TRAFFIC CONTROL IN A MULTIPLEXED COMPUTER SYSTEM

by

JEROME HOWARD SALTZER

S.B., Massachusetts Institute of Technology
(1961)

S.M., Massachusetts Institute of Technology
(1963)

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June, 1966

Signature of Author ______________________________
Department of Electrical Engineering, May 13, 1966

Certified by ______________________________
Thesis Supervisor

Accepted by ______________________________
Chairman, Departmental Committee on Graduate Students

TRAFFIC CONTROL IN A MULTIPLEXED COMPUTER SYSTEM

by

JEROME HOWARD SALTZER

Submitted to the Department of Electrical Engineering on May 13, 1966, in partial fulfillment of the requirements for the degree of Doctor of Science.

ABSTRACT

This thesis describes a scheme for processor multiplexing in a multiple user, multiple processor computer system. The scheme is based upon a distributed supervisor which may be different for different users. The processor multiplexing method provides smooth inter-process communication, treatment of input/output control as a special case of inter-process communication, and provision for a user to specify parallel processing or simultaneous input/output without interrupt logic. By treatment of processors in an anonymous pool, smooth and automatic scaling of system capacity is obtained as more processors and more users are added. The basic design has intrinsic overhead in processor time and memory space which remains proportional to the amount of useful work the system does under extremes of system scaling and loading. The design is not limited to a specific hardware implementation; it is intended to have wide application to multiplexed, multiple processor computer systems. The processor traffic controller described here is an integral part of Multics, a Multiplexed Information and Computing Service under development by Project MAC at M.I.T., in cooperation with the Bell Telephone Laboratories and the General Electric Company.

Thesis Supervisor: Fernando J. Corbató
Title: Professor of Electrical Engineering

ACKNOWLEDGEMENT

This thesis describes research done as part of the Multics development effort of Project MAC at M.I.T. The author is indebted to numerous people from Project MAC, the Bell Telephone Laboratories, and the General Electric Company who listened patiently to early iterations of this work and pointed out many overlooked difficulties. Chapter four, in particular, describes a problem which was worked out jointly by R.C. Daley, R.L. Rappaport, and the author.

Project MAC provided the environment and support for this thesis, and access to its interactive computer system as an aid in the composition of the actual document. The thesis was composed and reproduced on-line with the TYPSET and RUNOFF programs of the M.I.T. Compatible Time-Sharing System, and exists in computer accessible form.

A note of thanks is especially due the thesis committee, consisting of Profs. F.J. Corbató, R.M. Fano, and D.A. Huffman, as well as Prof. M. Greenberger. The time and effort taken out of such busy schedules to supervise the thesis are deeply appreciated.

Finally, and perhaps most importantly, an acknowledgement is due the author's wife Marlys, and daughters Rebecca and Sarah, who for several years have withstood the many problems of living with a graduate student who is a doctoral candidate.

J.H.S.
Waban, Massachusetts
May, 1966

CONTENTS

ILLUSTRATIONS

TABLES

Traffic Control in a Multiplexed Computer System

CHAPTER ONE

Introduction

In designing a computer utility, one is faced with two distinct classes of problems. The first class of problems is that of communication between people, by sharing algorithms and information, and of communication between the human and the computer. We term this class of problems intrinsic. The second class of problems is sharing of resources to lower the cost per user. We term this class of problems technological. Our choice of terms is deliberately intended to convey the notion that with appropriate advances in technology problems lying in the second class would not even exist; on the other hand, technological advances can only ease the solutions to the first class of problems.

The technological problem of resource multiplexing in a computer utility can be stated briefly as follows: Given a large computer system consisting of core memory, secondary storage, many input/output devices, and several processors; to design an operating system which allows effective multiplexing of resources among many independent users. The design must be flexible enough to allow for specialized needs of many computer installations without significant reprogramming, and it must scale up and down smoothly to allow easy growth of a computer installation.

The intrinsic problems of man-machine communication and information sharing in a computer utility can similarly be stated briefly: Given many users, and their private stores of data, algorithms, and other information; the system must provide access to, ability to manipulate, and controlled sharing of this information as flexibly as possible, while providing privacy for users or groups of users and protection of information against the accidental blunders of others.

In this thesis we describe a design for a traffic controller: the processor multiplexing and control communication section of an operating system. This traffic controller provides a workable solution, in a single package, to each of the following problems of the computer utility:

1. Multiplexing processor capacity among independent users.
2. Organizing multiple processors to allow reliability and expansion.
3. Keeping multiplexing overhead to a fraction of system capacity which is independent of system size.
4. Arranging for idle processes (1)* to contribute zero overhead in processor multiplexing time and core space.
5. Allowing different users to see different operating systems while running simultaneously.

---

* Footnotes, indicated in parentheses, appear at the end of each chapter. References, indicated in brackets, are collected together starting on page 117.

6. Permitting parallel processing (including input/output) to a single user.

7. Allowing communication of control signals between users.

The first four of these problems have to do with resource sharing, and we therefore class them as technological. The last three problems are examples of intrinsic problems.

Before going any further, we should first consider the reasons why the problems tackled by the traffic controller are interesting. First, multiplexing a processor among many independent users is an effective way of achieving an interactive but economical computer system. It is also a powerful technique which speeds production time of input/output limited jobs, and permits balancing of resources across a spectrum of jobs, none of which may be individually matched to the computer system.

Secondly, organized control of identical, multiple processors provides a technique for expanding system capacity without the need to over-reach whatever is the currently available technology in processor speed. Also, a properly organized multiple processor system provides great reliability (and the prospect of continuous operation) since a processor may be trivially added to or removed from the system. A processor undergoing repair or preventive maintenance merely lowers the capacity of the system, rather than rendering the system useless.

Third (and fourth), the ability of the basic design to scale over a wide range of system capacity, load, and number of processes, means that it may be used without modification as the basis for a one-processor, three-user time-sharing system, a multi-processor airline reservation system with 5000 agent sets, or a weather-prediction system performing dependent but parallel computations at thousands of "grid points."

Fifth, an organization in which each user sees a private supervisor, which may be different for different users so long as it follows the ground rules of traffic control, means that the system is easily applicable to so-called real-time or process control functions while simultaneously serving more standard interactive or over-the-counter users. This feature also aids greatly in debugging new versions of a supervisor while maintaining continuous operation of the system for regular customers.

Sixth, smooth inter-process control communication features open the way for implementation of languages which take advantage of, or allow expression of, parallelism in algorithms. With the same facility, the final requirement of intercommunication among otherwise independent users also becomes possible. Immediate applications abound; for example, a group of persons may work on the same project from typewriter consoles in different buildings. Viewing input and output initiation as another example of inter-process communication places all parallel operations on a symmetrical and identical basis. The complexity of organizing a large problem requiring parallel processing capabilities is thereby greatly reduced.

The interest in solutions to these problems is clear. The significance of the proposed traffic controller is that workable solutions to all of these problems are presented in a relatively small collection of interacting procedures.

The traffic controller and operating system described here are being implemented as an integral part of the "Multics" system (Multiplexed Information and Computing Service), which, as its name implies, is a computer system organized to operate as a public utility. The general organization and objectives of Multics have been described in a group of six papers given at the 1965 Fall Joint Computer Conference [1, 2, 3, 4, 5, 6]. The reader interested in exploring further the economic and technological justifications for the notions of a multiplexed computer system is referred to these papers, especially [1]. In this thesis we will make the assumption that the reader is familiar with issues such as reliability, accessibility, and a shared, community data base which underly the Multics concept. In particular, we assume a two-dimensional segmented address space implemented within the system hardware (2).

In a segmented address space, a processor generates a two-part address for all instruction and operand fetches. The first part of the address is a segment number, the second a word number within the segment. The segmented address space is implemented by means of special processor hardware, which refers to a map stored in core memory that gives the absolute core address of the base of each segment. This map is itself a segment, the descriptor segment; its absolute address is stored in a descriptor segment base register in the processor. The descriptor segment may contain missing-segment bits for some segments. If a program attempts to refer to one of these missing segments, the processor will fault to a supervisor procedure which can find the segment, load it into memory, and continue the interrupted program.

We further assume that the memory is paged. Paging allows each segment to be broken up and thereby fit into core memory wherever space is available, without the need for contiguous locations. Paging is accomplished by processor hardware which is very similar to the segmenting hardware described above. For a thorough discussion of the techniques of and motivations for segmentation and paging, the reader is referred to several recent papers [2, 7, 8]. The crucial feature provided by a two-dimensional address space is that a single segment in core may appear simultaneously in the address space of two different processors, with distinct segment numbers. The ability of independent users of the computer system to share segments of addressable memory is a cornerstone assumption in the design of the operating system.

In order to successfully make use of shared procedure segments, we assume that all procedures (at least those of the operating system) are pure, that is, they do not modify themselves. A segment containing only pure procedures can then safely be shared by any number of processes. Data used by a procedure appears in a distinct segment which may or may not be shared among processes, depending on the purpose of the procedure. As we will see, the operating system is made up of closed subroutines which call on one another. When a subroutine has finished executing it returns to its caller. A private data segment is used as a call stack to store the return location when a subroutine is called. If that subroutine calls another, the return location is placed on top of the call stack so that returns will be made in the proper order and to the proper location. The call stack may also be used for temporary storage needed by a procedure. By using pure procedures and a call stack throughout, any procedures in the supervisor may be called recursively, if such usage is appropriate.

Background.

Multiplexed computer systems are not new, but general organizations for such systems have not yet been described. Critchlow, in a review article [9] traces the evolution of multiprogrammed and multiprocessor computer systems, so we will not need to do so here. Most published work on processor multiplexing falls into three categories:

1. Techniques by which a programmer may specify parallelism in his programs [10, 11]. These papers offer suggestions for implementation of a system to make use of such parallel specification, but not a complete design.
2. System designs to multiplex one or more processors among strictly independent jobs. Codd [12] and Thompson [13] describe multiplexed operating systems designed to speed processing of batch jobs on the IBM 7030 and the Burroughs D825 computers, respectively. A similar system has been designed for the GE 635 computer. Several time-sharing systems (see, for example, [14]) have been designed to multiplex a single processor among

interactive console users. Although some of these designs allow inter-user procedure or data sharing, there is no provision for inter-process control communication. Ad hoc additions to these systems (3) have provided some means of inter-process control communication, but no general structure.

3. Highly specialized "real-time" systems in which the specific application of the system heavily outweighs other features of the design. Examples of system designs in this latter category are the SABRE airline reservation system, and Project Mercury control, both described in [15]. As would be expected, such designs solve their intended multiplexing problems, but unfortunately leave no general structure on which to build a system for a different application.

The proposed multiprogrammed operating system for the IBM system/360 series of computers [16] is probably the system appearing in the literature which is closest in concept to the work taken up in this thesis. That system permits a restricted inter-process control communication facility for processes working under the same job; it remains to be shown that it can be extended to a multiple processor configuration since details of the design have not yet been published.

Method.

In chapter two, we first briefly describe the organization of the entire Multics operating system, so that we may view the later discussion of processor traffic control in an appropriate perspective.

We will then study traffic control in three stages. First, we assume an abundance of core memory and processors so that multiplexing is not needed. This assumption makes it possible to isolate the fundamental problems of inter-process communication. Then, we study the technological problem of multiplexing a limited number of processors among many competing processes, again assuming sufficient core memory to carry out the multiplexing. Chapter three concludes with a complete design for the traffic controller. Finally, in chapter four, we explore a second technological problem, the consequences of core memory size limitations on processor multiplexing.

Chapter five reviews the entire traffic controller design and discusses techniques by which it may be evaluated when in operation. Included here is a discussion of the crucial issue of how the system "scales"; that is, the effect of expansion of the number of users, the presented load, the size of memory, and the number and speed of processors.

---

(1) A process may be loosely defined as a program in execution; a more careful definition will be given at the beginning of chapter three.

(2) The field of computation systems, and this thesis, are replete with technical jargon. This thesis uses wherever possible terminology consistent with current literature as exemplified by the cited references.

(3) For example, by allowing one process to appear to be an input/output device to another process, as in the CTSS inter-console message facility [17].

CHAPTER TWO

Organization of the Computer Utility

The term "computer utility" by its very nature implies marketing of a useful resource in a usable form. Although immense computing power, sharable secondary storage, and flexible access to input and output devices are indeed useful resources, the primary function of the computer utility is to organize such resources into a usable, and thereby marketable, form.

From one point of view the marketing of computer resources is much the same as the marketing of candy bars. The man on the street would be quite pleased to purchase his candy bar direct from the factory at the candy jobber's prices. On the other hand, his enthusiasm wanes when he discovers that he must take not one candy bar but a carload, and delivery will require six weeks. In much the same way the ordinary computer user is quite unprepared to tackle the problems of managing several processors, I/O interrupts, and disk track organization, even though his particular problem might require sizable amounts of computer time, input-output, and secondary storage space.

Again using the candy bar example, we observe that the candy bars pass through several hands: the jobber, the wholesaler, the distributor, before they turn up on the drugstore counter. At each of these levels the product of the previous level is transformed into a resource with a wider market. The carload of candy bars is wholesaled in gross cartons; the distributor once a week provides the drugstore with boxes of 24 candy bars. Finally, the man on the street wanders in and purchases just one, whenever he likes. In a very similar manner, we may view the resources of the computer utility as being transformed three times, each time producing a resource that is successively more "marketable":

1. Starting with the basic hardware resources available, the "hardware management" procedures have the function of producing hardware independence. They do so by simulating an arbitrarily large number of "pseudo-processors" each with a private segmented address space (which may contain segments shared with other pseudo-processors), easy access to a highly organized information storage hierarchy, and smooth input/output initiation and termination facilities. The resulting resource is independent of details of hardware or system configuration such as processor speed, memory size, I/O device connection paths, or secondary storage organization.
2. Working with these pseudo-processors and the information storage hierarchy, the "resource management" procedures allocate these resources among "users", providing accounting and billing mechanisms, and reserving some of the resources for management services, such as file storage backup protection, line printer operation, and

storage of user identification data.

3. Finally, these allocated and accounted resources can be used by the ultimate customer of the computing utility either directly by his procedures or to operate any of a large variety of library commands and subroutines. Included in this library are a command language interpreter, a flexible I/O system, procedures to permit simple parallel processing, language translators, and procedures to search the information storage hierarchy and dynamically link to needed programs and data.

We now wish to study each of these transformations in more detail.

Hardware Management.

The basic hardware resources available to the utility are the following:

1. One or more identical processors.
2. Some quantity of addressable primary (probably core) memory. The processors are equipped with hardware to allow addressable memory to appear to be paged and segmented. It is not necessary that all possible memory addresses correspond to core locations. One might expect to have 100,000 words of core memory for each processor.
3. A substantial amount of rapidly accessible secondary storage. This secondary storage might consist of a large volume, slow access core memory, high speed drums, disks, data cells, or any combination thereof which proves to be economical. The total amount of accessible secondary

storage might be on the order of 100 million words per processor, although this figure can easily vary by more than an order of magnitude.

4. Channels to a wide, in fact unpredictable, variety of input and output devices, including tapes, line printers and card readers, typewriter consoles, graphic display consoles, scientific experiments, etc. In an installation committed primarily to interactive usage, one might find 200 typewriter channels, plus a few dozen other miscellaneous devices. Each of these channels can produce signals indicating completion or trouble. The signals are transmitted to the system in the form of processor interrupts.

5. Various hardware meters and clocks suitable for measuring resource usage.

The hardware management routines must do two very closely related jobs. First, they must shield the user of the system from details of hardware management. The user should be essentially unaware of system changes such as addition of a processor, replacement of processors by faster models, or replacement of a data cell by an equivalent capacity disk memory. Except for possible improvements or degradations of service quality, his programs should work <u>without</u> <u>change</u> under any such system modification. Second, the hardware management routines must handle the multiplexing of system resources among users in such a way that the users may again be unaware that such multiplexing is going on. Included in this second job is the necessary protection to insure that one user cannot affect another user in any way without previous agreement between the two users.

The strategy chosen here to implement this hardware management is the following. Using the hardware resources listed above and two major program modules, the traffic controller and the basic file system, simulate (by multiplexing processors and core memory) an arbitrarily large number of identical pseudo-processors, and an information storage hierarchy in which data files are stored and retrieved by name.

The information storage hierarchy is a tree-like structure of named directories and files which is shared by all users of the system. Access to any particular directory or file is controlled by comparing the name and authority of the user with a list of authorized users stored with each branch of the tree. This structure allows sharing of data and procedures between users, and also complete privacy where desired.

The pseudo-processors look, of course, very much like the actual hardware processors, except that they are missing certain "supervisory" instructions and have no interrupt capability. Each pseudo-processor has available to it a private two-dimensional address space. Within the address space are a number of supervisor procedures capable of carrying out the following basic actions upon request:

1. "Mapping" any named file or directory from the storage hierarchy into a segment of the address space. Files appearing in the information storage hierarchy are

identified by a <u>tree name</u> which is a concatenation of the name of the file within its directory, the name of the directory, the name of the directory containing this directory, etc., back to the root of the tree. As we will see below a utility program named the "search module" may be used to establish the tree name of a needed segment so that the map primitive may be used. The search module itself operates by temporarily mapping directories into addressable storage in order to search them. Use of the map primitive does not imply that any part of the file is actually transferred into core storage, but rather that the file is now directly addressable as a segment by the pseudo-processor. When the pseudo-processor actually refers to the segment for the first time, the basic file system will gain control through missing-segment and missing-page faults and place part or all of the segment in paged core memory. Except for the fact that the first reference to a portion of a segment takes longer than later references, this paging is invisible to the user of the pseudo-processor. The same file can appear as a segment in the address space of any number of processors, if desired; options allow the processors to share the same copy in core, or different copies.

2. Blocking, pending arrival of a signal from an I/O channel or some other pseudo-processor. A pseudo-processor blocks itself because the process which it is executing

cannot proceed until some signal arrives. The signal might indicate that a tape record has been read, that it is 3:00 p.m., or that a companion process has completed a row transformation as part of a matrix inversion.

3. Sending a signal (here known as a "wakeup") to another pseudo-processor or to an input/output channel. (From the point of view of a pseudo-processor, an I/O channel looks exactly like another pseudo-processor.) The wakeup facility, in combination with the ability for pseudo-processors to share segments, permits application of several pseudo-processors simultaneously by a single user. A user may thus specify easily parallel processing and input/output simultaneous with computation.

4. Forcing another pseudo-processor to block itself. This primitive, named "Quit", allows disabling a pseudo-processor which has gotten started on an unneeded or erroneous calculation.

All of these primitive functions are constructed as closed subroutines which are called using the standard call stack described in chapter one.

Figure 2.1 shows a typical hardware configuration of the utility, while figure 2.2 indicates the apparent system configuration after the hardware management procedures have been added. An important difference between these figures is that while figure 2.1 may change from day to day (as processors are repaired and a disk is replaced with a drum) figure 2.2 always is the same, independent of the precise hardware configuration.

When a pseudo-processor calls the "map" entry of the basic file system, the file system establishes a correspondence between a segment number of the pseudo-processor address space and a file name on secondary storage by placing an entry in a segment name table belonging to this pseudo-processor. It does not necessarily, however, load any part of the file into core memory. Instead, it sets a missing-segment bit in the appropriate descriptor word in the descriptor segment of the pseudo-processor. This bit will cause the pseudo-processor to fault if a reference is made to the segment.

Sometime after calling the "map" entry, the pseudo-processor may attempt to address the new segment. When it does so, the resulting missing-segment fault takes the pseudo-processor directly back to the segment control module of the basic file system, which now prepares for missing page faults by locating the file name corresponding to the segment number in the segment name table, placing the secondary storage location of the file in an active segment table, and creating in core memory a page table for the segment. This page table is filled with missing-page bits, and none of the file is actually loaded into core memory yet.

