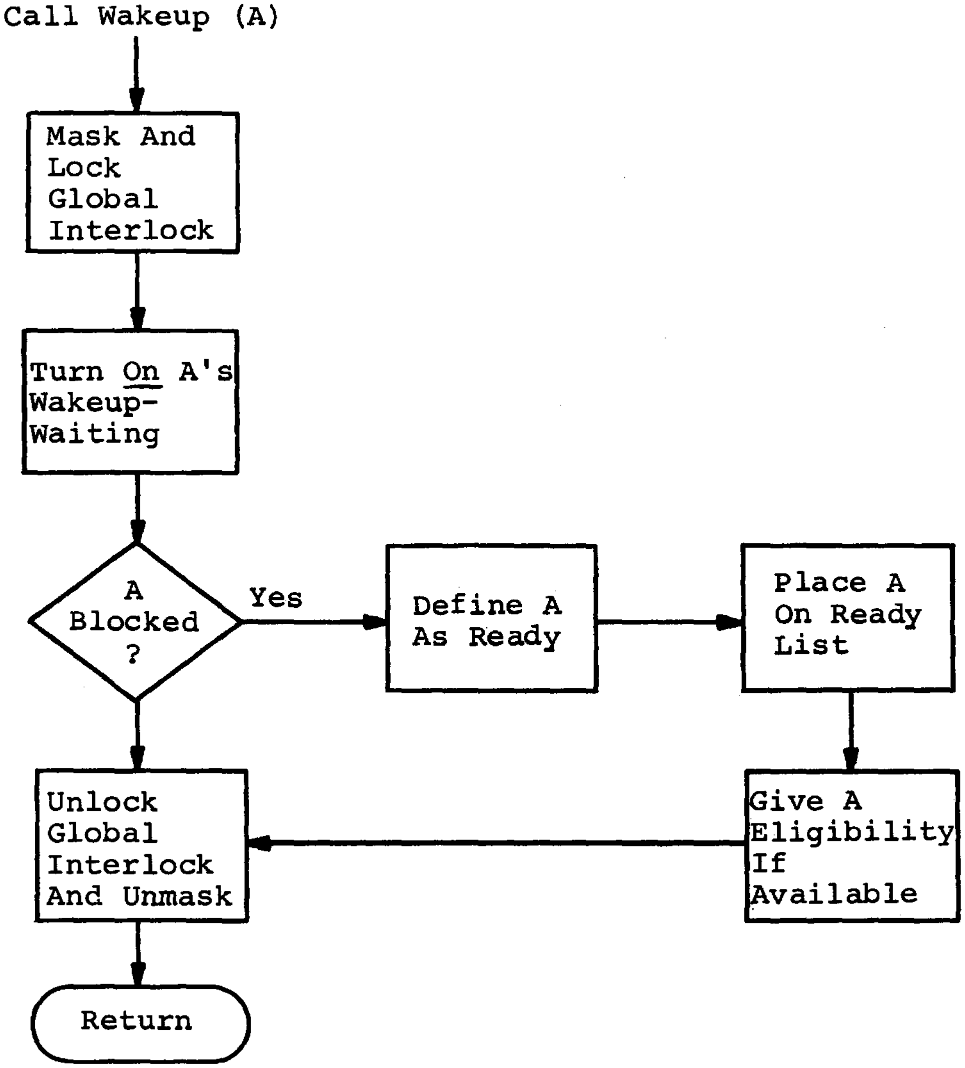
Call Wakeup (A)
Mask And Lock Global Interlock
Turn On A's Wakeup-Waiting
A Blocked ?
Yes
Define A As Ready
Place A On Ready List
Give A Eligibility If Available
Unlock Global Interlock And Unmask
Return

Figure II.2. Wakeup

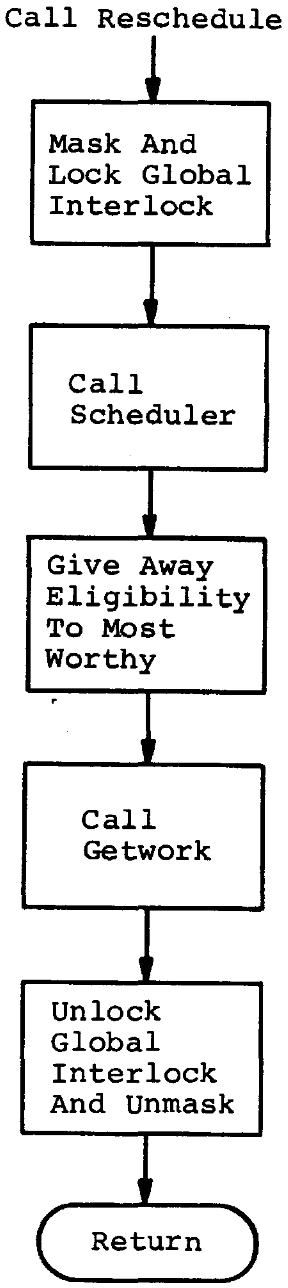
Call Reschedule
Mask And Lock Global Interlock
Call Scheduler
Give Away Eligibility To Most Worthy
Call Getwork
Unlock Global Interlock And Unmask
Return

