TIME SHARING SUPERVISOR PROGRAMS

Michael T. Alexander

The University of Michigan

Computing Center

May, 1969

Revised May, 1971

ABSTRACT

The structure of supervisor programs for time shared or multiple access operating systems is described. Those functions of the supervisor that are concerned with scheduling and resource allocation are emphasized. The four time sharing systems that are described are Control Program 67 (CP/67), Time Sharing System (TSS/360), and University of Michigan Multi-Programming Supervisor (UMMPS) for the IBM System 360 Model 67 and Multics for the General Electric 645. The emphasis is on describing and comparing the various supervisor programs, rather than recommending any specific approach. In general those aspects of the supervisor programs that are the direct result of some peculiarity of the hardware used are not included.

KEY WORDS

Time Sharing
Supervisor Programs
Scheduling Algorithms
Resource Allocation
Multi-Programming
Multi-Processing

TABLE OF CONTENTS

1. Introduction 1

2. Basic Concepts 2

3. Hardware Considerations 4
   3.1 Interrupts 4
   3.2 Relocation Hardware 4
   3.3 Protection Mechanism 7
   3.4 Monitoring of Main Storage Accesses 8
   3.5 Multiple Input/Output Paths 8

4. Supervisor Services 9
   4.1 Inter-Task Protection 10
   4.2 Resource Allocation 11
   4.3 Task Oriented Services in UMMPS 11
      4.3.1 Scheduling of Input/Output Resources ...12
      4.3.2 Processor Management 12
      4.3.3 Task Interrupts 12
      4.3.4 Inter-Task Communication 13
      4.3.5 Storage Management 13
      4.3.6 Miscellaneous Functions 13
   4.4 Interrupt Handling 13
   4.5 Task-Supervisor Interface 14

5. Structure of the Supervisors 17
   5.1 Attributes of the Supervisor 17
   5.2 Organization of Each Supervisor 18
      5.2.1 TSS Supervisor 18
      5.2.2 UMMPS 19
      5.2.3 CP/67 19
      5.2.4 Multics Supervisor 19

6. Storage Scheduling 21
   6.1 UMMPS Paging Algorithm 22
   6.2 Multics Paging Algorithm 31
   6.3°°CP/67 Paging Algorithm 32
   6.4 TSS Paging Algorithm 33

7. Processor Scheduling 36
   7.1 CP/67 36
   7.2 Multics 39
   7.3 UMMPS 41
   7.4 TSS 45
   7.5 Comparison of Processor Scheduling Algorithms 48

8. Input/Output Processing 50
   8.1 Organization of Input/Output Hardware 50
   8.2 Task Input/Output Control 52
   8.3 Supervisor Input/Output Control 54

9. Multi-Processor Considerations 59
   9.1 Organization of Multi-Processor Support 59
   9.2 Hardware Consideration 60
   9.3 Multi-Processor Support in UMMPS 61
   9.4 Inter-Processor Interference 62

FIGURES

1. Two Level Address Translation 6

2. PCB Format in UMMPS 23

3. Relation of PDP and UMMPS 28

4. CP/67 Processor Scheduling States 39

5. TSS Processor Scheduling Lists 48

6. Simple Input/Output Configuration 52

1. INTRODUCTION

These notes consider supervisor programs for several current time-sharing systems, where "time-sharing system" is used to indicate a system allowing simultaneous, on-line, conversational access by several users (who may be interractive terminals, small computers, or special devices) in competition for system resources, and the "supervisor programs" are those parts of the time-sharing system that control allocation of system resources to various users. The four separate time-sharing systems that have been selected for main consideration are Control Program/67 (CP/67), the University of Michigan Multi-programming Supervisor (UMMPS), Time Sharing System/360 (TSS/360), and Multics. These do not by any means represent an exhaustive list of current time-sharing systems, but rather were chosen because they represent several different approaches to supervisor programs for time-sharing. It will be seen that although the organization of the four supervisors considered is quite diverse, there is a quite considerable similarity among the algorithms used for controlling the use of these resources.

2. BASIC CONCEPTS

Since terminology in computing in general and time sharing in particular is not well established, a few terms will be defined. These are not the only terms used for these concepts, in fact the four systems considered do not all use them, but to avoid confusion, they will be used consistently throughout these notes.

The central concept of each of the four systems, and indeed of nearly all time-sharing systems, is the concept of a task (sometimes called a process or a job) which is the execution of a set of programs and subroutines. That is, a program is a (static) set of instructions and data while a task is the (dynamic) execution of a set of instructions operating on data. In a time-sharing system a task will usually have some specific purpose such as providing computing service for one user or controlling some specific portion of the input/output equipment, and each user of the system will have at least one task which is primarily responsible for providing service for that user. Some important attributes of tasks are (a) independence (they do not interract directly except in fixed limited ways), (b) parallel execution (each task executes in parallel with all other tasks), and (c) competition for system resources (in general there are not enough resources such as storage or processor time to give each task all it would take).

The entity which executes a program in behalf of a task is known as a processor (or a central processing unit or CPU). Many large time-sharing systems have more than one processor, creating interesting problems for the supervisor since two or more programs may be executing simultaneously. In particular the supervisor itself may be executing on more than one processor at a time. Some of the problems resulting from this will be considered in section 9.

Any medium for storing information so that it is available to the computing system on a random (or nearly random) access basis is referred to by the generic term storage. This normally includes some amount of main storage, usually core storage, in addition to a larger amount of slower auxiliary storage such as magnetic disks and drums or slower core storage. Usually (but not always) a processor is not capable of executing a program or referring directly to data stored anywhere but in main storage. In a time-sharing system a certain amount of the auxiliary storage is generally used to contain named private or shared collections of data (called files or data sets) for the various potential users of the system. This use of storage will not be explicitly considered here since it is not generally the direct responsibility of the supervisor

program.

Any portion of the computing system hardware which is primarily used for the transmission of information from one type of storage to another or between storage and the external world is known as input/output equipment. In nearly every case this transmission is asynchronous with respect to processors and takes place between core storage (usually main storage) and something else (e.g., a disk, a card reader, or a teletype).

3. HARDWARE CONSIDERATIONS

Several aspects of the hardware used for time-sharing system are important and will be described briefly here. This description applies primarily to the IBM System 360 Model 67 (on which CP/67, TSS/360, and UMMPS all run) and to the General Electric 645 (on which Multics runs), but several other computing system for time-sharing have similar features.

3.1 Interrupts

Whenever any event occurs which may be important to the supervisor, the hardware notifies the supervisor by means of an interrupt. An interrupt is simply a forced call of a predetermined subroutine along with a change in the state of the processor. The supervisor must provide subroutines to service each of the various types of interrupts which may occur.

The possible interrupts are generally divided into two categories: those due to some event external to the processor and those caused by some abnormal occurrence within the processor. The most common type in the first category indicates a change is the state of the input/output equipment, while the most common type in the second category is used by a task to request service of some kind from the supervisor.

Interrupts are the most important tool of the supervisor in performing its various functions in a time-sharing system and in a busy system the number of interrupts processed will be very large. For this reason it is necessary that the basic code used to process interrupts be efficient. It is estimated that removing one micro-second from each interrupt handler in UMMPS running on a large 360/67 could save as much as 20 seconds a day in processor time spent in the supervisor.

3.2 Relocation Hardware

The best known time-sharing hardware feature is the relocation hardware which, among other things, allows moving of programs in main storage without changing them. This facility is similar in general outline on the 360/67 and the 645, although the details are completely different. On both machines the address used by a program to refer to main storage (the virtual or logical address) is divided into three fields: the segment number, the page number, and the offset within the page. This division of the logical

address into three fields leads to the division of the logical address space referenced by a program into segments and pages within segments. There are as many possible segments as can be addressed by the segment number (although only a few will be active at any time) each of which can (but won't) be as big as can be addressed by the page and offset portions of the address. The division of the logical address into segments and pages is fixed in the 360/67, but in the 645 it may be changed among several possible ones by the supervisor; in fact the division of segments into pages can be disabled and the page and offset can be combined into a single quantity.

The segment number, page number, and offset are used as follows: the segment number is used as an index into a table in main storage (called the segment table or descriptor segment) to obtain a value which gives the status of the corresponding segment and the location of a page table for the segment. The status information may indicate that the segment is not available, in which case an interrupt occurs to notify the supervisor that some task attempted to access an unavailable segment; but if the segment is available then the page number is used as an index into the page table indicated by the segment table entry to obtain a page table entry. (This step may be skipped on the 645.) This entry again contains a status field and a pointer, which (if the status field indicates the page is available) points to the actual main storage address of the page. Again if the page is not available the supervisor is notified by an interrupt. The origin of the page is added to the offset given in the address to give the real main storage address. More precisely, since the page must start on an address that is a multiple of the page size, the offset is simply concatenated to the high order part of the page origin. See Figure 1 for a diagram of this process. This very superficial description leaves out many of the details of the implementation, most of which are included for efficiency. In particular it does not do justice to the complexity of the 645 in which the virtual addresses are not simply single quantities divided into three fields, but rather the segment number is stored separately.

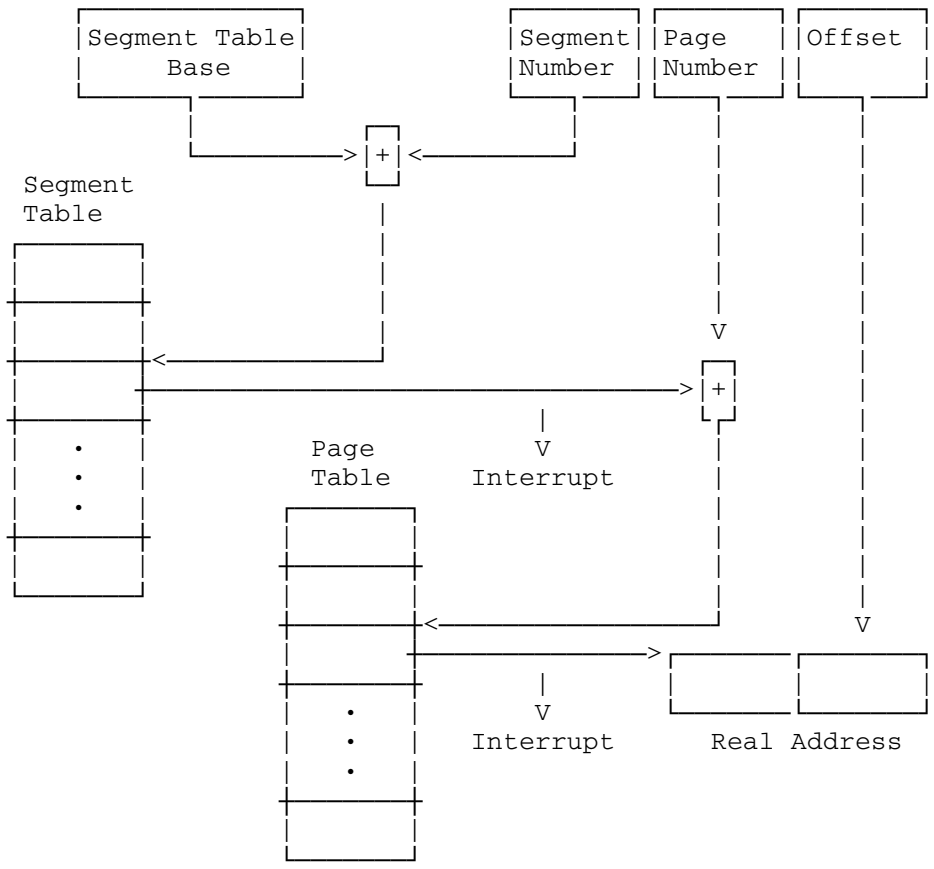

Figure 1: Two Level Address Translation

It is extremely important to note that this mapping from virtual addresses to real addresses (or interrupts) can be made different for each task operating in the system at any time by giving each task its own segment and/or page tables. This has several implications of importance. First it means that a given virtual address may have different meanings in different tasks. Also it means that information private to a task (i.e., existing only in that task's virtual address space) simply does not exist as far as other tasks are concerned, giving complete inter-task protection of private data.

Hardware Features

Since the segment tables of several tasks may contain entries pointing to the same page table, it is easy to share information between tasks, either with the same virtual address in each task or with a different segment number. Since the segment number of shared information may be different in different tasks, it is not possible for shared segments (which must have the same contents in all tasks) to contain pointers to other shared segments (or private segments of course).

In general the division of the virtual address into segments is used to divide the programs and data used by a given task into logically separate segments of information, while the division of segments into pages is transparent to the tasks and is used by the supervisor to efficiently manage main storage. The general area of proper use of segmentation is an extremely interesting and complex one but it will not be covered in these notes; for our purposes we will consider the virtual address space of a tasks to be divided into pages and consider how this division is used by the supervisor. See [1] and [3] for discussions of ways segmentation may be used.

