[illegible]

A MULTIPROGRAMMING OPERATING SYSTEM

THE DESIGN AND IMPLEMENTATION OF A MULTIPROGRAMMING OPERATING SYSTEM

by

HOON-LIONG ONG, M.Sc.

A Project
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree
Master of Science

McMaster University
December 1978

MASTER OF SCIENCE (1978)
(Computation)

McMASTER UNIVERSITY
Hamilton, Ontario

TITLE: The Design and Implementation of a Multiprogramming Operating System

AUTHOR: Hoon-Liong Ong, B.Sc. (Nanyang University)
M.Sc. (Lakehead University)

SUPERVISOR: Dr. Y.S. Kwong

NUMBER OF PAGES: vi, 80

ABSTRACT

A multiprogramming operating system (MOS) which is useful in operating systems education and research is designed and implemented in this project. The project includes the simulation of a hypothetical machine on a host computer system, and the design and implementation of a MOS for the simulated computer. A large number of compute-bound, balanced, and I/O-bound sample jobs have been created and run on the simulated system. Statistics are collected to measure the performance of the MOS.

ACKNOWLEDGEMENT

I wish to thank my supervisor, Dr. Y.S. Kwong, for his guidance and assistance during the preparation of this project.

TABLE OF CONTENTS

		Page
1.	INTRODUCTION	1
2.	MACHINE SPECIFICATIONS	3
	2.1 The Virtual Machine	3
	2.2 The Virtual Machine Instruction Set	4
	2.3 The Hypothetical Machine	7
	2.4 Slave Mode Operation	10
	2.5 Master Mode Operation	12
	2.6 Channels and I/O Devices	13
	2.7 Timer	14
3.	THE OPERATING SYSTEM	15
	3.1 The MOS Structure	15
	3.2 Input of Jobs	25
	3.3 Job Scheduling	30
	3.4 Interrupt Handling	31
	3.5 Output of Jobs	37
	3.6 The MOS Statistics	41

4. IMPLEMENTATION DETAILS 44

4.1 Overview 44

4.2 Representation of User Storage, Spooler Buffers, Backing Storage, and CPU Registers 45

4.3 Simulation of I/O Devices and Channels 47

4.4 Simulator for CPU Clock and Timer 51

4.5 Simulator for Paging Hardware 53

4.6 Implementation of Queue Data Structures and Resource Semaphores 53

4.7 Implementation of Processes 59

4.8 Updating of MOS Statistics 63

5. SAMPLE JOB STATISTICS AND THE MOS PERFORMANCE 65

6. CONCLUSION OF PROJECT 73

APPENDIX A: JOB, PROGRAM, AND DATA CARD FORMATS 75

APPENDIX B: PROGRAM LISTING 78

BIBLIOGRAPHY 79

1. INTRODUCTION

Research and development efforts in operating systems have produced a number of useful techniques and concepts. As a result, the subject is now established as an area of study in an academic computer science program. In a course on operating systems, students usually learn the basic techniques and the underlying principles. An operating system project would provide them with an opportunity to consolidate and apply some of the concepts and techniques taught in the course, and to become familiar with the different resources available in a system and their allocation schemes.

To prevent degrading the service provided to other users, students are not allowed to implement their operating system program on a real machine. Instead, a hypothetical machine is simulated on the real machine, and the operating system program will be implemented on the simulated machine. To reduce the cost and the complexity of a project, the hypothetical machine should be kept simple.

To illustrate how an operating system for a computer system can be constructed, a MOS for a simple hypothetical machine is designed and implemented in this project. The MOS has the major characteristics of a small computer system which can support multiprogramming.

The project consists of the following major components:
(1) the simulation of the hardware system,

(2) the implementation of a virtual instruction set, and

(3) the design and implementation of a MOS for the simulated machine.

The hardware system consists of a card reader, a printer, CPU, main memory, channels, paging and interrupt systems, and a drum which is used as a secondary storage device for the MOS designer and implementer. The detailed characteristics and components of the hardware system are described in Section 2.

The main function of the MOS is to process a batch of user jobs efficiently. This is accomplished by spooling input and output and to execute user's jobs in a multiprogramming environment. The main components of the MOS are given in Section 3.

Some details of implementation, such as the simulation of the hardware system, the implementation of resource semaphores, and the representation of queue data structures etc, are presented in Section 4.

To study the performance and response of the MOS to different input job characteristics, a large number of compute-bound, balanced, and I/O-bound sample jobs have been created and run on the simulated system. Statistics collected from these batches of sample jobs are tabulated and discussed in Section 5.

In the final section, some conclusions on the project are presented.

2. MACHINE SPECIFICATIONS

This section presents the configuration of the simulated machine [S1] which can be described in terms of the <u>virtual machine</u> viewed by a typical user, and the <u>hypothetical machine</u> used by the operating system designer. The former machine is called 'virtual' because it may not necessarily be implemented in hardware; it is a machine viewed by a user. For example, the PASCAL language describes a virtual machine. The PASCAL programmer only sees a machine that directly executes his PASCAL program. He is not concerned with the problems of register and storage allocation, I/O interrupts etc. The latter machine is called hypothetical because it is not real. The operating system implementer has to simulate it on an existing computer system and implement the operating system on the simulated machine. We shall call the host computer system which simulates the hypothetical machine a <u>real machine</u>.

2.1 The Virtual Machine

The virtual machine seen by a user is shown in Fig. 2.1. It consists of a CPU, a card reader, a line printer, and the main storage. The machine can execute programs written in a set of instructions given in Section 2.2.

The CPU has a five-byte general register R, a three-byte instruction counter IC, which contains the virtual machine address of an instruction, and a one-byte 'boolean' toggle C, which may contain either 'T' or 'F'.

The main storage consists of a set of 1000 five-byte words, with addresses from 0 to 999.

The card reader can read the first fifty columns of a card into the main storage by an input instruction. The line printer can print a new line of fifty characters by an output instruction. The first character on each line is interpreted as a printer control character with the following meanings:

blank	:	single spacing,
'0'	:	double spacing,
'1'	:	print on top of next page.

2.2 The Virtual Machine Instruction Set

Table 2.1 illustrates a small virtual machine instruction set proposed by Shaw [S1]. With this simple instruction set, a batch of compute-bound, I/O-bound, and balanced programs can be quickly written. For this reason, this virtual machine instruction set was implemented in our project.

Each virtual machine instruction consists of five bytes; the first two bytes and the last three bytes contain the operation code and the operand address of the instruction respectively. The operand address can address a virtual machine memory with a maximum of up to 1000 words.

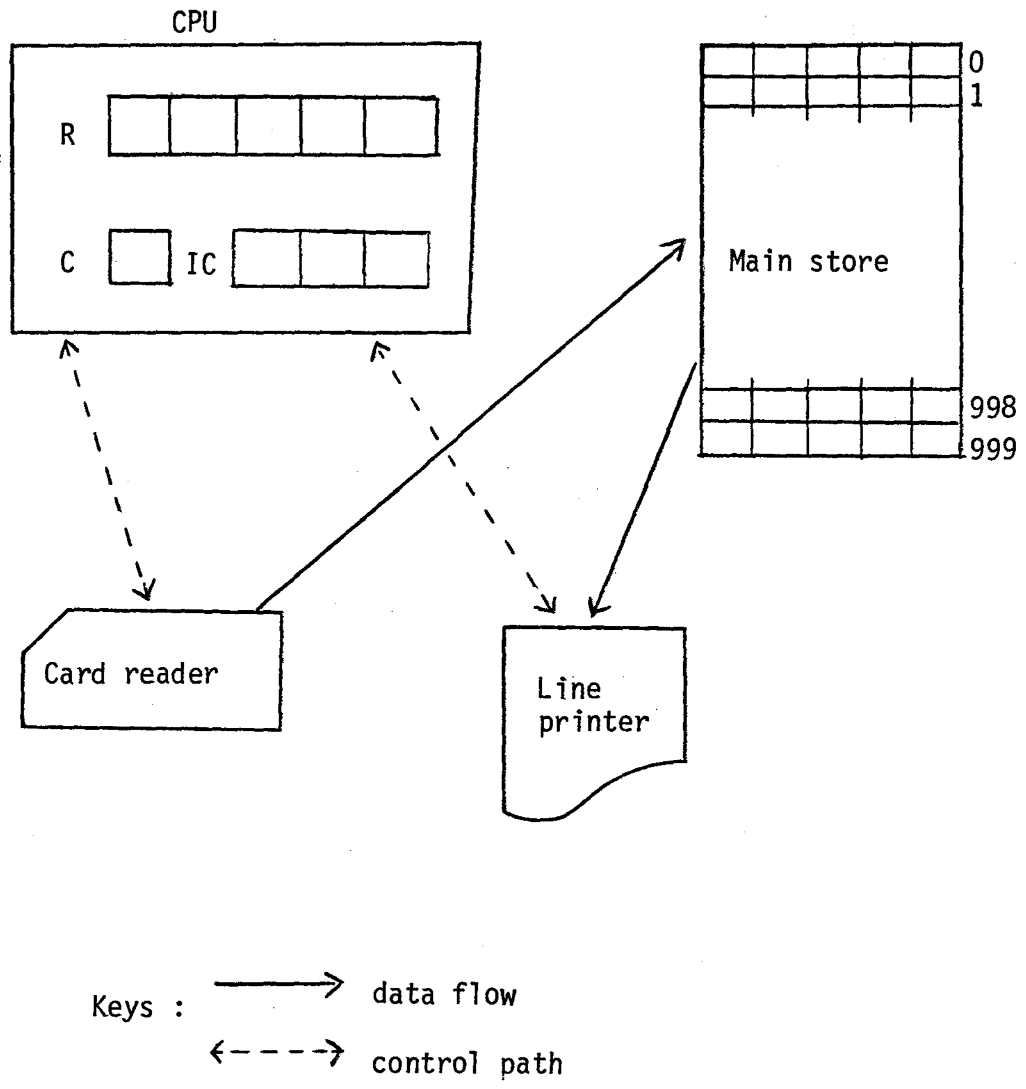


Fig 2.1 Virtual user machine

Table 2.1 Instruction set of virtual machine

Operator	Operand	Execution time	Interpretation
LR	$x_1x_2x_3$	1	R: = [α];
SR	$x_1x_2x_3$	1	α: = R;
CR	$x_1x_2x_3$	1	if R=[α] then C:='T' else C:='F';
BT	$x_1x_2x_3$	1	if C='T' then IC:=α;
RD	$x_1x_2x_3$	3	Read ([β+i], i=0,...,9);
WR	$x_1x_2x_3$	3	Write ([β+i],i=0,...,9);
H		0	halt

Notes:
1. $x_1,x_2,x_3 \in \{0, 1, \ldots, 9\}$;
2. $\alpha = 100x_1 + 10x_2 + x_3$;
3. $\beta = 100x_1 + 10x_2$;
4. [α] denotes the contents of location α;
5. the leading zeros of the operand field can be replaced by blanks.

The I/O operations are performed by the two virtual I/O instructions, RD and WR. The operand addresses of these two instructions are always multiples of ten.

It is assumed that all the compute-type instructions (LR, SR, CR, BT) are executed in one time unit, and the two virtual I/O instructions are executed in three time units, while the halt instruction H is executed in zero time units.

An example of a user program written in this instruction set is given in the Appendix A.

2.3 The Hypothetical Machine.

The hypothetical machine is a virtual machine viewed by the MOS designer. The components of this machine are illustrated in Fig. 2.2. The CPU may operate in either master or slave mode. In master mode, instructions from supervisor storage are processed by the high-level language processor (HLP). Any high level language that is available on the host system can be used. In this project, the language PASCAL was selected. In slave mode, instructions from user storage are processed by the virtual machine emulator which simulates the execution of the virtual machine instructions on the CPU.

The CPU has a time clock, a hardware timer, and a set of registers. The time clock is initialized to zero at system start time and it is incremented by one whenever a time unit has elapsed. The timer will be discussed in Section 2.7.