Figure II.3. Reschedule

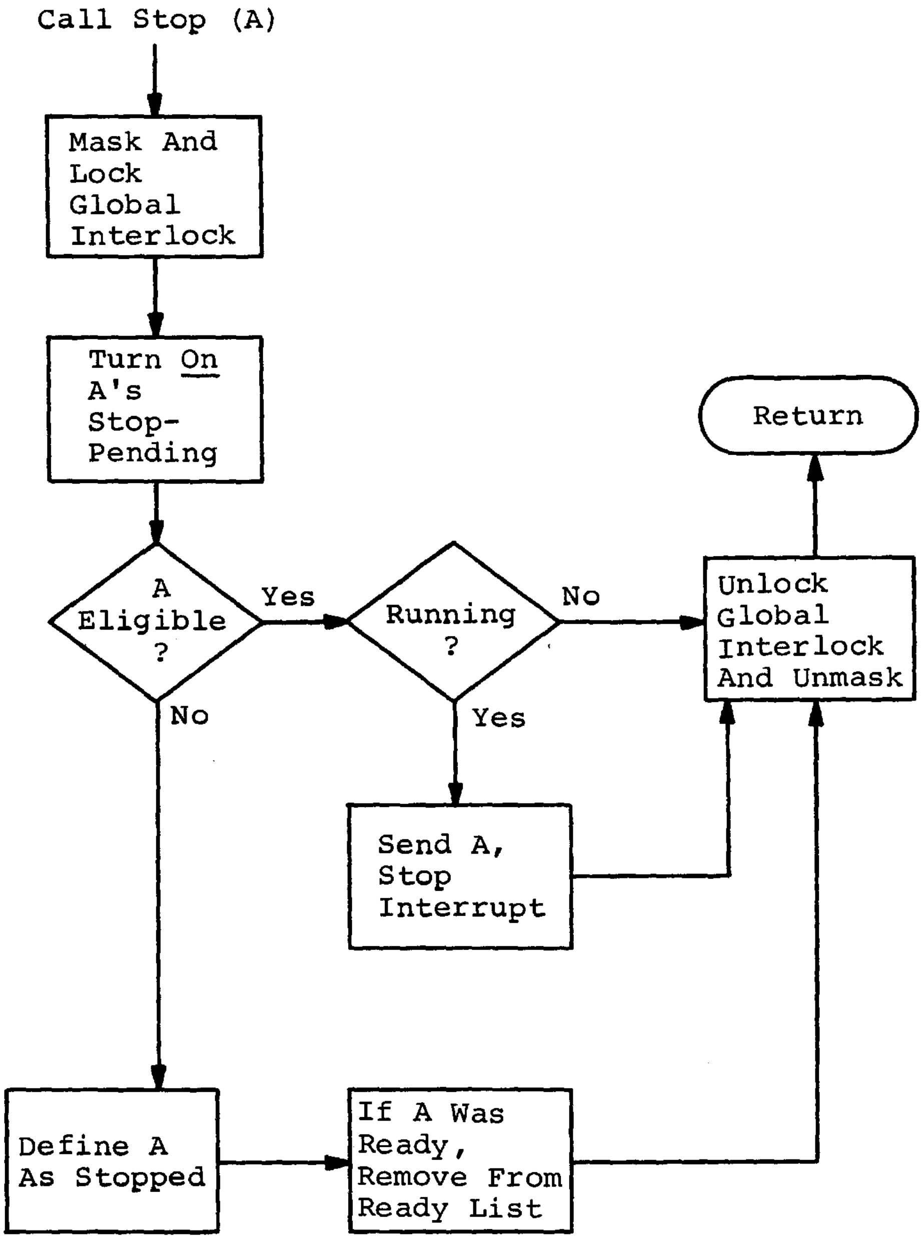

Figure II.4. Stop

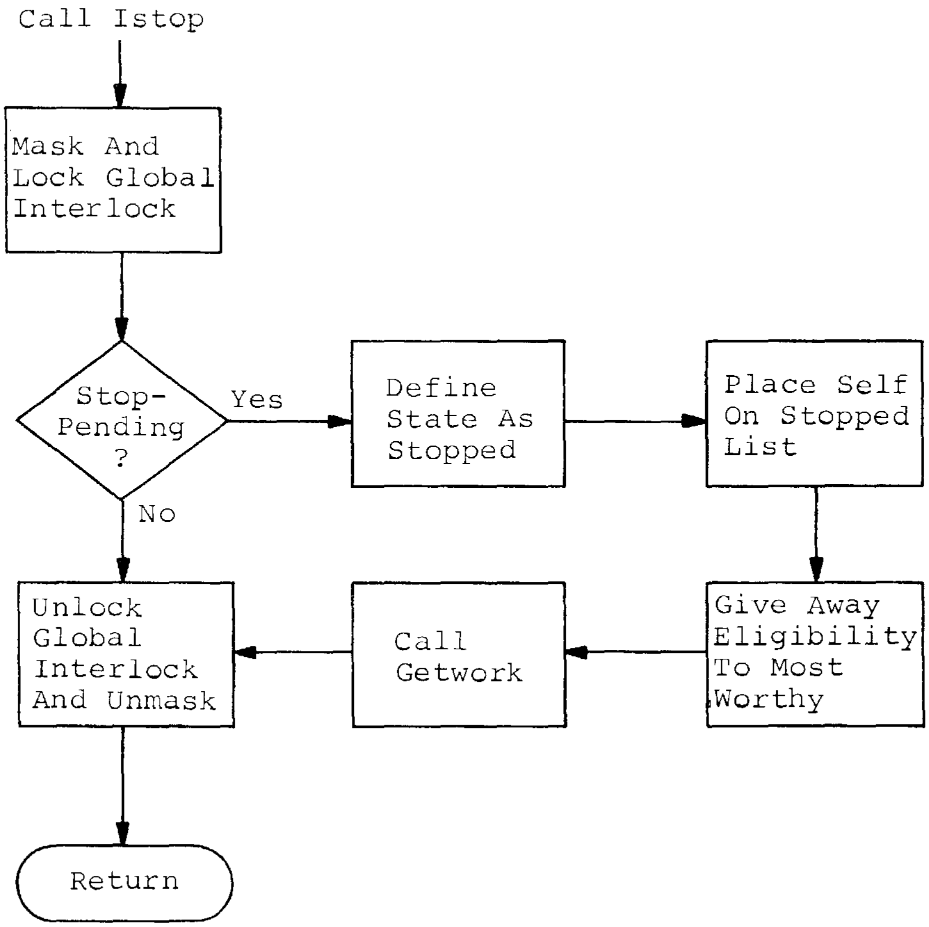
Call Istop
Mask And Lock Global Interlock
Stop-Pending ?
Yes
No
Define State As Stopped
Place Self On Stopped List
Give Away Eligibility To Most Worthy
Call Getwork
Unlock Global Interlock And Unmask
Return