The pseudo-processor is then allowed to continue its reference to the segment. This time, a missing-page fault takes the pseudo-processor to the page control module of the basic file system. Page control must locate two items: a space in core memory large enough for the missing page, and the location on secondary storage of the missing page. Establishing a space in core memory may require unloading some other page (possibly belonging to some other pseudo-processor) onto secondary storage. A policy algorithm in the "core control" module decides which page or pages in core are the best candidates for unloading, on the basis of frequency of usage of the pages.

Having established space in core memory for the page, and initiated the transfer from secondary storage, page control blocks the pseudo-processor pending arrival of the page. When the page is in, this pseudo-processor is re-awakened by the basic file system operating for some other process, page control returns to the point at which the missing-page fault occurred, and the pseudo-processor now completes its reference to the segment as though nothing had happened. Future references to the same page will succeed immediately, unless the page goes unused for a long enough time that the space it is holding is reclaimed for other purposes by core control. If the space is reclaimed, core control sets the missing-page bit in the page table <u>on</u>, and writes out the page onto secondary storage. A later missing-page fault will again retrieve the page.

As we will see in chapter four, some segments cannot take part in the paging in-and-out procedure; these segments must be "wired down" (that is, they are not removable) since their contents are needed, for example, in order to handle a missing-page fault. A general property of the file system organization is that a missing-page fault cannot be encountered while trying to handle a missing-page fault. The reason for this organization is not that a recursive missing-page fault handler is impossible to organize, but rather that the depth of recursion must be carefully controlled to avoid using up all of core memory with recursion variables (at least the call stack <u>must</u> go into a wired down segment.) The method chosen here to control recursion depth is to prevent recursive missing-page faults in the first place.

The method of implementing the secondary storage hierarchy, the "map" primitive, and core memory multiplexing has been described in a paper on the basic file system by Daley and Neumann [4] and the reader interested in more detail is referred to that paper. The multiplexing of hardware processors to produce many pseudo-processors is the function of the traffic controller, and is the subject of the remaining chapters of this thesis.

<u>Resource Management</u>.

The hardware management programs transform the raw resources of the computer system into facilities which are eminently more usable, but these facilities must be made available (allocated) to users of the system before those users can accomplish anything. Also, certain of the transformed facilities must be reserved for the system's own use in operation, administration, and preventive maintenance. Finally, a flexible, fair, and accurate accounting mechanism must be provided to determine how and by whom the system is actually being used.

The most important function of resource management is to define the concept of a "user" of the utility. A user, is, roughly, a person, working on a project, who signs out a portion of the system facilities by "logging in." He may work in concert with other users of the system on a single larger project, but his coming and going is independently noted in system logs. The definition of a person working on a project must be relaxed slightly to include the possibility of a so-called "daemon" user (1) which is not directly associated with a person. The definition of a daemon user is that it is automatically logged in to the system when the system is initialized; one cannot identify any particular person who claims to be this user. The daemon generally performs periodic housekeeping functions. (Most daemons, in fact, are creations of resource management, but there are also applications for customer-provided daemons.)

To get the flavor of the techniques used by resource management, we may consider the path followed in logging in from a typewriter console. One pseudo-processor is reserved for a daemon user to which we give the name "answering service". This pseudo-processor is given access to every typewriter channel which is not presently in use. The process operating on the pseudo-processor activates every attached typewriter channel so that the channel will return a signal when a console dials up, or turns power on in the case of direct connections. The process then blocks itself awaiting a signal from some typewriter channel. When a person dials up to a channel, that channel wakes up the answering service process which immediately brings into play two more pseudo-processors. One pseudo-processor is assigned the typewriter channel and a typewriter management process is initiated on that pseudo-processor. A "listener" process is initiated on the other pseudo-processor. The listener process reads from the typewriter by asking the typewriter manager process for the next line of input. The listener may have to wait if a line has not yet been typed. The listener can take any desired action upon the line, including establishing a process on yet another pseudo-processor to perform some computation. The programs executed by the listener and the typewriter manager come from the library, which is discussed in the next section, so we will not go into any further detail here. Their first action is, of course, to execute the "login" command to establish the identity of the user and his authority to use the system.

Logging in is accomplished by comparing the typist's credentials with a list of all authorized users which is stored in the secondary storage hierarchy. (As we will see, the storage hierarchy is used extensively for administrative purposes.) When a match is found, information stored there indicates this user's access privileges, authorities, and the section of the directory structure in which he keeps his private files. The system log (a file in the storage hierarchy) is updated to show that this user is logged in, and the typist may now begin typing commands.

The point of the description of logging in is to illustrate the techniques used in resource management, not the details. The most important feature of these techniques is that they are based on usage of the pseudo-processors and information storage hierarchy provided by the hardware management programs. They may, therefore, be debugged and replaced while the system is operating, in exactly the same way as any user program. They are also relatively independent of the configuration of the system.

A number of similar operations are carried out by resource management in other areas. For example, a daemon user continually copies newly created files in the storage hierarchy out onto tape for added reliability in case of some catastrophe. Another daemon user periodically wakes up and "checks out the system" by running test and diagnostic procedures. An example of an ordinary user dedicated to resource management is the operator in charge of detachable input and output devices such as tape and disk packs. At his typewriter console he receives messages requesting him to mount reels; he may reply when the reel is mounted or it cannot be found.

Finally, within every address space, certain resource management procedures are inserted in the path between a user procedure and the supervisor routines described under hardware management. These resource management procedures perform resource usage accounting for this process. A system of accounts is maintained within the storage hierarchy, which allows a project supervisor to allocate resources to group leaders who can in turn allocate to individual users. Every pseudo-processor draws on some account in this hierarchy. Also, among the library procedures available to any process are "system transaction programs" which allow the user to arrange special classes of service, sign up in advance for tape drives, etc.

Dynamic Linking, Hierarchy Search, and the Library.

So far, the hardware management procedures have insulated the user from the details of the system configuration and secondary storage management, and resource management procedures have established doors through which a user may enter and leave the system, and have his resource usage accounted for. Before the system is useful to the average user, however, a variety of utility and service (library) programs must be available to him. The library is merely a collection of procedures stored in one section of the information storage hierarchy. This library is built upon the foundations laid by hardware and resource management. It is flexible and open-ended, and procedures drawn from the library operate in exactly the same way as any user provided procedure drawn from elsewhere in the information storage hierarchy.

Fundamental to the usage of the system are dynamic linking and storage hierarchy search procedures. The pseudo-processor provided by hardware management has the capability of producing a linkage fault when a procedure attempts to refer to a segment which has never been mapped into addressable storage. When establishing a new pseudo-processor, one normally places a linkage fault handler in the new address space. When the new pseudo-processor encounters a linkage fault, the linkage fault handler (linker) locates the needed segment in the information storage hierarchy by calling the search module. The linker then maps the segment into addressable storage with the "map" primitive discussed earlier, and resets the inter-segment linkage pointer which caused the fault so that faults for that reference will not occur in the future.

By providing an appropriate algorithm to search the information storage hierarchy for needed segments, the user can arrange that a newly established pseudo-processor execute any desired sequence of procedure. The search may, of course, include those sections of the information storage hierarchy containing library procedures provided by the utility.

For example, consider the sequence of linkage faults and searches implicit in the logging-in procedure described earlier. The answering service establishes a new pseudo-processor to run the "listener" process, initially mapping into its address space the standard system linker, a search algorithm which looks at the system library, and a one-instruction procedure which attempts to transfer (through a linkage fault) to a program named "listen". The pseudo-processor is started at the planted transfer instruction. Of course, it immediately gets a linkage fault, and the linker calls the search module to locate the "listen" program. The search module finds a procedure by this name in the system library, the linker maps it into addressable storage, and the transfer instruction is continued. This time it completes execution, and the "listen" procedure is now in control of the pseudo-processor. As it calls on various subroutines, for example to communicate with the typewriter manager process, it gets more linkage faults, and triggers appropriate searches through the library. As needed, the address space of the pseudo-processor collects the subroutines and data segments required to operate a listener process.

An important library procedure is the "Shell", a command language interpreter which is called by the listener to interpret the meaning of a command line typed by the user. The Shell takes a typed command to be the name of a subroutine to be called with arguments, e.g., if the user types the command

PL/I ABCD

the Shell would take this to mean that it should call a subroutine named "PL/I" with one argument, the character string "ABCD". It therefore sets up linkage to a subroutine named PL/I (with a linkage fault in the path, of course) and attempts to call the subroutine. The resulting linkage fault causes a search of the library for, and linkage to, a procedure segment named "PL/I". When the PL/I compiler ultimately begins execution, it will similarly search for and link to the file named ABCD and (presumably) translate the PL/I program found there.

Among the library procedures commonly executed as commands are procedures to help type in and edit new files to be stored in the information storage hierarchy, translators for program files, and commands to alter the search algorithm, for example to search a portion of the hierarchy containing the user's own data and procedure segments. Note that through the mechanism of the Shell, any procedure segment, public or private, appearing in the information storage hierarchy and to which a user has access rights can be called as a command from the console.

Other library procedures include an input/output control system which allows symbolic reference to input and output streams and a substantial measure of device independence. These procedures include necessary inter-process communication facilities required to overlap input/output with other computation.

Through the mechanism of the linker and the search module an arbitrarily elaborate collection of utility programs may be established, yet all such programs are on an identical footing with the user's own programs. That is, they may be checked out and replaced while the system is in operation using the full resources of the system to aid in the checkout. The open-endedness of the library means that it is likely that there will be some users who never execute anything but procedures from the library. It is even possible, through the mechanism of access control provided in the information storage hierarchy, to have a user who, since he has no access to any compilers or input editors, can <u>only</u> execute commands found in some library.

Summary.

We have in this chapter seen a brief overview of several aspects of the organization of a computer utility. In this overview, we have seen how the raw resources of the system are successively transformed, first into configuration- and detail-independent resources consisting of pseudo-processors and an information storage hierarchy, secondly into allocated and accounted resources ready to be put to work, and finally, through a linker, search module, and system library, into a full scale, flexible operating system with a multitude of readily accessible utility procedures. Our overview has necessarily been too broad-brush to go into much detail on how these various techniques are implemented. The reason for the overview has been to give enough of a framework so that we can study in detail the particular problem of processor multiplexing, one of the fundamental aspects of hardware management. Chapter three begins our study of this topic.

---

(1) "dae·mon, n. in Greek mythology, any of the secondary divinities ranking between the gods and men; hence, 2. a guardian spirit." (Webster's New World Dictionary, 1958.)

CHAPTER THREE

Traffic Control in the Computer Utility

In chapter two we divided the operating system of a computer utility into three layers: hardware management, resource management, and the library. In this chapter we split the layer of hardware management into memory (core and secondary storage) management and processor management. We take up the detailed study of processor management, assuming that the memory management modules--the basic file system--already exist and operate as was briefly described in chapter two. Our general strategy here will be to begin by assuming that there are no technological problems of processor multiplexing. After examining the intrinsic problems which remain, we introduce the technological problems one by one.

SECTION ONE: THE CONCEPT OF "PROCESS"

In the sections which follow, "traffic control" will be described as the problem of multiplexing a limited number of processors among many processes and providing inter-process communication. We should therefore first define precisely our use of the term "process."

A process is basically a program in execution by a processor (1). This definition, while it appears to be precise, is in fact somewhat vague because the terms "program" and "processor" can be given widely varying interpretations. Although the IBM 7094 central processor, and a program coded in the FAP language, are a good example of the terms "processor" and "program" (and could be used to help provide one concrete definition of a process) we may observe other examples of "processors" and "programs."

For instance, one can consider the M.I.T. Compatible Time-Sharing System [17] to be a "processor" whose instruction set consists of system commands. One may give this processor a program in the form of a list of commands (RUNCOM); he may then talk of his "process" proceeding from command to command in his program. The fact that the implementation of each of his commands may actually cause five data channels and two central processing units to execute instructions simultaneously is irrelevant to this particular definition of a process.

We may thus conclude that the essential element in the definition of a process is a statement about the capabilities of a processor; the processor is not necessarily one implemented in hardware, but rather a composite processor made up of the hardware and programs of the system in which the process is executed. The composite processor may have either more or fewer apparent capabilities than the actual hardware processors which are the basis of the system.

As was described in chapter two, the fundamental technique of the traffic controller is to simulate an arbitrarily large number of pseudo-processors, each with its own two-dimensional address space. Each pseudo-processor is given the following capabilities:

1. Accessibility to a private segmented address space for instructions and data. This address space may include segments accessible to other pseudo-processors.
2. An instruction repertoire including the usual arithmetic, logic, shift, and conditional branch instructions.
3. Ability to "fault" (a type of conditional subroutine jump) upon execution of certain instructions.
4. Ability to call on supervisor procedures to extend the defined address space of the pseudo-processor, and to communicate control signals with other pseudo-processors and input/output channels.

If one likes, the last ability can be described as an extension of the instruction repertoire of the hardware processor.

The one capability of commonly described operating system pseudo-processors which is not included in our pseudo-processor is the "interrupt," or "courtesy call," a jump to a special subroutine in response to an arbitrarily timed (asynchronous) signal, for example, from an input/output channel. As we will see, the ability to use several communicating pseudo-processors provides a flexible and easy to use facility to replace the interrupt.

Our definition of a process is now clear. A process is a program in execution by a pseudo-processor. The internal tangible evidence of a process is a pseudo-processor stateword, which defines both the current state of execution of the process and the address space which is accessible to the processor. There is, then, a one-to-one correspondence between processes and statewords, and also between processes and address spaces. It will in fact be convenient to make use of this correspondence and identify a process with its address space. In terms of the two-dimensional segmented address space hardware described in chapter one, every process is identified with a descriptor segment. If we further assume for simplicity that descriptor segments are not shared between processes, we may establish a one-to-one correspondence between processes and descriptor segments. The stateword of a process includes a pointer to the descriptor segment of the process.

To maintain our definition of a process despite the realities of processor and memory multiplexing, we will place within the address space of every process a set of procedures--the traffic controller--which exercise further capabilities of the actual processor. Most of the instructions of the traffic controller will in fact be carried out within the address space of this process, but they are viewed as part of the implementation of the pseudo-processor.

SECTION TWO: TRAFFIC CONTROL WITH DEDICATED PROCESSORS

Our strategy of discussion of the general problem of processor multiplexing is to start by assuming an abundance of actual processors and of core memory. In particular, we assume that there is a processor available to assign to every process, and that there is enough core memory available so that at least the current procedure of every process is resident in core. We will discard these assumptions later, but for the moment they allow us certain insights into the problems of traffic control: we are able to separate the intrinsic problems of inter-process control communication from the technological problems of processor and core memory multiplexing.

Inter-Process Control Communication.

The intrinsic problem of inter-process control communication is to provide a means for two or more processes to work in parallel, cooperating on a single computation. This cooperation requires that the processes be able to synchronize their operation, that is, one process must be able to wait for a signal from another. The signal, when it comes, means that the first process may continue, for example because some input data it needs has now been prepared. (Another problem of inter-process control communication is that of turning off a process which has gotten started on an erroneous or unneeded computation. We postpone consideration of this problem until section three of this chapter.)

We start by considering a single process. This process follows a programmed path in its procedures, performing whatever computations are indicated there. Let us formalize slightly the structure of the program being executed by the process to see if we can determine what are its needs for traffic control. We may picture the process as having a work queue which is a list of "tasks" to do stored in a segment accessible to the process. We envision the process looking in the work queue, discovering a task there, and performing the computation indicated. When finished with this task, it goes back to its work queue for the next task. Let us assume that there is some mechanism by which some other process can add tasks to the work queue. If the second process should add a task to the work queue while the first process is executing another task, it is apparent that the first process will not discover the existence of the new task until it has completed its previous task. According to our assumptions about the capabilities of the processor executing this process, there is no way for the second process to force the first one to stop what it is doing and look at its work queue. A process can only follow its program.

Suppose the process should finish executing the last task in its work queue. What should it then do? It could loop by continually looking in the work queue, and finding nothing look again. When another process adds a new task, the process will discover it immediately and begin computation. A different solution is for the process to block itself until some new work arrives. The ability to block a process is a traffic control function which we will conceive as a closed subroutine:

```
call block;
```

In the dedicated processor system this subroutine can be implemented in hardware by a halt instruction. We will see later, when we study processor multiplexing, that the call to block will be taken to be an opportunity to give the processor to another process.

An immediate complication arises if a process blocks itself. When a new task is added to the work queue of a process which has blocked itself, the process must somehow be unblocked ("wakened".) Unblocking means that the closed subroutine "block" returns to its caller.

We thus conclude that for a single process, the only "traffic control" feature which is needed is the ability for a process to block itself, and for another process to be able to waken the blocked process. We may define two execution states of a process, _running_ and _blocked_. A process is _running_ if it is executing instructions, it is _blocked_ if it is waiting for a signal to continue execution.

The Two-Process System.

The next level of complexity we wish to consider is the two-process, two-processor system. Since each process has its own processor, we do not yet need to become involved in issues of processor multiplexing. Assume for the moment two identical one-process systems as described above are placed side by side. If the two systems are independent, there are no particular complications. The two-process system is of interest, however, because we wish to provide some means of communication between the two processes. We provide this communication by means of an area of core storage which appears in the address space of _both_ processes--a common segment. If the common segment is _read-only_ (neither process can alter its contents) then there is still no communication between the processes. They are however, sharing a section of memory which may contain either data or procedure.

If either process can alter the contents of the common segment we have a means of communication between the processes. Let us name the two processes "A" and "B", and assume that "B's" work queue is a common segment between "A" and "B". Suppose that "B" is running, working on some previous task in the work queue, and "A" adds something to the work queue. Then, when "B" is finished with its earlier task it will discover the task from "A" in the queue, and perform it.

Suppose "B" finishes its last task and calls block. If "A" now comes along and places new work in the queue, "B" will not look at the queue, because he is blocked. "A" must do something to cause "B" to wake up. In addition to data communication, an inter-process <u>control</u> communication function is needed. We claim this as a traffic control function, and provide another closed subroutine named <u>wakeup</u>.

With the two-process system,

```
call wakeup;
```

means unblock the other process. Again, it is simple to wire such a function into the hardware of the system, and "call wakeup" becomes the execution of a single instruction by "A". To generalize to a system with many processes and an equal number of processors, we need merely add the concept of a process(or) identification tag, and generalize the traffic controller entry (or hardware instruction) to:

```
call wakeup(B);
```

if "B" is the identification tag of the process to be wakened.

An example of a two-process system.

We consider now a specific example of a two-process system. In this example, process "A" is performing a lengthy computation, and occasionally produces numbers to be typed on the typewriter. Process "B" is operating the typewriter. The work queue of process "B" is a buffer into which "A" puts the results he wishes to have typed. To operate the typewriter, "B" executes a "type-out" instruction, which takes 100 milliseconds, and types one character.

If "A" falls behind, "B" will type out everything in its work queue, and block itself. Then, if "A" puts a new message in "B's" queue, he must wake up B. For simplicity, he always wakes up "B"; we make the convention that the wakeup call does nothing if "B" is already running. Figure 3.1 is a flow diagram of the two-process system.

What if process "B" falls behind? This means "A" is producing results at an average rate faster than "B" can print them. If the work queue is long enough, this does not hurt anything. The situation is detected when "A" discovers that the work queue is full. ("Full" is a relative concept, based perhaps on a consideration of how far ahead of the typewriter we wish the computation to get.) A solution is to put as the last entry of the queue, not a piece of data, but a special flag, which means "help, I am blocked;" after which "A" blocks itself. By agreement between "B" and "A", when B discovers this flag at the end of his queue, he will wake up "A". We may redraw the flow diagrams of "A" and "B" as in figure 3.2 to take into account this possibility.

We may now ask, "Why use two processes instead of one?" One reply is that after the computation of a result and a decision to print it, we can logically do

1. computation for typewriter code conversion, and
2. waiting for the "type-out" instructions,

simultaneously with the computation of the next result. If we were satisfied to do everything serially, we could do so by using only one process. The serial approach would mean that the total job time would be longer, and that the typewriter would often pause while awaiting the next computation.

