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 without change 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 tree name 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 on, 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 must 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.

Resource Management.

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 only 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