3.3 Protection Mechanism

Another area of the hardware which is closely related to relocation is the area of protection. As mentioned above, the relocation itself gives inter-task protection for private data; however, some mechanism is required to provide protection for sensitive data within a task and to ensure that the supervisor maintains control of the machine.

The first requirement is met by assigning an access right to every reference by a program to a virtual storage location. The access right is determined by the attributes of both the program making the access and the data in question. On the 360/67 this mechanism is essentially the standard System 360 storage protect mechanism which assigns a "key" (which can be set by the supervisor) to both the program being executed and to all blocks of main storage. If the keys of the current program and the data being accessed are not the same, access is restricted either to read only or to no access. On the 645 each segment for each task has a mode indicating whether it can be read as data, changed, or executed by the task. If it can be executed the mode is further subdivided to indicate the privilege of the program contained in the segment. The mode of a segment is indicated in the segment table entry for the segment and to change the mode of any or all segments, the supervisor need only change the segment table pointer to point to a different segment table. This means that in Multics, each

task may in fact have several different segment tables for different levels of protection.

In order to guarantee that the supervisor maintains control of the machine certain instructions which change the state of the machine in some basic way must be made unavailable to the tasks. Hence both machines (and in fact most current computers) have a non-privileged mode (sometimes called user mode, slave mode, or problem program mode) in which certain instructions cause interrupts if executed, and all tasks operate in this mode. The instructions which are not allowed are those which affect the relocation or protection mechanisms or the input/output state.

3.4 Monitoring of Main Storage Accesses

A minor point which should be mentioned in connection with hardware features is the automatic monitoring of every reference by any component of the hardware to every block of main storage. Whenever a component of the hardware (a processor, an input/output unit, or anything else) refers to a block of main storage, a bit associated with that block is set. In addition if the block is changed a second bit associated with the block is set. These bits (which are associated with the storage key on the 360/67 and the page table entry on the 645) can be tested and reset by the supervisor, allowing it to find out which blocks of main storage have been referenced and changed since the bits were last reset.

3.5 Multiple Input/Output Paths

Another feature of the 360/67 which is much more important on it, although present to a limited extent in other models of the System 360, is the availablity of several "paths" from main storage to an input/output unit. It is possible to have up to eight possible sets of equipment (including channel control units, data channels, and control units) which can be separately assigned to transmit data between a particular set of input/output equipment and main storage. Section 8 describes how this affects the supervisor.

4. SUPERVISOR SERVICES

There are several functions that the supervisor of a time-sharing system performs. As mentioned above, the tasks in a time-sharing system are independent and protected from one another and the enforcement of this rests largely with the supervisor and is one of its more important functions. A second function of the supervisor is to provide an interface between the tasks and the hardware of the computing system. Modern computing hardware is designed to be flexible and efficient rather than easy to use, particularly the area of input/output equipment. For this reason most supervisor programs provide a modified interface to the hardware for the tasks under their control. Also to obtain best use of a large computing system it is essential that some program be responsible for allocating portions of the system to various uses in an intelligent manner. This function of the supervisor is closely related to its job of acting as an arbitrator to the tasks in their competition for system resources.

Before looking at the services provided to the tasks by the supervisor, it is important to have a good understanding of the nature of a task in the systems we are considering. Although most current operating systems have the concept of task, the exact meaning of this varies widely from system to system. In the case of the systems we are considering here it is very much as if the separate tasks were running on separate computers and were completely unrelated. In fact CP/67 makes a special effort to treat the tasks as if this were true. The vast majority of the tasks active at any given time in one of these systems will each be serving the needs of a single "user" of the system, and for each user there will be a task responsible for him. All of the tasks in this category are treated as nearly the same as possible by the supervisor.

There will often be a few tasks active at any time that serve functions that are not related to serving any specific user. These tasks will be created by the system to perform some function that is required to allow the other tasks to run normally. An example of such a task is the PDP task described below in section 6.1.

In the case of UMMPS, every task in the system has a name which is specified when the task is created, either by the operator or by another task. This name determines the entry point of the program that the task is to execute when it has been initialized, and for this reason describes the general nature of the task. If the program specified by a given name is reentrant, it is possible to have many tasks active with that same name. Such is the case with the tasks

serving users, all of which are named MTS. In addition to specifying the entry point of the program the task is to execute, the name specifies whether the task is to be absolute or relocatable. An absolute task is run with the relocation hardware turned off and is allowed to execute certain supervisor calls that are prohibited from the relocatable tasks. These tasks all are system tasks that are not related to any specific user (for example the PDP).

4.1 Inter-Task Protection

The job of protecting the tasks from each other is made somewhat easier by the special hardware mentioned above — particularly the relocation hardware and the storage protect mechanism; but in spite of this much care and consideration must be given to this problem.

One point that seems almost too obvious to be mentioned, but which is often overlooked anyway, is that all information which is necessary for the proper functioning of the supervisor must be maintained in storage that is protected from the tasks. All of the systems considered here do a fairly good job of this, although the completeness of the safeguards vary from system to system.

In particular it is essential that all information necessary to remove a task from the system, and reclaim all of the system resources of which it was the owner, is maintained in storage which can be guaranteed to be safe. Then it is possible to assure that no matter what happens to the task, at least it is possible to remove it from the system without causing any permanent damage. This is very important for a system which may expect to run continuously for many hours. If when a task crashes it were necessary to loose system resources until the next system shutdown, the system performance might be seriously degraded. As a result of this consideration it is necessary that all allocation of system resources to a task be done by the supervisor so that it will have a complete record of the current allocation to each task.

When the supervisor allocates a system resource to a task it must be sure to maintain ultimate control of the resource so that it can revoke the allocation later; it must never unconditionally allocate part of the system to a task. For instance when a processor is given to a task it must be certain that an interrupt will eventually occur to give control of the processor back to the supervisor. In the case of less important resources, such as input/output devices, it may only be necessary to assure that it is possible to get them back if necessary (for instance if

requested to do so by the system operator).

4.2 Resource Allocation

In most computing environments it is not possible to give each task all of the resources it needs all of the time. In fact if this is always possible, then the system is probably not fully loaded and the work could be handled with less hardware. The job of deciding when to allocate each resource to each task requesting it falls in the realm of the supervisor because it is the only part of the system which maintains a global enough view of the situation to be able to do this intelligently. In fact nearly every function of the supervisor falls into the general area of scheduling the allocation of resources. The requirements of the tasks for system resources fall into three general categories: processor time, storage use, and input/output equipment. The methods used by the supervisor to control the allocation of each of these forms the topic of a section of these notes.

In performing this basic function of controlling the allocation of system resources to the tasks requesting use of them, the supervisor program could operate with two somewhat contradictory goals: to minimize the total time tasks must wait for requested resources, and to maximize the utilization of all system resources. These goals are contradictory because to switch control of a resource from one task to another always requires use of some resources. For example to switch control of a block of main storage from one task to another usually requires saving the information in the main storage and restoring the information needed by the new user of it, and this procedure requires use of processor time, input/output equipment time, and main and auxiliary storage. Hence it is sometimes advisable to require a tasks to wait longer than the minimum possible time for a resource in order to reduce the number of times it is switched from one task to another. Since time-sharing systems generally are interracting with human users, any waiting time which is comparable with human reaction time is usually considered adequate. The biggest exception to this is tasks which are providing some system related function and which require faster service for this reason.

4.3 Task Oriented Services in UMMPS

The supervisor in a sense never initiates any work in the system; when no tasks are active, the system is idle even if the supervisor is active. Hence the primary purpose

of the supervisor is to provide services for the tasks. The services provided by the UMMPS supervisor to the tasks it controls fall into several general categories:

4.3.1 Scheduling of Input/Output Resources

This function requires the largest amount of code in the supervisor, not so much because it is extremely difficult, but rather because of the multitude of special cases required. The tasks are able to request use of specific input/ouput devices (such as card readers, printers, telephone lines, etc.), and when they have received permission to use one of them they can queue requests for input/output operations on it. The supervisor will execute these operations as facilities become available and will notify the task when the operations are complete. In the case of some abnormal event in connection with the input/output operation the task will be notified and further operations on that device will be inhibited until the task takes some corrective action. For this purpose several supervisor functions are provided. (This aspect of the supervisor is described in Section 8.)

4.3.2 Processor Management

The UMMPS supervisor provides a mechanism by which a task can wait for some event to occur, where an event is defined to be a certain set of bits in some byte in the task's virtual storage being all zero. The task can indicate whether the event will also result in a signal being transmitted to the waiting task or whether a periodic check should be made to see if the event is complete. It is also possible for a task to voluntarily relinquish a processor to another task that needs it (if any exists). In CP and TSS a task can wait only for any interrupt, not for a specific event.

4.3.3 Task Interrupts

Certain events (such as the completion of an input/output operation) may cause an interrupt to the task to be generated, that is the current status of the task will be saved and it will be forced to execute at a certain predetermined location. The supervisor provides facilities which allow the management of this mechanism, such as a way to set up certain interrupts, a way to return to the point at which the interrupt occurred, and a way to delete saved return information.

4.3.4 Inter-Task Communication

Although tasks are generally completely independent, it is sometimes necessary for some communication to take place between them. There are facilities provided by which a task may send a signal to another task or get or receive information from another task. Also there are a set of supervisor subroutines that can be used to synchronize several tasks by causing them to wait until some resource is available.

4.3.5 Storage Management

Subroutines are provided to allocate and release virtual storage for the tasks. Note that what is being allocated in virtual storage not real storage; only the address space is being allocated and the actual storage will not be allocated until it is needed as described in Section 6 below. This is an area in which UMMPS is not so general as Multics or TSS: in TSS it is possible for several tasks to allocate storage that is shared between them; while in Multics it is possible to specify that the storage allocated is to be effectively preset with the contents of some data set that has been stored in the system previously. The data set is actually read into main storage only as the parts of it are referenced, not when it is allocated. As in TSS it is possible to allocate shared storage, now with the added complication that it may also be "preset."

4.3.6 Miscellaneous Functions

The supervisor is also responsible for providing certain small housekeeping functions such as maintainance of the time of day. Also it is of course responsible for the creation and destruction of tasks.

4.4 Interrupt Handling

The basic tool of the supervisor in performing all of these diverse functions is the interrupt mechanism. In CP, TSS, and UMMPS (but not in Multics) this is the only way in which the supervisor can be entered; it is never entered from a task except by way of an interrupt.

The possible causes of interrupts on the 360/67 are

1. Change in the state of input/output equipment

2. Signal from another processor

3. Signal from the operator

4. Expiration of a preset time interval

5. Malfunction detected in some component of the hardware

6. Abnormal condition in the program being executed

7. Specific request for an interrupt by the program being executed

Interrupt types 1 through 5 are due to conditions external to the program being executed, while types 6 and 7 are due to that program and are sometimes called faults instead of interrupts.

In TSS and UMMPS interrupt type 7 (called a supervisor call or SVC) is usually used to request some specific supervisor service, while interrupt type 6 (called a program interrupt) generally indicates a program error, except that missing page interrupts are categorized as program interrupts. However in CP SVC's are never processed by the supervisor and program interrupts are used for all requests for service by the tasks (more will be said on this below).

In general it is possible to think of the supervisor in CP, TSS, or UMMPS to be simply a set of subroutines for processing interrupts, although this is not exactly true in the case of TSS. Even in the case of Multics interrupts play an important part in the operation of the supervisor, although it is designed to hide this fact as much as possible.

4.5 Task-Supervisor Interface

The supervisor in each of these systems has a well defined interface with the tasks. However the particular interface chosen varies from one to the next. Logically the supervisor exists between the tasks and the hardware of the computing system and can be thought of either as an extension of the task to deal with the hardware or as an extension of the hardware to deal with the task. The particular approach used by each of the systems is at least partly determined by the nature of the hardware they use. On the 360/67 the supervisor programs are entered by interrupts which either request service or indicate task errors, while on the 645 the hardware is such that the supervisor may be entered either by an interrupt or by a direct transfer by a task to a new segment. Hence it is

natural for the Multics supervisor to be considered to be part of the task, in fact just a set of programs and data bases some of which are shared among all tasks. On the other hand the instructions on the 360/67 which cause interrupts to request supervisor service can easily be considered to be "extended machine instructions" and the supervisor appears to be almost part of the processor the task is running on. In fact in CP/67 and TSS/360 the programs and data which form the supervisor are not included in any tasks virtual storage (incidently making them immune to modification by a task). Note that even in the 645 the supervisor is sometimes entered by an interrupt (for instance when an unavailable segment or page is referenced), but that in these cases things are set up so that it appears to be running in some task anyway.