Figure II.5. Istop

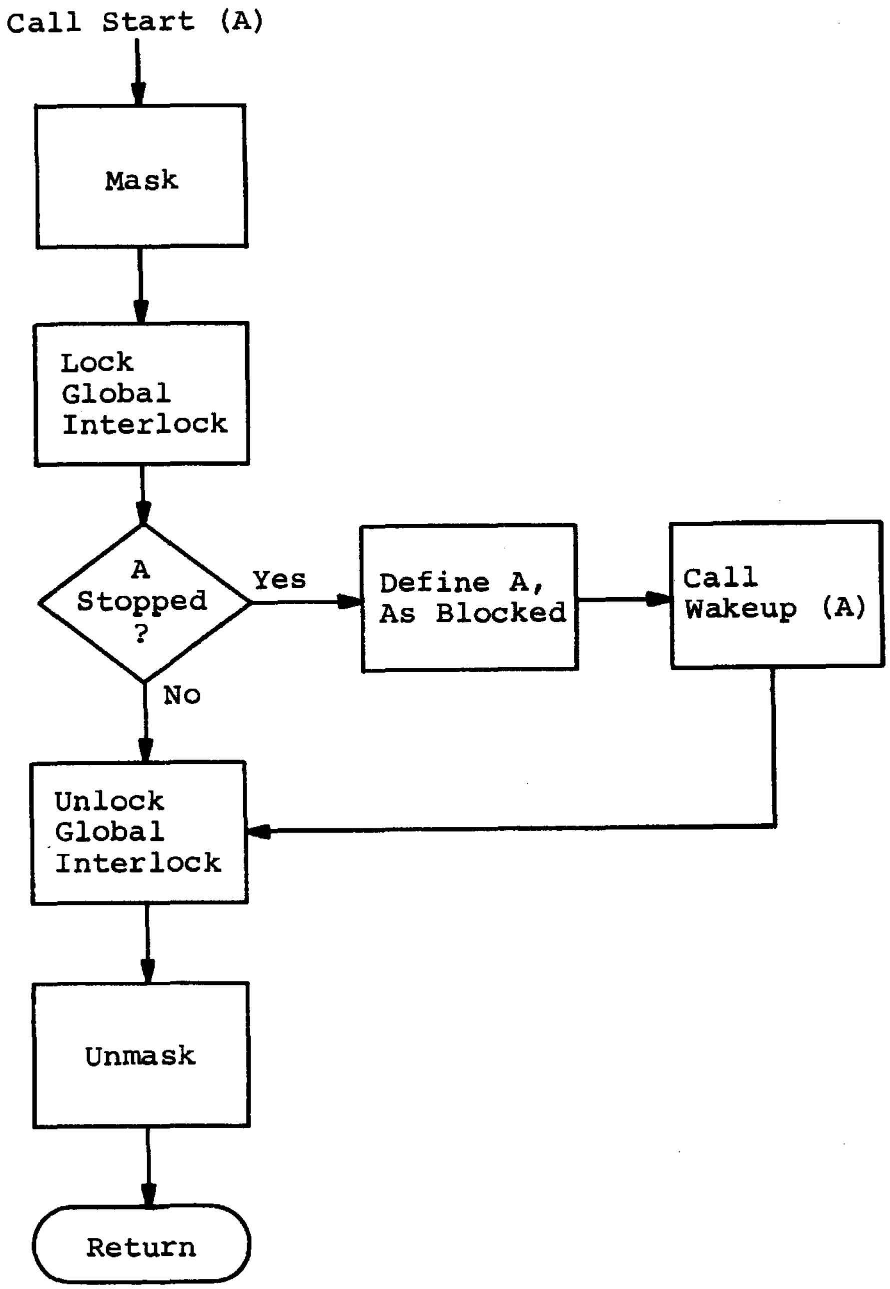

Figure II.6. Start

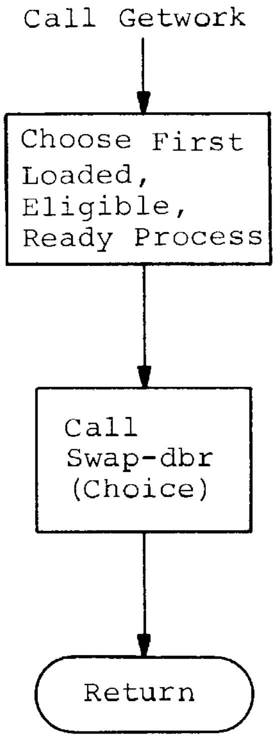

Figure II.7. Getwork

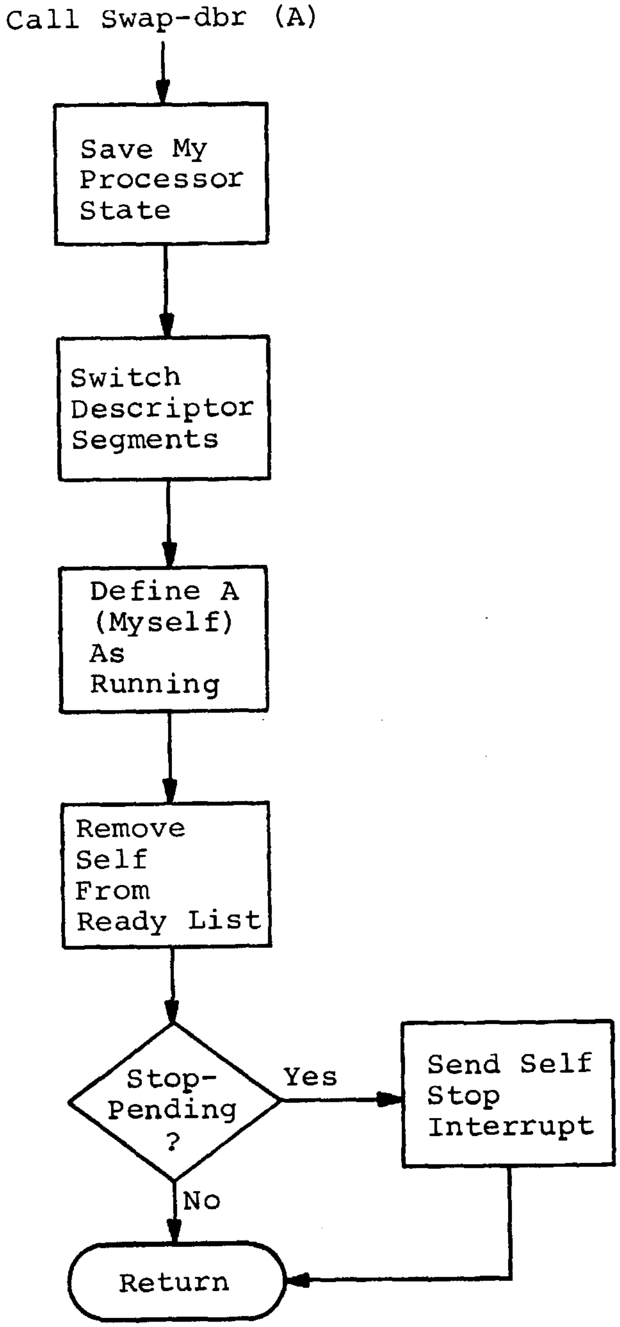

Figure II.8. Swap-dbr