The CPU registers of interest are:

C : a one-byte 'boolean toggle',

R : a five-byte general register,

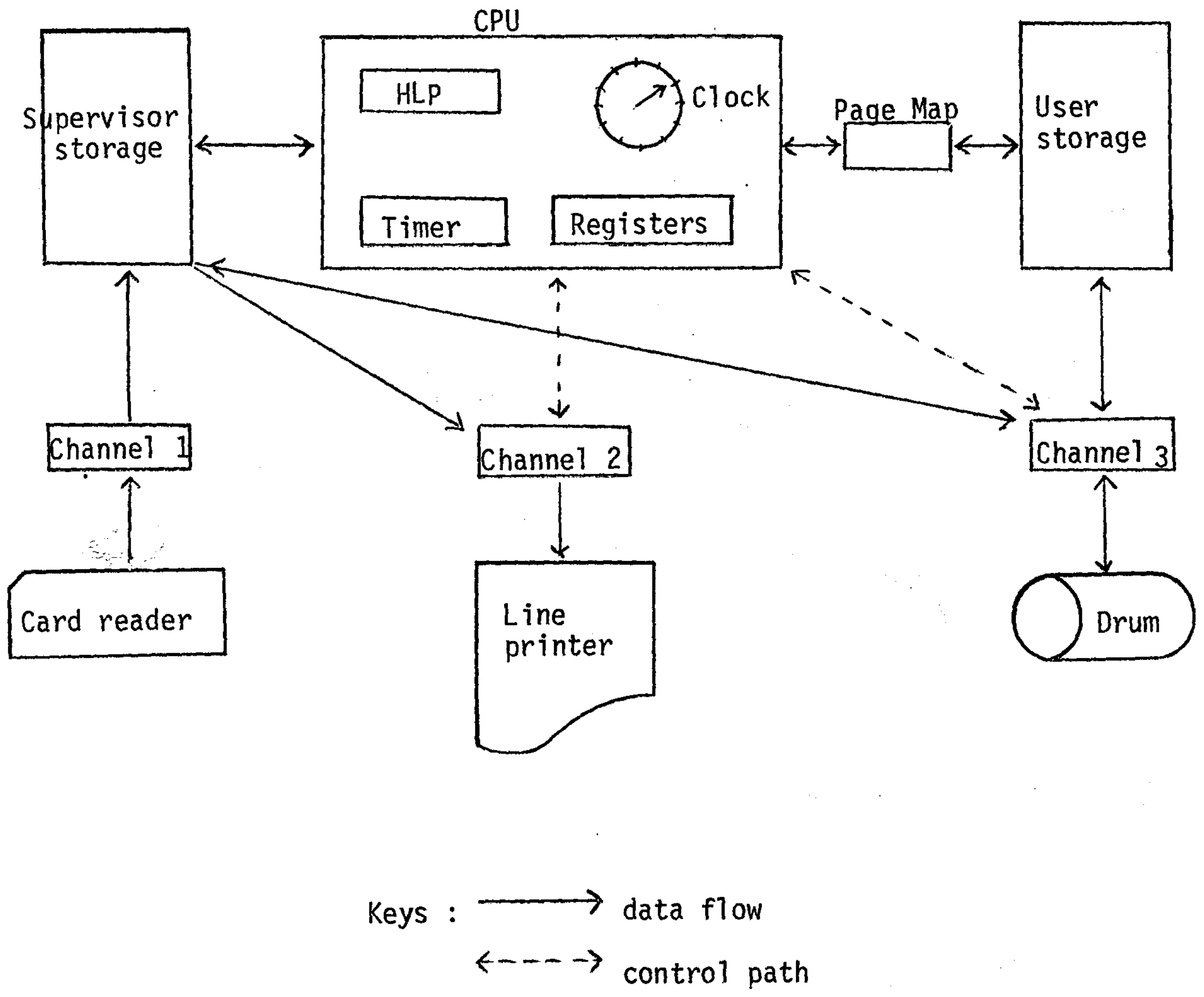


Fig 2.2 The hypothetical machine

IC	:	the virtual machine instruction counter,
IR	:	a five-byte instruction register,
PI	:	protection interrupt register,
SI	:	supervisor interrupt register,
IOI	:	a three-bit I/O interrupt register,
TI	:	timer interrupt register,
PTR	:	page table register,
MODE	:	mode of CPU, 'MASTER' or 'SLAVE'.

The function of these registers will be described in the Sections 2.4 - 2.7.

User storage contains 3000 five-byte words. It is divided into 300 ten-word blocks for paging purposes. A user storage location is addressed by the ordered pair (page-frame-number, offset), where page-frame-number points to a page frame in the user storage and offset is the displacement of a word in that page.

The supervisor storage is loosely defined as that amount of storage required for the MOS implementation.

The backing storage is a high-speed drum of 1000 tracks, with one ten-word block per track. A transfer of one block of data to/from the drum takes two time units.

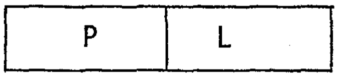
The card reader and line printer both operate at the rate of three time units for the I/O of one record. The record size for the card reader is the same as the virtual machine reader, while the record size for the printer is 51 bytes. The extra byte can only be used by the MOS designer/implementer for reformatting the user program and input data (refer to Section 3.5).

Channels 1 and 2 connect the supervisor storage to the card reader and line printer, respectively. Channel 3 is used to connect the drum to the supervisor and user storage.

2.4 The Slave Mode Operation.

User programs are read in from the card reader and transferred to the drum. A user program is ready to run after it has been loaded into the user storage. All user programs are executed in slave mode and the first instruction of a program must always appear in location zero of the virtual machine.

User storage addressing while in slave mode is accomplished through paging hardware. The PTR register consists of the following two parts:



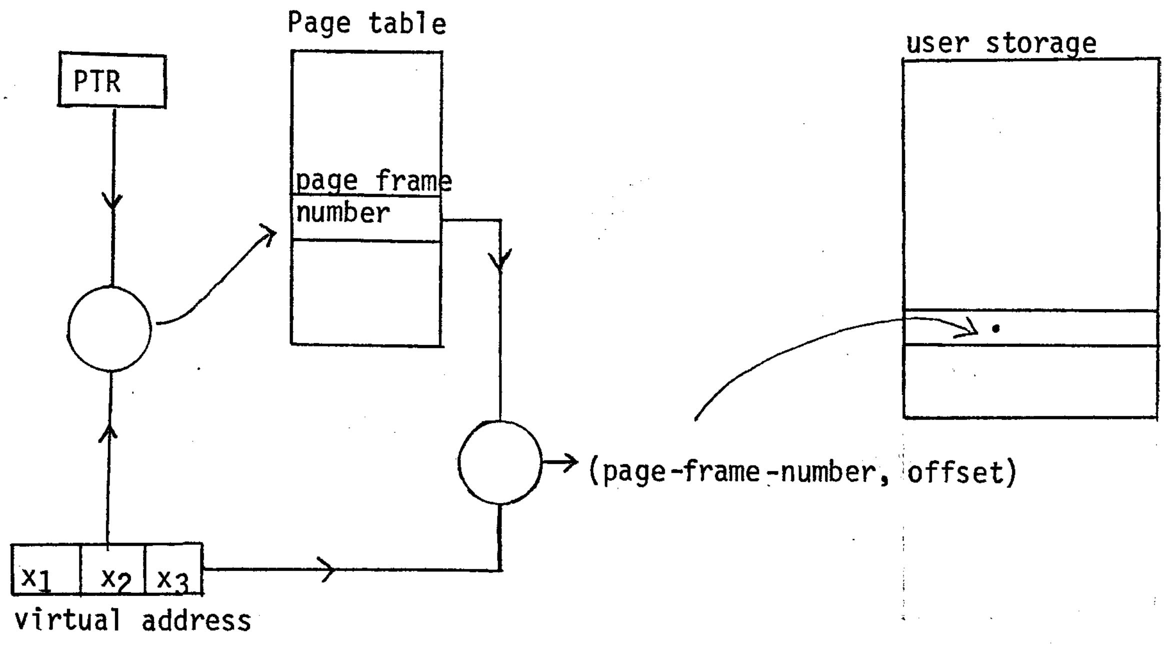
where P is the number of user storage page frames allocated to the running user program and L contains the page table base location in the supervisor storage.

As illustrated in Fig. 2.3, the virtual address $x_1x_2x_3$ is mapped into the user storage through the following transformation:

$$x_1x_2x_3 \longrightarrow (\text{page-table } [10x_1+x_2+1],\ x_3) \qquad (2.4.1)$$

where page-table [α] denotes the contents of the α^{th} entry of the page table.

Each user program can only access those user storage locations which are allocated to them, and any attempt to



where offset = x_3

Fig 2.3 Paging hardware

address a location outside its address space would cause a protection interrupt (i.e. if $10x_1 + x_2 + 1 > P$).

An instruction in the user storage is fetched into the instruction register IR and then decoded. The interrupt registers PI, SI and TI will be set to the following values if any of the associated interrupt events arise :

PI	=	1	:	protection interrupt;
PI	=	2	:	invalid operation code;
SI	=	1	:	input instruction (RD) encountered;
SI	=	2	:	output instruction (WR) encountered;
SI	=	3	:	halt instruction (H) encountered;
TI	=	1	:	timer interrupt

The state of the slave process is then saved in the supervisor storage locations and the CPU is switched to the master mode.

2.5 The Master Mode Operation.

Master mode programs residing in supervisor storage have access to the user storage and the CPU registers. They consist of a set of procedures for scheduling job execution, allocating and accounting the resources in the system, etc. Master mode instructions are assumed to be executed in zero time units except that a master mode program may wait until the CPU clock has been advanced a specified number of time units to change the current state of the MOS.

The CPU is not interruptable while it is executing a supervisor program. The contents of the interrupt registers can be interrogated and reset by the interrupt servicing routines. The CPU can switch back to slave mode by loading the state of a slave process stored in the supervisor storage, and set the CPU mode to SLAVE.

2.6 Channels and I/O Devices

A channel descriptor and a device state descriptor are associated respectively with each channel and device of the hypothetical machine. The channel descriptor contains information about the current state of the channel, the device to which it is connected, device record block size, I/O transfer rate, etc. The device state descriptor contains information about the current state of a device and the result of the I/O operation (i.e. end of file, invalid I/O command, or the I/O has been performed successfully).

A channel can be activated by the CPU to instruct the connected device to perform an I/O operation when it is free. The I/O transmission occurs completely in parallel with CPU activity.

The IOI is a three-bit I/O interrupt register, with bits 1, 2 and 3 corresponding to channels 1,2 and 3, respectively. A bit is set to indicate an I/O interrupt when the associated channel has completed an I/O operation. It will be reset by the I/O interrupt servicing routine after the interrupt has been serviced.

2.7 Timer

The timer hardware decrements the supervisor storage location TM by one whenever the CPU clock has advanced by one time unit. The timer interrupt register TI is set to one to indicate a timer interrupt whenever TM is decremented to zero. If the timer interrupt is raised in slave mode, the state of the slave process will be saved and the interrupt will then be serviced by the <u>timer interrupt servicing routine</u>; otherwise, no action will be taken.

3. THE OPERATING SYSTEM

We begin this section by presenting a brief description of the MOS structure. It is then followed by a detailed discussion on some of the software components of the MOS. These include input of jobs, job scheduling, interrupt handling, output of jobs, and the MOS statistics.

3.1 The MOS Structure.

The main objective of the MOS is to process a batch of user jobs efficiently. In this MOS, several jobs are residing in user storage concurrently. However, only one of them is being executed at any time by the CPU. If a running job issues an I/O command to a channel, it must wait until the channel has completed the I/O operation before it can continue. Instead of allowing the CPU to idle during this period, it can be switched to execute another user job which is ready to run. With proper scheduling, the amount of CPU and channels idling time can be reduced.

The MOS, including the user jobs executing within it, can be logically described as a set of processes that operate almost independently of one another, but compete for the limited available resources. The term process used in this project report refers to the activity resulting from the execution of a program by a processor (CPU or channels). To distinguish the activities of executing master mode programs and user programs in the system, the processes in the MOS are divided into two categories: _supervisor processes_ and _user processes_. The supervisor processes are responsible for scheduling and controlling the user jobs in the system, providing the means for communication and synchronization among processes, allocating and accounting for all hardware and software resources. A user process refers to the activity resulting from the execution of a user job on the virtual machine. Since a user process is executed in slave mode, it is sometimes referred to as a _slave process_.

After system initialization, the whole system is controlled and driven by a supervisor process called the _basic supervisor_. As illustrated in Fig. 3.1, the basic supervisor repeats the following three steps until a batch of user jobs have been processed:

1. Call the CPU scheduler to select a process from the ready queues. If all the ready queues are empty, the idling process which forces the CPU to idle for a certain number of time units will be selected.
2. Run the selected process until it is either blocked or terminated.
3. Examine the interrupt registers and call the appropriate interrupt handling routines to service the interrupts.