There are, of course, many other applications for a two-process system. In our example, we have the ability for the process computing results to get ahead of the slower typewriter. Another example would be typewriter input in which a typist can type requests to a computing process; by using two processes, the typist can type ahead in those cases where he has requested an exceptionally long computation. Such a facility is extremely important for smooth man/machine interaction since the human being is rarely matched perfectly with the computation rate of his program.

Channel Logic.

It will be profitable to introduce one further complication to our example, since we have begun a discussion of input/output. If we require that "A" do typewriter code conversion on his computed results and place in the work queue for "B" only numbers "ready to go", it is then possible to replace process "B" with a channel. A channel is really nothing more than a simple processor with a wired-in program such as the one that process "B" was following. It is activated by placing work in a queue and sending it a wakeup signal. The channel is capable of returning a wakeup signal when it is done, or when it runs into trouble, if appropriate entries are made in its queue. We will learn in the next section that one of the jobs of the traffic controller in the complete, multiplexed system is to translate processor interrupts originating from hardware devices attached to the system I/O channels into wakeups for the appropriate processes.

A Critical Race Between Processes.

Before going on to a more complex system, we may examine an especially awkward problem of multi-process traffic control which shows up even in our two-process, dedicated processor model. Looking again at the method of communication between "A" and "B", we find that process "A" goes through two steps:

1. Put task into work queue of process "B".
2. Wake up process "B".

Process "B", when it needs work, also goes through two steps:

1. Look in work queue, find it empty.
2. Call block.

Since process "A" and process "B" are running simultaneously, there is a distinct possibility that the order of events in time is the following:

1. "B" finds work queue empty.
2. "A" places task in queue.
3. "A" wakes up "B".
4. "B" calls block.

By our convention that wakeup does not affect a running process, process "A's" attempt to wake up "B" was ignored. Unfortunately, "B" has blocked itself and missed the signal intended to wake it up. This problem is similar in nature to the "critical race" of switching networks. Its resolution requires some help from the traffic controller; the primitive functions _block_ and _wakeup_ so far specified are not sufficient to resolve the problem. Our solution is to invent a "wakeup waiting" switch which is associated with process "B". Whenever process "A" calls wakeup, whether "B" is running or blocked, "B's" wakeup waiting switch is turned _on_. When "B" calls block, block returns immediately if the wakeup waiting switch is _on_. In addition, "B" is given access to its wakeup waiting switch to reset it.

With this addition to the machinery of the traffic controller, process "A" goes through the following two steps:

1. Put task into work queue of process "B".
2. Wakeup process "B", turning wakeup waiting switch _on_.

while process "B" does the following:

1. Reset the wakeup waiting switch to _off_.
2. Look in queue, find it empty.
3. Call block, which returns if wakeup waiting is _on_.

It does not matter now what the specific time relationships of "A" and "B" are, process "B" will never miss a wakeup signal.

The essential requirement of any solution to this race problem is some way for process "B" to check a variable accessible to both processes, then get blocked before the other process can possibly change the value of the variable. This requirement dooms all attempts of the processes to arrange interlocks themselves without the aid of a special feature in the Traffic Controller. (This last statement is not quite true; it is possible for process "A" to delay by doing busy work for a fixed length of time at an appropriate point so as to permit "B" to get blocked. This type of solution lacks generality, since any change in the characteristics of the system might require a new delay value to be inserted for every such timing dependent interlock.)

One of the reasons for choosing the particular pair of primitives block and wakeup is that all critical races which might occur in inter-process control communication are concentrated in a single race, which is resolved by a single simple mechanism.

Summary of Traffic Control Needs So Far.

Up to this point, since we have only considered dedicated processor systems, we have been able to restrict the traffic control needs of these systems to that of inter-process control communication. The problem of inter-process control communication is, briefly, that there must be a way for one process to signal another that there is work to do; there must also be a way for a process to wait for such a signal. We have proposed that the traffic controller provide three facilities for inter-process control communication:

1. Entry point block, to wait for a signal.
2. Entry point wakeup, to send a signal.
3. The wakeup waiting switch, to resolve signaling races.

In passing, we also introduced the concept of a process identification tag; this latter feature is needed in systems of more than two processes to identify the process being wakened in a call to wakeup.

SECTION THREE: PROCESSOR MULTIPLEXING

In this section we discard the assumption that a separate processor is dedicated to every process in the system. Instead, we assume that a limited number of processors must be shared (multiplexed) among the many processes. The job of the traffic controller is to make the multiplexed system look like a dedicated processor system by creating one pseudo-processor for each process. To this end, it will need to use two system facilities not directly available to a process: the ability to switch a processor from the address space of one process to that of another, and the ability to snatch a processor away from a running process and give it back later (the system interrupt.) Since there is competition for processors, the ideas of queuing, priority, and pre-emption must be introduced. Finally, an important task of the traffic controller in a multiplexed system is to provide an interface with input/output channel hardware to translate I/O signals into wakeups for the appropriate processes.

As we proceed, we will discover that the traffic controller logically splits apart into two major pieces: the "system interrupt interceptor," which provides the interface with the processor interrupt hardware, and the "process exchange," which actually performs processor multiplexing.

In the previous section, we made the observation that in a dedicated processor system inter-process communication could be implemented easily in hardware. When multiplexing is introduced, however, the desired flexibility of implementation--the scheduling algorithm, for example--precludes any realistic opportunity of avoiding a traffic controller based partly on hardware and partly on program. This is not to say that hardware implementation of the traffic controller described here is impossible, but rather inadvisable.

In this section, we continue to make the assumption that sufficient core memory is available for at least the current procedure of every process. In the next chapter we will explore the added complications when this assumption cannot be made.

System Interrupts.

A system interrupt is a hardware signal directed to a specific processor which causes that processor to interrupt its activities, store its state in an interrupt stack (similar to the call stack described in chapter one), and begin fetching instructions from a special subroutine corresponding to the interrupt. Connected with system interrupts are several features which are indispensable to operation of a multiplexed, multi-process system. We describe these features briefly here.

Associated with each processor are one or more _interrupt cells_. When a condition arises which requires that a processor be interrupted, one of these interrupt cells may be set _on_ by some active device--an I/O channel or a processor. Each interrupt cell has an established _priority_ relative to every other interrupt cell. It is possible for a processor to set one of its own interrupt cells _on_.

Whenever an interrupt cell is set _on_, an _interrupt signal_ is sent to the processor. The processor may choose to ignore temporarily, but remember, the interrupt signal by operating in _inhibited_ mode. Upon arrival of an interrupt signal, or if inhibited, when the processor leaves the inhibited mode, the interrupt signal causes the processor to reset the highest priority interrupt cell which is _on_, store its state, and jump to a special procedure corresponding to the interrupt cell which was reset.

In addition to the general inhibiting mode mentioned above, a processor may _mask_ individual interrupt cells to prevent generation of an interrupt signal when the masked interrupt cell is set _on_. If an interrupt cell which is _on_ is unmasked, it immediately generates an interrupt signal.

We will often refer to a hardware device known as an _interval timer_. This device has the following properties:

1. There is a separate interval timer for each processor.
2. The interval timer is a register which may be loaded and stored by the supervisor program.

3. The interval timer counts down on a regular basis, for example at a fixed time rate, or whenever the processor makes a memory access.
4. Whenever the interval timer counts to zero it sets a system interrupt cell belonging to its processor; it continues counting into the negative numbers.
5. It is possible to disable the timer interrupt mechanism so that no interrupt will occur if the timer register passes zero.

Processor Switching.

If there are many processes and few processors, it follows that there must be some mechanism by which processors are switched from one process to another. We ignore for the moment the problem of deciding that a particular switch should be made, and concentrate instead on the mechanics of switching. An implication of switching a processor from one process to another is that somehow the address space of the new process must be acquired by the processor. This acquisition requires saving and reloading a portion of the processor stateword by a special hardware instruction.

Recall from chapter one that the address space accessible to a processor is defined by the contents of a <u>descriptor segment base register</u> in the processor, which contains the absolute address of the base of a <u>descriptor segment</u>. The descriptor segment, in turn, contains the absolute address of the base of each segment accessible to the processor.

A processor must be very careful when acquiring a new address space to make sure that it does so in an orderly manner. Consider, for example, what happens if a processor is executing instructions from segment number 28, and is fetching an instruction from location 1172 in that segment. If the instruction orders the processor to reload its address space pointer (descriptor segment base register), the processor will do so, and then go on to fetch the instruction in location 1173, segment 28, in the <u>newly acquired</u> address space. Unless we have planned ahead, the instruction found there may not be the appropriate one at all. There are at least three alternative ways of planning ahead:

1. Build a special address-space-switching instruction into the processor. This instruction would deposit the complete processor stateword, including instruction location counter, and reload a complete new stateword, including an instruction location counter.
2. Simulate the address-space-switching instruction by program. This can be done by disabling the two-dimensional address space hardware (switching to "absolute" addressing mode) completely for long enough to reload the descriptor segment base register and transfer to the proper point in the new address space.
3. Arrange that the procedure containing the address space acquisition instruction be a common procedure appearing in every address space in the system, and that it have the same segment number in every address space.

The third solution is completely adequate as long as we assume that there is a one-to-one correspondence between address spaces and processes in the system. For simplicity, we adopt both this assumption and the third alternative. In every process, then, one segment number is reserved. It contains a procedure we will name "Swap-DBR", short for "swap descriptor segment base register." This procedure is called with one argument, the identification tag of the process to switch to:

```
call Swap-DBR(J);
```

Swap-DBR is a closed subroutine, called with a standard call stack as described in chapter one. Figure 3.3 illustrates the operation of this subroutine in the case where process "K" calls to switch control of the processor to process "J". The blocks representing steps of Swap-DBR are drawn twice for clarity, once in process "J" and once in process "K". It is understood, of course, that Swap-DBR is really a shared common procedure. The dotted line indicates the path of the processor; at the step "LDBR" (Load Descriptor Segment Base Register) the processor jumps to the address space of "J".

Note in this figure that since Swap-DBR obtains its return location from the top of the call stack in process "J", it returns to the location at which "J" called Swap-DBR. This location clearly does not have to be the same as the location from which "K" called Swap-DBR. If any other process ever subsequently calls Swap-DBR(K), control will reappear in process "K" immediately following the LDBR instruction in the address space of "K". It is clear then, that Swap-DBR, now operating for process "K", and using "K's" call stack, will return to the point in process "K" from which Swap-DBR was originally called.

One further question needs to be answered before ending our consideration of process switching: how does Swap-DBR know how to acquire the address space of process "J"? There must be a table relating process identification tags and descriptor segment base register values. This table, a segment common to all processes, is known as the process table. It contains an entry for every process in the system. The process identification tag is the index key to this table. As we explore the implications of processor multiplexing we will find that this table is the primary data base of the traffic controller, and that it contains considerably more information about a process than just the location of its descriptor segment. With one item stored here we are already familiar: the wakeup waiting switch, which was described in section 2 of this chapter.

Processor Dispatching.

If there are many processes and few processors, it follows that not all unblocked processes can really be running. We must expand our notion of execution states of a process to include a third possibility: a "ready" process. Thus a process may be in one of three execution states:

1. Running. A running process is executing on some processor.
2. Ready. A ready process would be executing if only a processor were available.

3. Blocked. A blocked process has no use for a processor at the moment; it is waiting for a wakeup signal.

From the notion of a ready process, we may invent immediately the "ready list", a list of all ready processes. This list is the basis for dispatching a processor when it is released by another process. We can now begin to perceive the dim outlines of an implementation of processor multiplexing within the framework of "block" and "wakeup". A call to the traffic controller to wake up a blocked process means "put it on the ready list." When a process calls to block itself it means "I am temporarily abandoning the processor. Switch it to some process on the ready list."

If we place in the process table entry for each process a three-way switch to indicate the execution state of the process (the running/ ready/ blocked switch) we may then draw a flow diagram of the closed subroutines Block and Wakeup as in figure 3.4. In the implementation of Block, once a decision has been made (in the example, by process "K") as to what process should be run next (process "J"), "K" calls Swap-DBR to execute the switch. Recall that control vanishes from process "K" somewhere in the middle of Swap-DBR. Not until

1. some other process puts "K" back in the ready list (by calling Wakeup(K)), and
2. some other process, calling Block, switches control to "K"

will Block, in process "K", receive a return from Swap-DBR. At that time, Block returns control to the program in process "K" which originally called it. Thus every process in the system moves back and forth among the running, ready, and blocked states, as indicated in figure 3.5.

One further comment may be made about the organization of the procedure named "Block." The last three steps of the flow diagram (in which we locate a ready process, change its execution state to running, and switch to it by calling Swap-DBR) can be collected together into a closed subroutine, named "Getwork". Although at this point such modularity simply places these three steps under a mnemonic name, we will find in the next section that the subroutine named Getwork is again needed to implement scheduled processor pre-emption. We emphasize again that Block, Getwork, and Swap-DBR are closed subroutines using the standard call stack.

Processor Scheduling.

Within the framework of processor multiplexing so far described, we can begin to look at the problem of processor scheduling; that is, deciding which processes should be allowed to run at any given time. We are not interested here in deciding particular questions of scheduling policy, but rather in providing a framework with as much flexibility as possible in which to establish scheduling policies.

There are clearly two opportunities to make scheduling decisions in the administration of the ready list: when a process is placed in the ready list, and when the time comes to take one out. Arranging an appropriate scheduling mechanism must be a compromise between the interests of flexibility of scheduling policy and efficiency of administration. An organization in which a processor spends 40% of its computing capacity deciding what to do next is undesirable. We may thus dismiss immediately any schemes which require any significant computation which is proportional to the number of processes on the ready list. This requirement tends to exclude computation at the time of choosing a process to run. Instead we will consider only techniques of scheduling in which the priority of a process is established at the time it is placed in the ready list. We will maintain the ready list "in order" so that the dispatcher (Getwork) need merely choose the process at the head of the ready list.

There are still at least three alternative forms of scheduling, and our present objective is to determine what framework is appropriate to allow all of these alternatives. The simplest scheduling technique is first-come, first-served; it is implemented by placing new processes at the end of the ready list, in order of arrival. If in figure 3.4 we change the step "Put J in ready list" to "put J at end of ready list" we have established this technique. The next more elaborate scheduling technique is "fixed priority." Here, every process in the system has a fixed priority number relative to every other process. (The priority label can be stored in the process table.) Procedure Wakeup places a process in the ready list in order according to the value of its priority label. Neither of these two alternatives makes any great demand on the organization of the traffic controller.

The third scheduling technique, however, does. We may name this technique the "computed priority" technique. Here, whenever a process is placed on the ready list, its priority is computed according to some algorithm which may, for example, take into account the amount of system resource that the process has already used, or the length of time other processes have been waiting in the ready list. In this case the process is placed in the ready list in order according to the value of its computed priority label. We may now make the observation that each process should schedule itself, since each process knows factors which influence its needs for a processor. Strictly speaking, it is impossible for a process to schedule itself, since it cannot be allowed to execute until it has been scheduled. We can, however, again take advantage of our assumption of a one-to-one correspondence between address spaces and processes. We do so by making the convention that the priority computation algorithm for process "A" is a closed subroutine named "Schedule" within the address space of process "A". Process "B" may then wake up "A" by switching temporarily to the address space of "A", calling the procedure named Schedule (which will put "A" in the ready list at an appropriate point) and then switching back to the address space of "B".

The intent to allow a process to provide its own scheduling algorithm should not be construed to mean that the user of the system is to dictate his own scheduling policy; such an arrangement would surely be putting the rabbit in charge of the lettuce. We intend instead to obtain two important features:

1. The data base on which scheduling decisions for a process are made can be private to the process; the alternative would be system-wide accessibility to every scrap of information which might possibly be needed to make the scheduling decision. With a private scheduler it becomes that much easier to modify the scheduling algorithm to use an additional piece of data without relocating the data to a system-wide table.
2. It becomes possible for two different processes to use totally different scheduling algorithms. While both scheduling policies are certainly provided by the system, an administrative authority to use a non-standard scheduler is a very flexible and powerful tool to obtain easily a special grade of service. A process tending a real-time experiment, for example, might be allowed to use a scheduler which always puts it at the head of the ready list. Conversely, a process which is purchasing an extremely low grade of computer service (presumably at a less expensive rate) might be assigned a scheduler which habitually places him at the end of the ready list. It is also possible to try an experimental scheduling policy on one or a few processes without forcing this policy on all users of the system.

By analogy, we might consider the "traffic control" techniques used in regulating automobile traffic on an expressway. One could attempt to impose a "master controller" which keeps track of the position of every automobile and orders lane changes in an optimum fashion. It is much more practical instead to allow each driver to make such decisions on the basis of some standard laws, the position of his automobile relative to a few others nearby, and his own desires as to speed and which exit to take. In this analogy we see both the "limited overhead" aspect of each decision, and also the ability for different drivers to use different policies, as long as they fit into the same general legal framework. By requiring that even non-standard schedulers be provided by the system the chances of extending the analogy to an irresponsible driver who ignores the legal framework are minimized.

We may implement our rule that "each process schedules itself" by complicating the Wakeup procedure slightly. In place of the step (in figure 3.4) containing "Put J in ready list" we substitute "call Ready-Him(J)". "Ready-Him" is a second entry into the segment containing procedure Swap-DBR. (Remember that this segment takes care of all address space switching, and is therefore in a fixed location in every process.) Ready-Him goes through four steps (assume that "J" is waking up "K"):

1. Load descriptor segment base register to switch to the address space of "K".
2. Call procedure Schedule in the address space of "K".
3. Load descriptor segment base register to return to the address space of "J".
4. Return to caller (Wakeup) in "J".

Process "K" has been successfully scheduled; process "J" may now go about its business.

At the risk of disclosing the existence of modules not yet discussed, figure 3.6 is a block diagram of the complete process exchange. In this figure, the solid arrows represent closed subroutine calls; dotted lines are data paths. The two modules named "Restart" and "Quit" will be explained shortly.

<u>Pre-emption</u> <u>Scheduling</u>.

Our traffic control framework is now almost complete. The one major piece of machinery left to install comes from the answer to two closely related questions:

1. What if when a process begins to run, it merely runs and runs, without ever calling Block? If this happens one processor does not take part in the multiplexing.
2. What if a process is added to the ready list by a scheduler which thinks that this process is far more important than any process presently running? Must the important process wait until some other process decides to call Block?

A first reaction to these question might be, "so what?" The real problem these questions raise, however, is significant: how to guarantee adequate response time to requests for processor time. The priority scheduling mechanism so far described is not sufficient to provide guaranteed response in an environment where running time of a job cannot be predicted. In addition to priority scheduling, a mechanism must be available to return a <u>running</u> process to the <u>ready</u> state; a processor must be pre-empted. The mechanism used is, of course, the system interrupt.

For traffic control purposes we will reserve three interrupt cells from each processor and name them <u>internal</u> interrupts to distinguish them from interrupt cells set by input/output devices, the <u>external</u> interrupts. For reasons we will see below, the internal interrupts are given lowest priority. Since the traffic controller may decide to trigger an internal interrupt, a processor always masks internal interrupts whenever it enters the traffic controller. (The reason for this masking will become clear later.) The three interrupt cells are used for:

1. Processor interval timer runout.
2. Pre-emption by the scheduler.
3. Quit--one process turns off another.

The problem of a long-running process is solved, then, by having the processor interval timer trigger an interrupt cell if a process runs "too long." What constitutes "too long" is left up to the scheduler of that process. We extend the ready list to contain pairs of entries: a process to run, and a running time limit. When the dispatcher (Getwork) chooses a process from the head of the ready list, it loads the processor interval timer with the specified time limit. When the interval timer runs out, it sets the internal interrupt cell reserved for it, thereby interrupting the processor.