The interface between CP and its tasks is the easiest to describe: it is simply and exactly the same as the interface between the System 360 hardware and a program executing on a 360. Hence the instructions normally used to call the supervisor in System 360 are not processed by CP at all but are simply returned to the task as a simulated interrupt to be processed by whatever "supervisor" there is in the task. Also there are tables for each task giving its virtual machine's status, such as whether it is in privileged state (i.e., can execute all machine instructions) or not. However the real machine is never put into privileged state while a task is executing, and any privileged instruction which is executed by a task causes an interrupt and is simulated by CP if the virtual machine for that task is in privileged state. This means that those interrupts that are used to call the CP supervisor are just those interrupts that are due to the attempt to execute a privileged instruction — the same interrupts in any other of the systems are considered to be an error.

The interface between the tasks and the TSS supervisor is similar to the hardware interface on the System 360, but it has been changed to suit the environment. Instead of simulating a virtual machine that looks like a real 360, the TSS supervisor simulates one that has been altered a bit to make the interface more efficient. All request for supervisor service are the result of supervisor call interrupts rather than privileged operation interrupts as in CP. Furthermore the format and meaning of certain of the built in parts of the virtual machine are different than in the real 360. For instance the number and meaning of the interrupts are slightly different and some of them return significantly more information than is returned by the corresponding interrupt on the real 360. In spite of these alterations the interface still looks remarkably like the real 360 interface.

Supervisor Services

The interface between UMMPS and its tasks is even more removed from the real 360 interface. As in TSS all requests for supervisor services are the result of supervisor call interrupts, not privileged operation interrupts. Again there are interrupts which can be directed to the task, but now they are not at all like the interrupts in the real 360. Furthermore they are not a very important part of the interface. The usual case is for some specific service to be requested by the task and if necessary the task will be made to wait until the service is completed.

It is possible for a task executing under the control of UMMPS to request the the next SVC or program interrupt that occurs in that task is not to be processed by the supervisor, but rather is to be sent to the task unprocessed. It is also possible for a task to request that its relocation tables be temporarily changed so that its address space corresponds to that of a "real" 360, either a standard 360 or a model 67. Using these facilities it is possible for a program running in a task under UMMPS to simulate a standard 360 much the same way that CP does, and in fact such a program exists. It is used to allow OS/360 and other stand alone programs and systems to be run under the control of UMMPS so that it is not necessary to shut the system down to run a few OS jobs. This facility also allows UMMPS to be run under itself, providing a powerful tool for developing the system.

In Multics the interface between the tasks and the supervisor in a sense does not exist, since the supervisor is simply a set of subroutines in each task. The supervisor always runs in some task even if it is processing an interrupt. To do this it requires a stack for calls that is hidden from the task and used only for interrupt processing. However most requests for service from the tasks are made with ordinary calls.

5. STRUCTURE OF THE SUPERVISORS

The four systems being considered differ widely in the basic organization of their supervisor programs. In general the three systems for the 67 are basically similar in this respect (but in practically no other) while Multics is radically different.

5.1 Attributes of the Supervisor

Certain attributes of a supervisor are important to an understanding of its organization.

1. It may or may not run with the relocation mechanism described above enabled.

2. It may or may not run entirely in privileged state, that is, in the state in which all machine instructions are legal.

3. It may or may not run with interrupts (from input/output equipment, etc.) enabled.

It is possible to disable the relocation mechanism mentioned above so that the address used by a program is the same as the actual main storage address. This is done while a task is running only in UMMPS and there only for special tasks, but TSS, CP, and UMMPS run this way in the supervisor. On the other hand, since the Multics supervisor is considered part of the task, it runs with relocation enabled. The advantages of the Multics approach are that the supervisor is not fixed in any areas of main storage, and in fact part of it need not remain in main storage at all times, although some portions of it must remain in main storage to service missing page interrupts.

The supervisors for TSS, CP, and UMMPS all run in the privileged mode at all times allowing them complete freedom to execute any instruction. Again Multics is different in this respect since most of its supervisor runs in non-privileged mode, although certain portions of it must run in privileged mode to execute those instructions not available in non-privileged mode. Since most of the supervisor runs in non-privileged mode, it is somewhat protected form errors in itself. This distinction and the previous one between Multics and the 360/67 systems is largely due to the distributed supervisor concept in Multics that puts the supervisor in each task instead of off by itself as in other systems.

TSS and Multics supervisors run most of the time with input/output and other external interrupts enabled. This means that some mechanism must be maintained to process interrupts occurring while other interrupts are being processed. In Multics the supervisor is recursive and maintains a stack for this purpose, while in TSS an interrupt occurring while processing a previous interrupt is simply queued for later action. This queueing method in TSS is central to the whole supervisor and will be discussed below. The supervisors for CP/67 and UMMPS both run with all interrupts disabled, allowing them to complete processing one interrupt before another one occurs on the same processor. The biggest disadvantage of this is that no interrupt can be guaranteed service in less time than it takes to process the longest interrupt. Also, since some interrupts can not be completely processed immediately, some queueing mechanism is needed to handle actions that must be delayed by the supervisor, even if other interrupts can not occur.

5.2 Organization of Each Supervisor

It will help in understanding the algorithms used by the supervisors to look at the general organization of each of them. The algorithms employed by these four supervisors to implement scheduling of system resources are similar in several cases even though the overall organization is quite different among them.

5.2.1 TSS Supervisor

The TSS supervisor consists primarily of a mechanism for queueing interrupts as they occur, a program to scan this set of queues looking for work, and a set of programs which are called to perform the processing required when a non-empty queue is found. Most of these programs run with interrupts enabled and all of them run with the machine in privileged state and relocation disabled. All entries to the supervisor are made via interrupts (either task generated or generated outside the processor) and the supervisor appears in no tasks virtual address space. When the queue scanner finds no work for the supervisor to do it goes to a procedure which will give the processor to a task requiring it if any exists, using an algorithm which is described in some detail below.

5.2.2 UMMPS

The UMMPS supervisor consists of a set of subroutines some of which are called as the result of hardware interrupts and some of which are internal. It runs in privileged state, with interrupts disabled, and with relocation turned off. All interrupts are processed to completion as nearly as possible before more interrupts are allowed in the same processor, but a queue is maintained for those things which must be delayed. All entries to the supervisor are made by interrupts as in TSS, and when an interrupt is completely processed (and no work which was delayed can be done) the supervisor gives the processor to a task if possible. To the tasks UMMPS appears to be an extension of the 360/67 hardware and the instructions used to call it appear to be extended machine instructions.

Unlike the other systems, UMMPS allows some tasks to run with the relocation hardware disabled. This allows these tasks to refer to any main storage location without any address translation taking place. The tasks running this way are all special tasks providing some specific service such as controlling the unit record equipment.

5.2.3 CP/67

The CP supervisor is essentially the same as the UMMPS supervisor at this gross level except for the meaning of many of the interrupts. In particular those interrupts which are used in TSS and UMMPS by a task calling the supervisor are not processed at all by the CP supervisor but are merely passed back to the task by simulating an interrupt for the task, and the privileged operation interrupts which in TSS and UMMPS are considered task errors and are passed to a special task subroutine are used by a task as the primary method of calling the supervisor.

5.2.4 Multics Supervisor

The Multics supervisor is a set of programs and data which are part of all tasks. It always runs as part of some task and hence operates with interrupts enabled, relocation enabled, and in non-privileged state except when it cannot do so. The supervisor is recursive to allow it to handle interrupts occurring within itself. In fact in Multics the entire notion of a supervisor becomes rather ill defined and nebulous since there is no entity separate from all tasks. In spite of this, the functions of resource allocation and scheduling must still occur and it is the programs that implement these and certain other functions which are

referred to as the supervisor and which we will consider here.

Overall Supervisor Structure

6. STORAGE SCHEDULING

The method used by all of these systems to schedule the use of main and auxiliary storage is usually known as demand paging. This term is used to describe a system of storage management in which information is stored partly in main storage and partly on several types of slower and less expensive auxiliary storage and in which the information is moved between these two (or more) levels of storage in units smaller than the total storage of one task. To understand why this method is chosen one must bear in mind two points: that information must be stored in main storage before a processor is able to utilize it, and that moving information between main storage and auxiliary storage is a process that is relativelly expensive in terms of processor time and other system resources. For these reasons among others it is desirable to store much of the information contained in the system as a whole on more inexpensive auxiliary storage and move only that part of it that is currently required into main storage. The relocation hardware described earlier allows this to be done easily since it is possible to have any subset of the pages associated with any task in main storage at any time, and furthermore these pages may be located anywhere in main storage.

These consideration alone would justify this organization of storage, however demand paging has further advantages above and beyond these in that it is possible for the total storage available to a task to be much larger than actual amount of main storage available in the system. In fact by use of the segmentation hardware described earlier it is possible to consider all the data stored in the system and available to a task to be simply an extension to its virtual address space, as if it were all in some very large virtual storage. This is done by considering each logical grouping of data (data set or file) to be "mapped" into one or more segments in virtual storage. Then the mechanism described below is used to move the data from external storage to main storage as it is needed. This approach is used to some extent by TSS and to a much larger extend by Multics, in which it plays a very important part in the basic design of the system.

One pitfall which must be avoided at all costs (which has not always been the case) is the error of considering this vast virtual storage as if it were all real storage, each portion of which is directly addressable in an equally short time. This is not the case and programs which indiscriminately reference a very large virtual storage will suffer the consequences of requiring many transfers of pages between auxiliary storage and main storage at an excessive penalty in terms of system overhead. This effect is

responsible for a large part of the poor reputation which demand paging systems enjoy today. However when properly used the mechanism of treating all of the data available to a program as an extension of virtual storage is an extremely powerful one.

The methods used to control the transfer of pages between main and auxiliary storage in CP, UMMPS, and Multics are similar and the UMMPS method will be described followed by a summary of differences. The TSS method is quite different and will be discussed separately.

In all of these systems (including TSS) a page must be in main storage to be used by a task and if an attempt is made to use a page which is not in main storage an interrupt will occur to notify the supervisor of this event. When this type of interrupt occurs the supervisor must find an available main storage block to hold the requested page and either move the page from auxiliary storage if it is stored there or allocate the page in the main storage block if it is a new page. In any case the decision of which pages are to be moved to main storage is not generally very difficult since each request is handled when it occurs.

There are two cases in which pages must be moved to main storage even though they were not explicitely requested by a task. This can occur if certain pages must be in main storage before a task can even be started on a processor or if the supervisor attempts to "pre-read" certain pages to reduce the time a task must wait for the moving of pages. Only Multics makes use of "pre-reading" but both TSS and Multics require certain pages to be in main storage before a task can be started. These request are handled much as if they were page request interrupts which occur when a task is considered for the use of a processor and hence do not affect the algorithm much.

6.1 UMMPS Paging Algorithm

In UMMPS the function of moving pages between main storage and auxiliary storage is divided between the supervisor and one special task (called the Paging Drum Processor or PDP) running under control of the supervisor. The auxiliary storage medium used by UMMPS is one or more drums or disks. These devices are under complete control of the PDP which handles the function of constructing channel programs to read and write pages on them and which notifies the supervisor when a page has been read or written. In this context "reading" a page means moving it into main storage and "writing" it means moving it to the auxiliary storage medium, i.e., the drums or disks. The choice of

which pages are to be read or written is completely up to the supervisor while the actually reading and writting is up to the PDP.

The basic unit of information used in communication between the supervisor and the PDP is the Page Control Block (PCB) which contains all the information concerning the status of a single page. All PCB's contain a main and auxiliary storage address for the page, status bits indicating the state of the page, scratch area for use while moving the page, and a pointer field that is used to chain the PCB's on various queues. The format of the PCB's is given in Figure 2.