All the supervisor processes can be described by a set of master programs together with their data structures in the supervisor storage. Supervisor processes have access to user

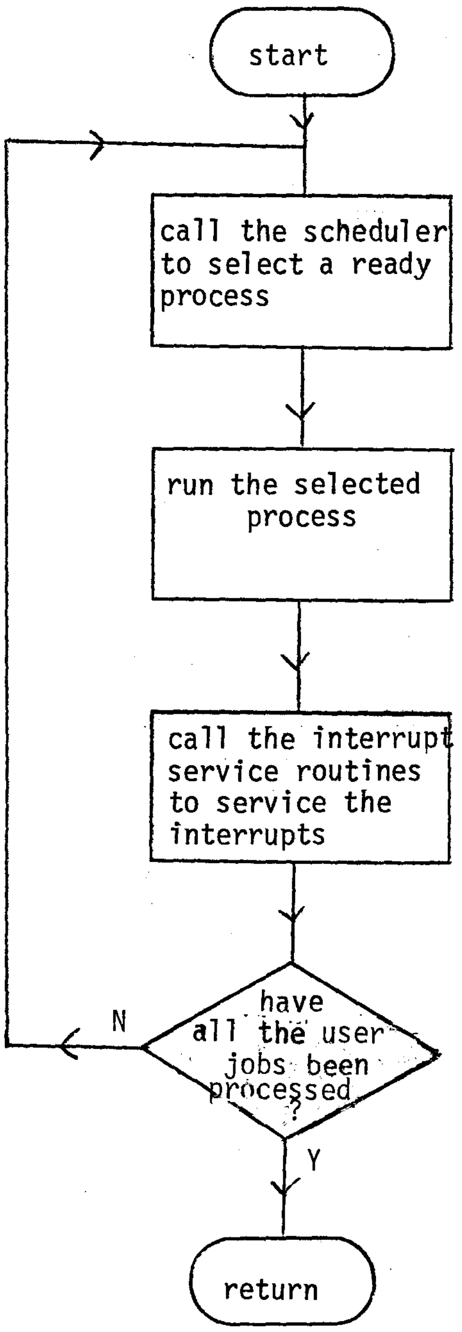

start
call the scheduler to select a ready process
run the selected process
call the interrupt service routines to service the interrupts
have all the user jobs been processed ?
N
Y
return

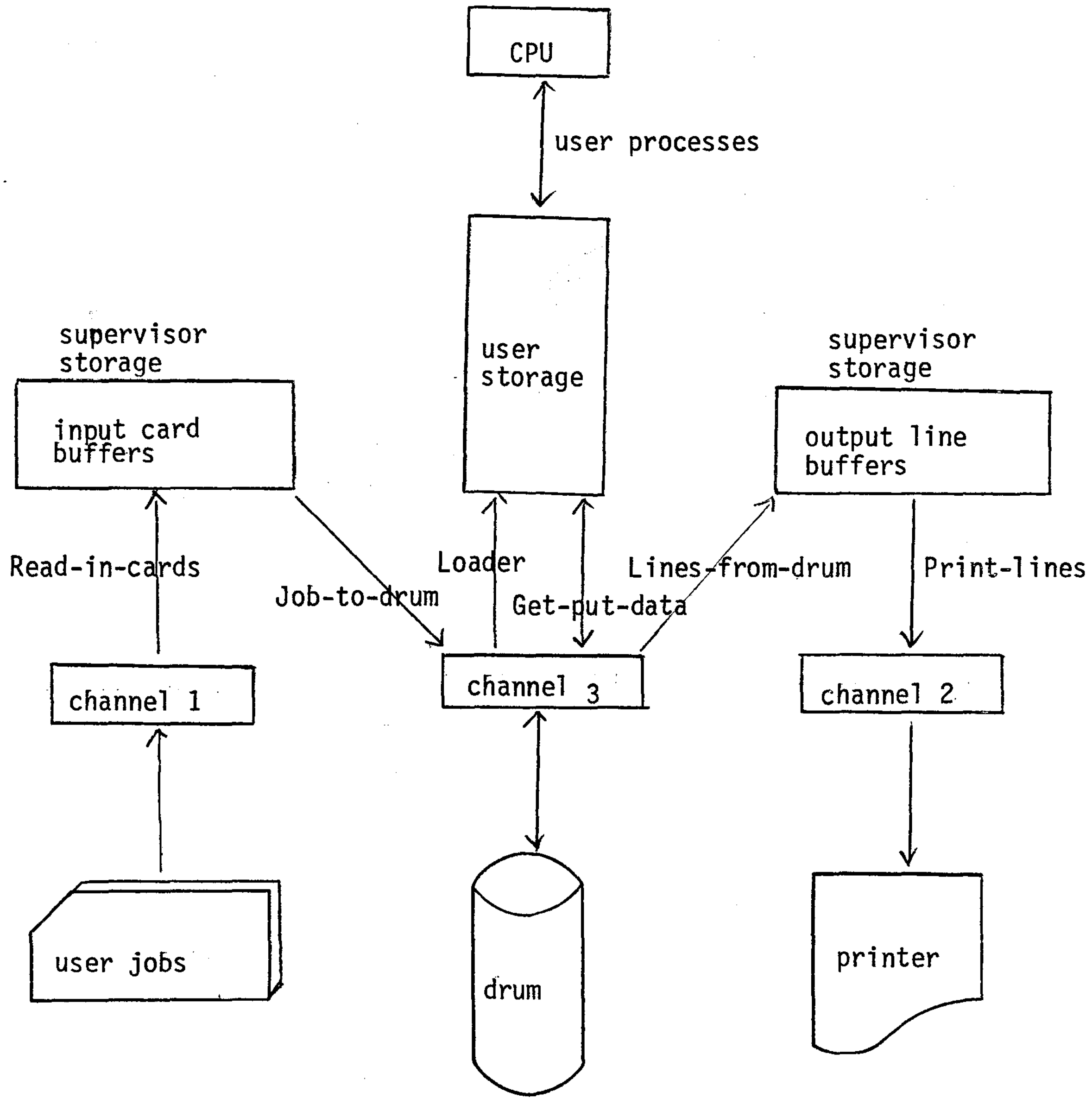
Fig 3.1 The basic supervisor control chart

storage and the CPU registers. The following major supervisor processes are included in the MOS:

1. <u>Read-in-cards:</u>	Read cards into supervisor input buffers.
2. <u>Job-to-drum</u>:	Create a job descriptor for a new user job and transfer the job to the drum.
3. <u>Loader</u>:	Load a job into the user storage.
4. <u>Get-put-data</u>:	Perform the user I/O operation.
5. <u>Lines-from-drum</u>:	Transfer the source program, and output data of a terminated job from the drum to the supervisor output buffers.
6. <u>Print-lines</u>:	Print output lines from the supervisor output buffers on the printer.
7. <u>Terminate:</u>	Perform a job termination.
8. <u>Idling</u>:	Force the CPU to idle until an I/O interrupt is raised from any of the channels.

As shown in Fig. 3.2, a user job J will pass sequentially through the following phases:

1. J is read into card buffers by the Read-in-card process.
2. J is transferred to drum by the Job-to-drum process.
3. J is loaded into the user storage by the Loader process.
4. J is then ready to run and becomes a user process j until j terminates, either normally or as a result of an error.
5. I/O requested by j will be performed by the Get-put-data process.



Key : $\longrightarrow$ flow of data

Fig 3.2 Major supervisor and user processes control chart

6. J's output, including job statistics, system messages, and its original program are transferred from the drum to the output line buffers by the Lines-from-drum process.
7. Data in the output line buffers are printed on the printer by the Print-lines process.

The interactions among processes are controlled by a set of logical resource semaphores. A logical resource is anything that can cause a process to enter a logically blocked state [S2].

The logical resources used by the MOS can be classified as follows:

1. Channels 1, 2 and 3: Used to transfer data between the I/O device and memory.
2. Empty-card-buffers, full-card-buffers: The input card buffers are characterized by the two logical resources empty-card-buffers and full-card-buffers. They are used by the Read-in-cards and Job-to-drum processes to transfer data between the two I/O devices, reader and drum.
3. Empty-line-buffers, full-line-buffers: The output line buffers are characterized by the two logical resources empty-line-buffers and full-line-buffers. They are used by the Lines-from-drum and Print-lines processes to transfer data between the two I/O devices, drum and printer.

4. Free-core-frames: Free user storage page frames.
5. Free-drum-frames: Free drum blocks.
6. Waiting-jobs: User jobs created by the Job-to-drum process to be run on the CPU.
7. End-jobs: Terminated user jobs to be printed on the printer.
8. User-process-identifiers: The process identifiers to be assigned to a newly created user process.

The producers and consumers of these logical resources are illustrated in Table 3.1.

A semaphore is associated with each of the logical resource [D2,B3]. Each semaphore is represented by a counter which denotes the number of available resources, and a waiting queue which contains the processes blocked by this logical resource.

Two operations are allowed on each semaphore, namely, the wait and signal operations. When a process requests a logical resource, the associated semaphore wait operation will be performed. If the resource is available, the semaphore counter will be decremented by one and the resource is allocated to the requesting process; otherwise, the requesting process will be entered into the semaphore waiting queue. When a resource is released by a process, the semaphore signal operation is performed. The resource is allocated to one of the processes in the semaphore waiting queue if it is not empty; otherwise, the semaphore counter is incremented by one. The implementation of the resource semaphores and their wait and

Table 3.1 Producers and consumers of the logical resources

Resources	Consumers	Producers
Channel 1	Read-in-cards	Read-in-cards
Channel 2	Print-lines	Print-lines
Channel 3	Job-to-drum, Loader, Get-put-data, Lines-from-drum	Job-to-drum, Loader, Get-put-data, Lines-from-drum
empty-card-buffers	Read-in-cards	Job-to-drum
full-card-buffers	Job-to-drum	Read-in-cards
empty-line-buffers	Lines-from-drum	Print-lines
full-line-buffers	Print-lines	Lines-from-drum
free-core-frames	Loader	Terminate
free-drum-frames	Job-to-drum, Loader	Get-put-data, Terminate
waiting-jobs	Loader	Job-to-drum
end-jobs	Lines-from-drum	Job-to-drum, Terminate
user-process-identi-fiers	Loader	Terminate

signal operations will be discussed in Section 4.

A priority level in the range from 1 to maxpriority is assigned to each of the processes in the system, where maxpriority is the highest priority level selected by the MOS implementer. In this project, the highest priority level, 3, is assigned to the supervisor processes Read-in-cards, Job-to-drum, Loader, Get-put-data, Lines-from-drum, and Print-lines. Priority levels for the user processes are determined at job creation time, and is computed by a priority algorithm which we shall discuss in detail in the next section.

As shown in Fig. 3.3, a process can be in one of the following states: ready, blocked, running, suspended, or terminated. Initially, all the processes are in the terminated state. Supervisor processes are in the ready state after they have been initiated at system start time. User processes will be in the ready state after they have been created by the Loader process. There is a ready queue in the system for each level of priority. A process in the ready state is in one of them according to its priority level. Ready processes are competing for the CPU, and one of them will be selected by the CPU scheduler. The scheduling policy is to serve the highest-priority processes first and employ the FIFO discipline within the same priority ready queue. As a result, a ready low priority process has to wait until all the higher priority processes are blocked, suspended or terminated.

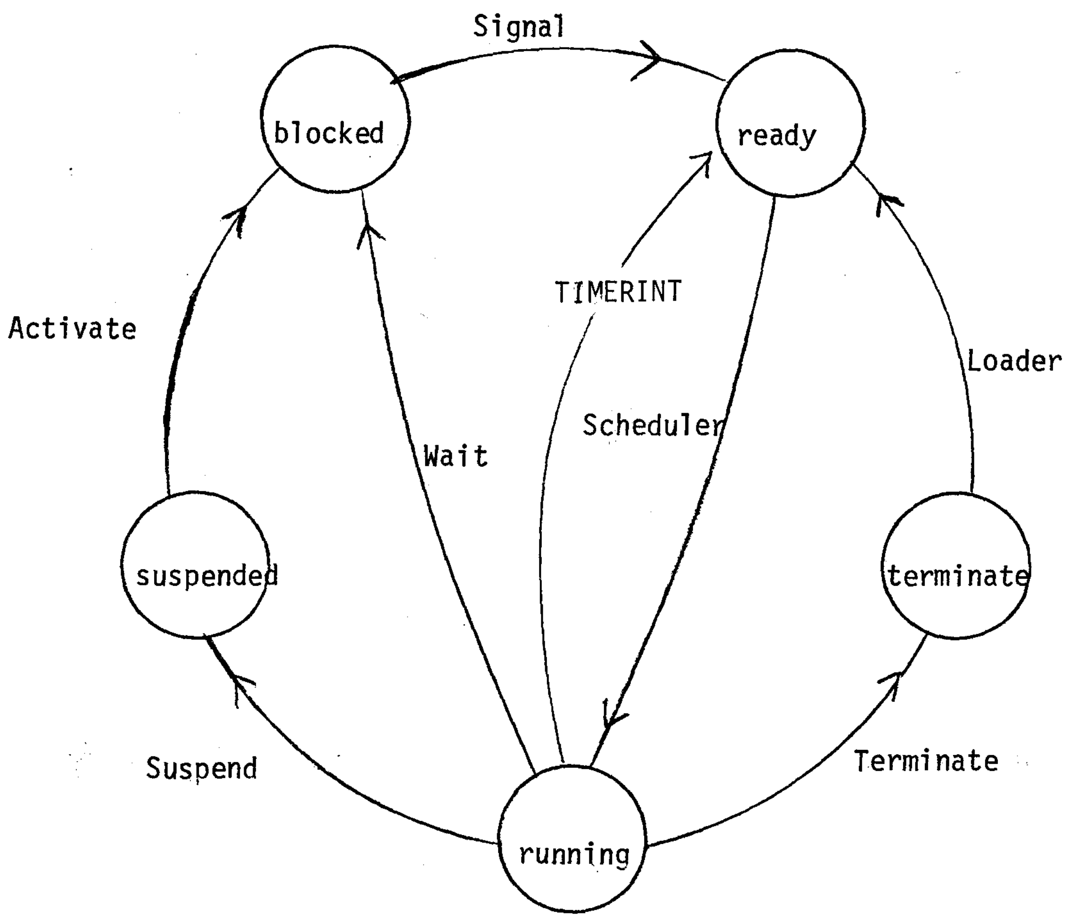


Fig 3.3 Process status

To prevent deadlock,when a process requests a particular resource it may be desirable to defer the allocation of the resource for sometime ; even though the resource is available at the time of request. The requesting process can be suspended by the Suspend process. The suspended process will remain in the suspended state until it is activated by the Activate process.