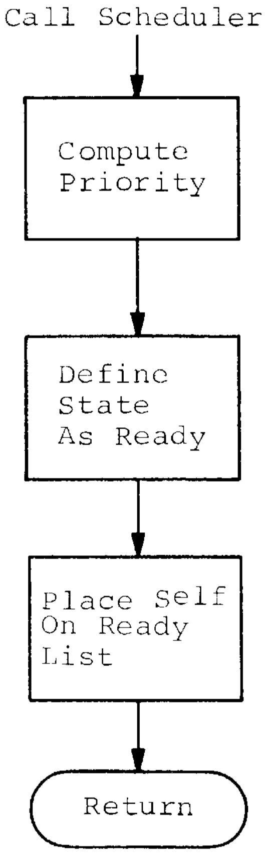

Figure II.9. Scheduler

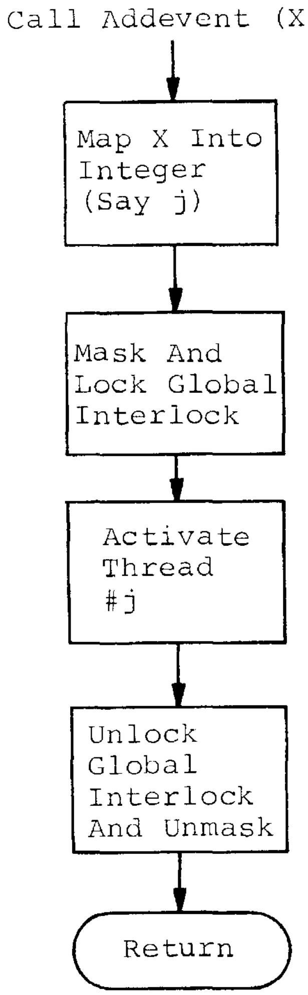
Call Addevent (X)
Map X Into Integer (Say j)
Mask And Lock Global Interlock
Activate Thread #j
Unlock Global Interlock And Unmask
Return