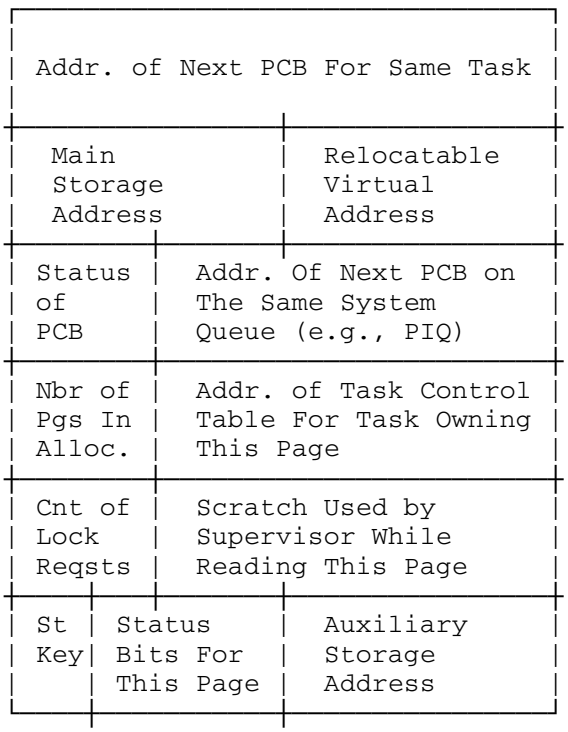

Figure 2: PCB Format In UMMPS

To control the interraction between the supervisor and the PDP there are five supervisor subroutines called only by the PDP and four queues of PCB's used to pass information between the supervisor and the PDP. The four queues are:

1. Page In Queue (PIQ) - contains PCB's for all pages which have been requested to be read into main storage but which the PDP has not started reading yet.

2. Page In Complete Queue (PICQ) - contains PCB's for all pages which the PDP has completed reading (or allocating if no reading was necessary) but of which the supervisor has not been notified.

3. Page Out Queue (POQ) - contains PCB's for all pages which are in main storage and which could be removed if necessary to make space for more pages.

4. Release Page Queue (RPQ) - contains PCB's for all pages which have been released by there owning tasks but which the PDP has not released yet.

The five supervisor calls (SVC's) used only by the PDP are:

1. Get Real Page (GETRP) - used to request a main storage block into which to read (or allocate) a page that must be brought to main storage.

2. Free Real Page (FREERP) - used to notify the supervisor that a main storage block that was previously allocated to a PCB is now available for reallocation. Also used to notify the PDP that a page was reclaimed while it was being written.

3. Get Write Pages (GETWP) - used to request one or more pages from the Page Out Queue which will be removed from main storage.

4. PDP Wait (PDPWAIT) - used to notify the supervisor that the PDP has no more work to do temporarily.

5. Get Queues (GETQS) - used to return the PIQ and the RPQ to the PDP.

Note that although there are four queues for PCBs, it is not necessary for a PCB to be on one of them at all times. In normal operation, most PCBs (including those that correspond to pages on auxiliary storage that are not being paged in or out) will be on no queue.

Perhaps the best way to learn how these are used is to follow an example page request through its processing. The functions of the PDP and the Supervisor in this process are diagramed in Figure 3. When the supervisor determines that a page must be moved to main storage it will place the PCB for that page on the end of the PIQ and start the PDP if it is currently idle. When the PDP has completed whatever work it is doing at that time it will call the GETQS subroutine which will return to it the PCB for the page that was requested (and any other pages which must be brought to main storage or released, i.e., which are on the PIQ or the RPQ).

At this point the PDP will place each of the PCB's from the PIQ on local queues which exist for each disk and for each relative position on each drum. For each drum there are 9 queues corresponding to each of the nine possible locations for a page around the circumference of the drum. This division into nine queues allows more efficient use of the drum by reading the pages in the same order they appear on the drum.

This process of ordering the page reads and writes by the position of the page on the drum is usually called "slot sorting" and works as follows. Since the drum is a rotating storage medium, there is a limited set of pages that can be read or written at any given time: the set that is under the read/write heads at that time. In the case of the drums used with the 360/67, there are 9 rotational positions at which pages start on the drum and 100 pages at each position, i.e., there are 9 "slots" on the drum and each one contains 100 pages. Whenever a page read or write is completed for slot "n", the pages that can be read or written soonest are those in slot "n+1" (or 1 if "n" is 9). By placing the read requests on 9 separate queues by slot, the PDP can build channel programs to take advantage of this fact. Also since a page can be written anywhere that another page is not already stored, the PDP will allocate a place for a page to be written in some slot that is not otherwise being used in the channel program that it is constructing.

If the page is not currently stored on any auxiliary storage device (i.e., it has never been used before) an SVC GETRP will be executed immediately to attempt to get main storage space for it. If this is successful the PCB will be placed on the PICQ to indicate that it is now available, but if not it will be placed on a special local queue in the PDP so the call to GETRP can be repeated later.

When all PCB's received from GETQS have been processed (by either placing them in a local queue or calling GETRP) the PDP will build a channel program for a drum to read a

page from each of the nine positions ("slots") for which there is an outstanding read request, first calling GETRP to obtain a main storage block to read each page into.

If there is not sufficient main storage available to allocate a block to read into, GETRP will refuse to allocate a block for the PDP and no more read requests will be added to the channel program. In this case, since main storage is almost full, GETWP will return pages to be written and main storage will become available. This situation occurs very infrequently in actual operation because a few pages are removed from main storage whenever it is in danger of approaching this point.

When all slots that have read requests have been filled, the PDP will call GETWP to attempt to get enough pages to fill all remaining slots with writes. The supervisor may decide to not give the PDP as many as it asked for if there is enough room in main storage, but whatever pages are received will be written by the channel program constructed by the PDP.

If a page to be removed from main storage has not been changed since the last time it was read from an auxiliary storage device and the copy on the auxiliary device is still valid, it will not be written again but instead FREERP will be called immediately to release it and GETWP will be called again to get another one to replace it in the channel program.

A channel program is constructed for a disk to read or write a single page at a time. No attempt is made to read or write more than one page in a single channel program for a disk.

When all write requests have bee processed, the completed channel program will be queued for execution on some available path to the correct device using the standard input/output SVC's. When an interrupt occurs indicating that the channel program is complete the PDP will scan the PCB's used in constructing it and call FREERP to release the main storage blocks for all pages written, while putting all PCB's for pages read on the PICQ. The supervisor will notice that there are pages on PICQ the next time it is entered and will restart the tasks that were waiting for them. Meanwhile the PDP will repeat the process of getting queues, constructing channel programs, and starting them.

Note that the process of building channel programs goes on in parallel with the process of handling interrupts from earlier channel programs and that at any time there will be page requests in every stage of completion.

If a page that is being written on an auxiliary device is required in main storage before FREERP is called to release it, the copy already in main storage will be used and the PDP will be notified when it calls FREERP that the copy on the drum is not valid. This saves the time required to read the page into main storage in this case. This should not happen very often if the the algorithm used to select pages to be written is satisfactory.

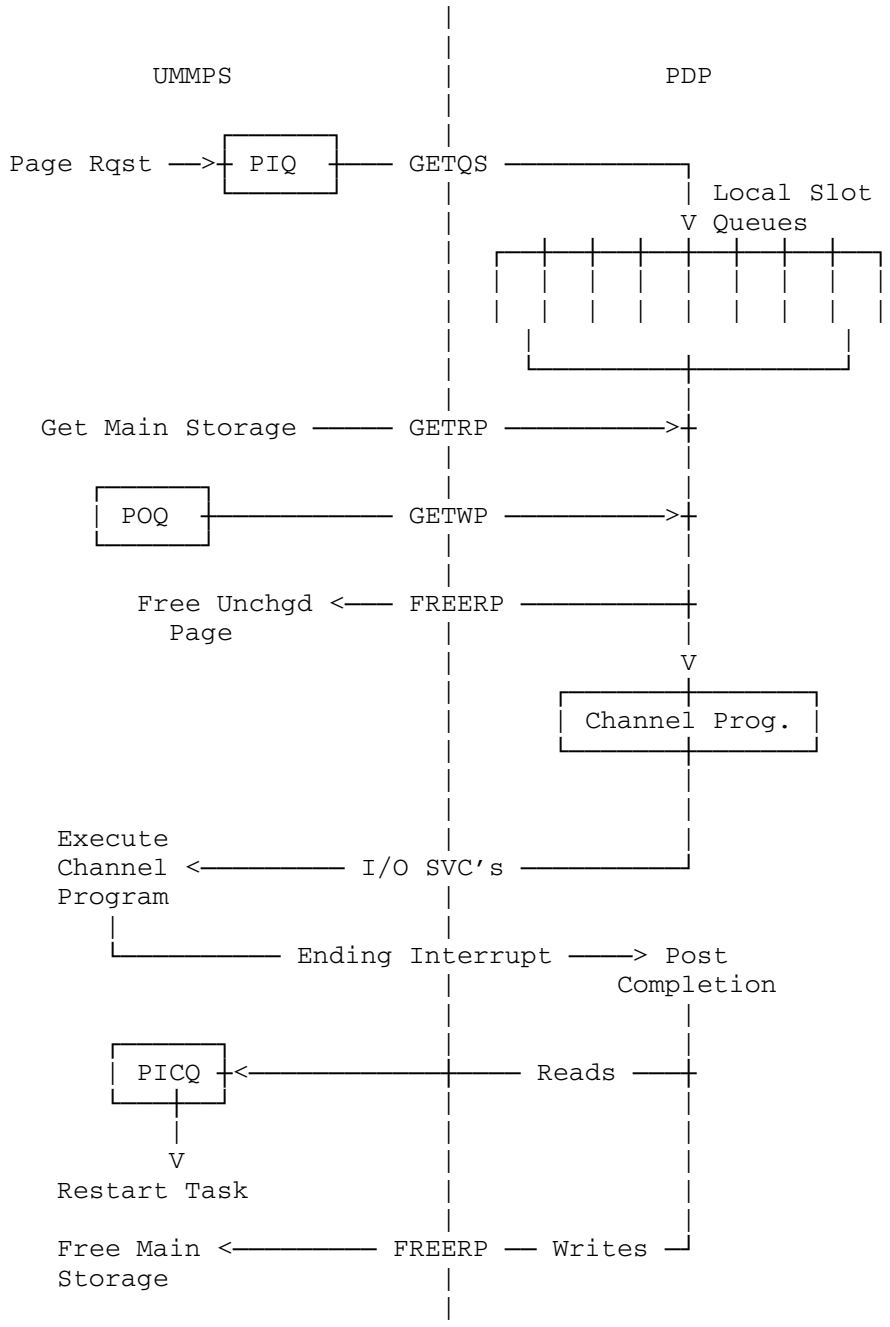

Figure 3: Relation of PDP and UMMPS

In summary, a page is transferred to main storage in the following manner:

1. The supervisor places the PCB for the page on the PIQ.

2. The PDP calls GETQS and obtains that PCB and all others which are on the PIQ or RPQ.

3. The PDP places the PCB on a internal queue corresponding to its auxiliary storage location.

4. If the page is on a drum, the PDP constructs a channel program to read the page together with pages from every position on the drum for which there is a read request, filling in the other positions with writes. If it is on a disk, the PDP constructs a channel program to read that one page.

5. The channel program constructed in 4 is executed on some available path to the device by the supervisor.

6. When the channel program is completed, the PDP places the PCB on the PICQ.

7. The supervisor notices that the page is on the PICQ and restarts the task that was waiting for it.

A page is removed from main storage by the following process:

1. The PDP calls GETWP to request pages to fill up holes in a channel program constructed to read zero to eight pages from a drum or to request a page to write to an idle disk. A page will be written to a disk only if all drums are completely full.

2. If main storage is almost full, the supervisor will take certain pages from the POQ and give them to the PDP.

3. If the page has not been changed since the last time it was written on an auxiliary storage device, FREERP will be called immediately to free the main storage block since the copy on the auxiliary storage device is still valid.

4. If the page has been changed since the last time it was saved in auxiliary storage, the PDP will include the page in its channel program.

5. The channel program will be executed as in 5 above.

6. When the channel program is completed the PDP will call FREERP to release the main storage block occupied by the page.

An examination of the process above will show that the critical step in determining overall performance is the subroutine GETWP, which must decide which pages to remove from main storage. This is the case since it is just those pages which will have to be read if they are subsequently needed by some task.

The following algorithm is used for this function:

1. When a page is brought to main storage it is placed on the top of the POQ if it is eligible for removal from main storage.

2. When GETWP is called it first checks to see if the amount of free main storage exceeds a system parameter, and if so it returns no pages to be written.

3. If there is relatively little main storage available GETWP starts at the top of the POQ looking for pages that can be removed from main storage.

4. Before it removes a page it tests to see if the bit is on which indicates that the page has been referenced and if it is (meaning that some processor or input/output device has referred to the page since the bit was last reset), GETWP will reset the reference bit and put the page on the bottom of the POQ. Note that when a page is first read and placed on the top of the POQ its reference bit is set and hence it will not be removed by GETWP the first time it is scanned.

5. If the reference bit is not on, the page is removed from main storage.

6. GETWP will continue looking at pages until it has as many as were requested or until there

are no more on the POQ.

This use of the information concerning whether a page has been referenced allows those pages which are being used to be kept in main storage while the ones which are idle will be removed. Multics makes use of the same information in a somewhat different way. This information was not originally used by UMMPS (the oldest page in main storage was removed each time) and when it was first utilized, the improvement was quite dramatic.