To prevent a process from monopolizing the CPU, a time slice is assigned to each of the user processes in the system. A user process is executed until either its quantum runs out, it becomes blocked, it terminates, or it is interrupted. After processing for the duration of its time slice, it is returned to the ready queue. Since all the supervisor processes except the idling process are executed in zero time units, and the idling process is always terminated by an I/O completion interrupt in a finite number of time units, the CPU will never be monopolized by a supervisor process.

3.2 Input of Jobs.

User jobs are read in from the card reader. The reader operates at a speed much slower than the CPU. To avoid having the CPU wait for its data from the reader, and allocating a card reader and printer to each job executing in a multiprogrammed system, jobs are collected on auxiliary storage prior to their execution and their outputs are also written on auxiliary storage during processing. By this means, the job scheduler has more freedom in selecting jobs and the I/O operations are faster. This decreases the amount of times a job would be in the main storage. The subsystems that read jobs into auxiliary storage and print job outputs from auxiliary storage to the printer are called input and output spoolers, respectively.

The input spooler has ten buffers in the supervisor storage. Each of them is capable of holding the contents of one input card (the first fifty columns of a card). It is controlled by two supervisor processes: the Read-in-cards process and the Job-to-drum process. The Read-in-cards process requests the Channel process to transfer a card from the reader into the input spooler buffer via channel 1 while the empty-card-buffers resource is available. The I/O request is carried out by performing the wait operation of the semaphore associated with the channel 1 resource. The Read-in-cards process then waits for the I/O completion interrupt and it will be entered into the appropriate ready queue after the interrupt has been serviced. This process is repeated until the whole batch of jobs has been read in.

The Job-to-drum process examines the contents of the input spooler buffer produced by the Read-in-cards process. If it is a job card, a job descriptor for the user job is created. If it is a source program card or input data card, the contents of the buffer is transferred to the drum via channel 3. The drum frames which store these data are recorded in the source queue and input-data queue respectively.

All the pertinent information is kept in the job descriptor.

A job descriptor contains the following information:

1. Entries appearing on the job card: these include job name, user account number, and estimated resource limits such as time estimate, line estimate, user storage estimate etc. Details are given in APPENDIX A.
2. Job information: these include the priority level assigned to the job and the <u>source</u>, <u>input-data</u>, <u>output-data</u>, and <u>reserve-drum-frames</u> queues. These queues contain information about where the user's original program, input data, and output data are stored in the backing store and the drum frames reserved by this job (to be used for storing output data and will be discussed in the next section). Some of these queues might be empty initially. They will be updated as the job passes through the input spooling and processing phases.
3. Accounting and system information: these include the time when a job is read into the system, the CPU and I/O run times, and any messages produced by the system. This information is used to update the system statistics and produce a system report for the user job.

The priority level of a user job is computed by taking the weighted average of the three resource priority components P_i,P_o, and P_t associated with the number of input data cards, line estimates, and time estimates respectively. The <u>weighting factors</u> for the three priority components P_i,P_o,P_t are selected by the MOS implementer. The three factors 0.25,0.25, and 0.50 are used in our project.

As illustrated in Fig 3.4, the priority for a particular resource is a step function of the resource requirement estimated for the job, and can be computed as follows ;

$$\text{priority} = \max(1, \left\lceil \frac{R - r}{R} \times \text{maxpriority} \right\rceil)$$

where maxpriority is the highest priority level allowed for a job in the MOS, R is the maximum resource limit, and r is the estimated resource limit.

For example, suppose the following parameters are used in the MOS implementation :

maximum number of input data cards allowed for a job = 200,
maximum number of output lines allowed for a job = 200,
maximum execution time allowed for a job = 1000.

If a user job is supplied with the following information:

input data cards = 10,
line estimate = 50,
time estimate = 500,

then we have $P_i = 3$, $P_o = 3$, $P_t = 2$. The weighted average of these three resource priority components = 0.25 x 3 + 0.25 x 3 + 0.50 x 2 = 2.5, and the priority level for this user job will be rounded up to 3.

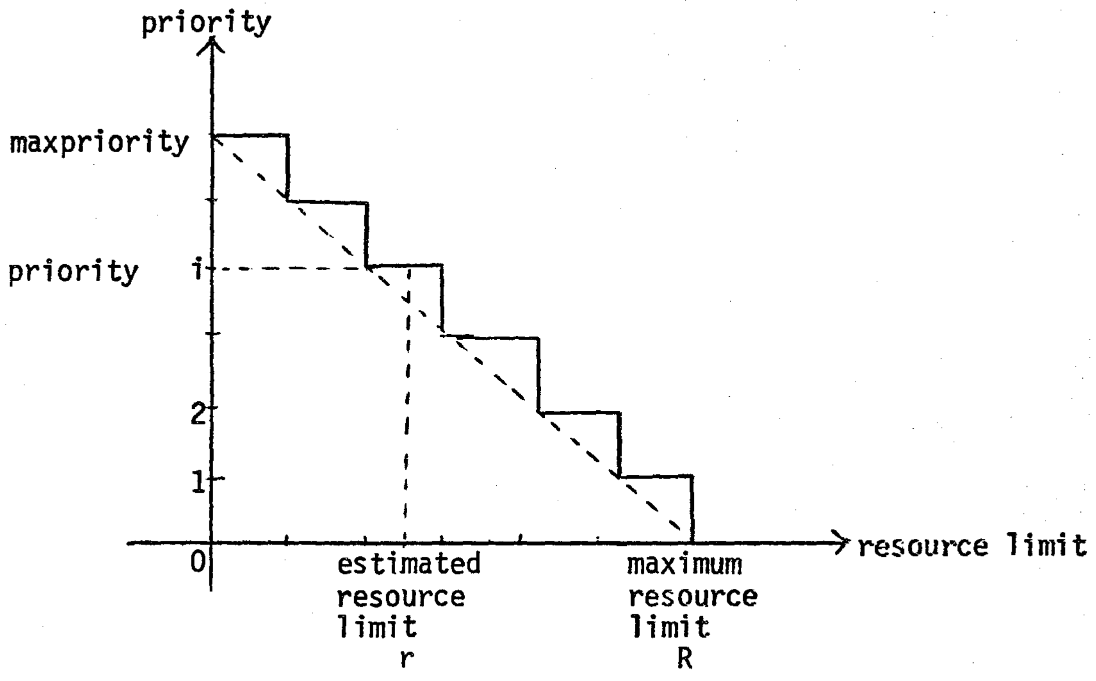


Fig 3.4 Resource priority component of a resource

To avoid system deadlock, the Job-to-drum process will be suspended when the number of free-drum-frames is less than a certain limit (20% of the drum capacity is used in our implementation), and it will be activated when some drum frames are released by some other processes in the system.

3.3 Job scheduling.

The user storage is divided into a number of ten-word page frames. User programs and data can be scattered throughout it on a page basis. The free page frames are recorded in the free-core-frames queue.

In this project, a maximum of up to ten user processes can be in the system simultaneously. Each user process is identified by a unique number in the range from 8 to 17 (numbers 1 to 7 are reserved for supervisor processes). Free user process identifiers are recorded in the free-process-id queue and controlled by the resource semaphore user-process-identifiers. When the free-process-id queue and the waiting-jobs queues are not empty, the Loader process will be activated and try to load a user job into the user storage.

There are three possible classes of user-jobs in the waiting-jobs queues. The Loader process uses the highest-priority-first-fit scheduling rule to select a job from the waiting-jobs queues and loads it into the user storage. The scheduling rule can be described as follows:

1. Start from the first non-empty highest priority waiting-jobs queue.
2. Examine the estimated user storage and estimated lines entries of the job descriptor in the queue, and select the first one with estimated resources less then or equal to the available resources.
3. If none of the jobs can be selected, try the next non-empty lower priority waiting-jobs queue.

4. Repeat steps 2 and 3 until a user job is selected or the waiting-jobs queue is exhausted.

If none of the jobs are selected, the Loader process will be suspended. It will be activated when either additional user storage page frames, or drum frames become free or new user jobs are available.

By using the <u>highest-priority-first-fit</u> scheduling policy, a low priority job may have to wait for a longer time in the queue. Having selected a user job to be run, the user program part will be loaded into the user storage. The loaded user job is assigned a <u>process identifier</u> taken from the free-process-id queue. The user job is then ready to run and it becomes one of the user processes in the system.

Several user processes may be in the system simultaneously. These user processes may activate the Get-put-data process to transfer their output data to the drum, and thus free-drum-frames may be requested by the Get-put-data process at run time. To prevent system deadlock, the number of estimated drum frames to store the output data are allocated to the user job at job loading time. They are recorded in the <u>reserve-drum-frames</u> queue and kept in the job descriptor.

3.4 <u>Interrupt Handling</u>.

The contents of the interrupt registers are examined to check for interrupts at the end of every instruction cycle if the CPU is executing a user process. If the CPU is executing a supervisor process, the interrupt registers will only be examined when the supervisor process is blocked or terminated.

As we have mentioned in Section 2, there are four possible types of interrupts in the MOS:

1. Program: protection, invalid operation code.
2. Supervisor: RD, WR, H.
3. I/O: completion interrupts.
4. Timer: TM decremented to zero.

The program and supervisor interrupts can happen only in slave mode, while I/O and timer interrupts can occur in both master and slave modes. Several of these events may happen simultaneously.

When an interrupt occurs in slave mode, the instruction counter IC, the boolean toggle C, and the general register R of the CPU will be saved in the process descriptor of the interrupted user process and the CPU is then switched to the master mode to handle the interrupt.

As shown in Fig. 3.5, the contents of the four interrupt registers, IOI, PI, SI, and TI will be examined in turn. If they are not zero, the four interrupt service routines <u>IOINT</u>, <u>PROGINT</u>, <u>SUPINT</u>, and <u>TIMERINT</u> will be called to service the interrupts. If an interrupt was raised in slave mode, the interrupted user process will be either entered into the appropriate ready queue or terminated by the appropriate interrupt service routine. After the interrupts have been serviced, the CPU will then be allocated to a new process selected by the CPU scheduler.