Figure II.10. Addevent

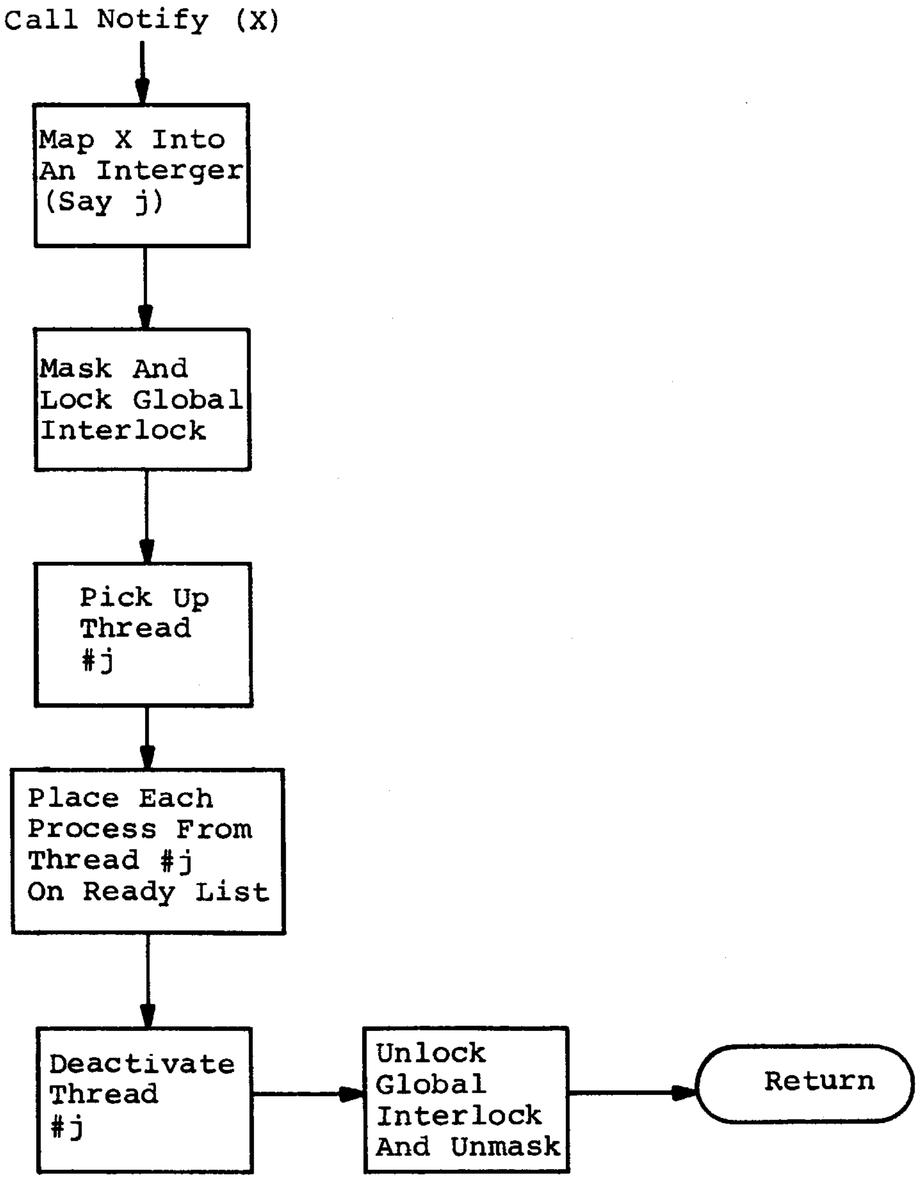
Call Notify (X)
Map X Into An Interger (Say j)
Mask And Lock Global Interlock
Pick Up Thread #j
Place Each Process From Thread #j On Ready List
Deactivate Thread #j
Unlock Global Interlock And Unmask