If the timer runout interrupt should occur, the processor will suddenly find itself in the system interrupt interceptor, a traffic control module. This is an indication that the process should return to the ready state. We therefore provide a procedure named "Restart" which is to be called by the system interrupt interceptor whenever a timer runout interrupt occurs. Restart merely

1. Calls the scheduler to put this process back on the ready list.
2. Calls Getwork to turn the processor over to the highest priority process available.

The other problem, that of a scheduler convinced that the process it has just placed at the top of the ready list is more important than any presently running process, is solved with the same mechanism: force the process to call Restart, by causing an interrupt. A scheduler, then, when it schedules a very high-priority process may wish to inspect even the list of running processes, and set the pre-emption interrupt cell of one of the processors. Since we have been careful to arrange that:

1. All processors are equivalent ("anonymous"), and
2. the particular processor executing the scheduler is masked for pre-emption as long as it is in the traffic controller,

the scheduler need merely pick the lowest priority process from the list of processes currently running. It is entirely possible that the scheduler will choose the processor on which it is now executing. If so, the instant that this processor exits from the traffic controller it will be interrupted; the system interrupt interceptor will call Restart, and the high-priority process will be on its way. If the scheduler chooses a different processor, some other process will meet the same fate.

It has not been our intent here to become involved in issues of scheduling policy, but rather to provide a framework within which many policies can be implemented. Among the scheduling policies which may be easily incorporated within this framework are a simple round robin or a multi-level priority queue based on processor or total resource usage. Any priority-computation algorithm which calculates a fixed queue number or priority label can be used.

The Quit Module.

In a practical system it often turns out that a process gets into a loop, begins producing large quantities of unneeded output, or uses up more resources than its owner has agreed to pay for. In any of these cases, it is necessary to "turn off" the process. The Quit module is provided for this purpose; it can force a running or ready process into the blocked state. If one process makes the observation (we do not ask how) that another process should be turned off, it may

call Quit(K)

to shut off process "K". The procedure Quit follows is straightforward:

1. If the process in question is already blocked, nothing need be done. Quit returns to its caller.
2. If the process in question is running, the Quit module resets the Wakeup Waiting switch for the process, and generates a system interrupt, the "Quit interrupt," for the appropriate processor. The meaning of the interrupt is that the process should call Block; in the description

of the system interrupt interceptor below, we see exactly how this call comes about.

3. If the process being blocked is ready, it is merely removed from the ready list and its Active Process Table entry modified to show that it is blocked.

If a later change of heart occurs, the process which has been quit can be restarted by calling Wakeup for it.

One of the responsibilities of the resource management procedures of the operating system is to insure that a typewriter user, for example, can always get a signal through to some process requesting that a looping process be "quit".

Although the Quit module does not call other process exchange modules, it should be considered part of the process exchange because its activities must be coordinated and interlocked with other process exchange modules.

Review

In section three we introduced a wide variety of ideas and it may be useful to review them briefly here before proceeding.

The problems of processor multiplexing are technological. That is, they stem from the fact that we wish to share resources for economy. A solution to the problem of processor sharing must be able to:

1. Dispatch processors to waiting ("ready") processes when a process blocks itself.
2. Provide a technique of priority scheduling of processors among processes whose running times are unknown a priori, with pre-emption to help guarantee response time.

3. Perform the mechanics of switching from one process to another in such a way that each process may have a different operating system, including a private scheduler, if desired.
4. Control the overhead cost of multiplexing so that it does not grow combinatorially with the complexity of the system.

In order to achieve multiplexing of a few processors among many processes, it is necessary to use two special hardware devices, the system interrupt and the ability to switch a processor from one address space to another. Processor switching is accomplished by a module named Swap-DBR, which must appear in the same segment in all address spaces. When a process calls Swap-DBR(K), the processor disappears from this process, to reappear in module Swap-DBR in the address space of process K. The processor then returns to the last place in process K which called Swap-DBR. The process table contains a list of process tags and descriptor segment base register values for Swap-DBR.

A scheduler maintains a ready list, an ordered list of processes ready to execute, each with a time limit. A process leaves the running state by calling the module Getwork, which locates the highest priority process in the ready list and calls Swap-DBR. There are two reasons why a process may leave the running state: it may desire to block itself, or it may be pre-empted by another process. In the first case the process calls the Block module voluntarily. In the second, a system interrupt forces it to call the Restart module.

Scheduling of a process (placing it in the ready list) is accomplished by a module named "schedule" in the address space of the process. If a process wishes to wake another up, it calls the module named Wakeup which, by calling the subroutine Ready-Him, switches to the address space of the awakening process, calls schedule, and switches back to the address space of the caller This technique allows each process to be scheduled by a scheduler which makes a limited overhead decision which may be based in part on factors known only to the process being scheduled. One option available to a scheduler is to pre-empt a running process by triggering a pre-emption interrupt in the appropriate processor.

Finally, an entry named "Quit" is provided to allow one process to turn off another which has gotten into a loop or is otherwise performing a valueless service.

All of these modules are closed subroutines which are called, and call on one another, using a standard calling stack. These modules together form the process exchange. We next wish to examine that part of the traffic controller which interfaces with the system interrupt hardware, the system interrupt interceptor.

SECTION FOUR: THE SYSTEM INTERRUPT INTERCEPTOR

The traffic controller becomes involved in interrupt handling for two reasons:

1. The system interrupt is the tool by which pre-emption decisions of the scheduler may be enforced.
2. In order to allow a process to communicate with an input/output channel, the traffic controller must intercept signals coming from the I/O channel and direct them to the correct process.

In the discussion of the process exchange, we saw that three kinds of system interrupts were generated by the process exchange itself:

1. Timer runout interrupt (ordered by scheduler).
2. Pre-emption interrupt (ordered by scheduler).
3. Quit interrupt (signal from another process).

These three system interrupts are <u>internal</u> interrupts. In addition to the three internal system interrupts, the hardware input and output devices of the system generate a large variety of interrupts to indicate completion, progress, or trouble. These are <u>external</u> interrupts. An important distinction between the interrupts generated by the process exchange and the externally generated interrupts is that the former are directed to the process running at the time while the latter are usually of interest to some other process, the one which initiated the I/O action, for example.

Since the external interrupts generally "belong" to some other process of unknown and possibly higher priority, they are given priority over the internal system interrupts. All system interrupts are given priority over scheduled work. By appropriate arrangement of procedure, the effect of this arbitrary priority assignment on scheduling and response times can be minimized. The procedure executed at the instant of the external system interrupt is only enough to determine which process has the real interest in this interrupt, and to reflect a wakeup signal to that process.

Flow within the system interrupt interceptor.

The system interrupt interceptor is automatically entered by all system interrupts. After passing through a short piece of code which saves the processor state in an interrupt stack, the interrupt interceptor masks further interrupts of equal or lower priority. Up to this point, execution has been in inhibited mode. Once the processor state is saved and the mask is set, the unmasked interrupts may be permitted; the processor leaves inhibited mode.

The system interrupt interceptor now calls an appropriate procedure, known as an interrupt handler. Upon return from the interrupt handler, the processor state is restored (including the previous state of the interrupt mask) and control returned to the point at which the interrupt happened. Figure 3.7 is a flow diagram of the system interrupt interceptor.

The interrupt handlers are brief and they must be carefully coded with certain restrictions. They cannot contain programmed faults, including page-not-in-core faults, or depend on further interrupts (for example by calling Block.)

It is important to realize that the system interrupt interceptor is executed as a part of the process which happens to be running on the processor at the time of the system interrupt.

A system interrupt interrupts the processor, not the process; the internal system interrupts do, however, imply that the traffic controller should change the execution state of the process. The distinction between a processor interrupt and the change of execution state of a process cannot be over-emphasized. We may note, however, that the processor time used to service an external system interrupt can be metered, and charged to the process responsible rather than the one which answered the interrupt.

The handler invoked for an external interrupt is in principle fairly simple: decode the meaning of the external interrupt and call wakeup for the appropriate process or processes. (It is entirely possible that a single system interrupt may represent an event of interest to several processes.) Since the external interrupt decoding procedure is executed as a part of the interrupted process, all procedures and data necessary to decode the meaning of the interrupt must be available to all processes.

Procedures for internal interrupts.

The internal interrupt handlers are more complex since internal interrupts imply that some special action by the Traffic Controller itself is needed. The internal interrupts are given the following priority:

1. Pre-emption interrupt.
2. Time-out interrupt.
3. Quit interrupt.

The pre-emption and time-out interrupts are handled identically. The handler for the pre-emption interrupt calls the Restart entry of the process exchange. The Restart entry changes the state of the process to ready, reschedules the process, and calls Getwork to give the processor to the highest priority process available. Sometime later, the process which called Restart will come to the top of the ready list, and obtain a processor. At that time, Restart will return to the system interrupt interceptor. The system interrupt interceptor then returns directly to the point of the pre-emption interrupt.

The lowest priority system interrupt is the quit interrupt. This interrupt means that the process now running should immediately revert to the blocked state. The procedure for this interrupt is straightforward:

1. Unmask the processor.
2. Call entry point Block in the process exchange.

The process which has just blocked itself is now at the mercy of the originator of the quit interrupt. A wakeup signal from another process will cause a return from Block to the quit interrupt handler in the system interrupt interceptor. This return is interpreted to mean that the process should be allowed to continue, so the system interrupt interceptor returns directly to the point of the quit interrupt.

---

(1) Dennis and Van Horn [11] have used the words "locus of control within an instruction sequence," to describe a process; the alternative term "thread" (suggested by V. Vyssotsky) is suggestive of the abstract concept embodied in the term "process."

CHAPTER FOUR

Traffic Control with Limited Core Memory

The traffic controller design of chapter three is based on one very important premise: that sufficient core memory is available for the procedures and data bases required to do processor multiplexing. The procedures of the traffic controller constitute a trivial issue since they are shared among all or nearly all processes. The data bases are another matter, however, since they tend to be repeated once for every process, or are proportional in size to the number of processes. For every process with its own private address space there must be a descriptor segment and data about the process. This data base consists of the process stateword, the call stack, and the wakeup waiting switch; a centrally located table of descriptor segment pointers is also necessary.

In this chapter our objective is to find out what special efforts are required to minimize the quantity of such data which must be kept in core memory at all times. We will see that with appropriate and carefully designed intercommunication between the traffic controller and the basic file system, it is possible to reduce the core requirements of most processes to zero when they are blocked, thus opening the way to an almost unlimited number of processes "in the system." Similarly, we will find that the core memory requirements of even a ready process consist of only a few table entries.

The reasons for such an objective are two-fold. First, we wish to be able to balance the capacity of the processors with as small an amount of core memory as possible. Secondly, we would like to be able to have an arbitrarily large number of processes "in the system", but not presently demanding processor time, without increasing the core memory size needed to balance the processor capacity.

Since core memory multiplexing is within the province of the basic file system, it may be profitable to review briefly the techniques used in core memory management. The reader is advised to reread the section on core memory management in chapter two if he is not thoroughly familiar with the basic file system. In that section, recall, we noted that the basic file system operates with one system-wide table, the active segment table, and one table per process, the segment name table. The active segment table contains pointers to the secondary storage location of each segment for which a missing-page fault might occur (each "active" segment); it is the primary source of data when a missing-page fault occurs. Since as mentioned in chapter two the organization of the basic file system precludes recursive missing-page faults, the active segment table is a segment which is "wired-down". That is, it must not take part in core multiplexing; it remains in core at all times.

The segment name table belonging to a process contains a list of all segments belonging to this process; it provides a correspondence between the segment number used to address the segment and the tree name of the file in secondary storage. The segment name table does not need to be wired-down, but it must be active, since a missing-segment fault encountered when looking in the segment name table would be impossible to handle.

Core Memory Needed by a Running Process.

With this background, we can now outline the core memory requirements of a running process. (For reference, table I is a catalog of the data bases mentioned here.) First, the process must be capable of handling a missing-page fault in order to retrieve pieces not in core; this implies that the part of its descriptor segment describing the page control and traffic control modules of the file system is in core, as well as the procedures of page and traffic control. Since page control refers to the active segment table to find pointers to pages on secondary storage, that table must be in core. Since page control wishes to block the process while awaiting the arrival of the page from secondary, it must be possible for it to call Block in the traffic controller without getting a missing-page fault. Since Block calls Getwork, which looks at the ready list, the ready list (and process table entries for processes appearing in the ready list) must be in core memory.

Before giving the processor away to another process, Swap-DBR must save the process stateword including the status of the call stack which represents the trail by which the process got to Swap-DBR. This information is needed so that when this process regains a processor it can return to its work. This information could be saved in the process table entry for this process. We will find, however, that once the process leaves the running state, its stateword does not need to stay in core memory. We therefore make up a special segment private to the process, the process data segment. Whenever a process is running, its process data segment must be in core memory so that the process can deposit its stateword and leave running status if necessary. When executing in the hardware management procedures, the process data segment is used as the calling stack.

It is also necessary that a running process be able to handle a missing-segment fault. Since segment control must look at the process' segment name table to interpret the meaning of the missing-segment fault, it must be able to access the segment name table without getting another missing-segment fault. It follows that the segment name table of a process must be "active" whenever the process is running. By definition, "active" means that there is an entry in the active segment table and a page table in wired down core.

Summarizing, then, a running process must have the following information in core:

1. Certain pages of its descriptor segment, containing descriptors of traffic control and page control.
2. A page table for the segment name table.
3. A process data segment, to receive the process stateword.
4. Entries in the active segment table for the segment name table and the process data segment.

5. Process table entry for itself.

In addition, we found that the ready list and process table entries for processes on the ready list must also be in core, so that the running process can leave the running state if it has to, following a missing-page fault.

When a process has all five items mentioned above in core memory, we say it is loaded; we may summarize the core requirements of a running process by saying that a running process must be loaded.

Figure 4.1 is a schematic illustration of a loaded process. This figure is simplified for intelligibility; for example, the traffic controller and basic file system would probably consist of several segments instead of one each as shown. Page tables are not shown; it is assumed that all segments are paged, however. Abbreviations are explained on the diagram. The arrows leading out of tables represent table entries containing pointers to the objects indicated. The descriptor segment is divided into universal descriptors shared by all processes, and private descriptors belonging to one or a few processors.

Core Memory Needed by a Ready Process.

The core memory requirements of a ready process are determined by our ability to change its status to running and switch to it, when it comes to the top of the ready list. If the process happens to be loaded, there is certainly no problem in switching to it. We can, however, switch to a process which is missing part of the list of loaded items, if we can arrange to recreate them at the instant of the switch or to have the process retrieve them itself.

Let us examine each of the pieces of information making up a loaded process to see which, if any, a ready process can get along without. Consider first the descriptor segment. The segment descriptor words appearing in the descriptor segment of any process may be divided into three classes, namely

1. Descriptors pointing to traffic control and basic file system segments which are shared by all processes in the system.
2. Descriptors pointing to segments known to this particular process by virtue of their appearance in the process' segment name table.
3. Descriptors containing missing-segment bits. Unused segment numbers and segments which have not been used for a long time will have a missing-segment bit.

We may observe first that any descriptor of the second category can by replaced by one containing a missing-segment bit; if the missing-segment bit later causes a fault, segment control can rebuild the descriptor by reference to the segment name table. We may observe next that all descriptor segments will have an identical set of descriptors of the first category. From this line of reasoning, it follows that we may discard the descriptor segment of any process which is not running, and recreate its descriptor segment whenever it becomes necessary to change it to running state. The descriptor segment may by recreated (in Swap-DBR) by creating an empty, wired-down segment from a pool of free core, and copying into it the contents of a descriptor segment "template". This template, built up at the time the system is initialized, is a data segment accessible to all processes in the system. It contains all the descriptors of the first category, above, plus missing-segment faults for all other descriptors.

We have complicated slightly the subroutine Swap-DBR, which must now check to see if the process it is switching to is loaded and, if not, establish a new descriptor segment for the process. We place in the process table a flag which, if on means that the process is not loaded. Figure 4.2 is a flow diagram of Swap-DBR including the added complication of checking the loaded flag.

Consider next the segment name table of a ready process. We must preserve the ability for the process to take missing-segment faults when we switch to it, so it can fill in the rest of its descriptor segment by itself. When a missing-segment fault occurs, segment control looks into the segment name table to look up the file name corresponding to the segment number which caused the fault. For this reference to the segment name table to succeed, there must be a page table in core for the segment name table. (Page control can retrieve the individual pages of the segment name table on missing-page faults as long as the secondary storage location of the segment name table appears in the wired-down active segment table.) However, we can again arrange to discard this page table when the process is not actually running by the following strategy: After switching to the process, but before any missing-segment faults can occur, Swap-DBR can call segment control directly (at a special entry point) and ask it to make up a page table for the segment name table. This is possible only if the location of the segment name table on secondary storage is still in core in the active segment table so that segment control does not have to search for it in the directory hierarchy (using missing-segment faults).

Again, we have complicated Swap-DBR in order to cut down on the minimum core requirements of a ready process. After switching to the process, Swap-DBR must check to see if it is loaded and, if not, ask segment control to create a page table for ("activate") its segment name table. Figure 4.3 is a flow diagram of Swap-DBR showing this latest addition. As an indication of things to come, we have drawn the block containing "call segment control to activate the segment name table" outside Swap-DBR in a special module named the process bootstrap module.

Consider finally the process data segment. This segment is needed for two reasons: as a call stack when executing in the traffic controller, and to provide a place for Swap-DBR to leave the stateword when the process leaves running state. Again, there is a strategy by which we can remove this data base from core when the process is not running. The strategy works as follows: suppose we try to switch control to a process which does not have its process data segment in core. We do so planning that the process bootstrap module should retrieve the information itself by means of missing-segment and missing-page faults after activating the segment name table. This would work except for one detail: when the bootstrap module gets a missing-page fault, page control will want to block the process until the page arrives from secondary storage. Blocking the process requires depositing its stateword in the process data segment, which is the very thing being retrieved. We can resolve this problem by having Swap-DBR create an empty "interim" process data segment before switching to an unloaded process. The interim process data segment is used by the process bootstrap module as a call stack, and by Swap-DBR to store the process state whenever the process leaves running status while trying to retrieve its real process data segment. Figure 4.3 is a complete diagram of Swap-DBR.

In review, then, a ready process must have the following personal information in core so that it can bootstrap itself back to the loaded state:

1. An entry in the active segment table for its segment name table.
2. An entry in the process table for the process.

When a process has these two table entries in core, we say it is an <u>active</u> process. We can summarize the core requirements of a ready process by saying that a ready process must be active, but it does not have to be loaded.

<u>When Blocked Processes Must be Active</u>.

A line of reasoning essentially the same as that above leads us to the conclusion that a blocked process must also be active in order to accept a wakeup signal. The information in core for an active process is the minimum quantity of information which must remain in core such that the process can survive by itself and bootstrap itself in. If the information in the process table and active segment table is removed from core, it is inevitable that some other process will have to provide substantial aid in getting this process back in operation. Reconstructing either of these table entries requires retrieval of information from the secondary storage hierarchy, by means of missing-segment and missing-page faults. As shown in the previous section, when switching to a ready process it is not in general possible for the preceding user of a processor to provide this help, so we require that all ready processes be active.

The situation for a blocked process is quite different. In most cases, there is no reason why a process calling the wakeup entry cannot first check to see if the process being awakened is active and, if not, retrieve the information necessary to activate it. The cases in which the calling process cannot provide this aid, in fact, are only two:

1. The calling process is operating in the basic file system, has discovered a previously requested page is now in core, and is attempting to wake up the blocked process to inform him of this fact.
2. The calling process is operating in the system interrupt interceptor, and is attempting to wake up the other process because a system interrupt has arrived.

Both of these cases represent situations in which the basic file system cannot tolerate a missing-page fault, the first because a missing-page fault has already occurred for the process doing the call, and the second because the interrupt may have occurred while handling a missing-page fault. (Recall our restriction of no recursion on missing-page faults.)