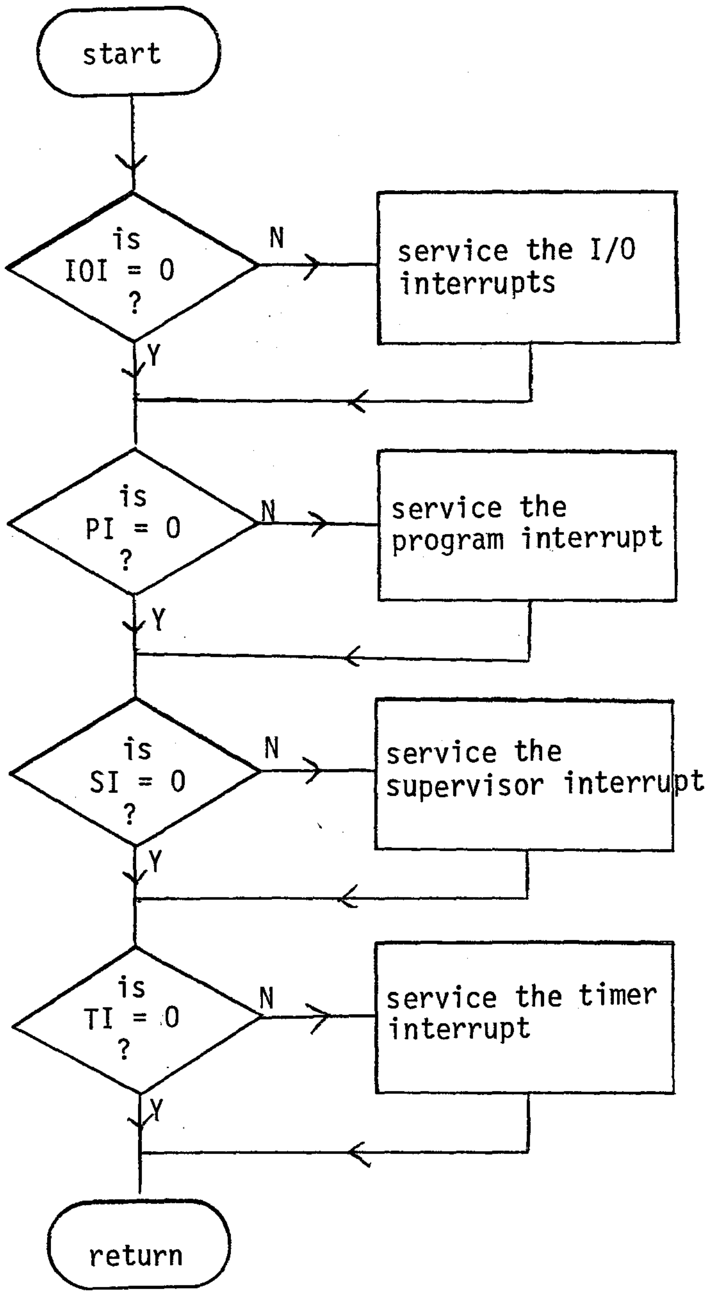

start
is IOI = 0 ?
N
service the I/O interrupts
Y
is PI = 0 ?
N
service the program interrupt
Y
is SI = 0 ?
N
service the supervisor interrupt
Y
is TI = 0 ?
N
service the timer interrupt
Y
return

Fig 3.5 Interrupt handling flow chart

The four interrupt service routines can be described as follows:

1. I/O interrupt service routine (IOINT): As illustrated in Fig. 3.6, the three bits of IOI interrupt register are examined one by one. If it is set, the status of the associated I/O device is checked to see what kind of I/O interrupt was caused by this device. If it is an I/O completion interrupt, the process that was requesting the I/O operation will be entered into the ready queue and the busy flag in the asosciated channel descriptor is set to false to indicate that the channel is free now. If it is an unusual I/O interrupt, the appropriate action will then be taken. For example, if it is an end of file interrupt initiated by the card reader, the reader status will be set to a normal state and the Read-in-cards process will be terminated.
   After the I/O interrupt has been serviced, the signal operation of the associated resource semaphore will be performed, so that the free channel may be allocated to another requesting process.
   The three bits of the IOI register are set to zero after all the I/O interrupts have been serviced.
   If the interrupt was raised in slave mode, the interrupted user process is then entered into the ready queue.

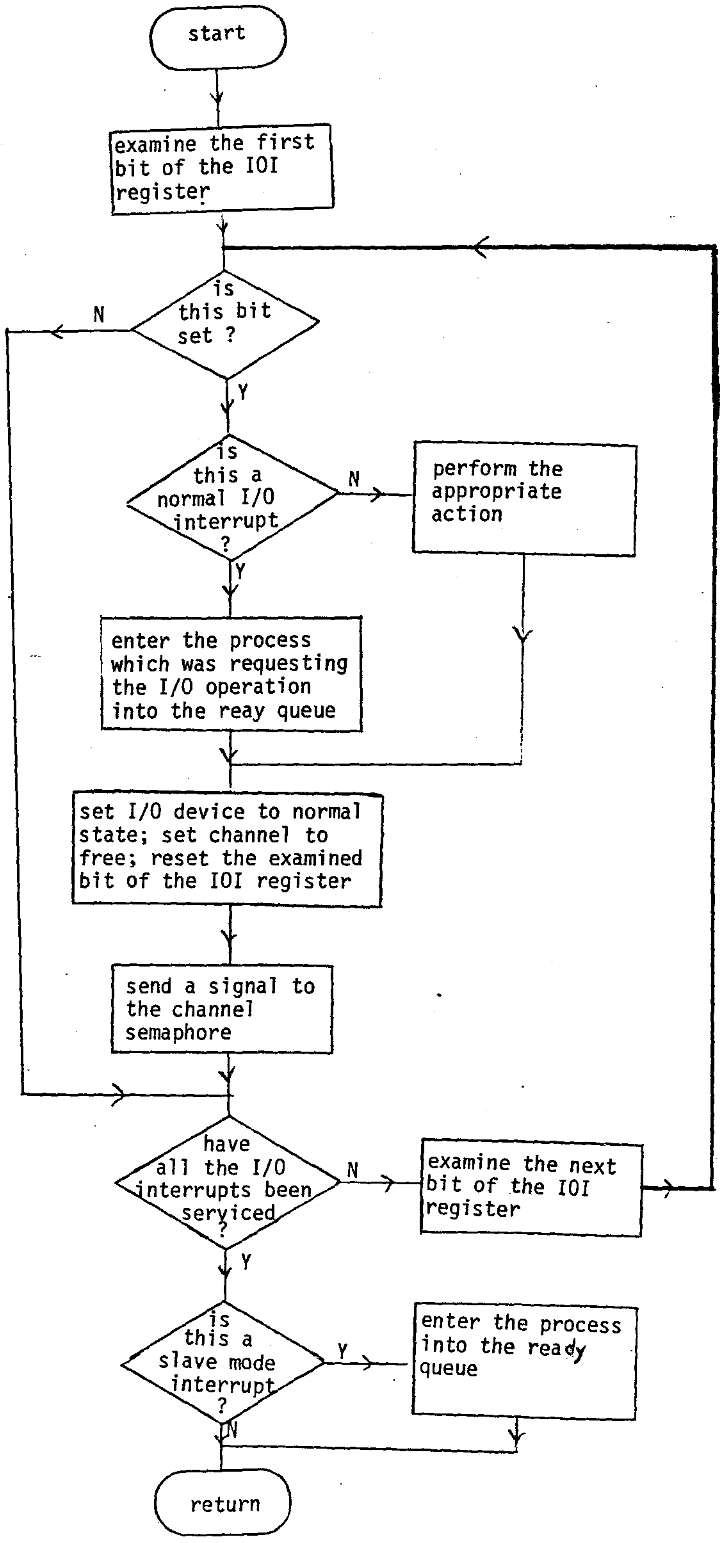

start
examine the first bit of the IOI register
is this bit set ?
N
Y
is this a normal I/O interrupt ?
N
perform the appropriate action
Y
enter the process which was requesting the I/O operation into the reay queue
set I/O device to normal state; set channel to free; reset the examined bit of the IOI register
send a signal to the channel semaphore
have all the I/O interrupts been serviced ?
N
examine the next bit of the IOI register
Y
is this a slave mode interrupt ?
Y
enter the process into the ready queue
N
return

Fig 3.6 I/O interrupt service routine

2. Program interrupt service routine (PROGINT): As shown in Fig. 3.7, there are two kinds of program interrupts, protection or invalid operation code interrupt. The appropriate error message will be written on the job descriptor of the interrupted user job. The Terminate process will then be activated to release the user storage and unused drum frames to the free-core-frames queue and free-drum-frames queue respectively. The user job is then entered into the end-jobs queue waiting for the output spooler to print it on the printer. The PI register is set to zero after the interrupt has been serviced.

3. Supervisor interrupt service routine (SUPINT): As illustrated in Fig. 3.8, there are three kinds of supervisor interrupts: the read, write, and halt supervisor interrupts. In case of user I/O interrupt, the user-storage-page-frame number and drum-frame number (i.e. the locations in the main and backing store involved in the data transfer) are set in the process descriptor of the user process. The user process is then entered into the user-I/O-request queue which is followed by sending a signal to the Get-put-data process to perform the I/O operation. If an exceptional condition arises from an user I/O request, e.g., end of file encountered from an input request or the line limit exceeded

from an output request, then an error message will be written on the job descriptor of the user job and the Terminate process is activated to transfer the user job to the end-jobs queue.
If the supervisor interrupt is a halt supervisor call, the user job will be transferred to the end-jobs queue by the Terminate process.
The supervisor interrupt register is set to zero after the interrupt has been serviced.

4. Timer interrupt service routine (TIMERINT): The action of this routine is illustrated in Fig. 3.9. If the interrupt occurs in the master mode, except for setting the timer interrupt register to zero, no action will be taken. If it happens in the slave mode, the time used (CPU plus I/O time) by the user process is checked to see whether the time limit has been exceeded or not. If it has, a message will be written on the job descriptor of the user job and the Terminate process will be activated to transfer the user job to the end-jobs queue; otherwise, the user process will be entered into the ready queue.

3.5 Output of Jobs.

The output spooler is the software module which reads the user's original program and output data from the drum to the supervisor output buffers and prints it on the printer.

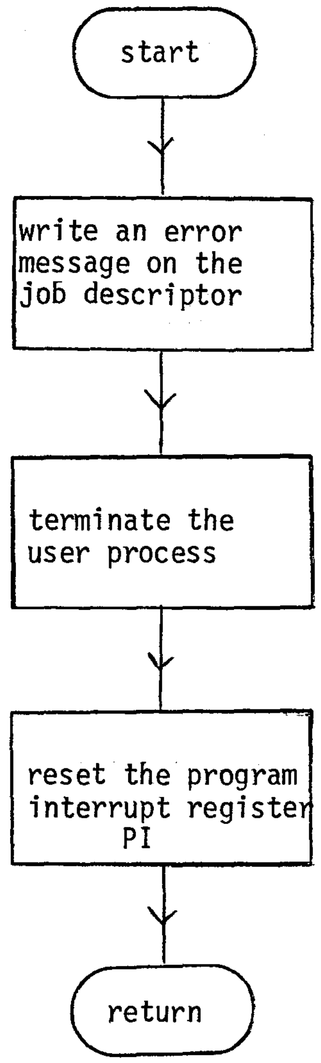

start
write an error message on the job descriptor
terminate the user process
reset the program interrupt register PI
return

Fig 3.7 Program interrupt service routine

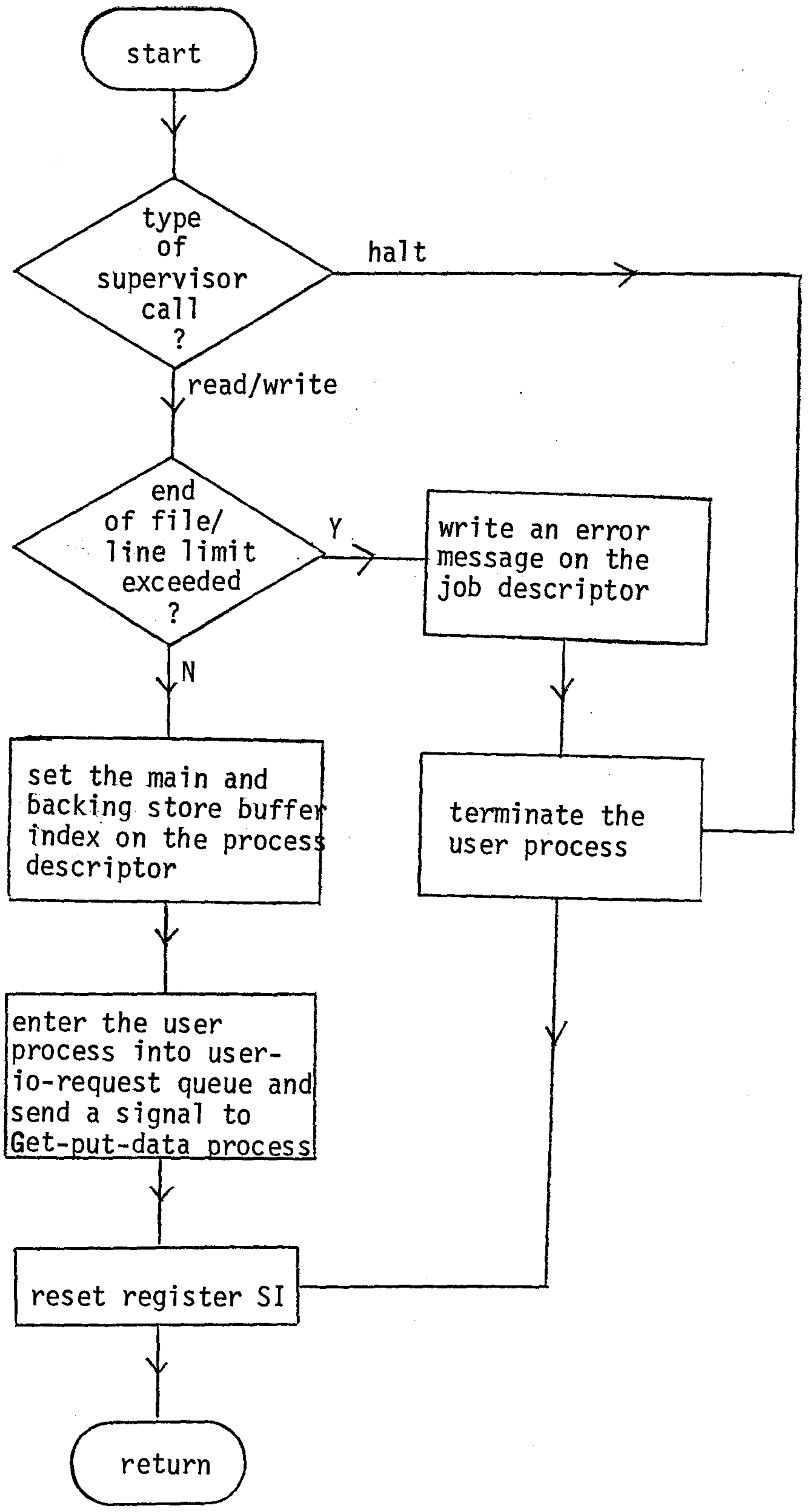

start
type of supervisor call ?
halt
read/write
end of file/ line limit exceeded ?
Y
N
write an error message on the job descriptor
terminate the user process
set the main and backing store buffer index on the process descriptor
enter the user process into user-io-request queue and send a signal to Get-put-data process
reset register SI
return