Return

Figure II.11. Notify

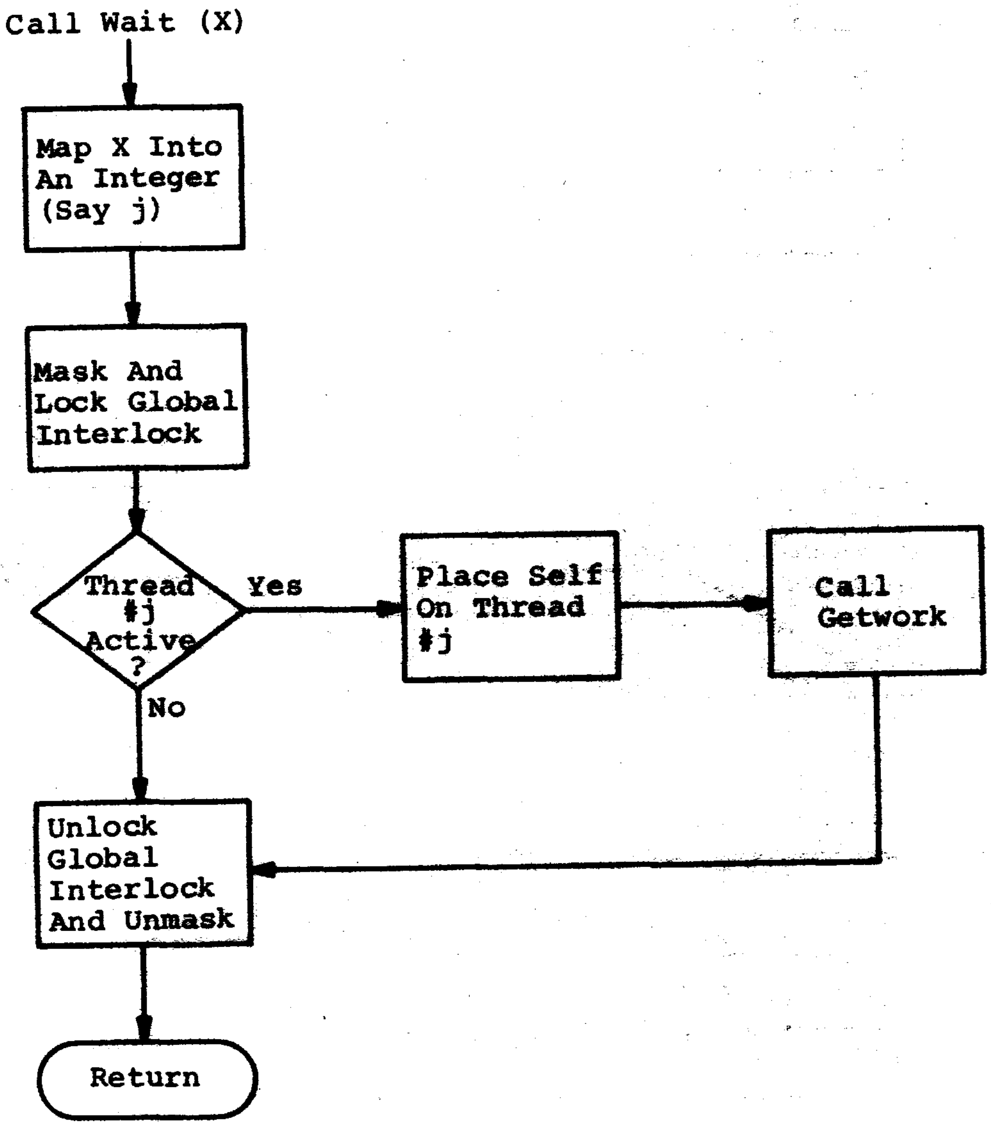
Call Wait (X)
Map X Into An Integer (Say j)
Mask And Lock Global Interlock
Thread #j Active ?
Yes
No
Place Self On Thread #j
Call Getwork
Unlock Global Interlock And Unmask
Return