6.2 Multics Paging Algorithm

The processing of page requests by Multics is similar in spirit to that of UMMPS but different in detail. When a page must be transferred to main storage, a subroutine is called which first calls a subroutine to allocate main storage block to contain the new page and then calls a subroutine similar to the PDP which processes the actual read request. If the subroutine which must allocate main storage determines that a page must be removed to make room available, it will select a page according to an algorithm nearly the same as the one GETWP in UMMPS uses and then call a subroutine which will accomplish the removal. The processing of the actual input/output is handled by a set of subroutines which run in several tasks and operate much the same way as the PDP.

There are at least two important differences between Multics and the other systems being considered. First the auxiliary storage available to Multics is used as the medium of storing all user and system data as well as a mechanism of relieving main storage. This is done by considering all data to be stored in segments which may be attached to any authorized task. When a segment is removed from main storage, the location in auxiliary storage to which it is moved is permanently recorded by the system so that if the user requests it again at a later time it can be retrieved easily. This means that the mechanism for allocating and accounting for auxiliary storage is somewhat more complicated than in any of the other systems.

The second difference is that an attempt is made to determine which pages each task is using most heavily at any time. Certain information about the last 200 pages read into main storage for each task is recorded at all times and periodically three subsets are selected from among this set of 200 pages on the basis of this information.

The "purge set" is made eligible for removal from main storage by reseting the reference bit for each page in it.

The pages in this set are presumably those that the task does not need very much.

The size of the "working set" is used as an estimate of the amount of main storage tha task needs to execute effectively. The individual pages in the working set are not treated in any special way.

The "pre-page set" contains those pages important enough to be moved to main storage as soon as the task becomes eligible for use of a processor.

The information recorded for each of the last 200 pages moved to main storage includes the reference and change bits, the current and permanent location of the page on external storage, and whether it has been read more than once in the last 200 reads for this task.

Note that this procedure assumes that each task will periodically have no pages in main storage, a situation that will normally occur only in moderate to heavy paging. If certain pages are more or less permanently in main storage because not enough paging is being done to flush them out, they will never appear among these 200 pages. However they will probably be among the most heavily used pages and should be included in the working set and perhaps the pre-page set.

6.3 CP/67 Paging Algorithm

The processing of page requests by CP is similar to but more limited than both UMMPS and Multics. As in Multics the main storage is allocated at the time the request is generated and the subroutine that does this will also initiate writes if necessary. Unlike UMMPS or Multics, CP will limit the number of reads that may be outstanding at any time and if more are requested they will be defered to keep from overloading the system.

One of the differences between CP and UMMPS or Multics is the method of choosing which page to remove from main storage. This algorithm underwent a major change in CP some time ago. Both the old and the new algorithm will be described. In the old algorithm the blocks of main storage were scanned looking for a page to remove in the following order of priority:

1. Vacant block

2. Unreferenced and unchanged page

3. Unreferenced and changed page

4. Referenced and unchanged page

5. Referenced and changed page

In addition each main storage block had a "FIFO" flag which was set when it was allocated, and no page in a main storage block with this flag on was removed. If all blocks had the FIFO flag on, they were all turned off and the scan was repeated. The flag was an attempt to keep from allocating a main storage block to a task and then immediately releasing it to another task, a function which is served in UMMPS and Multics by having the pages eligible for removal on a queue ordered in part by when a page was allocated.

The more recent CP algorithm (current as of July, 1969) works as follows. When a block must be allocated in main storage, a round-robin scan of the blocks in main storage is commenced, starting with the block after the last one allocated. This scan will make up to two passes over all the blocks in main storage looking for a block that can be allocated to the page to be read in. On the first pass any block that is available or that contains a page belonging to a task that is not on either of the two processor queues (see section 7.1) will be selected. If this pass does not find a block into which to read the page, the next pass will select the first block that is not permanently allocated to some specific page. This has the affect of looking for a block that is not very important first, but if no such block can be found, take any one.

This change to the paging algorithm was made because it was felt that the older one required too much processor time in the supervisor to implement. This may have been the case, but it seems that the new one is so primitive that serious paging problems can be expected in any but a very light load.

Another important difference between CP and the other three supervisors is that the auxiliary storage address of a page in CP is fixed (as are the number of pages assigned to each task) when the task is created. This means that when a page is removed from main storage it is always written back in the same place and no optimization of drum or disk transfer can be made by filling in write requests around reads. Similarly each read in CP is handled as a separate request and no ordering on the basis of rotational position on the drum or disk is done.

A further distinction is that there is no task involved in the process; all the work including actual input/output

operations is done in the supervisor.

6.4 TSS Paging Algorithm

The algorithm used to control paging in TSS is almost completely different than in any of the other system. The most important difference is that unshared pages are removed from main storage only when the task owning them is removed from consideration for use of a processor, that is, when it reaches a point know as Time Slice End. This may occur because the task has used up its allotment of processor time or for any of several other reasons to be described in Section 7.4. At this time the pages belonging to the task which are in main storage are written to auxiliary storage (if they have been changed since the last time they were written) and the main storage is made "pending". This means that if another page needs the main storage it will be released for the use of the new page, but if the original task makes a request for the same page again before the storage has been reused then it will be reallocated to it. In actual practice there are several pending lists for different categories of pages so that the ones most likely to be reclaimed will be left around.

If a page is requested that can not be found on a pending list, the request is first passed by the queue scanner to a subroutine that attempts to find main storage block for the page. This subroutine first looks for an unassigned block, then for a pending block in each of the categories. If there is too little main storage available, it will call a subroutine that will write out some pages that are shared between two or more tasks and which have not been referenced, or if this does not produce enough storage it will write shared pages which have been referenced. If this fails it will then attempt to force a time slice end for some other task, thereby forcing its pages to be removed from main storage; or if this is not possible it will force a time slice end for the task requesting the new page, thus forcing it to give up some of its own main storage for the new page.

Certain tasks may be forced to give up main storage blocks when they accululate more than a certain number of them. This is called "page stealing" and allows a task to reference a larger number of pages in one time slice without being forced to time slice end.

Periodically (every n time slices) the system will write out shared pages that have not been referenced, even if main storage is not full.

This method ties the scheduling of page transfers closely with the scheduling of processor time since the primary reason for writing pages is that the task owning the pages has temporarily lost its access to a processor. This is the only case among the four systems in which there is a close connection between page scheduling and processor scheduling although there is some connection in each of them, if only implicitly.

TSS recognizes two levels of auxiliary storage and will attempt to assign each page written to the catagory most appropriate to it. If the faster and smaller storage (drum storage) becomes nearly full, the subroutine responsible for allocating auxiliary storage will select an inactive task for "migration," which means that its pages which are on the drum will be moved to slower (disk) auxiliary storage. The pages of a task that waits for a console interraction will automatically be migrated at that time.

7. PROCESSOR SCHEDULING

The method used by these systems to schedule the use of processors differs radically from system to system. In spite of this, all of the systems have certain goals in common with regard to processor scheduling. Stated in the most general terms these are to allow the task requiring a small amount of processor time to be serviced quickly while requiring that all tasks with the same status get about the same amount of time. A secondary consideration in all the systems is to make sure that no tasks waiting for processor time must wait too long for it. The algorithms used by each of the systems to achieve these goals will be discussed, with the simplest being considered first. One thing common to all of them is that there is no fixed priority among tasks such as is found in systems (e.g. OS/360) which cater to batch or real time operation.

7.1 CP/67

At the same time that the paging algorithm of CP was changed, the processor scheduling algoritmh was also changed. The old algorithm will be described first followed by a desription of the new one.

The old algorithm used by CP to schedule the processor (CP is unique among these systems in that it will handle only one processor) is the simplest and most standard — as much as any scheduling algorithm can be considered standard. This algorithm is quite similar to the one used by the Compatible Time Sharing System [9] of Project MAC and MIT. A task running under CP was in one of three states with respect to processor scheduling:

1. Running — task is the one currently running.
2. Ready — task could be running if no other task was running.
3. Waiting — task is waiting for some event to occur.

All tasks were on one of several queues each of which represented a level of priority in the race for a processor. The maximum level queue a task could occupy was set when the task was created and the task was put on top of this level whenever it completed an interraction with a console. This attempted to assure that a task which is interracting with a console would get processor service fast enough to keep the user happy.

Whenever the processor was to be given to a task, the queues were scanned from highest priority to lowest looking for a ready task. Within any queue the scan was round-robin; that is, the task next considered was the one after the task last considered. If a ready task was found it was given the processor after first setting up a timer interrupt to occur after an amount of time depending on the queue which the task was in and the amount of processor time it had already used while in that queue.

When the tasks allowed time in a queue had expired, the task was moved to the next lower queue if it was not in the lowest queue already. Hence, the more time the task used the lower its priority. To make sure that no task got completely left out for a long time, a scan of all queues was made every fifteen seconds looking for a ready task which had not received any processor time since the last scan. If such a task was found it was moved to the next higher queue unless it was in the highest queue allowed for that task.

There was some feed-back from processor scheduling to storage scheduling in CP: periodically (every minute) the percent of the time the processor had been idle while at least one task was waiting for page transfers to be completed was computed. If this number was too small, the maximum allowed number of simultaneous page reads was increased; if it was too large, the maximum number was decreased unless it was already at a set minimum. This was an attempt to keep the system from becoming completely paging bound.

It is easy to see which aspects of this algorithm attempted to meet each of the goals set forth above. A task completing an interraction with a console was placed in a queue in which it would receive a relatively small amount of processor time quickly, but if it needed more time it would be moved to a lower priority queue where it would not interfere with other tasks needing a small amount of time. Any given queue was scanned round-robin which attempted to guarantee equal service to all tasks in that queue. Furthermore any task which got no processor time at all for fifteen seconds had its priority increased so it would have a better chance the next fifteen seconds.

The newer scheduling algorithm used by CP (again current as of July, 1969) uses two queues. Q1 contains all tasks that are not putting a heavy processor time load on the system, while Q2 contains all tasks which require a large amount of processor time. The number of tasks that can be in either of these queues at any time is limited so that there are two other queues of tasks waiting to get into

either Q1 or Q2. This leads to 5 possible states for a task:

1. In Q1
2. Waiting to get into Q1
3. In Q2
4. Waiting to get into Q2
5. Dormant and not requiring a processor

A task may be in Q1 or Q2 even though it is not currently runnable, for example if it is in page wait. When the processor is to be given to a task, a runnable task in Q1 is given first priority, followed by a runnable task in Q2, except that if Q1 is not full a runnable task waiting to get into it may be moved to Q1. A task is allowed 0.4 seconds of processor time in Q1 before it is moved to Q2 and it is allowed 5 seconds of time in Q2 before it is removed from the queue to allow another task to be moved into it. A task is put back into Q1 (or waiting to get into Q1) whenever it does a read from the console. These operations on the queues in CP is illustrated in Figure 4.

The processor scheduling algorithm of CP was changed for much the same reason as the paging algorithm was changed: to reduce the amount of processor time required by the supervisor. The new processor scheduling algorithm seems to be similar to the old one, but not so general, and it probably contains many of the good features of the old one with less overhead.

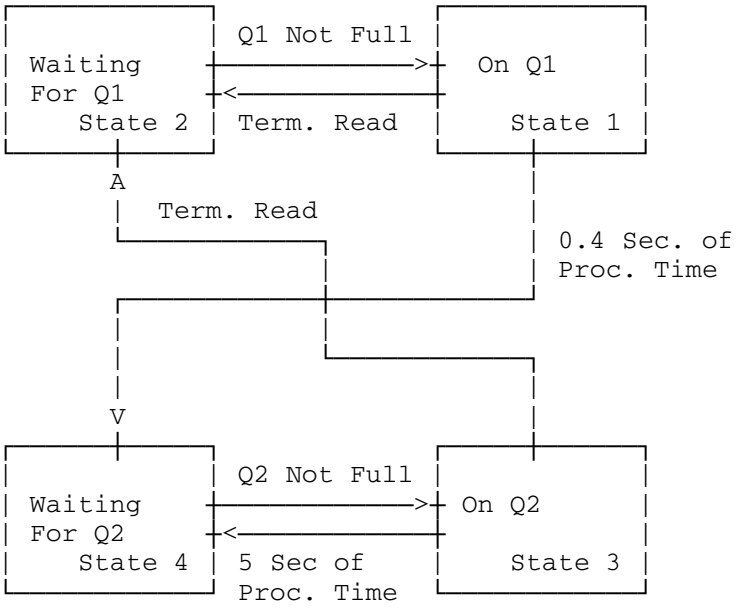

Figure 4: CP/67 Processor Scheduling States

7.2 Multics

The processor scheduling algorithm of Multics is similar to the old algorithm of CP, but it is somewhat more complex. There are a set of queues which contain tasks which could use a processor if one were available, and a task enters at a high priority queue and works down to a lower priority queue as it uses more time. However, two new concepts are added to this: a task may or may not be loaded and it may or may not be eligible. The first condition pertains to whether the pages needed to run the task are present in main storage and is really only an incidental detail as far as processor scheduling is concerned. The eligibility of a task is very important for processor scheduling and will be considered below.