We therefore make a note in the process table if a process is expecting a wakeup from either of these two sources, and agree to leave such processes active. We may, however, safely deactivate any blocked process which is not waiting for such a wakeup. When a process is deactivated, there is no longer any information whatsoever in wired-down core memory pertaining to the process; the number of deactivated processes in the system is limited only by the amount of secondary storage the system is willing to devote to the tables needed to remember them.

As a summary of the various state transitions which a process can undergo, we repeat the state diagram of figure 3.5 taking cognizance of unloading and deactivation in figure 4.5.

Core Memory Management with Processor Multiplexing.

Processor multiplexing adds a new dimension to the problem of multiplexing core memory usage. In addition to the ability to write out little-used pages it is now possible both to unload and deactivate processes to free up core space. We will also discover that it is possible to postpone the loading of a process if there is insufficient free core available.

The first step in managing the core requirements of the traffic controller is to divide the process table into two tables: the active process table and the known process table. The active process table is wired-down to core and contains an entry for every active process. All other processes appear in the known process table, which may be paged out on to secondary storage. To activate an inactive process prior to waking it up, one must first locate the information describing the process in the known process table and copy it into a vacant slot in the active process table. This operation, of course, will generally result in one missing-segment fault for the known process table and any number of missing-page faults. The next step in activating the inactive process is to call the basic file system and ask it to find the segment name table of the process being activated. Locating in secondary storage the segment name table will again, in general, require missing-segment and missing-page faults.

Management of a wired-down table requires some guarantee that the table will not grow without bound. In the case of the active process table, we may control its size by assigning an upper limit to the number of allowed active processes. Once this number is reached, if someone wishes to activate a process he must first look through the active process table for a blocked process to deactivate. The activation and deactivation is, of course, part of the responsibility of the traffic controller and is actually carried out by the Wakeup module when it discovers that it has been asked to wake up an inactive process. The decision as to which blocked process to deactivate is in the province of a policy module called by Wakeup. The active segment table is managed in a similar fashion.

Unloading of a process is carried out by the core control module of the basic file system when it needs space in a fashion similar to unloading a single page. The decision to unload a process rather than a page, and which process to unload, is again handled by a policy module called by core control.

Reloading an unloaded process, recall, is done by Swap-DBR when the process comes to the top of the ready list. This action poses a most interesting problem. Suppose that at the top of the ready list are a large number of unloaded processes. If left to its own devices, the traffic controller will begin loading the first one, only to have it immediately block itself waiting for its process data segment to come in. The traffic controller will then assign the again free processor to the next process, and begin it loading also, and the next, and the next. In the fashion of a sorcerer's apprentice, traffic control will rapidly fill up all of available core memory with processes trying to load themselves.

We can forestall such a circumstance by realizing that the decision to reload a process is really a commitment of a considerable amount of core memory resource, both for the special segments needed to load it, and also for private segments which the process will begin to use as soon as it is loaded. We therefore give to core control, the submodule of page control which is in charge of core resource commitments, the privilege of responding "no" when Swap-DBR requests core memory space to build a descriptor segment in preparation for loading a process. Core control can make a decision to give a "no" response on the basis of the amount of available core and the number of processes already loaded.

What should Swap-DBR do if it receives a "no" response from core control? The meaning of this response is that core is too full to reasonably commit resources to load another process. Swap-DBR therefore gives an error return to its caller, Getwork. Getwork, upon receiving this error return, can then run down the ready list looking for a loaded process to run instead. If no loaded process is found, we have a situation similar to that when the ready list is empty: there is no useful work for the processor to do at the moment. As we will see in chapter five this condition may indicate that the resources of the system are not well matched to the load.

The simplest way to handle this condition is to place the processor in some sort of a busy loop, for example searching the ready list over and over for a loaded process. Eventually some other processor, or this one upon taking a system interrupt, will add a loaded process to the ready list. This last strategy depends upon the fact, not previously mentioned, that the basic file system uses a daemon process to manage the secondary storage devices (the "drum daemon") and it is careful never to unload this daemon process. If core is overcrowded to the point that core control refuses the loading of any new processes, it will have already begun output to free up some core memory. Therefore, whenever a processor goes into a busy loop because it cannot load a ready process, it is guaranteed that there will soon be an output completion interrupt from the secondary storage device asking to wake up the drum daemon.

Summary

Again, we have introduced a host of ideas and would do well to pause and summarize them. The fundamental problem addressed in this chapter has been to discover techniques by which processor traffic control can be accomplished without usurping all of core memory for the traffic controller and its data bases. The procedures of the traffic controller are fixed in size and shared among all processes; the real concern is with the data bases. We identify two kinds of data bases which cause difficulty: those private tables which are repeated once for every process in the system, and shared tables which contain entries for every process.

For a process to be running, it must be loaded. A loaded process has a descriptor segment in core, and other private tables necessary to allow it to take missing-segment faults. It also appears in wired-down tables so that it may properly handle missing-page faults.

A ready process does not need to be loaded, but merely active, meaning that it still has in core the wired-down table entries permitting it to take page faults. In this way, the process can be switched to running status in short order when it comes to the top of the ready list; it can then retrieve on its own the other tables it needs to operate outside the traffic controller. In order to switch control to an unloaded process, Swap-DBR must first create empty descriptor and process data segments from a pool of free core memory and copy a descriptor segment template into the descriptor segment. The template contains descriptors for the traffic controller, basic file system, and the process bootstrap module, which bootstraps the rest of the process back into core memory.

Blocked processes do not even need to be active unless they are to be reawakened by arrival of a system interrupt or by the basic file system. An inactive process requires no information whatsoever to be stored in wired-down core tables, but it requires some effort and missing-page faults by another process to reactivate it.

As demand for core fluctuates, policy algorithms determine which pages to write out, and processes to unload or reactivate. One important policy algorithm refuses to allow too many processes to be loaded at once; when the traffic controller is told by the basic file system that it cannot run an unloaded process, it instead searches for a loaded one to run. It is possible that a processor may not be able to find any useful work to do even though the ready list is not empty, since beginning to work on an unloaded process would merely tend to overload the core memory resources.

With this examination of the interaction between core memory and processor multiplexing, our detailed design of the traffic controller is complete. As a final step, we next wish to consider some properties of this particular traffic control design: how it reacts to a mismatched load, and how it scales in size.

CHAPTER FIVE

System Balance and Scaling

In the previous chapters we have been concerned exclusively with the organization of a computer utility, stressing the detailed requirements of processor multiplexing. In this chapter we stand back from the resulting design and look at it from two related aspects: system balance and system scaling. We use a limited definition of system balance, namely the relation between processor capacity and memory size for a given presented load. System scaling is the ability of the design to scale in capacity up or down over a wide range of presented loads.

In doing so, we purposely raise many more questions than we attempt to answer. One intent, in addition to indicating implications of the design, is to indicate interesting areas of exploration for the future.

System Balance.

One of the first problems confronting the administrator of a computer system is to find out how well his system is working. In this section we propose some simple tests which help answer whether or not processor and core memory are being used effectively. Our comments only scratch the surface of a difficult problem, and completely ignore many important aspects of system balance such as whether or not the input/output channel capacity of the system is appropriate. A very simple model of the system consisting only of processors, core memory, and a "large enough" secondary storage and I/O capacity is the basis for the following comments. It is assumed that the load presented to the system is somehow homogenous and non-varying.

The particular balance problem we are concerned with here is to determine whether or not a system's core memory and processors are balanced relative to each other, and to the presented load. In addition to measurements of the characteristics of the presented load and of the operation of the system, we can perform certain simple but very revealing experiments. For example, since the core memory is paged and allocation is extremely flexible, it is possible to vary its effective size almost continuously by removing a page at a time from the pool of allocatable pages. One problem is that to a certain extent, it is possible for a shortage, say of core memory, to be taken up by a surplus of processor capacity. One other problem which is characteristic of the paged memory strategy used by the basic file system is that no matter how much memory is available, it will all be used. Finally, one must realize that the solution to a recognized imbalance may lie in any of several directions. For example, a shortage of core memory might be corrected by 1) purchasing more core memory, 2) readjusting a core memory multiplexing algorithm, 3) discouraging large programs by appropriate charges, or 4) improving a popular translator program to produce shorter object code. Any or all of these techniques, plus others, may be appropriate for a given situation.

The problem of deciding whether or not balance has been achieved is complicated by the possibility that a mis-tuned policy algorithm is causing "thrashing", that is excessive overhead caused by unnecessary processor switching or page swapping. Such thrashing can turn a resource surplus into a resource shortage. Before meaningful measurements can be made of system balance, it is necessary to convince oneself that the balance measurements will not be distorted by thrashing.

Thrashing can be detected by an appropriate set of measurements also. Consider first the case of core memory. If the core multiplexing algorithm permits too many processes to be loaded for the size of core memory available, those processes will fight very hard for the remaining core space for their private pages; as a result the average age of a page being written out for "lack of usage" may become quite short. If pages are written out only to be read in again an instant later, thrashing exists. We may postulate the following rough rule of thumb to indicate whether nor not core thrashing is occuring: If the average age of pages being written out is less than the average time that a process remains loaded, the chances of writing out a page which is still in use are very high, and we have prima facie evidence of thrashing caused by the core management algorithm. One appropriate correction to make for this situation might be to reduce the average number of loaded processes, as was described in chapter four.

A similar thrashing problem exists for processors. If, in an effort to improve response time to short requests, the processor scheduling algorithm places too short a time limit on most processes, the predominant cause of processor switching will be timer runout rather than release of a processor by a process blocking itself. Each timer runout introduces one extra scheduling operation and an extra dispatching operation later, when the process is picked up again. In addition to increasing overhead, if timer runouts are the predominant cause of processor switching, it is possible that average response time is actually degraded. For an intuitive notion why this is true, consider 10 processes each of which need 5 seconds of processor time. If each is run to completion, followed by the next, the first process will be served after five seconds, the second after 10, etc., the last after 50. On the other hand, suppose that each process is served for only 1 second, then the processor is switched to the next, etc., in a round robin. In this case, the first process to enter the system will not get out until 46 seconds have passed, the last still leaving at 50. If processor switching adds overhead, the delay times would be even more.

We conclude that it can be unprofitable to have pre-emption occur very often. Again, we may postulate a rough rule of thumb to detect processor thrashing: If more than half (or some other appropriate threshold) of processor switching operations are generated by running out of scheduled time, we may claim prima facie evidence of processor thrashing. Again, one possible correction to reduce thrashing is clear: adjust the processor scheduler to increase the average time limits it sets.

On the other hand, timer runouts are the primary technique available to the processor scheduling algorithms to improve response time on short requests. To guarantee adequately short response time for short requests for processor time it may well be necessary for the scheduler to push the system in the direction of processor thrashing by shortening its time limits. The tradeoffs required to produce a given quality of service while avoiding excessive overhead and undue delays for some jobs are an interesting area for exploration.

Once one is convinced that neither the core multiplexing nor the processor multiplexing algorithms are thrashing unnecessarily, it becomes possible to ask questions about system balance. To a certain extent, the importance of having processor and memory capacity balanced depends on how closely the presented load exhausts total system capacity. In a very underloaded system, severe imbalance may have little effect. In a system operating at the threshold of overload, a slight mis-allocation of resources can be very damaging. In order to talk sensibly about the state of balance of core and processor capacity, let us assume that the load on the system has been adjusted (by regulating the number of logged-in interactive users, for example) to the point that response time is just adequate. We can then look at the state of balance of the system. The primary tool to use here is the measurement of processor idle time. Recall from chapter four that a processor may be idle for one of two reasons:

1. There is actually no work to do.
2. The ready list contains work, but the core multiplexing algorithm refuses to allow any more processes to be loaded.

When a processor is idle for the first reason, we have a potential case of processor overcapacity. On the other hand, a certain amount of processor overcapacity may be essential to provide responsive service under peak load conditions. Again, the desire for service quality must be carefully weighed against the desire to maintain only enough processor capacity.

If the percentage of total time spent by processes in idling for the second reason is very great, we have evidence that the allowable load is being limited by the amount of core memory available. Reducing processor capacity will have very little effect on total system capacity or service quality under these conditions. On the other hand increasing only memory size will increase total system capacity or service quality.

The related problem of detecting core memory overcapacity is a little bit trickier since, as mentioned before, a paged core memory multiplexing strategy tends to use up all available memory, no matter how much there is. On the other hand, this very flexibility of allocation of memory can be turned into an experimental tool. One need merely "turn down" the size of core memory a little at a time by removing small blocks of memory from consideration by the core multiplexing algorithm. As the appropriate memory size is reached, processor idle time will begin to mount and the desired information of where memory "undercapacity" begins has been obtained.

We thus have several simple handles and tools available for detecting whether or not the resources of the system are well matched to the job they are trying to do. First, simple measurements indicate whether or not the scheduling algorithms are thrashing; after they are appropriately adjusted, and the load is adjusted to give reasonable response time, one can determine the state of balance of the system by measurements and an appropriate experiment. It is essential, of course, that the necessary performance monitoring "meters" be included in the traffic control and core control procedures. It is most important to realize that tuning a system requires consideration not only of hardware efficiency, but also of service quality as measured by the distribution of response times for various size computation requests. One must also be prepared for the possibility that the presented load will change, either in total resources used or in detailed character. In either case, the picture of system balance would be expected to change also.

System Scaling.

Our final area of inquiry is the range of system capacity and organization is permitted by the traffic control scheme. Two specific features in the traffic controller are directly concerned with the problem of scaling. First, the decisions of the scheduler are limited in overhead; that is, the amount of computation required to schedule a process does not depend on the number of other processes or processors in the system. Secondly, the interface with the basic file system has been organized in such a way that a process not presently making demands on the system can be effectively ignored. An idle process requires no space in core memory, even in tables, and does not increase the overhead of processor multiplexing.

What then, are the limitations on scaling which remain inherent in the design of the traffic controller? There are at least two distinct kinds of limitations which arise.

First, an important "constant" of the traffic controller is the total amount of computation required to schedule a process by a call to wakeup, plus the amount of computation involved in dispatching a processor by a call to Block. We will use the term _process switch time_ to denote this sum, while recognizing that it may be more appropriate to measure computation by counting processor instructions rather than measuring the time required to execute the instructions. The average amount of computation a process does before it calls block, (_average running time_) which is a characteristic of the presented load, combines with the process switch time to determine the overhead of processor multiplexing. (We are ignoring as a trivial complication the fact that extra process switching caused by pre-emption increases overhead also.)

If we claim that an idle processor is not contributing to the total computation performed by the system, it is clear that the fraction of total computation spent in multiplexing overhead does not depend on the size of the system in any way, or even on the speed or number of processors. It depends only on the relative values of these two "constants", one a characteristic of the traffic controller, the other a characteristic of the load.

From this fact we glean one clue about the way the system scales: it may not be profitable to split a single computation among several parallel processes if each of the parallel processes will have a run time short relative to the process switch time. We have here a limitation on the nature of the presented load, rather than the total computation required by the load; acquiring extra processor capacity, for example, will not reduce the fraction of overhead, although it will provide the ability to absorb the overhead.

The second kind of limitation inherent in the design of the traffic controller is found in the accessibility to the ready list. When the scheduler places a process in the ready list, and when the dispatcher (Getwork) removes a process from the ready list, there is a brief period during which the contents of the ready list must not be changed by another processor. During this period, the processor which is using the ready list <u>locks</u> the list by setting a lock cell non-zero. Each processor checks the lock cell before accessing the ready list, and if it finds the lock <u>on</u>, it must loop, waiting for the cell to go off, which signifies that the first processor has finished with the ready list. (Programming a multiprocessor interlock is a non-trivial task, and is usually done with the aid of special processor or memory hardware.) Since the ready list is only available on a one-at-a-time, first-come, first-served basis, it represents a potential bottleneck if there is more than one processor. The seriousness of the bottleneck depends on the amount of computation performed while the ready list is locked and the frequency of scheduling and dispatching operations. If, for example, it is observed that for a certain presented load processors do 1% of their computation with the ready list locked, it is clear that in a 100-processor system the ready list is virtually always locked by some processor and it represents a major bottleneck. On the other hand, in a two processor system only one scheduling operation out of 100 would result in a wait for the ready list to become free.

Competition among processors for the use of the ready list presents a problem similar to the problem of competition for the use of memory ports. One avenue of solution to this problem might be to break up the ready list into a number of sublists, each of which is locked separately, and arrange some strategy to insure that low priority processes from one list are not served before high-priority processes from another.

There are at least two important directions in which the system may scale almost indefinitely without any hindrance from the traffic controller whatsoever. First, the number of idle processes in the system may grow to a quantity bounded only by the amount of secondary storage required to remember them. By design, idle processes cost the system nothing in the way of core memory space or processor time. Scaling in this direction allows construction, for example, of an airline reservation system with 5000 agent sets in which each agent set is serviced by one process, and at most several hundred are active at once.

Secondly, if one adds the needed processor and memory capacity, there may be any number of so-called "scientific" jobs in the system--processes with extraordinarily long average running times. Such processes do not bring into play either of the fundamental limitations of the traffic controller since they generate virtually no scheduling operations themselves.

The important conclusion to be drawn from our discussion of scaling is that the basic scheme for traffic control can be used as the basis of a wide variety of types and sizes of computer systems; real-time, production, payroll, interactive work, etc. As technology of computer hardware evolves, the traffic controller represents a starting point for future designs; its fundamental limitations in scaling will need to be re-evaluated in the light of such hardware evolution.

A few final comments are in order concerning the "distributed supervisor" concept on which the traffic controller has been based. Recall that the hardware management procedures are segments appearing in the address space of each pseudo-processor. Although these procedures can be shared among all processes, there is no reason why every process must use the "standard version" of any procedure except Swap-DBR, which carries out mechanics of process switching. As long as the procedures respect the conventions of the system-wide data bases (process table, ready list, etc.) they can be distinct and can carry out distinct policies.

This organization is of interest to our discussion of scaling for two reasons. First, it provides the means of limiting the cost and complexity of decisions made by even a "standard version" of some supervisor policy-making module, by limiting the amount of information about a process which must appear in a system-wide data base. The processor scheduler, for example, by always executing in the address space of the process being scheduled, has access to the process data segment and any other private information about the process needed to make a scheduling decision. This information does not need to be placed in a table accessible to all processes.

Secondly, the distributed supervisor organization permits different processes to have different copies of the hardware management modules, and therefore to see radically different operating systems. This flexibility permits a system to be used, for example, for real-time or process control applications while simultaneously serving more routine customers. Again, since the scheduler for a process is the one provided by the process, the relative priorities of different operating systems can be distinct and independent. This same organization permits simple checkout of a "new" supervisor by one or a few system programmers without affecting continuous operation of the "old" supervisor to regular customers.

The distributed supervisor, then, contributes to the range and variety of customers and applications to which a single computer installation may be applicable.

CHAPTER SIX

Summary of Ideas

In the words and flow charts of chapters three and four we have described a relatively brief collection of algorithms, collectively known as the traffic controller, which are intended to provide workable solutions to each of the following problems of processor traffic control raised by a computer utility:

1. Processor multiplexing. This includes both sharing of processors among many users to provide interactive response (sometimes called time-sharing) and switching among procedures in response to interrupts so as to keep both processors and I/O equipment as fully utilized as possible (sometimes called multi-programming.)
2. Multiple processor organization. The problem here is to organize the system so as not only to increase its capacity by adding processors, but also to insure that operation can continue without program changes in the event that a processor breaks down.
3. Size and overhead. The overhead cost of processor multiplexing should not grow out of proportion to the size of the system as that size is increased.
4. Distributed supervisor. If each user can see his "own" supervisory system which may be different than the one

others see, the way is opened for simultaneous service to real-time or process control functions, regular computing customers, and a system programmer checking out a "new" system.