Fig 3.8 Supervisor interrupt service routine

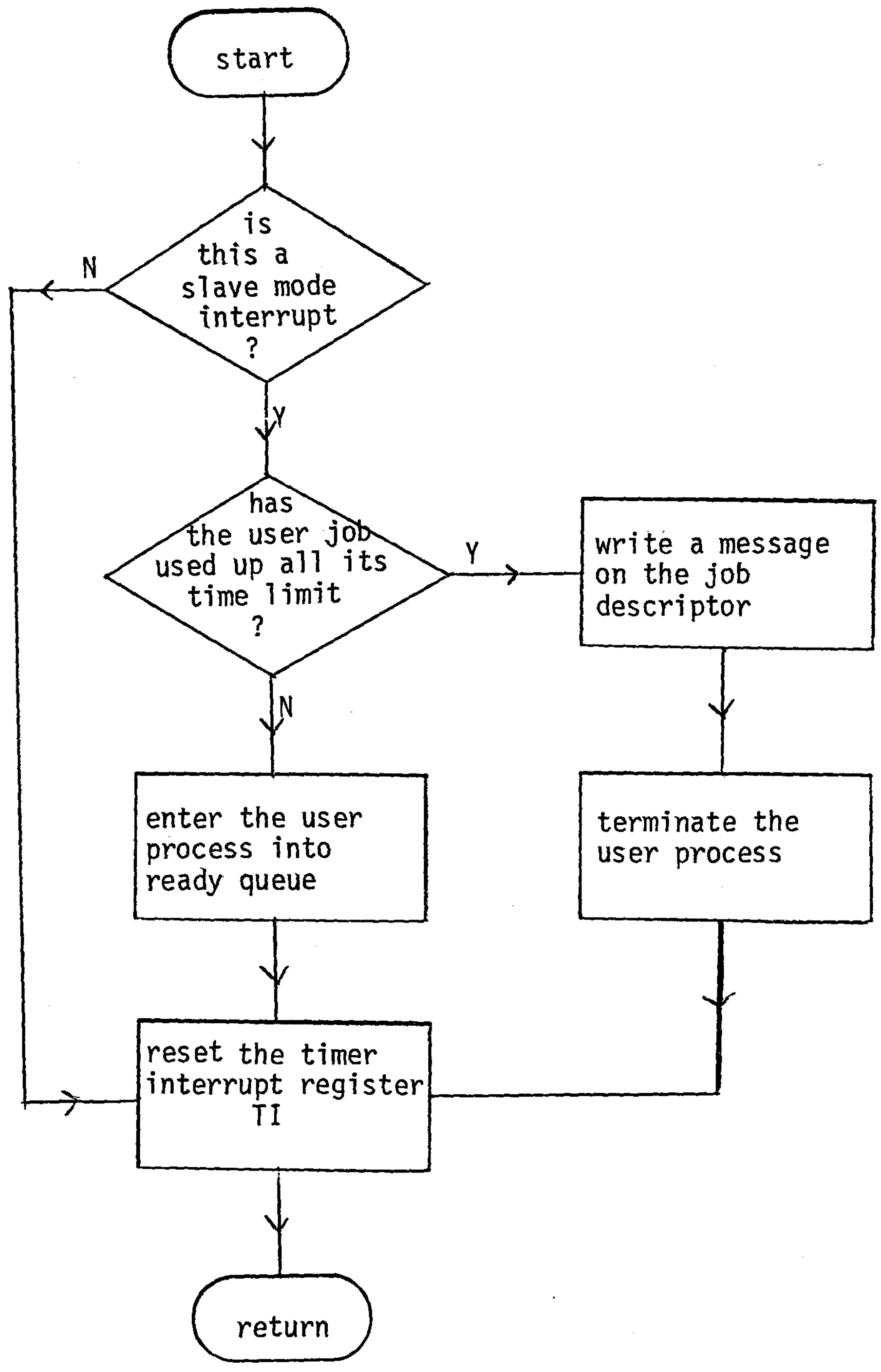

start
is this a slave mode interrupt ?
N
Y
has the user job used up all its time limit ?
Y
N
write a message on the job descriptor
enter the user process into ready queue
terminate the user process
reset the timer interrupt register TI
return

Fig 3.9 Timer interrupt service routine

The accounting and system messages for each user job are also printed on the printer.

The output spooler has ten buffers in the supervisor storage, each of which is capable of holding 51 characters. It is controlled by two supervisor processes: the Lines-from-drum process and the Print-lines process. The Lines-from-drum process removes the job descriptor of a terminated user job from the end-jobs queue and produces a system report for this job. The report contains the error messages that were written on the job descriptor and also the user job statistics (number of cards read in, number of lines printed, user storage required, CPU and I/O times used by the job).

The user's original program is fetched from the drum and reformatted into the printer format by adding a printer control character to each block of data fetched from the drum. The user's output data fetched from the drum has only fifty characters. A blank in inserted into the last character position of the spooler buffer.

On the completion of the reformatting of the data in the spooler buffer, the Lines-from-drum process sends a signal to the resource semaphore <u>full-line-buffers</u> to activate the Print-lines process which prints the contents of the output spooler buffer on the line printer via channel 2.

3.6 The MOS Statistics.

The performance of the MOS can be measured from the statistics collected by the MOS. The statistics maintained by the MOS software are as follows:

1. Resource statistics: Fig. 3.10 shows the graph of a resource semaphore counter plotted against the CPU clock during a run; from this graph, we define the <u>total availability</u> of a logical resource as the area under the curve and the <u>average availability</u> of a logical resource as its total availability divided by the total run time. If the maximum capacity of a logical resource is defined, then we define the <u>utilization</u> of a logical resource as

$$1 - \frac{\text{average availability}}{\text{maximum capacity of the logical resource}} .$$

   The total availability of a logical resource is recorded whenever the wait and signal operations of the logical resource semaphore are performed. These statistics are printed at the end of a run.

2. Job characteristics: the run time, elapsed time, user storage required, input length, and output length for each user's job are recorded by the Lines-from-drum process. The mean values of these statistics and the distributions of user' jobs (i.e. number of users' job belonging to each priority classes) are also printed at the end of a run.

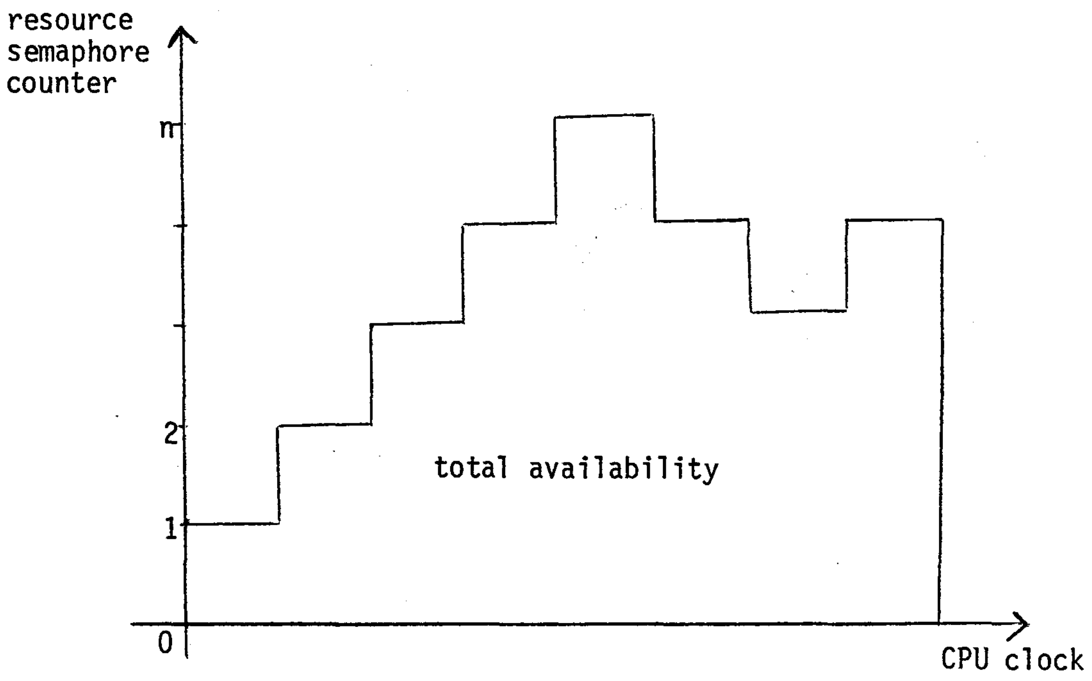


Fig 3.10 Graph of resource semaphore counter against CPU clock

4. IMPLEMENTATION DETAILS

The MOS is implemented in a modular and structured manner as a set of procedures written in PASCAL. As a result, the software modules of the operating system can easily be identified, maintained and modified. Some of the programming techniques and data structures used are discussed in the following sections.

4.1 Overview

In this project the following major modules have been implemented:

1. Simulators for hardware: these include the user storage, backing storage, CPU registers, reader, printer, drum, channels, CPU clock, timer, and paging system.
2. An emulator which simulates the CPU of the virtual machine.
3. The MOS: it includes a set of interacting processes and a set of procedures to handle the interrupts and the allocation and administration of resources in the system.

The size and time parameters of all the hardware (virtual machine size, user storage and drum size, I/O transfer rates, instruction times,word size,etc) and some of the software parameters (priority level range, system buffer size, maximum number of concurrent processes,etc) are represented by symbolic names in the constant section of the PASCAL program. It is therefore very easy to change these parameters to study the MOS behavior under different hardware and software assumptions.

The MOS statistics and user program listings are written on two separate files. As a result, it offers an option to the user to determine whether the contents of one or both files are to be listed on the printer of the real machine.

4.2 Representation of User Storage, Spooler Buffers, Backing Storage, and CPU Registers.

The user storage is divided into pages. The pages and input spooler buffers are of the same size; each of them can hold ten words. We represent them by the following array structure:

```
type    word       =  packed array [1..wordsize] of char;
        wordindex  =  0.. pagesizeminus1 ;
        textbuffer =  array [wordindex] of word;
        corebufindex = 1.. maxcorebuf;
var     memory:  array [corebufindex] of textbuffer;
```

where maxcorebuf = number of user storage pages + number of input spooler buffers + 1. The user storage and input spooler buffers are represented by the first number-of-user-storage-pages locations and the next number-of-input-spooler-buffers of the array *memory* respectively. The last entry of the array memory is a temporary buffer area used by the Lines-from-drum process to store the data fetched from the drum, which are then reformated and transferred to the output spooler buffers.

The output spooler buffers are represented by the following array structure:

```
        type line  =  packed array [1..linebufsize] of char;
        var linebuffer: array [1..maxlinebuf] of line;
```

where _linebufsize_ is the number of characters per line to be printed, and _maxlinebuf_ is the number of output spooler buffers.

The backing storage can be represented as either an array structure or a file structure. Since drum frames are accessed randomly, it is not efficient to implement the backing storage as a sequential file in PASCAL (direct access files are not allowed). It is therefore better, in our opinion, to implement the backing storage as an array structure if its size is not very large. An alternative solution is to implement the backing store on a direct access file, but an assembly subroutine for accessing has to be written to coordinate with the PASCAL program. For the sake of simplicity, the drum frames in our implementation are represented by the following array structure:

```
type   drumframe  =  1.. noofdrumframes;
var    backingstore:  array [drumframe] of textbuffer.
```

Instead of specifying the sizes of user storage, spooler buffers, and backing storage in the constant section, our first intention is to pass these parametric values through some parameter cards; but because of the language restriction in PASCAL (dynamic arrays are not allowed), all the parametric values are set in the constant section and the whole program is recompiled whenever a change is desired.

Each of the CPU registers is represented by a variable of the type simple, packed array, or record in PASCAL.

4.3 Simulation of I/O Devices and Channels

The card reader, line printer, and drum are simulated by a set of routines. Each individual device has a device descriptor which contains information about its current state and the result of an I/O operation.