Figure II.12. Wait

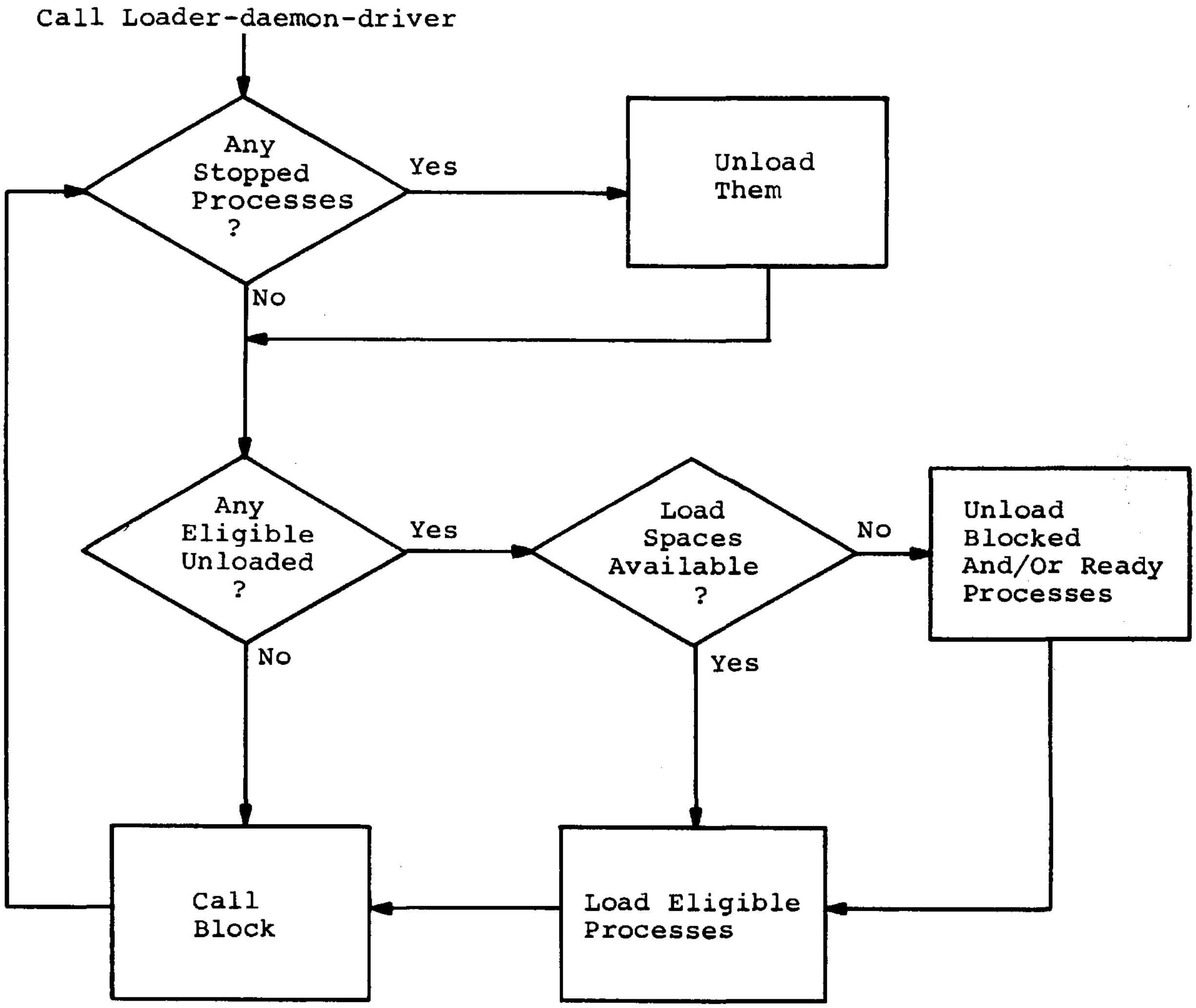

Figure II.13. Loader-daemon-driver Program

REFERENCES

Abbreviations used in the references:

AFIPS	American Federation of information Processing Societies
FJCC	Fall Joint Computer Conference
ACM	Association for Computing Machinery

References, in order cited:

1. Saltzer, J.H., "Traffic Control in a Multiplexed Computer System", Sc.D Thesis, M.I.T. Department of Electrical Engineering, May, 1966.

2. Glaser, E.L., et al., "System Design of a Computer for Time Sharing Application," AFIPS Conf. Proc. 27 (1965 FJCC), Spartan Books, Washington, D.C., 1965, pp. 197-202.

3. Dennis, J.B., "Segmentation and the Design of Multiprogrammed Computer Systems", Journal of the ACM 12, 4 (Oct. 1965), pp. 589-602.

4. Daley, R.C., and Neumann, P.G., "A General-Purpose File System for Secondary Storage", AFIPS Conf. Proc. 27 (1965 FJCC), Spartan Books, Washington, D.C., 1965, pp. 213-229.

5. Madnick, S.E., "Multi-Processor Software Lockout", to be published in the Proceedings of the ACM National Conference of August, 1968.

6. Daley, R.C., and Dennis, J.B., "Virtual Memory, Processes, and Sharing in Multics", Pesented at the Symposium On Operating System Principles, held at Gatlinburg, Tennessee, Oct. 1967, Sponsored by the ACM.

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Massachusetts Institute of Technology Project MAC	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP None

3. REPORT TITLE

Implementing Multi-Process Primitives in a Multiplexed Computer System

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

M.S. Thesis, Department of Electrical Engineering, September 1968