5. Parallel processing ability. Any user of the system should be able to specify parallelism in his algorithms, both to speed a compute-bound algorithm, and to provide input/output simultaneous with computation.

6. User control communication ability. Independent users should be able to send control signals back and forth to each other so as to be able to utilize effectively the data-sharing facility provided by the file system.

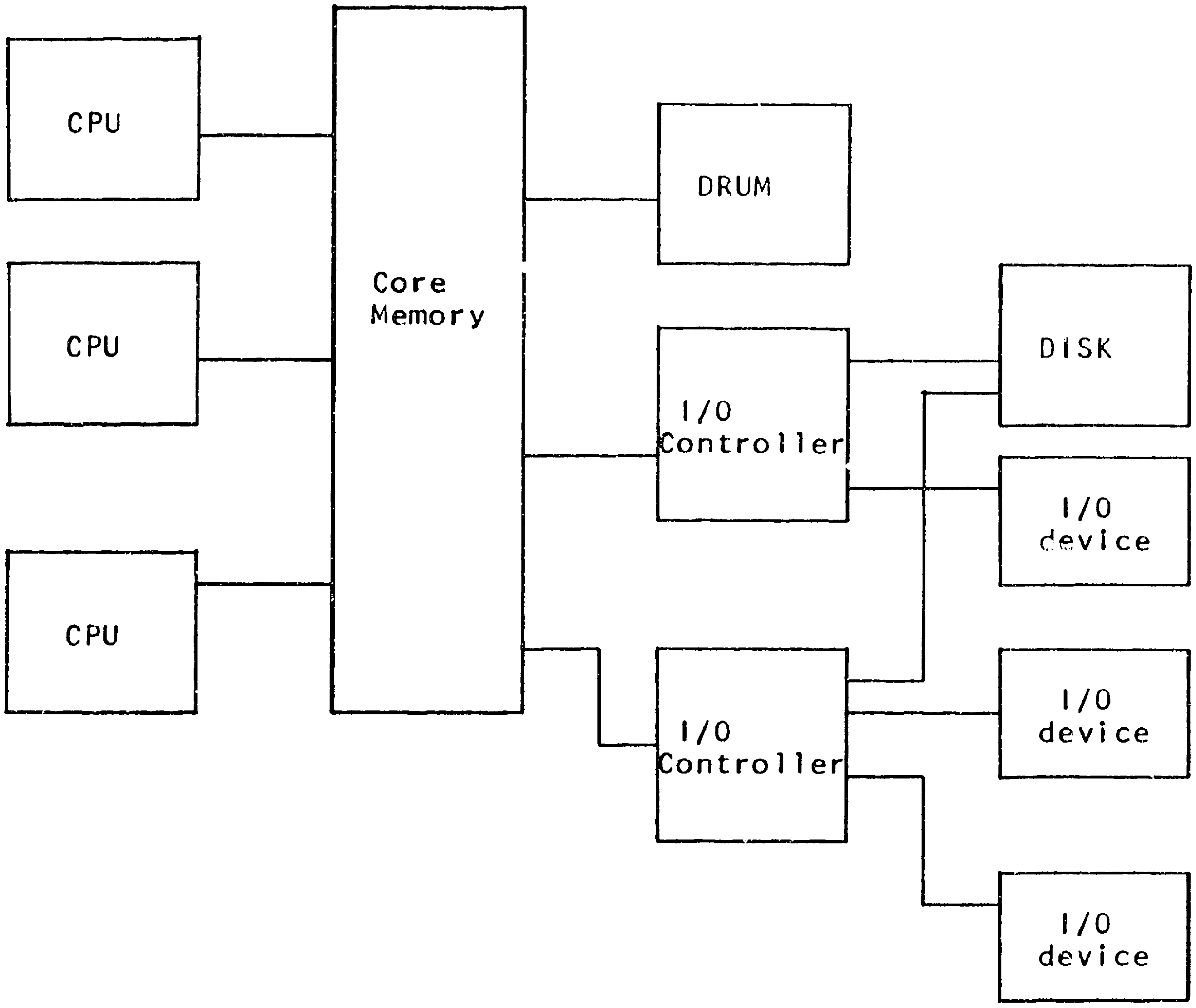
We have described the first three of these problems as technological, with the implication that the problems would not exist in a sufficiently advanced technology. The last three we describe as intrinsic, meaning that these problems will exist in some form in the computer utility no matter how advanced a technology is reached.

To meet these objectives, we have designed a traffic controller which simulates an arbitrarily large number of pseudo-processors, each with its own two-dimensional address space. The traffic controller appears as a group of segments in that address space; it contains entries which furnish the inter-process control communication necessary for parallel processing and inter-user communication. It also intercepts interrupts from I/O devices and converts them into inter-process control signals.

The multiplexing of the traffic controller is organized around a ready list, an ordered queue of processes ready to run, and a priority scheduler which places processes in the ready list in response to control signals. All processors service the ready list independently. Time limits on tasks in the ready list are used to trigger processor pre-emption; by setting appropriate time limits the processor scheduler can control response time to requests for computation. For fast response a scheduler can pre-empt a processor directly.

To limit the processor overhead cost of multiplexing, the scheduler is a procedure in the address space of the process being scheduled. An intricate interface between the traffic controller and the basic file system limits the core memory overhead cost of multiplexing in such a way that there may be an indefinitely large number of idle processes in the system.

Together, the procedures of the traffic controller can form the basis for a wide variety of operating systems on a wide variety of computers. As computer technology advances, the basic scheme described here may be used as a starting point for more advanced designs.



(lines show communication paths.)

Figure 2.1 -- Typical hardware configuration.

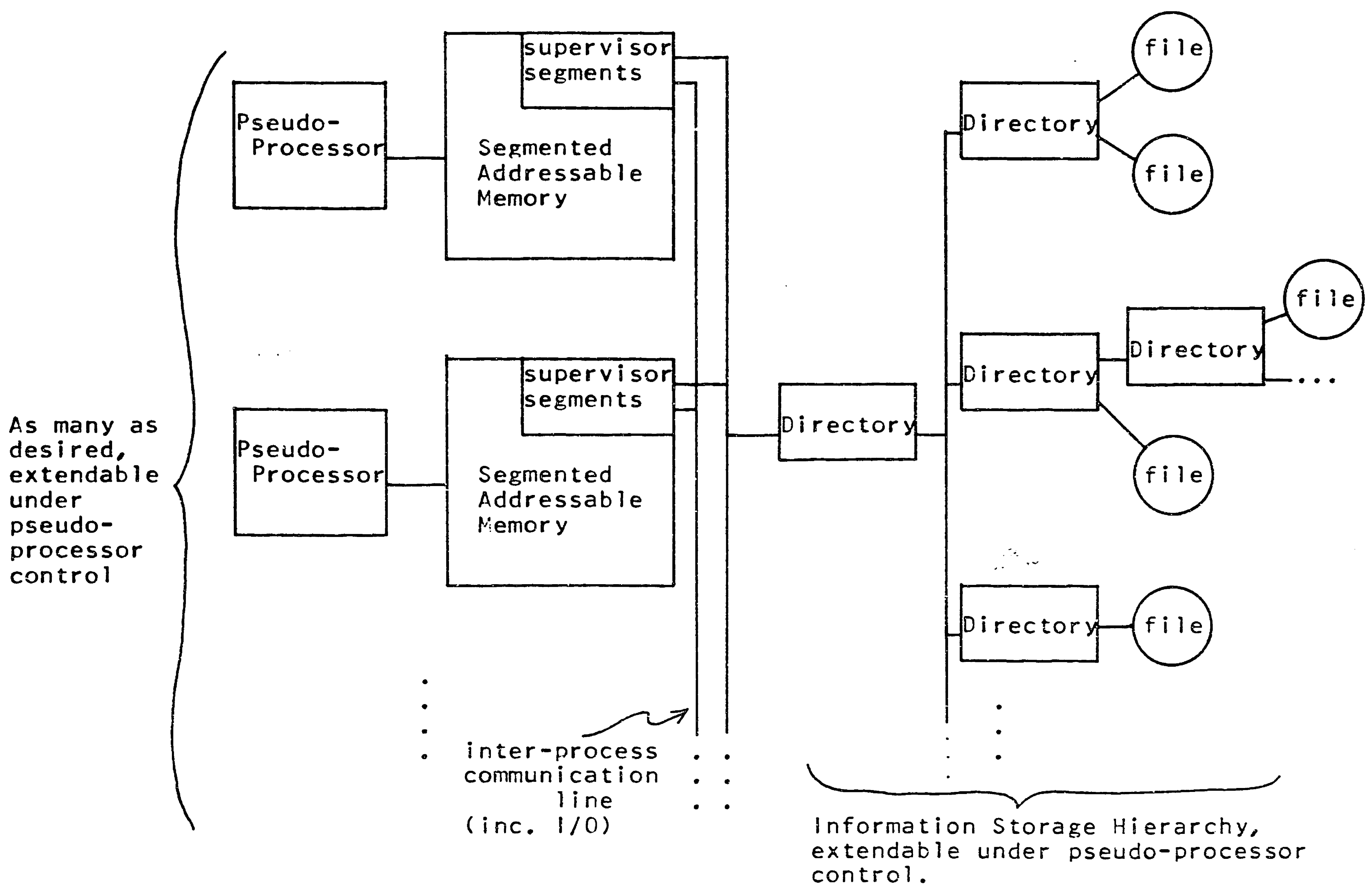

As many as desired, extendable under pseudo-processor control
Pseudo-Processor
Segmented Addressable Memory
supervisor segments
Pseudo-Processor
Segmented Addressable Memory
supervisor segments
inter-process communication line (inc. I/O)
Directory
Directory
file
file
Directory
Directory
file
file
Directory
file
Information Storage Hierarchy, extendable under pseudo-processor control.

Figure 2.2 -- Apparent system configuration after hardware management.

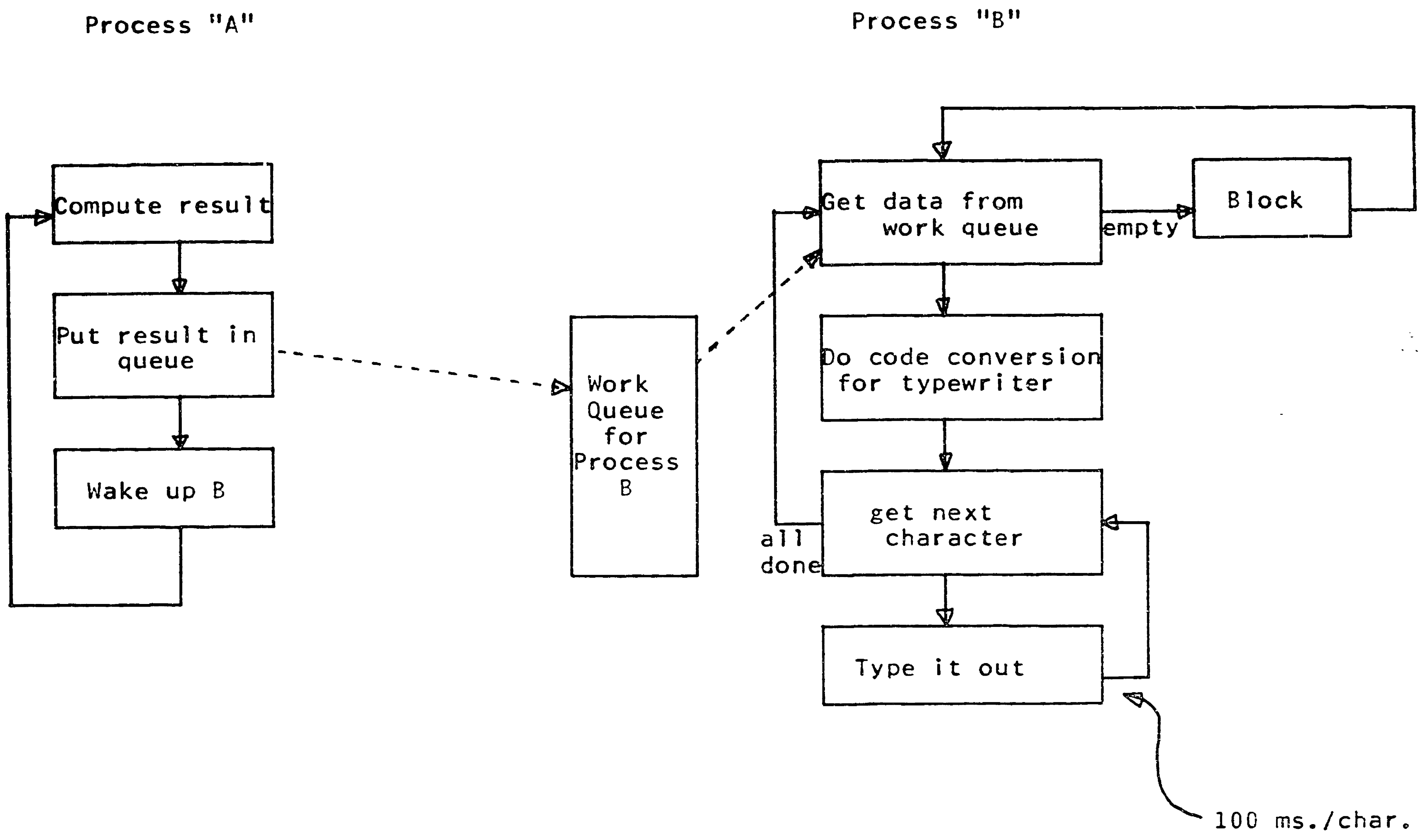


Figure 3.1 -- Flow diagram of a two-process system.

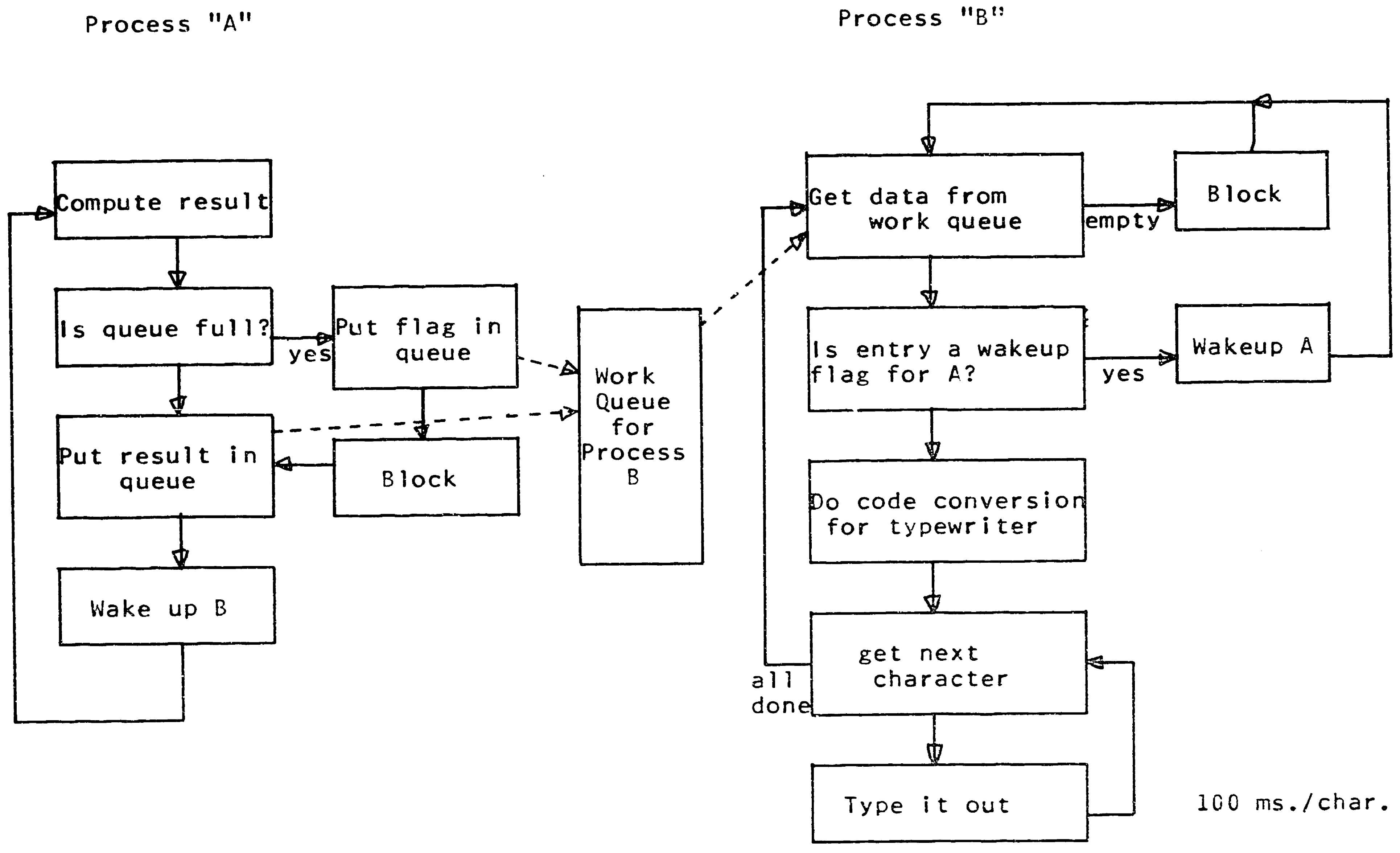


Figure 3.2 -- Two-process system with "full queue" provision.

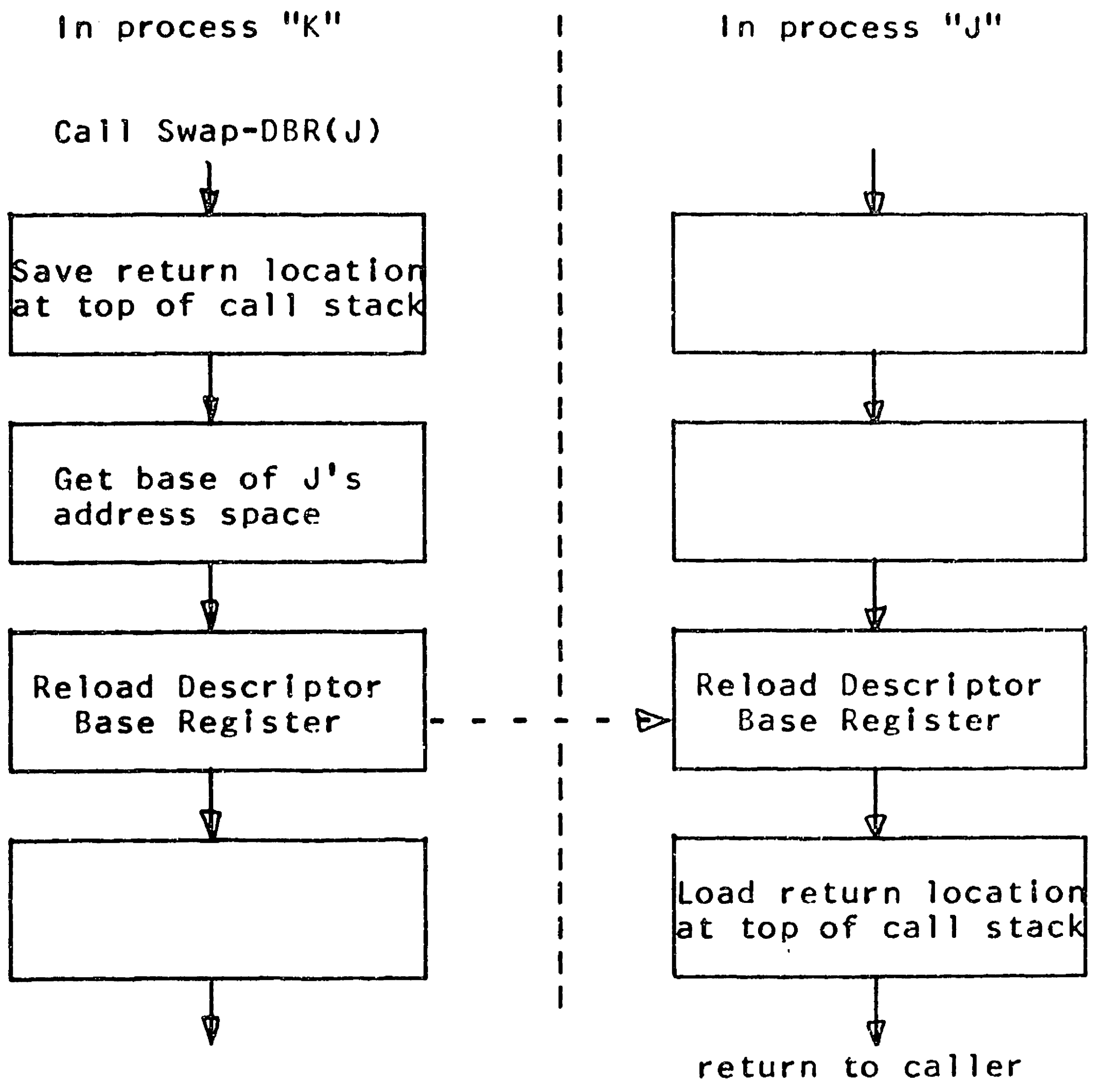

In process "K"
In process "J"
Call Swap-DBR(J)
Save return location at top of call stack
Get base of J's address space
Reload Descriptor Base Register
Reload Descriptor Base Register
Load return location at top of call stack
return to caller

Figure 3.3 -- Flow of control in Swap-DBR.