The device descriptor is represented by the following record structure:

```
type  devicedescriptor  =  record
                             busy :  boolean;
                             status : (complete, endoffile,
                                        errcommand);
                           end
```

As shown in Fig 4.1, a device is activated by setting its busy flag. When the busy flag is set, the device examines the type of I/O command specified in the command buffer. If the command is valid, the necessary I/O operation is performed; otherwise, an error will be indicated in the status of the device descriptor. Any unusual condition occuring on a device should be cleared before attempting to call the device simulation routine; otherwise, the system will be aborted. The busy flag is reset after the I/O operation has been performed.

The three I/O devices are connected to three channels. Each channel has a channel descriptor which contains all the necessary information about the current state of the channel and the device to which it is connected.

The channel desdriptor is represented by the following record structure:

```
type  channeldescriptor  =  record
                               devicename: (reader, printer,drum);
                               recordsize: integer;
                               blocktransferrate: integer;
                               busy:  boolean;
                               process: processid;
                               readytime: integer;
                            end
```

where recordsize is the number of characters to be transferred per I/O operation; blocktransferrate is the time required to transfer a block of data between the main store and the device buffer.

When a process requests I/O, the wait operation of the associated channel semaphore is performed. If the channel is free, the identifier of the requesting process is set in the channel descriptor, and the channel busy flag is then set to true to activate the associated channel. A channel is controlled by the Channel process. As shown in Fig. 4.2, when the channel is activated, the Channel process extracts the main/backing store buffer index and the type of I/O command from the process descriptor of the requesting process. This information is passed over to the appr127orpiate device simulation routine, which then performs the specified I/O operation.

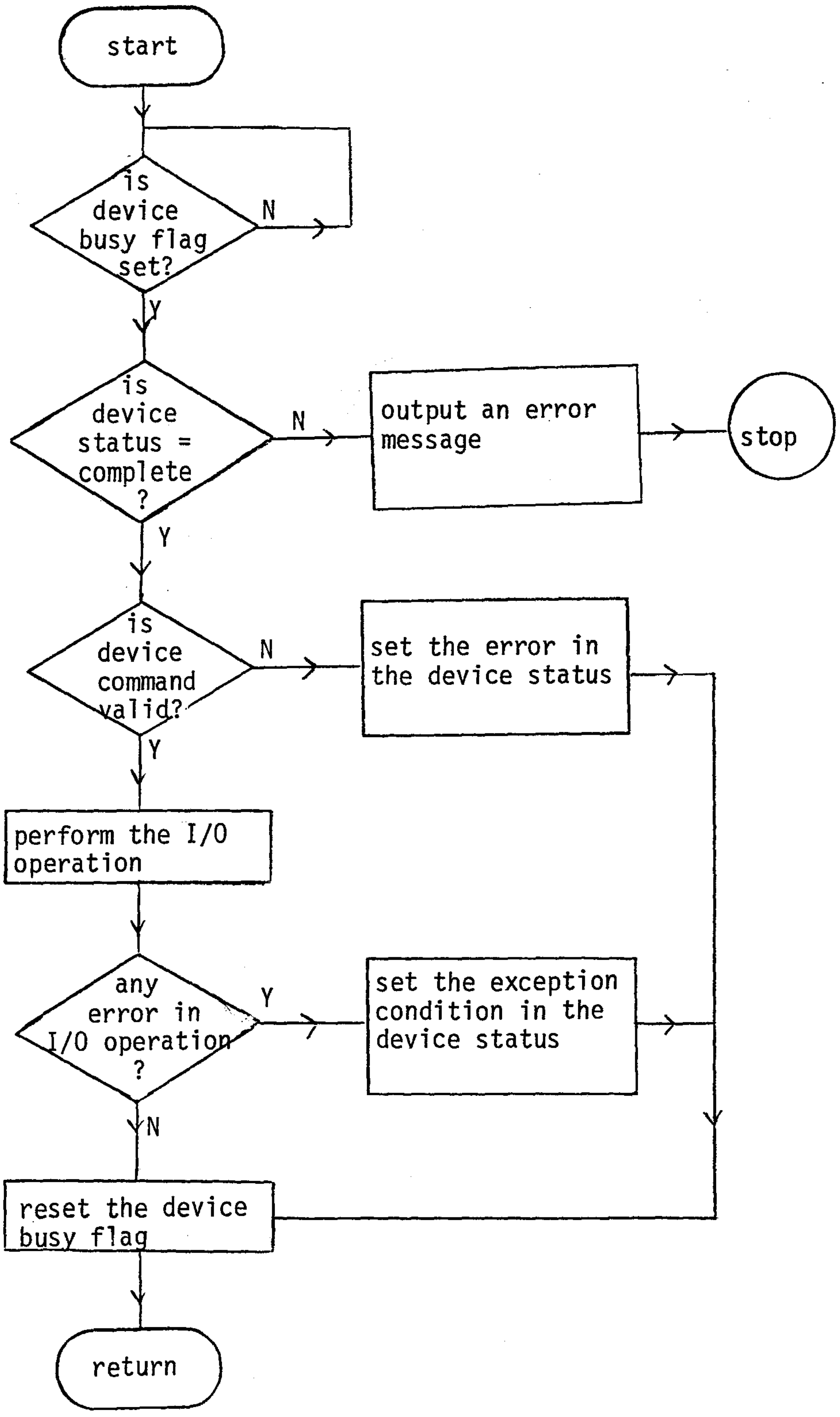

start
is device busy flag set?
N
Y
is device status = complete ?
N
output an error message
stop
Y
is device command valid?
N
set the error in the device status
Y
perform the I/O operation
any error in I/O operation ?
Y
set the exception condition in the device status
N
reset the device busy flag
return

Fig 4.1 Device simulation routine

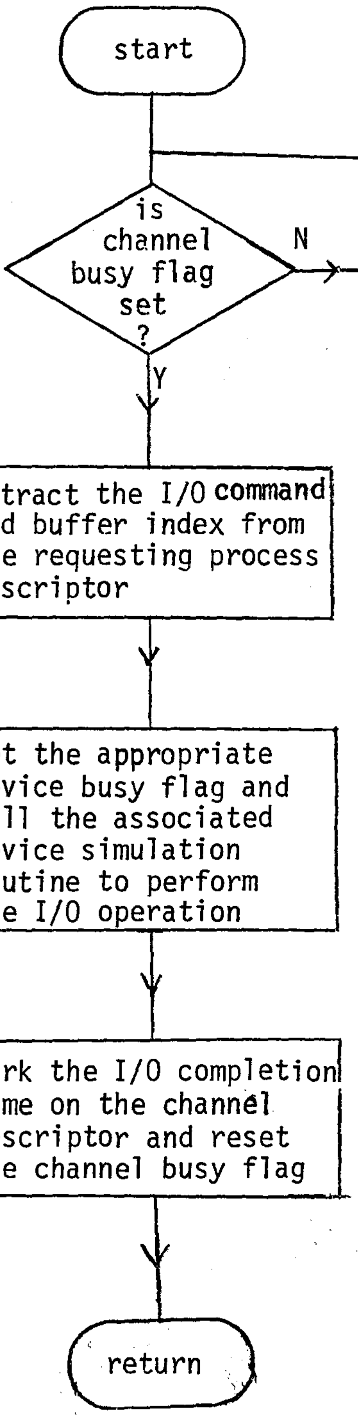

start
is channel busy flag set ?
N
Y
Extract the I/O command and buffer index from the requesting process descriptor
set the appropriate device busy flag and call the associated device simulation routine to perform the I/O operation
mark the I/O completion time on the channel descriptor and reset the channel busy flag
return

Fig 4.2 Channel control process

The Channel process resets the busy flag after the requested I/O operation has been performed.

4.4 Simulator for CPU Clock and Timer

The CPU clock plays an important role in the simulated MOS. From the current time in the clock and the I/O completion interrupt time recorded in the channel descriptor, the CPU knows when to service the I/O interrupt. Thus, the parallel processing activity of the CPU and channels can be simulated.

The CPU clock is represented by an integer variable. It is updated at the end of every virtual instruction cycle and when the Idling process is called to force the CPU to wait for a certain number of time units (the difference between the smallest I/O interrupt due time recorded in the channel descriptors and the current CPU clock time).

The timer hardware is simulated by the procedure Timer. As shown in Fig. 4.3, the Timer procedure performs the following tasks:

1. Decrements the supervisor location TM by t units of time, where t is the virtual instruction execution time or the CPU idling time.
2. Updates the CPU clock.
3. Sets the timer interrupt register TI and the I/O interrupt register IOI if the corresponding types of interrupts are due to occur.

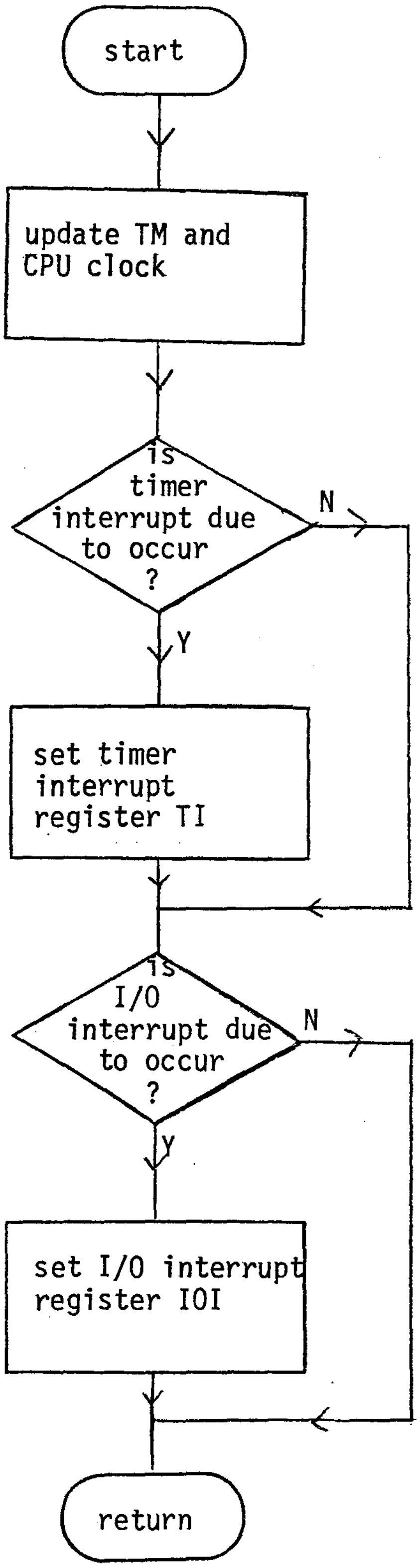

start
update TM and CPU clock
is timer interrupt due to occur ?
N
Y
set timer interrupt register TI
is I/O interrupt due to occur ?
N
Y
set I/O interrupt register IOI
return

Fig 4.3 The timer simulator

4.5 Simulator for Paging Hardware

As we have mentioned in Section 2.4, user storage addressing while in slave mode is accomplished through paging hardware. As illustrated in Fig. 4.4, the paging hardware is simulated by the procedure NLMAP. A virtual address is mapped into user storage through the transformation defined in the equation (2.4.1) (pp.10). The protection interrupt register PI will be set if an invalid virtual machine instruction is detected.

4.6 Implementation of Queue Data Structures and Resource Semaphores.

As shown in Fig. 4.5, all the queue data structures are represented by linked lists. The counter is an integer variable which indicates the number of entries in the queue. The first and last are two pointer variables which point to the first and last element respectively of the queue.