5. AUTHOR(S) *(Last name, first name, initial)*

Rappaport, Robert L.

6. REPORT DATE November 1968	7a. TOTAL NO. OF PAGES 133	7b. NO. OF REFS 6

8a. CONTRACT OR GRANT NO. Office of Naval Research, Nonr-4102-(01) b. PROJECT NO. NR-048-189 c. RR 003-09-01 d.	9a. ORIGINATOR'S REPORT NUMBER(S) MAC-TR-55 (THESIS) 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. AVAILABILITY/LIMITATION NOTICES

This document has been approved for public release and sale;
its distribution is unlimited.

11. SUPPLEMENTARY NOTES None	12. SPONSORING MILITARY ACTIVITY Advanced Research Projects Agency 3D-200 Pentagon Washington, D.C. 20301

13. ABSTRACT

In any computer system, primitive functions are needed to control the actions of processes in the system. This thesis discusses a set of six such process-control primitives which are sufficient to solve many of the problems involved in parallel processing, as well as in the efficient multiplexing of system resources among the many processes in a system. In particular, the thesis documents the work performed in implementing these primitives in a particular computer system — the Multics system — which is being developed at M.I.T.'s Project MAC.

During the course of work that went into the implementation of these primitives design problems were encountered which caused the overall program design to go through two iterations before program performance was deemed acceptable. The thesis discusses the way the design of these programs evolved during the course of this work.

14. KEY WORDS

Computers
Machine-aided cognition
Multiple-access computers
Multi-process primitives
On-line computers
Process-control primitives
Real-time computers
Time-sharing
Time-shared computers