A task in Multics may be in any one of five execution states:

1. Running — task is currently using a

processor.

2. Ready — task could use a processor if one were available.

3. Waiting — task is waiting for some event for which notification will be broadcast.

4. Blocked — task is waiting for specific notification of some event.

5. Stopped — task is no longer in competition for a processor and is in a state in which it can be removed from the system.

In addition there are three loading states

1. Loaded — task has enough information in main storage to be run.

2. Being loaded/unloaded — task is either being loaded or being unloaded.

3. Unloaded — task has none of the required pages in main storage.

Finally there are two eligibility states

1. Eligible — task is to be considered for a processor.

2. Ineligible — task is not be considered for a processor because there are too many tasks already vying for them.

It is important to note the relationship between these states: the execution states and loading states are independent except that a running process must be loaded, and the eligibility state applies only to running, ready, or waiting tasks. There is an upper bound enforced for both the number of loaded and the number of eligible tasks.

All ready ineligible tasks are kept on a set of queues which are used in the same way as in the old CP algorithm. A task is put on a high priority queue after an interraction with a console and moves to lower priority queues as it uses more processor time. The amount of processor time allowed on each queue is greater than the amount of time allowed on the immediately higher priority queue. The amount of processor time allowed a task on the highest priority queue is called a "quantum" and the amount of time allowed on queue n is 2*n quanta. Each task has a lowest and highest

allowed queue number which establish its general range of service.

The most important difference between this scheduling algorithm and the old CP algorithm is the concept of eligibility. The number of eligible tasks in the system is limited to reduce the contention for main storage. This is done by computing the working set size for each eligible task as described in section 6.2, and by requiring that the total size of all working sets of all eligible tasks can be no larger than a certain value. A task will be made eligible if it is the highest priority ready, ineligible task and the total size of all working sets of eligible tasks is less than the maximum. It will maintain its eligibility until (1) it uses a specified amount of processor time, (2) it enters the blocked or stopped (but not wait) state, or (3) it is pre-empted (see below).

There is a priority associated with each eligible task such that the task that has been eligible longest has highest priority. This means that a task will receive better and better service the longer it is eligible, until it has used the maximum processor time allowed. This should reduce paging by allowing tasks to accumulate more pages in main storage as they remain eligible longer.

When a task is removed from the eligible state, the last 200 pages read into main storage for it are divided into the three subsets described in 6.2.

When a task is made eligible it will be loaded if it is not loaded already, hence the number of loaded tasks is always at least as large as the number of eligible tasks. A task will be unloaded if an eligible task must be loaded and there are the maximum number of loaded tasks. Finally a task is selected for use of a processor only if it is the highest priority ready, eligible, and loaded task.

7.3 UMMPS

The processor scheduling algorithm of UMMPS is somewhat similar to that of CP and Multics but is different in several significant details. Instead of the multiple queues for ready tasks only one queue is used and it may contain tasks in any of several states. The basic states that a task may occupy in UMMPS are:

1. Running — currently running on some processor.

2. Ready — could use a processor if one were

available.

3. Wait — waiting on some event.

4. Page wait — waiting for a page to be brought to main storage.

All tasks which are running or ready and some tasks which are waiting will be on the processor queue.

In UMMPS a task may be put into wait state either until any interrupt is directed to the task or until certain bits in some byte of its virtual storage are zero. In the first case and in the second case when it is known that an interrupt will cause the task to be added to the processor queue when the wait is complete, the task is removed from the processor queue during the wait; but if the task will not be notified when the wait is done, it is left on the processor queue and treated as if it were a ready task except that it is not given a processor. This would occur if, for instance, the task is waiting for some byte in shared storage to be reset to zero and whichever task will clear that byte will not notify the task(s) waiting for it to be cleared that it has been cleared. The task will be removed from the processor queue during a wait in the following cases:

1. The wait was initiated by the supervisor itself for an input/output operation or a page read.

2. The byte defining the wait is in the tasks private virtual storage.

3. The tasks specifically requests to be removed from the queue during the wait.

In case (3) it is up to the task to ensure that it is properly notified when the wait is complete. For this purpose there is a supervisor subroutine which may be called by any task and which will place some other task on the processor queue if it is not already there. (The task is not otherwise affected.) There is an overhead associated with leaving a waiting task on the processor queue (because the task must be periodically checked to see if its wait is up) and for this reason all of the more common waits are such that the task is removed from the queue.

Whenever a task is added to the processor queue for any reason it is added to the top of the queue, thus making it the next task to be given a processor. This means that very quick service is usually given to interrupts and to tasks

which have just had a page brought to main storage. It has been observed that a task requesting a page will often request another one soon, and if it is given a processor right away it will select quickly which page is next to be read. A disadvantage of this is that several tasks which each want many pages available in main storage will force the system to continually read and re-read the pages they need. This effect is largely overcome by the privileged task mechanism described below.

Another desirable result of putting new tasks on top of the processor queue is that it is possible for ordinary tasks to control input/output devices fairly efficiently even though this requires rapid interrupt response. An extreme example of this is the PDP which controls the paging drums as described in section 6.1. This task does not receive any special treatment as far as processor scheduling is concerned, yet it can easily keep up with the interrupts from the paging drums.

Each task is alloted a certain amount of processor time (a time slice) and when this time is used up the task is taken from wherever it is on the processor queue and placed on the bottom, after it has been given a new time slice. It is very important to note that this is the only time the task is given a new time slice; it is <u>not</u> given a new one if it goes into wait state for any reason, hence no matter how many times it goes on and off the processor queue it will eventually run out of time and relinquish its place to another task. It is possible for a task to request that it be prematurely forced to the end of the processor queue and given a new time slice, but any waits or page waits by the task do not affect the time slice in progress.

This mechanism obviously partly defeats the advantages mentioned above which allow efficient input/output management by ordinary tasks. This does not seem to be a major problem, but if it were, certain tasks which are known to be input/output limited by their nature (for example the tasks used to drive card readers and printers or the PDP) could be given a very large time slice so they would never be forced to the bottom of the queue.

A further aspect of processor scheduling in UMMPS which was alluded to above is the privileged/non-privileged task mechanism. (The choice of names for this is somewhat misleading since it has nothing to do with what the task is allowed to do, but rather only affects how much processor time and paging the task is allowed.) This mechanism is designed to do what the eligibility mechanism of Multics or the maximum number of concurrent read requests in CP is designed to do; namely, to reduce the possibility of having

too many tasks vying for a processor and main storage. This mechanism works as follows in UMMPS:

1. Whenever a task is initially added to the processor queue it is added as a "neutral" task. This means that no assumption is initially made concerning whether the task will require many pages in main storage or not.

2. When a task accumulates more than a certain number of blocks of main storage it reaches a decision point. The next time it requests a main storage block it is either made non-privileged or privileged, depending on other tasks in the system.

3. If the task reaches this decision point and the number of main storage blocks allocated to privileged tasks is less than the maximum allowed then the following things are done:

   (a) The task is made privileged, meaning that it is allowed to get as many blocks of main storage as it wants.

   (b) The task is given an extra long time slice.

4. If, when the task reaches the decision point, there are already the maximum number of main storage blocks allocated to privileged tasks then this task is made non-privileged. This means that the task is not allowed to have a processor again until some privileged task leaves that state.

5. A task that is privileged remains so until either it uses up its (extended) time slice, it voluntarily asks to be placed at the end of the queue, or it enters wait state except page wait. When a task leaves privileged state it is made neutral, unless it is at time slice end and still has a large number of main storage blocks, in which case it is made non-privileged.

6. When a task leaves privileged state a non-privileged task can now be made privileged.

7. A non-privileged task maintains its place on the processor queue relative to other non-

privileged tasks, and when it is started again it is made privileged, not neutral.

The maximum blocks that will be allocated to privileged tasks and the threshold are set depending on the amount of main storage available.

For the purpose of counting the main storage blocks allocated to privileged tasks, each such task is counted as having a number of blocks equal to the threshold if it temporarily has less than that number. This limits the total number of privileged tasks to the maximum number of pages for privileged tasks divided by the threshold. This mechanism is required because when a non-privileged task is made privileged (e.g., because a privileged task used up its time slice) it will often have few or no pages in main storage.

In addition to the queue mentioned above which controls which tasks may use a processor at any time, there are two queues for each task which control that task's use of a processor: the local processor queue and the wait queue.

The local processor queue is used to control task interrupts and each entry contains information about the tasks status at some point in its execution. Each time an interrupt is passed to a task, this queue is pushed down and a new entry is added to the top, and when a task is given a processor the information in the top entry is used to start the task executing.

The wait queue parallels this local processor queue and contains one entry for each level of the queue at which a wait condition is outstanding. This means that a task can wait on some event, be interrupted for something else, can wait on something at that level, and when it returns to the original level the first wait will still be in force.

There are supervisor subroutines callable by the tasks to remove the top entry from the local processor queue (return to the point of an interrupt) and to remove all levels below the top (throw away return information). It should be noted that these queues do not play any part in the scheduling of processors to tasks, but only what happens when a task gets a processor.

7.4 TSS

TSS is the only one of these systems to have a table driven processor scheduling algorithm. The state of each task relative to processor scheduling is defined by an entry

in a table called the schedule table. Since there can be up to 256 entries in the schedule table any task can be in any of up to 256 states. A schedule table entry contains several types of information summarized below.

1. The priority of the task in this state. This is used only for assigning a priority in processor scheduling.

2. The processor time allowed before time slice end, expressed as a quantum length and the number of quanta.

3. A quantity known as "delta to run" which is essentially the time the task will wait for a processor without being considered behind schedule.

4. A flag indicating whether the task may be pre-empted.

5. The maximum number of main storage blocks and page reads allowed before the task is forced to time slice end.

6. The maximum time a task is allowed to wait for an interrupt before it is forced to time slice end.

7. The maximum number of page reads that can occur in a quantum before the task is considered paging bound.

8. The next schedule table entry to be used in each of several cases.

As perhaps can be seen already, the concept of time slice end is very important in processor scheduling in TSS. This term refers to any event which forces the task to be removed temporarily from consideration for use of a processor, such as using too much processor time (called normal time slice end), too many pages requests, waiting too long for an interrupt, waiting for console interraction, too little main storage available (see Section 6.4 above), etc.

When a task reaches time slice end its pages which are in main storage will be written onto auxiliary storage unless so few tasks are in the system that this one is run again immediately. In addition if the time slice end is due to a wait on a console interraction, the pages of the task which are on fast auxiliary storage (drum) will be moved to slow auxiliary storage (disk). Since time slice end will

free up some main storage, it is used as described in Section 6.4 by the subroutine which allocates main storage when too little of it is available.

For the purposes of processor scheduling three lists or queues are maintained by TSS: the dispatchable list, the eligible list, and the inactive list. The dispatchable list contains those tasks that are authorized to use a processor; the eligible list contains those tasks that are waiting to get onto the dispatchable list; and the inactive list contains those tasks that are waiting for an interrupt.

The task will be moved from the dispatchable list to the inactive list when a time slice end due to (1) request by the task, (2) waiting on console interraction, or (3) waiting too long for an interrupt. The task will be moved from the dispatchable list to the eligible list for any other time slice end. A task will be moved from the inactive list the eligible list when it receives an interrupt. Finally a task will be moved from the eligible list to the dispatchable list when there are few enough tasks on the dispatchable list, enough main storage is available to hold the pages used by the task last time slice, and the task is the highest behind schedule task on the eligible list. If there is a behind schedule task on the eligible list which can not be moved to the dispatchable list (because too little main storage is available or too many tasks are on the dispatchable list), an attempt will be made to find a lower priority pre-emptable task on the dispatchable list and if one is found it will be forced to time slice end.