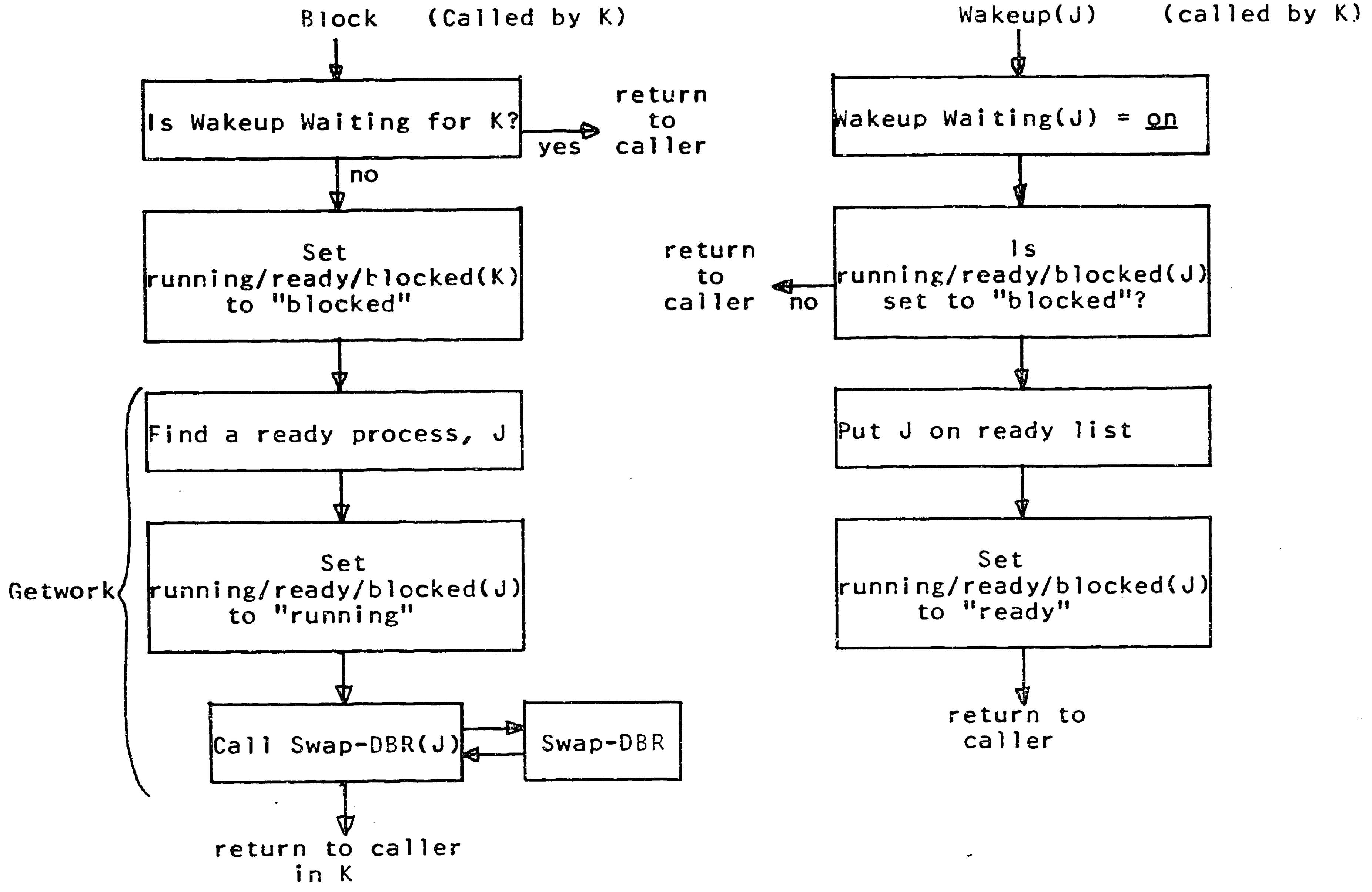


Figure 3.4 -- Flow diagram of Block and Wakeup.

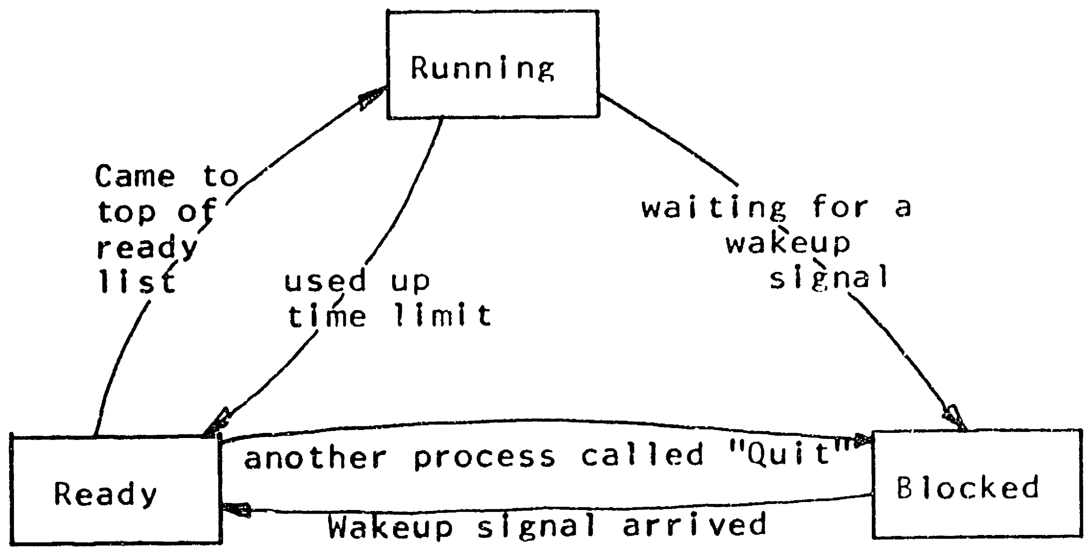

Running
Came to top of ready list
used up time limit
waiting for a wakeup signal
Ready
another process called "Quit"
Wakeup signal arrived
Blocked

Figure 3.5 -- Execution state transitions.

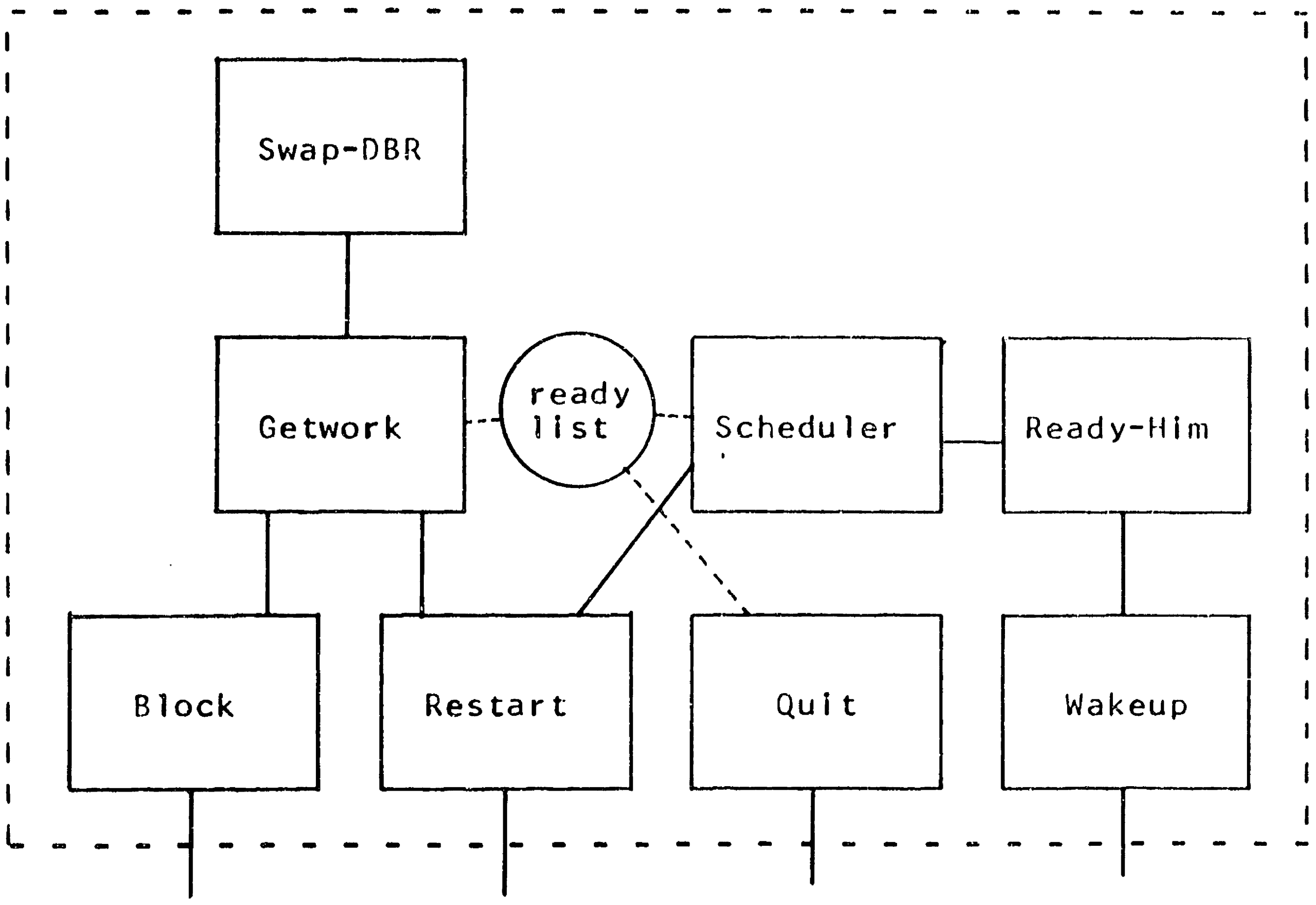


Figure 3.6 -- Block diagram of process exchange.

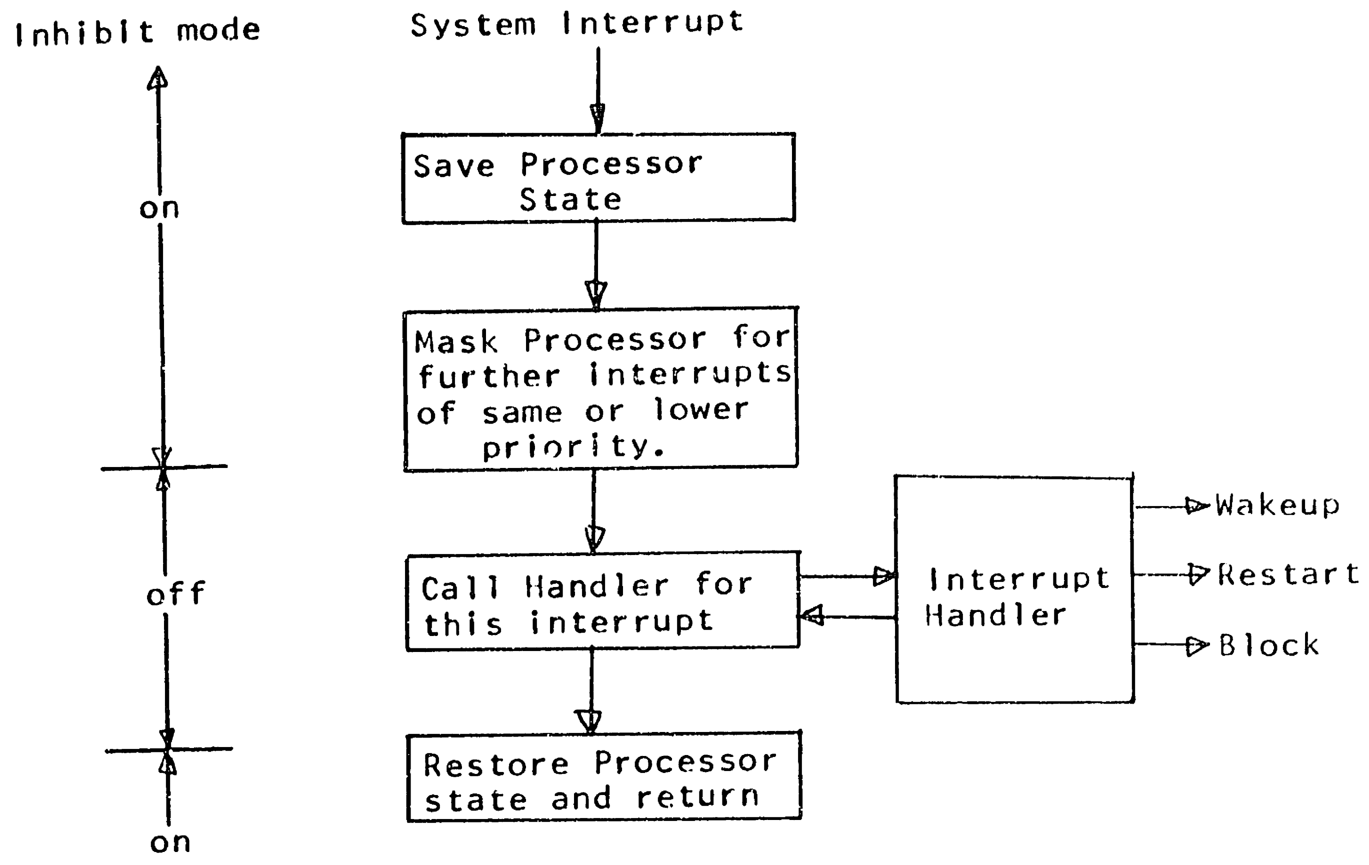

Inhibit mode
on
off
on
System Interrupt
Save Processor State
Mask Processor for further interrupts of same or lower priority.
Call Handler for this interrupt
Restore Processor state and return
Interrupt Handler
Wakeup
Restart
Block

Figure 3.7 -- Flow diagram of the system interrupt interceptor.

Table I. Catalog of tables used in processor and core multiplexing.

System-Wide Tables

name	number of entries	indexed by	contents of entry	usage
process table	1/process	process identification number	pointer to descriptor segment running/ready/blocked switch wakeup waiting switch	processor multiplexing data
active segment table	1/segment for which page table is in core	segment name	secondary storage location of segment	to find secondary storage location during missing-page fault

Private Tables, repeated once per process

name	number of entries	indexed by	contents of entry	usage
process data segment	1	———	process stateword	defines state of process when not running
segment name table	1/segment number used by process	segment number	name of segment	to find name of segment during a missing-segment fault.
descriptor segment	1/segment number used by process	segment number	segment descriptor word	map used by processor hardware to provide segmented address space

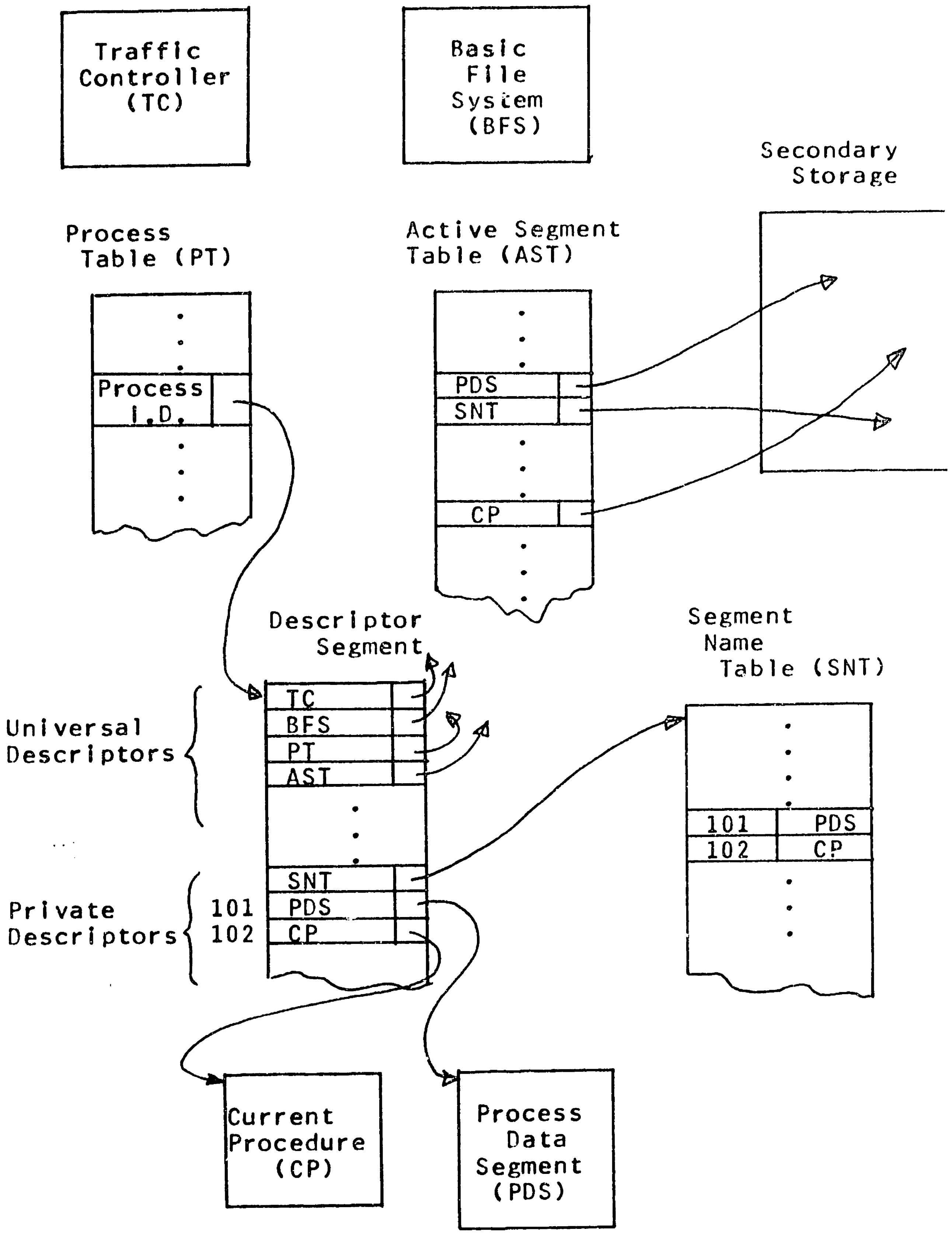


Figure 4.1 -- Schematic diagram of a running process.