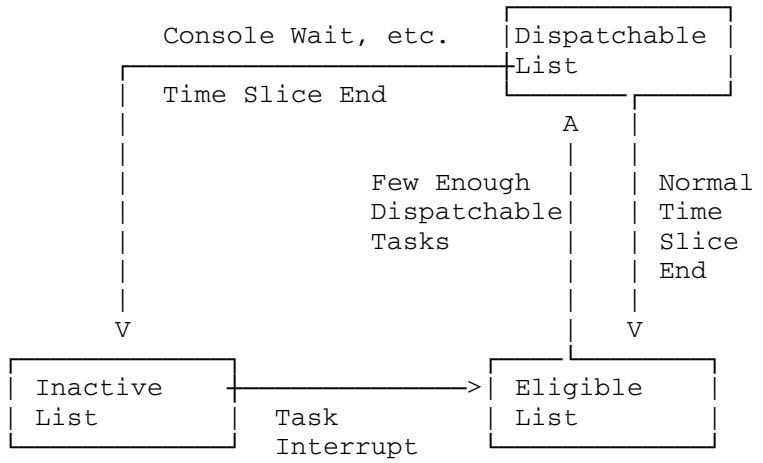

Figure 5: TSS Processor Scheduling Lists

The order of the tasks on the eligible list is determined by the priority from the schedule table entry and a quantity known as the scheduled start time, which is computed from the delta to run in the schedule table entry. The order on the dispatchable list is paging bound tasks first followed by compute bound tasks. The task to get an available processor is the first one on the dispatchable list that is not waiting for an interrupt and has no outstanding page requests.

7.5 Comparison of Processor Scheduling Algorithms

Although the processor scheduling algorithms used by these three systems are quite dissimilar, they do have a number of things in common.

All of them except UMMPS use a number of queues of decreasing priority and increasing processor time for jobs that need a great deal of processor time (assuming that the TSS schedule table has been set up this way). UMMPS achieves somewhat the same effect by accumulating processor time over wait periods as well as running and ready periods, although this is not quite the same thing.

TSS is the only one of the systems that has a close connection between storage scheduling and processor scheduling, the rest of them are willing to let each area take care of itself as much as possible. In spite of this all the systems have some mechanism to keep from getting too

many tasks in main storage at once. Multics does this by limiting the number of eligible tasks, while TSS makes a number of tests on each individual task as well as attempting to avoid starting a task unless enough storage is available to hold all the pages the task used the last time it ran.

The newer CP algorithm sets a fixed maximum on the number of tasks considered for the processor, while the older one limited the number of pages that could be read simultaneously, an approach that was once tried in UMMPS and abandoned in favor of attempting to detect those relatively few tasks that were using an excessively large number of main storage blocks and limiting the number of them that can run simultaneously.

The scheduling algorithm of TSS is the most general since by properly constructing the schedule table, any of a wide variety of algorithms could be implemented, including the ones used by Multics or CP and perhaps the one used by UMMPS.

All in all it can be seen that the parallels between these systems in the area of processor scheduling, while not as great as in the area of storage scheduling, are significant.

8. INPUT/OUTPUT PROCESSING

This section describes the processing of input/output requests and applies TSS and UMMPS since the features of the supervisor being described apply to the 360/67 rather than the 645 and CP does not have the same flexibility in this area.

8.1 Organization of Input/Output Hardware

On the 360/67 the input/output equipment is divided into a three (or sometimes four) level heirarchy. The top level of this heirarchy (if there are three levels) is the channels, which are general devices for transmitting data between main storage and some external destination. The channels are nearly completely independent of the type of device with which they are communicating at any time and any channel may be used to communicate with any device. There are two general types of channels which are distinguished by whether they can handle several simultaneous low speed transmissions (multiplexor channels) or only one potentially higher speed transmission (selector channels). Typically a 360/67 will have about four channels (one multiplexor and three selector) per processor.

The next lower level in the input/output heirarchy is the control unit, which is a unit responsible for interfacing the various specific input/output devices to the channels. The control units handle all of the specific peculiarities of the individual devices so that the channels may be independent of these peculiarities. Whereas there are only two types of channels on a 360/67, there may be many different types of control units, since a different type is required for each type of device interfaced.

The lowest level in this heirarchy is the individual input/output devices. There may be as many as 200 of these in a large 360/67 installation and they may be of as many as twenty different types. Typical input/output devices are card readers, printers, magnetic tape units, magnetic disks, or interfaces to telephone lines.

Whenever the state of any of the components of this heirarchy changes in any significant way, the supervisor is notified by an interrupt indicating the particular channel, control unit, and device (if any) which is affected by the change and the nature of the change. At this time the supervisor can take whatever action is necessary to notify the tasks affected and can also initiate any input/output requests which can now be started. The only other reason for an input/output interrupt is some asynchronous event

which may be of interest to the supervisor, for example a user at a terminal striking an attention key to indicate that he wants service of some kind. This kind of interrupt normally requires no action by the supervisor except notification of the task involved.

The fourth (and highest) level in this heirarchy, which is present in all multi-processor 67's and in some single processor 67's, is the channel controller, which is a unit which interfaces between the channels and the rest of the system, allowing the channels to be independent of the processors. This unit is not generally significant directly to the programming of input/output support, except that without it solving the problem of input/output control in a multi-processor 67 would be much more difficult. This is the case because without the channel controller, each individual channel would interface to a specific processor and only that processor would be able to control that channel or receive interrupts from it. In fact this is the way the multi-processor 360/65 is organized and partly as a result of this the input/output control for that system is both more complicated and less general than that of either TSS or UMMPS. In spite of their importance to input/output programming in multi-processor 67's, the channel controllers are almost completely transparent to the supervisor.

It is important to note that any given control unit may be attached to up to two channels and any given device may be attached to up to four control units. To execute an input/output operation on any device requires the use of a control unit to which it is attached and a channel to which that control unit is attached. Such a combination is called a "path" to the device, and any device can have up to eight paths to it. In the small configuration shown in Figure 6, device D1 has one path, devices D2 and D3 have four paths, and devices D4 and D5 have two paths. The concept of a path is important to an understanding of input/output processing in UMMPS or TSS.

The importance of the channel controllers can clearly be seen by considering that without them the specific paths available to a device would depend upon which processor is processing the input/output request. This is true since without the channel controller only certain channels would be accessible to each processor, and paths to a device passing through any other channels would not be available to that processor. Referring to Figure 6, without the channel controllers the paths passing through C1 and C2 would be accessible to only processor 1, while the paths passing through C3 and C4 would be accessible to only processor 2. This means that processor 1 would have paths to D1, D2, and D3 but not to D4 or D5 while processor 2 would have paths to

all devices except D1. This sort of situation is not at all unlikely to occur without the channel controllers and obviously complicates input/output programming.

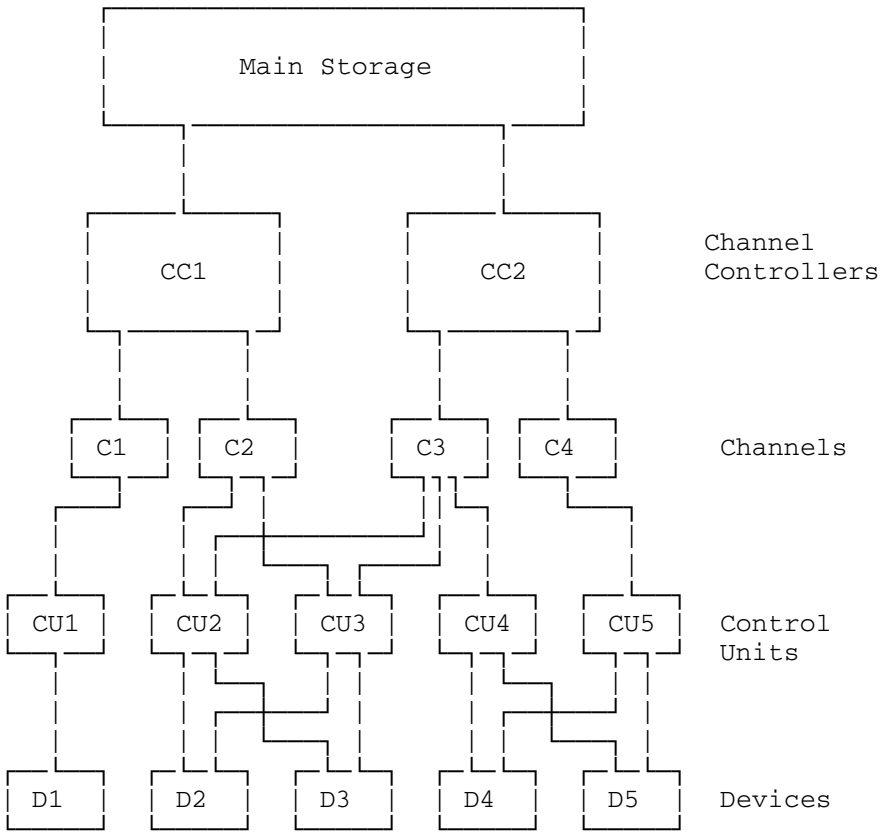

Figure 6: Simple Input/Output Configuration

8.2 Task Input/Output Control

The basic entity in the programming of input/output for the 67 is the device, which is the unit that the task deals with directly. A device may be allocated to a task by the supervisor, either with exclusive control or to be shared with other tasks, and from that time until the device is released by the task (either voluntarily or forcibly) the task is allowed to control it through calls to appropriate

supervisor subroutines. In any case nothing will be done to or with the device unless the task specifically requests it to be done, and all recovery from abnormal conditions is the responsibility of the task. This contrasts with the action of some other systems (for example OS/360) which include some of the device error recovery as part of the supervisor. In UMMPS all input/output is initiated at the request of the tasks except for the operator's console; the supervisor never initiates other input/output on its own.

The basic operation of a task with respect to a device it owns is the queuing of an input/output request for that device. This input/output request is defined by a set of commands to the channel, control unit, and device which will be used to execute the request. The supervisor will maintain a queue of requests for each device and will execute them in the order given when equipment becomes available. At any given time the queue will be divided into two sections: the portion that has been completed but about which the task has not been notified (see below for a description of how the task is notified), and the portion that has not been completed. In UMMPS the first entry on the second portion of the queue is called the "active entry" and it represents the input/output request which the actual device is working on or is about to work on, while in TSS two separate queues are kept. An entry is deleted from the queue by UMMPS when it has been completed with no abnormal conditions and the task has been notified of this.

The task can be notified by UMMPS of the completion of an input/output request in one of two ways depending on the option selected by the task: the task can wait until the current top request on the input/output queue for a specific device is complete or the task can receive an interrupt when the next request on the queue for a particular device is complete (this is the method used by TSS). In either case if an abnormal condition is detected with respect to an input/output request, the active entry in the input/output queue for that device is prevented from moving further (i.e., further operations on that device are not started) until the task has been notified of the abnormal condition and taken some action with respect to it. The actions open to the task at that point in UMMPS are to either ignore the abnormal condition or to save the input/output queue while some recovery is attempted. After the queue has been saved the task may do one of several things:

1. Execute other input/output requests for the device.

2. Retry the request that caused the original problem.

3. Ignore the error after having possibly done other operations on the device.

4. Delete the request that caused the trouble and go on to the next one.

8.3 Supervisor Input/Output Control

The supervisor's main responsibility with regard to input/output processing is to schedule the operations requested by the tasks on the available paths to the devices. In order to execute an input/output request, the supervisor must be able to allocate a channel (possibly including channel controller) and a control unit to the request. If the channel used is a selector channel, then the device must have exclusive use of the channel and control unit; but if a multiplexor channel is used, then it usually need not have exclusive use of the channel. However it is necessary to guard against overloading the multiplexor channel by executing too many operations on it simultaneously, and in some cases a control unit attached to a multiplexor channel will not support more than one operation at a time. Considerations such as these make it much more complicated to decide when a path to a device is free and when it is busy. Also complicating the problem is the fact that there may be several possible paths to a device each of which shares equipment with some paths to other devices.

In the face of this complexity the supervisor's job is to make sure that no section of the input/output equipment sits idle when it could be working and that no input/output request waits longer than necessary before being started. Some devices require better service than others and if two devices are competing for the same channel, the one with the higher priority will be given first consideration, while within a particular priority level the scheduling is round robin to give each device equal service.

The general strategy used to implement these goals is to attempt to restart any idle part of the input/output equipment whenever an interrupt occurs indicating a change in its state. However because of the very large number of interrupts and the number of devices, it is not practical to look at each device whenever an interrupt occurs. The method used to avoid this will be set forth below.

An additional fucntion that the supervisor performs with respect to input/output processing is the relocation of channel programs. Any channel program passed to the supervisor for execution by a (relocatable) task will

contain relocatable addresses that must be translated into absoulte addresses required by the input/output hardware before the channel program is executed. In addition, the pages referenced by the channel program must be "locked" into main storage for the duration of the channel program so that the input/output equipment can refer to them.

Since locking these pages into main storage represents a significant demand on a scarce resource, the supervisor attempts to delay doing this as long as possible and to release the main storage blocks as soon as possible. A channel program will be translated into absoulte addresses and its pages locked into main storage only when it is the next channel program to be executed on a device that is idle, and the pages will be unlocked as soon as the channel program is completed.

Before considering the algorithms involved in scheduling the input/output equipment it is necessary to be aware of the tables maintained by the supervisor for this purpose. These tables fall into 4 general categories in UMMPS (and similar tables exist in TSS):

1. Device oriented tables contain the status of each device in use by any task, including the input/output queue for the device. In addition there is an indication of which channels the paths to the device start from.

2. Control unit tables (one per control attached to the machine) contain status of the control unit in addition to a list of the devices which are attached to the control unit.

3. Channel tables (one per channel attached to the machine) contain status of the channel and a list of the control units which are attached to the channel.

4. Interaction tables indicate which paths may interract with each other, i.e., which paths share components, so that if an interrupt indicates that a particular channel or control unit is now free, it is possible to determine which paths to which devices may now be free. Included with each such table is a count of the number of requests for input/output service that have not been filled yet within the group of paths indicated by this table. This count is included so that unnecessary scans of the input/output tables may be avoided, thus

reducing the overhead in the supervisor.

Note that more than one control unit table may point to a given device table and more than one channel table may point to a given control unit table.

When an interrupt occurs in UMMPS indicating a change in the state of the input/output equipment, the first order of business is to find out which components of the input/output equipment are now free and update the appropriate tables to indicate this. Then UMMPS performs any functions required to notify the task of the state of its input/output requests. When this is completed UMMPS goes to a section of code which attempts to start any pending input/output requests which were waiting for a component which is now free. (This is the same section that is executed whenever a new request is added to a device queue.) The operation of this section is as follows:

1. From the interraction tables find the list of all channels which could have been affected by this interrupt.

2. If the count of pending requests on these channels is zero quit.

3. Find the first (next) free channel in this list. If there is none quit.

4. Find the first (next) free control unit attached to the channel under consideration. Go to step 3 if none exists.

5. Scan for an idle device which has a request that needs to be started and which is attached to this control unit. This scan is round robin, i.e., it starts each time with the device after the last device considered the previous time this control unit was inspected. If no device needs service, go to step 4. If a multiplexor channel is being restarted, there will generally be a specific device to restart and only that device will need to be considered.

6. Attempt to start the operation pending on the device.

7. If the operation started then (a) if this is a selector channel go to step 3, or (b) if this is a multiplexor channel then quit if

only one device needs to be considered (the usual case) or go to step 5 if the control unit supports multiple operations and step 4 if it does not. The only time that it is necessary to start more than one operation on a multiplexor channel is when the channel was quiesced earlier for some reason (e.g., to allow a burst mode operation to be executed).

8. If the device is busy in spite of what the tables say (unlikely) then go to step 5.

9. If the control unit is busy then go to step 4. This is more likely to happen because the control units are not very consistent about notifying the supervisor when they are not busy. Also it may be necessary to repeat this whole process if a control unit indicates that it is busy but refuses to present an interrupt when it is free again (something which is allowed).

10. If the channel is busy (very unlikely) go to step 3.

11. If nothing is busy, but some status was presented by one of the components involved, then leave this section and go process the status as if there had been an interrupt.

This process is somewhat more complex than indicated to allow for such things as control units which are connected to two channels, but which will operate with only one of them at a time or control units which are connected to a multiplexor channel but require the channel to operate in burst mode (only one operation at a time).

This completes the description of input/output processing in UMMPS. The processing in TSS is similar in spirit except that when restarting the input/output system after an interrupt, it will scan the devices which could have been affected by the interrupt and for each one which has an outstanding request, it will try to find a path on which to execute the request. This is slightly less efficient since each time a path is needed it is necessary to start all over again from the top, while in UMMPS the channel and possibly control unit may already be known. This multiple path mechanism does not exist in Multics or CP; in Multics it is not necessary because of the hardware of the 645 (the actual processing of input/output is quite different and less interrupt oriented) and in CP only one path to any device is used except in a few limited cases.

It is possible to make this simplification in CP since it does not (currently) support multi-processor systems and the multiple paths are most common in that type of system.

9. MULTI-PROCESSOR CONSIDERATIONS

All of the systems being considered here support multiple processors except CP. This section will consider the aspects of the supervisor specifically intended to allow this.

9.1 Organization of Multi-Processor Support

There are at least three ways in which multi-processor support may be implemented: as separate systems for each processor, as a master-slave relationship between processors, or as a symmetric treatment of all processors. Each of these organizations has certain things to recommend it.

In a separate system organization each task in the system is assigned to a particular processor and always runs on that processor. Furthermore all input/output operations are controlled by the processor assigned to the task making the request for the operation and all input/output interrupts are directed to the processor that initiated the associated input/output operation or which controls the task owning the input/output device. Main storage is shared between all of the processors and only one copy of reentrant programs needs to be in main storage.

This organization is inefficient since it is possible for one processor to be overworked while the other ones are idle. This is possible since the work to be done is assigned in advance to a processor and can not be switched from one to another as the load shifts. Furthermore, since each processor has its own set of input/output equipment, there will not be a single pool of auxiliary storage available to all tasks in the system unless special code is included to allow sharing of input/output equipment among independent systems. Without this each user will have to have a processor assigned to him which will have access to the data sets belonging to that user. For these reasons none of the systems being considered use this organization.

In the master-slave organization, one processor is assigned the function of controlling the system and the other processors simply execute tasks, i.e., the supervisor always runs in only one of the processors. This means that the supervisor need not be reentrant and need not be as concerned with problems of multiple processors accessing common tables simultaneously. It is necessary for the processor receiving an interrupt requesting task service (which will always occur in the processor executing the task's program) to direct the interrupt to the processor

executing the supervisor. If there is any idle time in the control processor, it can also execute task programs then.

This organization has the disadvantage of allocating only one processor to the work of the supervisor, which may require more than one processor to accomplish with reasonable response time. Normally not more than one processor is required in the long run for the supervisor, but it is possible for many interrupts to occur in a short time and temporarily overload one processor. Also the processor executing the supervisor is very important to the system and any failure in that processor will probably bring the system down. Again this organization is not used by any of the systems being considered.

The symmetric processor organization treats all processors the same; every interrupt is handled by the processor on which it occurs and every processor executes task programs when it is not executing the supervisor. On the 360/67 interrupts associated with a program occur on the processor executing that program, while input/output interrupts occur on the processor that is least busy at the time. This automatically evens out the load on the several processors by assuring, for instance, that an input/output interrupt will be directed to the processor that is in wait state rather than one that is executing.

The disadvantages of this organization are that it requires the supervisor to be able to execute in several processors simultaneously and to handle the problem of simultaneous access to common tables. In spite of the difficulties all of the three systems use this method of multi-processor support.

9.2 Hardware Considerations

Several aspects of the hardware are important to multi-processor support. One that has already been discussed is the multiple path input/output configuration of the 360/67. The important aspect of this for multi-processor operation is the fact that the input/output programming is independent of which processor the supervisor is executing on. This may seem like an obvious thing, but some current multi-processor systems do not have this ability.

In order to make the supervisor reentrant it is convenient for each processor to have a certain amount of private storage which can be used to contain those things which are private to that processor. On the 360/67 this is done by assigning real page zero (i.e., the page with the real address 0 thru 4095) to a different actual storage

location for each processor. This assignment of page zero is independent of the address relocation described above and occurs after it is complete. Unlike that address relocation, it applies to input/output operations as well as the processors and is fixed rather than specified by tables in main storage.

In order to assure that only one processor is changing common tables at any time, it is necessary to have some way in which the processors can "lock" these tables and assure that no other processor is accessing them. This is done by providing instructions that simultaneously (on one storage cycle) test a storage location and set it to some predetermined value. Since the testing and the setting is done on the same storage cycle, it is not possible for another processor to access the same storage location between the two operations. To use this mechanism a storage location is assigned to each table (or other thing) which must be locked at some time. When this storage location is zero, the table is considered to be "unlocked," and if it is non-zero the table is "locked." If a processor needs to lock the table, it will use the special instruction to test this storage location and "simultaneously" set it. If the storage location is zero, it will be set to some non-zero value and the processor will have locked the table, but if it was already non-zero, the processor will get an indication that the table is already locked and can then either loop until it can lock the table or go on to something else.

In rare cases it is necessary for one processor to directly signal another one. For this purpose there is a method by which a processor can cause an interrupt on another one and a method by which it can start another one no matter what state it is in. In UMMPS the inter-processor interrupt is rarely used and the external start is used only once for each processor.

9.3 Multi-Processor Support in UMMPS

UMMPS is unusual in the respect that it was originally written to use only one processor like CP, but was later changed to use up to four. (It is rumored that a similar change is contemplated for CP.) This section will discuss the changes required to do this.

Only the supervisor portion of the system need be aware that there is more than one processor. This is true since no task will run in more than one processor at any given time, so that from the point of view of a task there is only one processor — the one on which it is running. In spite of

this, a task may switch from one processor to another very frequently, in fact whenever an interrupt occurs in that task. This independence of tasks from multi-processing made the conversion effort much easier since no program executed only by tasks needed to be changed.

The general approach to multi-processor support in UMMPS was to move all temporary and private locations of the supervisor to the storage that is private to the processor and to use locks as described above to guarantee the integrity of any information that needed to be common to all processors. These locks fall into two general categories: those that are set for a short time and on which the supervisor can wait, and those which are set for a long time and for which a queueing mechanism must be provided. In the first category are all locks that refer to input/output devices and tables and all locks that refer to processor and storage scheduling tables. In the second category are those locks which refer to a particular task, since these locks are set whenever the task is executing on a processor. It would have been possible to reset the lock after the task is selected for execution on a processor, but it was decided that it would be easier to leave the lock set.

A further distinction among locks is the type of information they refer to. Some locks refer only to an individual table entry (e.g., a device table entry) while others refer to a global quantity (e.g., all chains of PCB's). The global ones must be used in such a way that they are set for as little time as possible to avoid interference between the various processors, while this is not quite so important with the ones referring to only one specific quantity. There are 16 global locks and about 200 specific locks in UMMPS.

9.4 Inter-Processor Interference

The problem of interference between processors is an important one and must be considered when designing a multi-processor system. This problem can take one of at least two forms. The first is the lock interference mentioned above, while the other is interference in the hardware, primarily in storage accesses. The additional processors will put a rather heavy load on the main storage of the system and unless it is specifically designed to handle this a serious degradation in performance can result.

It is difficult to measure the amount of degradation due to interference between processors, but judging from measurements of overall supervisor behavior on a two processor 360/67, it is not a severe problem. In UMMPS

running on such a system the degradation is certainly less than a few percent and probably less than one percent.

Studies of lock byte interference in UMMPS running on two processors show that it is not a significant problem. Although locks are set very frequently (about 5000 times a second in a two processor system), the time required to set one is very short, ranging from 4.5 micro-seconds to 21 micro-seconds and averaging 6.3 micro-seconds for the 16 global locks.

Any degradation due to interference between processors is more than offset by the benefits resulting from multi-processor operation. These benefits are due to the ability to shift the load from one processor to another as its characteristics change and to share main storage and input/output equipment between processors. Experiments have shown that in many cases it is possible to run effectively far more than twice as many tasks one a two processor system than on half of it. This is particularly true of programs that require a large amount of main storage.

REFERENCES

1. Arden, B. W., O'Brien, T. C., and Westervelt, F. H., "Program and Addressing Structure in a Time-Sharing Environment," Journal of the ACM, 13,1 (Jan 1966), 1-16

2. Bayels, R. U., et al, Control Program-67/Cambridge Monitor System (CP-67/CMS), Program Number 360D 05.2.005, Cambridge, Mass, 1968

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

4. Dennis, Jack B. And Glaser, Edward L., "The Structure of On-Line Information Processing Systems," Proceedings of the Second Congress on the Information Sciences, November 1964, Washington D. C., 1965, 5-14

5. Multics System Programmer's Manual Project MAC, 1968

6. IBM System/360 Model 67 Functional Characteristics, Form A27-2719-0, IBM Corporation, New york, 1967

7. IBM System/360 Time Sharing System Resident Supervisor Program Logic Manual, Form Y28-2012-2, New York, 1968

8. Organick, Elliot I., A Guide to Multics for Sub-System Writers, Project MAC, 1969

9. Saltzer, J. H., CTSS Technical Notes, Project MAC Report MAC-TR-16, Boston, 1965

10. Vyssotsky, V. A., Corbato, F. J., and Graham, R. M., "Structure of the MULTICS Supervisor," Proceedings of the AFIPS 1965 Fall Joint Computer Conference, Part I, Washington D. C., 1965, 203-212