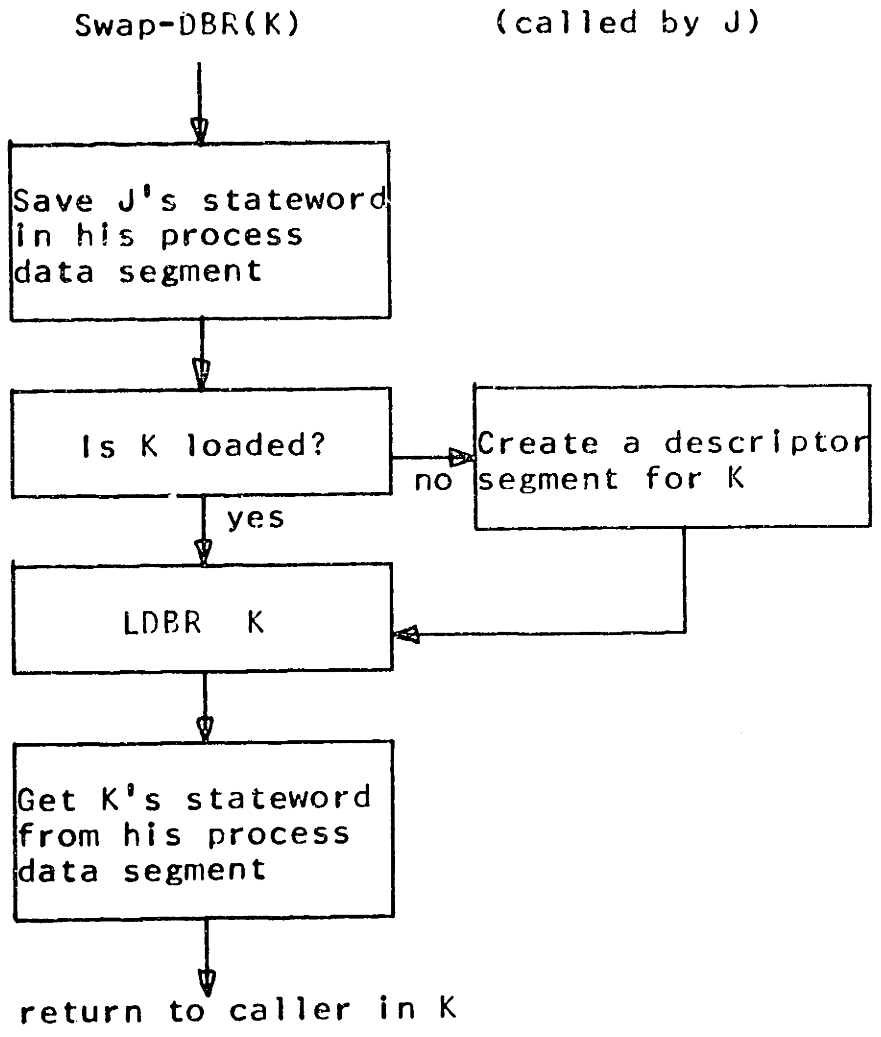

Swap-DBR(K)
(called by J)
Save J's stateword in his process data segment
Is K loaded?
no
Create a descriptor segment for K
yes
LDBR K
Get K's stateword from his process data segment
return to caller in K

Figure 4.2 -- Swap-DBR flow, to recreate descriptor segment.

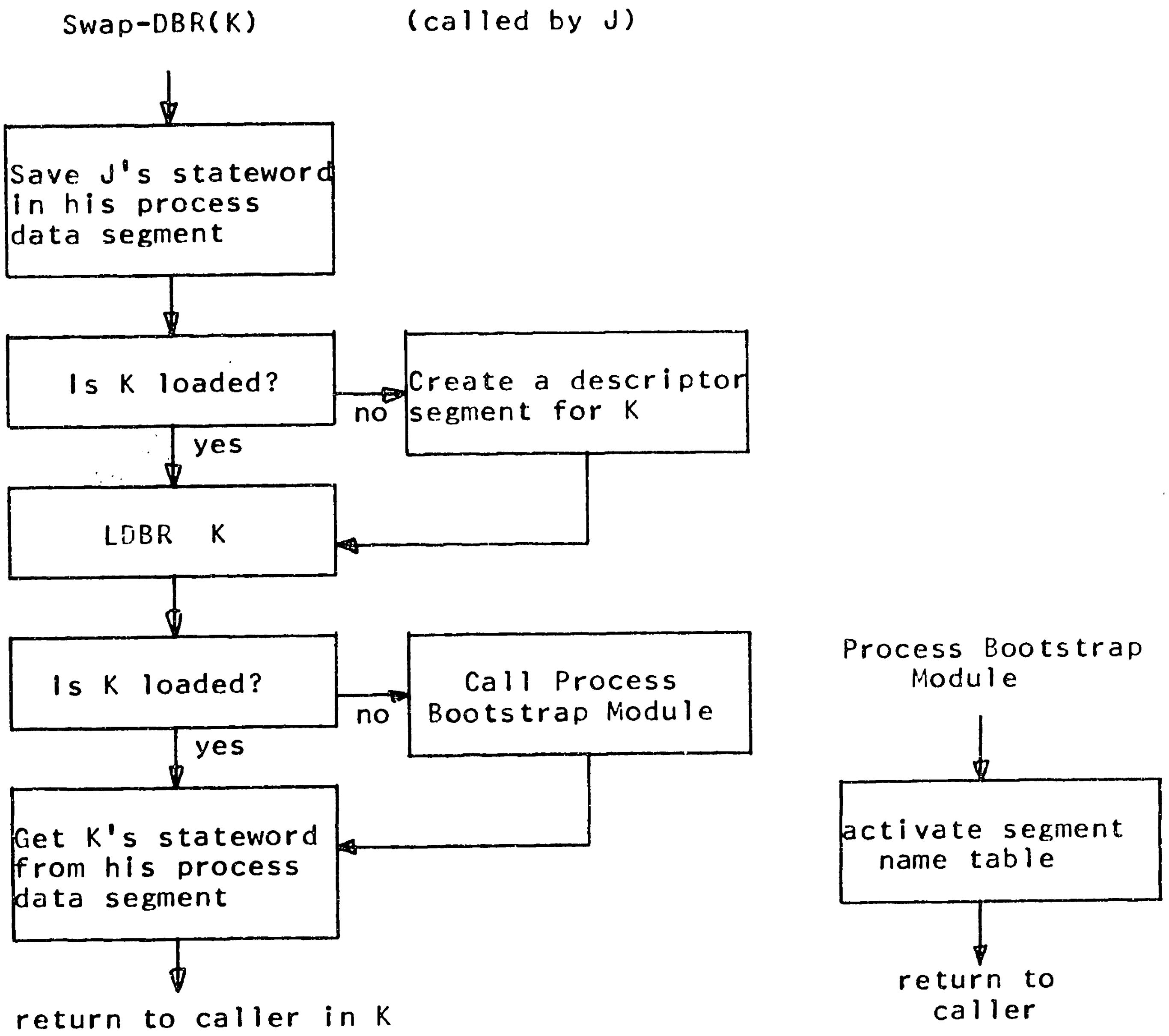

Swap-DBR(K)
(called by J)
Save J's stateword in his process data segment
Is K loaded?
no
Create a descriptor segment for K
yes
LDBR K
Is K loaded?
no
Call Process Bootstrap Module
yes
Get K's stateword from his process data segment
return to caller in K
Process Bootstrap Module
activate segment name table
return to caller

Figure 4.3 -- Swap-DBR flow, including process bootstrap module.

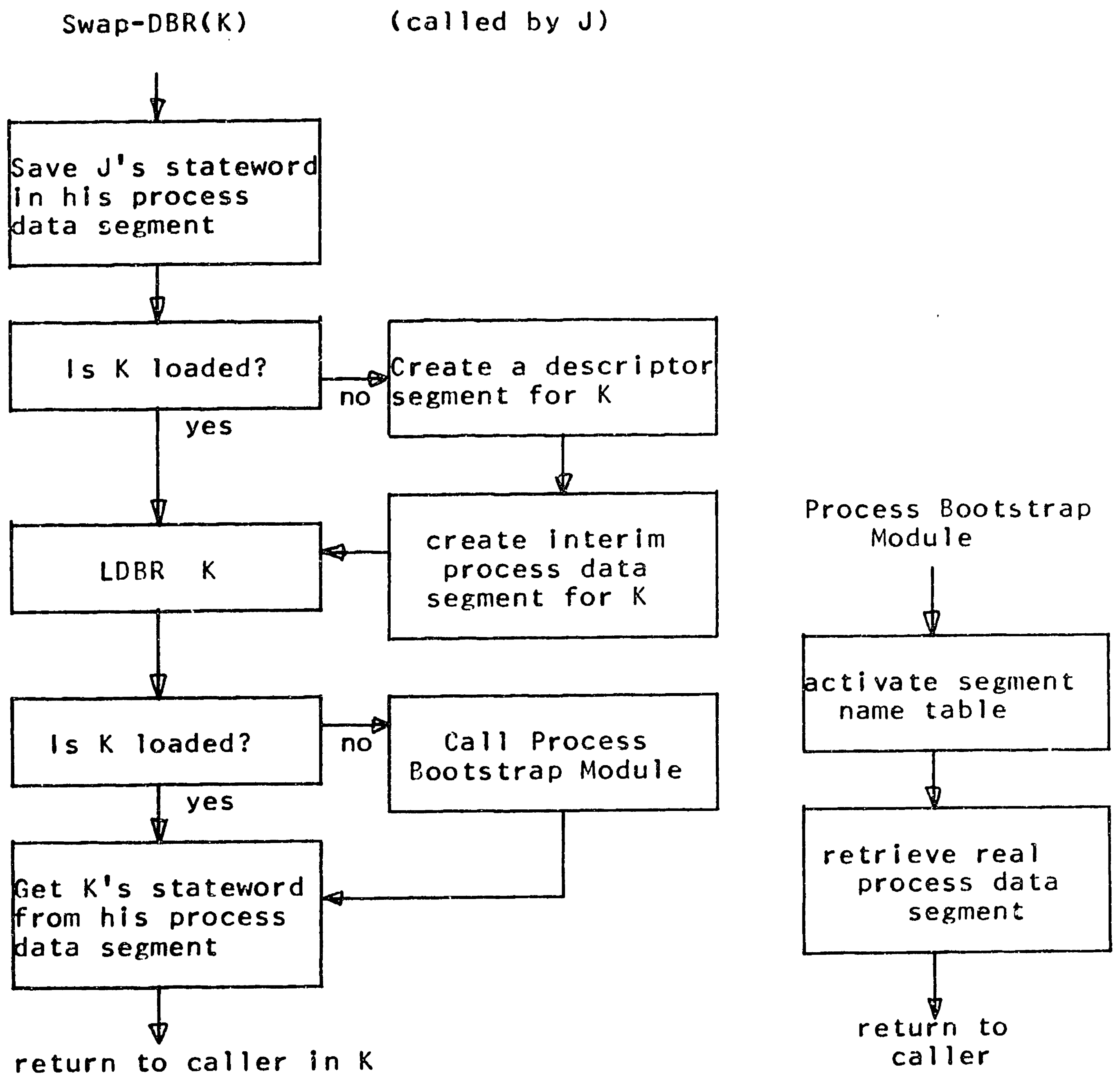

Swap-DBR(K)
(called by J)
Save J's stateword in his process data segment
Is K loaded?
no
Create a descriptor segment for K
yes
create interim process data segment for K
LDBR K
Is K loaded?
no
Call Process Bootstrap Module
yes
Get K's stateword from his process data segment
return to caller in K
Process Bootstrap Module
activate segment name table
retrieve real process data segment
return to caller

Figure 4.4 -- Complete flow diagram of Swap-DBR.

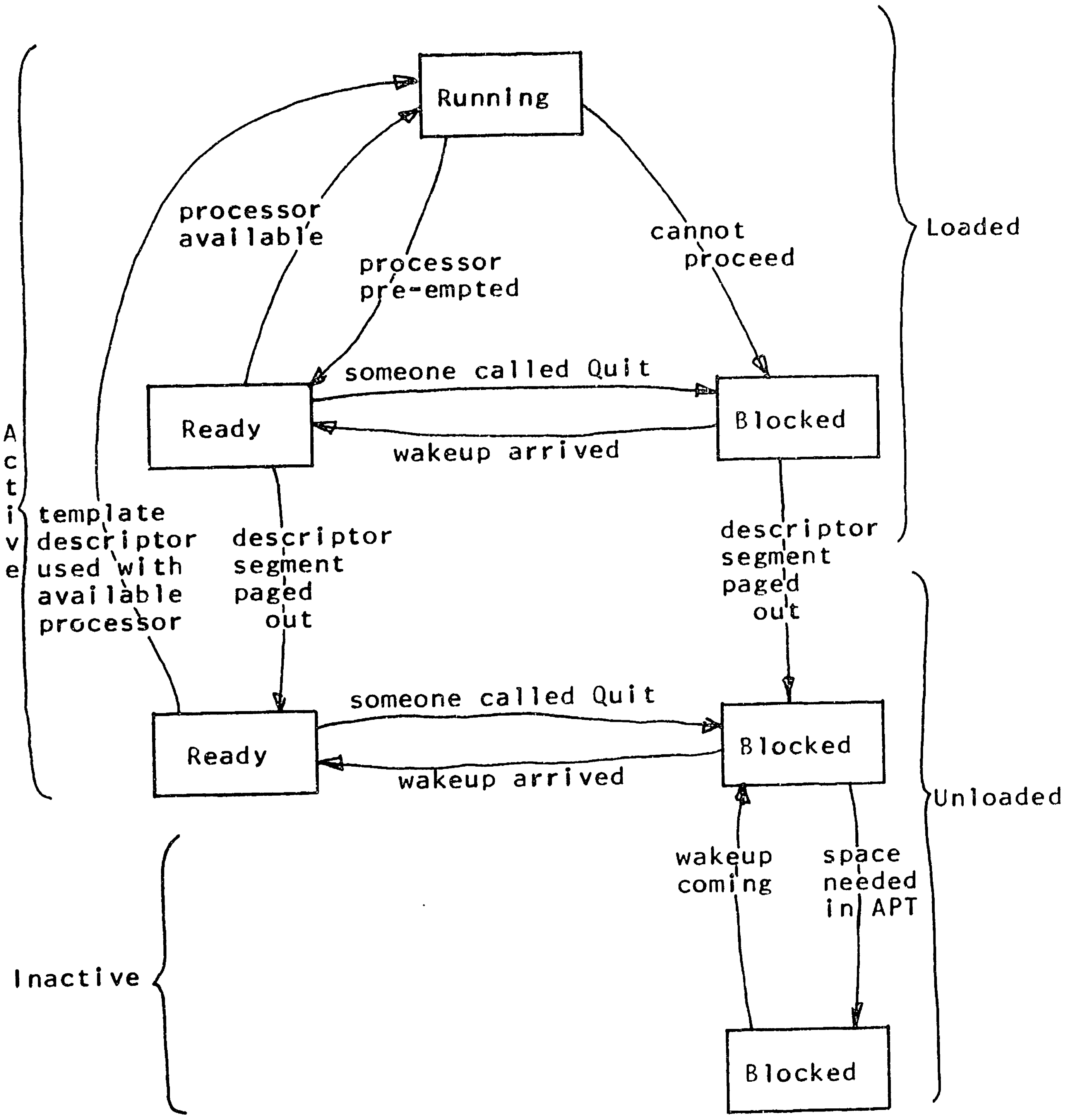


Figure 4.5 -- State transitions of a process.

REFERENCES

Abbreviations used in the references:

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

References, in order cited:

[1] Corbató, F.J., and Vyssotsky, V.A., "Introduction and Overview of the Multics System," AFIPS Conf. Proc. 27 (1965 FJCC), Spartan Books, Washington, D.C., 1965, pp. 185-196.

[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] Vyssotsky, V.A., et al., "Structure of the Multics Supervisor," AFIPS Conf. Proc. 27 (1965 FJCC), Spartan Books, Washington, D.C., 1965, pp. 203-212.

[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] Ossanna, J.F., et al., "Communication and Input/Output Switching in a Multiplex Computing System," AFIPS Conf. Proc. 27 (1965 FJCC), Spartan Books, Washington, D.C., 1965, pp. 231-241.

[6] David, E.E., Jr., and Fano, R.M., "Some Thoughts About the Social Implications of Accessible Computing," AFIPS Conf. Proc. 27 (1965 FJCC), Spartan Books, Washington, D.C., 1965, pp. 243-247.

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

[8] Arden, B.W., et al., "Program and Addressing Structure in a Time-Sharing Environment," Journal of the ACM 13, 1 (Jan. 1966), pp. 1-16.

[9] Critchlow, A.J., "Generalized Multiprocessing and Multiprogramming Systems," AFIPS Conf. Proc. 24 (1963 FJCC), Spartan Books, Baltimore, 1963, pp. 107-126.

[10] Conway, M.E., "A Multiprocessor System Design," AFIPS Conf. Proc. 24 (1963 FJCC), Spartan Books, Baltimore, 1963, pp. 139-146.

[11] Dennis, J.B., and Van Horn, E.C., "Programming Semantics for Multiprogrammed Computations," Comm. of the ACM 9, 3 (March 1966), pp. 143-155.

[12] Codd, E.F., "Multiprogramming", in Alt, F., et al., Advances in Computers, Vol. III, Academic Press, New York, 1962, pp. 77-153.

[13] Thompson, R.N., and Wilkinson, J.A., "The D825 Automatic Operating and Scheduling Program," AFIPS Conf. Proc. 23 (1963 SJCC), Spartan Books, Washington D.C., 1963, pp. 41-49.

[14] Corbató, F.J., et al., "An Experimental Time-Sharing System," AFIPS Conf. Proc. 21 (1962 SJCC), National Press, Palo Alto, 1962, pp. 335-344.

[15] Desmonde, W.H., Real-Time Data Processing Systems, Prentice-Hall, Englewood Cliffs, N.J., 1964.

[16] Witt, B.I., "The Functional Structure of OS/360: Part II, Job and Task Management," IBM Systems Journal, 5, 1 (1966), pp. 12-29.

[17] Crisman, P.A., editor, The Compatible Time-Sharing System: a programmer's guide, second edition, M.I.T. Press, Cambridge, 1965, section AG.1.04.

BIOGRAPHICAL NOTE

Jerome Howard Saltzer was born in Nampa, Idaho, on October 9, 1939. He attended public schools there, graduating from Nampa High School in May, 1957. He entered the Massachusetts Institute of Technology in September, 1957, where he studied Electrical Science and Engineering, receiving the degrees of S.B. (June, 1961) and S.M. (September, 1963). In June, 1961, he was married to the former Marlys Anne Hughes of Nampa, Idaho. They have two children, Rebecca Lee and Sarah Dawn.

Mr. Saltzer joined the staff of the M.I.T. Electrical Engineering department in February, 1961, as a teaching assistant; in July, 1963 he became an Instructor. He has taught subjects on circuit theory and computer programming systems; in June, 1965 he received a departmental teaching award. During the summers of 1961 and 1962 he was a staff engineer at M.I.T. Lincoln Laboratory, working in the area of computer waveform processing; since then he has been a consultant to the Lincoln Laboratory. In June, 1964, he became associated with M.I.T. Project MAC, where his research in multiplexed computer systems was the subject of his doctoral dissertation.

Mr. Saltzer is a member of Sigma Xi, Eta Kappa Nu, Tau Beta Pi, and the Association for Computing Machinery.

Publications

<u>Advanced Computer Programming</u>, M.I.T. Press, Cambridge, 1963 (With F.J. Corbató and J.W. Poduska)

"CTSS Technical Notes," Project MAC Technical Report TR-16, June, 1965.