A resource semaphore is represented by the following record structure:

```
type   semaphore  =  record
                       mutex: boolean;
                       counter: integer;
                       waiting: processqueue;
                     end
```

where mutex is a boolean variable used for controlling mutually exclusive access to the critical region of the resource; counter is an integer variable which represents the number of

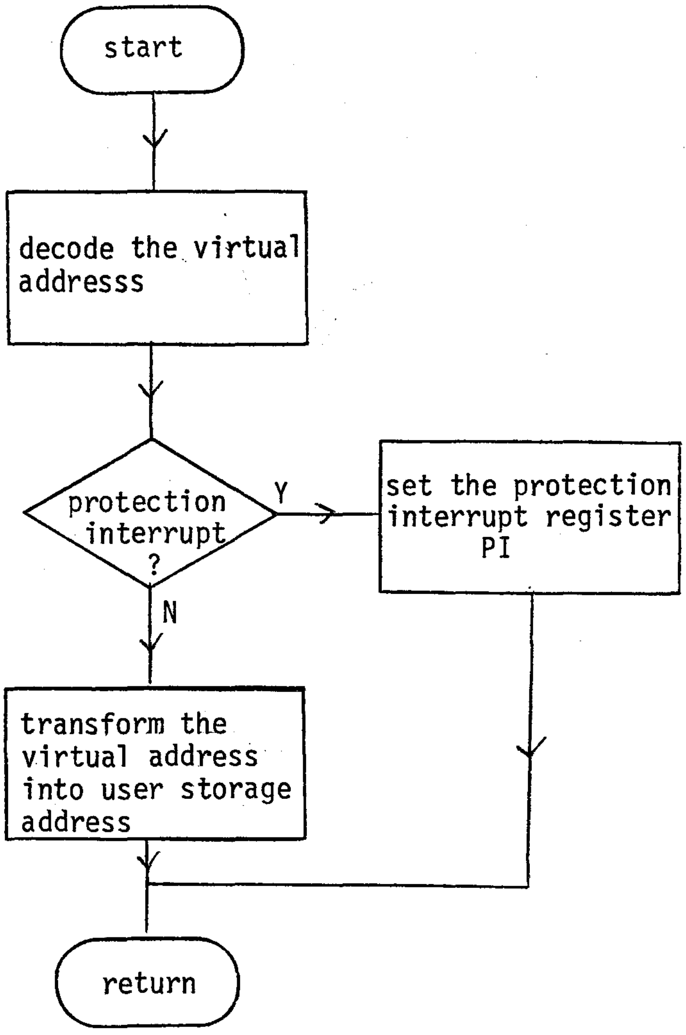

start
decode the virtual addresss
protection interrupt ?
Y
N
set the protection interrupt register PI
transform the virtual address into user storage address
return

Fig 4.4 Simulator for paging hardware

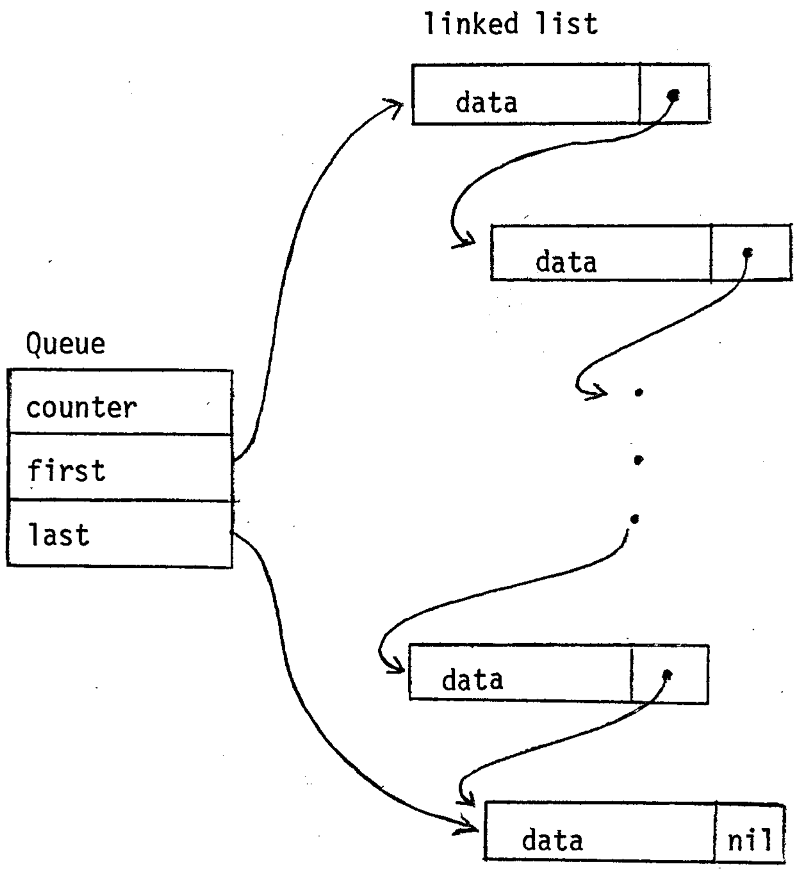


Fig 4.5 Queue data structure

available units of the resource; waiting is a queue of processes blocked by the resource and it is empty initially.

As shown in Algorithm 4.1, the two semaphore operations, wait and signal, are implemented by the procedure wait and the procedure signal, respectively. The function testandset tests and then sets the boolean variable mutex. It is used to prevent two processes simultaneously accessing the same semaphore. When the procedure wait is called by a process, the semaphore counter is checked. If it is greater than zero, it is decremented by one and the calling process continues; otherwise, the calling process is entered into the waiting queue and the status of the calling process is set to the blocked state. When the procedure signal is called, the semaphore waiting queue is examined. If one or more processes are waiting in the semaphore queue, one of them is transferred to the ready queue and its process status is changed to READY state; otherwise, the semaphore counter is incremented by one. The calling process continues in any case.

As shown in Algorithm 4.2, the wait and signal operations for a channel semaphore are implemented by the two procedures, request and release, respectively. The implementation of these two primitive operations is slightly different from Algorithm 4.1. In the procedure request, if the semaphore counter is greater than zero, then instead of simply decrementing the semaphore counter, the identifier of the requesting process is set in the associated channel descriptor and the channel is activated by setting its busy flag. Since the calling

Algorithm 4.1   The Semaphore Operations Wait and Signal

```
procedure  wait (RS: semaphore);
begin
  with RS do
    begin
      while testandset (mutex) do;
        if counter> 0 then counter: = counter -1 else
          begin
            enter (process, waiting);
            set the status of the calling process to BLOCKED state;
          end;
        mutex: = false;
    end;
end

procedure signal (RS: semaphore);
var
  blockedprocess: processid;
begin
  with RS do
    begin
      while testandset (mutex) do;
        if  not empty (waiting) then
          begin
             remove (blockedprocess, waiting);
             enter (process, ready);
             set the status of the removed process to READY state;
          end  else  counter: = counter +1;
        mutex: = false;
      end;
end
```

Algorithm 4.2 The Request and Release Operations for a Channel Semaphore

```
procedure  request (CS: semaphore);
begin
  with CS do
    begin
      while testandset (mutex) do;
        if  counter> 0 then
          begin
            counter: =  counter -1;
            set the identifier of the calling process in the
                  channel descriptor;
            activate the channel;
          end else enter (process, waiting);
        set the status of the calling process to BLOCKED state;
      mutex: = false;
    end;
end

procedure  release (CS: semaphore);
var
  blockedprocess: processid;
begin
  with CS do
    begin
      while testandset (mutex) do;
        if not empty (waiting) then
          begin
            remove (blockedprocess, waiting);
            set the identifier of the removed process in the
                   channel descriptor;
            activate the channel;
          end else counter: = counter +1;
      mutex: = false;
    end;
end
```

process has to wait until the I/O has been performed by the channel, its process status is set to BLOCKED state, and control is returned to the basic supervisor after the request operation. In the procedure release, if the semaphore waiting queue is not empty, then instead of entering the removed process into ready queue, its identifier is set in the channel descriptor, and the channel is activated. Although the implementation of the wait and signal operations for a channel semaphore is different from other resource semaphores, they are logically equivalent.

4.7 Implementation of Processes

A process can be specified by a procedure and data structure. The data structure associated with a process is the process descriptor, which defines the values of the processor registers and states of the process.

There are two kinds of processes in the MOS: the user processes and the supervisor processes. Each user process is associated with the activity of running a user job on the CPU. A user process is processed by the virtual machine emulator which simulates the execution of a user job on the virtual machine in slave mode. The operations of the virtual machine emulator can be summarized by the following procedure:

```
procedure VMEMULATOR;
  begin
    repeat
      fetch a virtual machine instruction into the instruction
         register;
      IC: = IC + 1;
      decode the instruction;
      execute the instruction;
      update the CPU clock and timer;
    until I/O interrupt or supervisor interrupt or protection
         interrupt or timer interrupt;
  end
```

A process descriptor is associated with each user process in the system. It contains the following information:

(1) priority level of the process;

(2) type of I/O request: read or write;

(3) main and backing store buffer indices;

(4) result of I/O request: invalid I/O command, end of file, or complete;

(5) process status: ready, running, blocked, suspended or terminated;

(6) values of the virtual machine CPU registers: IC, R, and C;

(7) contents of page table register;

(8) page table of the user process;

(9) job descriptor of the associated user job.

When a ready user process is scheduled to run on the CPU, the values of the virtual machine CPU registers which are stored in the process descriptor will be loaded into the corresponding registers of the CPU. These registers will be saved in the process descriptor when the process is interrupted.

Supervisor processes are run in master mode, and they are implemented by a set of procedures and process descriptors. Supervisor processes are executed by the HLP processor directly. They are not interruptable and can only be blocked by a logical resource or suspended to await for some condition at certain points. A blocked or suspended supervisor process will become active when the requesting resource has been allocated to it or when the awaiting condition has been satisfied. Since supervisor processes are processed by the high level language processor, we are not concerned with the instruction counter, general register, boolean toggle etc. Thus, instead of storing these CPU registers in the process descriptor, a variable which indicates the implemented procedure entry point is recorded in the process descriptor, the entry point is updated whenever the supervisor process is blocked or suspended. In addition to this entry point and the first five entities in a user process descriptor, a supervisor process descriptor contains other information depending on which supervisor process we are considering. A brief description of this additional information in a supervisor process descriptor follows:

1. Read-in-cards process:

   card-full-index: current input-spooler-buffer index used by the Read-in-cards process.

2. Job-to-drum process:

   new-job: the job descriptor of a new user job created by the Job-to-drum process;

   card-empty-index: current input-spooler-buffer index used by the Job-to-drum process;

   free-drum-delay: a boolean variable used to indicate whether the Job-to-drum process is blocked by the resource free-drum-frames.

3. Loader process:

   new-process: the user process identifier assigned to the user job currently being loaded.

   loading-page; an integer variable used to indicate the number of pages of the new user job that have been loaded by the loader process;

   sourceptr; a drum frame pointer points to the drum frame of the user job's source queue which is currently being loaded into the user storage.

4. Lines-from-drum process:

   user-job; the job descriptor which is currently being processed by the Lines-from-drum process.

   head-no; an integer variable used to indicate the current heading line number for the terminated user job in the system report produced by the

Lines-from-drum process.

print-heading: a boolean variable used to indicate whether the Lines-from-drum process is producing the heading or source/output data for the terminated user job.

Line-full-index: current output-spooler-buffer index used by the Lines-from-drum process.

listing: the source and output-data queues of the user job are combined into a single drum frames queue called listing. This queue is updated whenever a line has been reformated on the output spooler buffer by the Lines-from-drum process.

5. Print-lines process:

Line-empty-index: current output-spooler-buffer index used by the print-lines process

6. Get-put-data process:

user-process: the process identifier of the user process whose I/O request is currently being processed.

4.8 Updating of MOS statistics

The resource availability statistics are recorded and updated by the procedure updatestatistics, which is called by the procedures wait, signal, request, and release whenever they are called by a process.

Job characteristics statistics are maintained by the procedure _updatejobstatistics_. It is called by the Lines-from-drum process whenever a terminated user job is processed by the Lines-from-drum process.

The MOS statistics are written on a statistics file separated from the user jobs listing. It can be listed on the printer at the end of a run if required.

5. SAMPLE JOB STATISTICS AND THE MOS PERFORMANCE

In this section, we shall discuss the behavior of the MOS by examining the statistics of running several batches of user jobs of different classes.

To classify the user jobs into compute-bound, I/O-bound or balanced programs more precisely, we define the computation fraction of a user job as

$$\frac{\text{CPU time taken to run the job}}{\text{CPU time + I/O time taken to run the job}}$$

To study the behavior of the MOS, five batches of user jobs with computation fractions of 0.9, 0.7, 0.5, 0.3, and 0.1 have been created and run on the simulated hypothetical machine. The resource utilization and job characteristics statistics for these five batches of jobs are tabulated in Tables 5.1 and 5.2, respectively. To illustrate the resource utilization graphically, these statistics are plotted in Fig. 5.1. The analyses of these sample job statistics are summarized as follows:

1. The CPU was highly utilized while it was processing a batch of highly compute-bound jobs, and it was idling for most of the time while it was processing a batch of highly I/O-bound jobs.
2. Channels 1 and 2 are not very sensitive to the type of user jobs in the system, and they are only

Mean value / Job characteristics \ Computation fraction	0.9	0.7	0.5	0.3	0.1
Run time	269	319	384	477	635
Time in system	2660	2188	3772	4044	4404
User storage required (Words)	270	270	270	270	270
Input length (cards)	4.8	13.4	32.1	55.8	95.4
Output length (lines)	4.3	16.3	32.7	55.6	95.3
No. of concurrent user processes	3.6	3.4	6.5	5.2	3.7
Total system run time	26820	31322	42771	61042	92755

Table 5.2 Job Characteristics Statistics

Utilization / Resource \ Computation fraction	0.9	0.7	0.5	0.3	0.1
CPU	0.90	0.71	0.44	0.24	0.07
Channel 1	0.39	0.44	0.44	0.42	0.41
Channel 2	0.45	0.50	0.48	0.45	0.42
Channel 3	0.74	0.93	0.98	0.99	1.00
User storage	0.30	0.29	0.57	0.41	0.26
Drum frames	0.30	0.32	0.69	0.75	0.76

Table 5.1   Resource Utilization Statistics

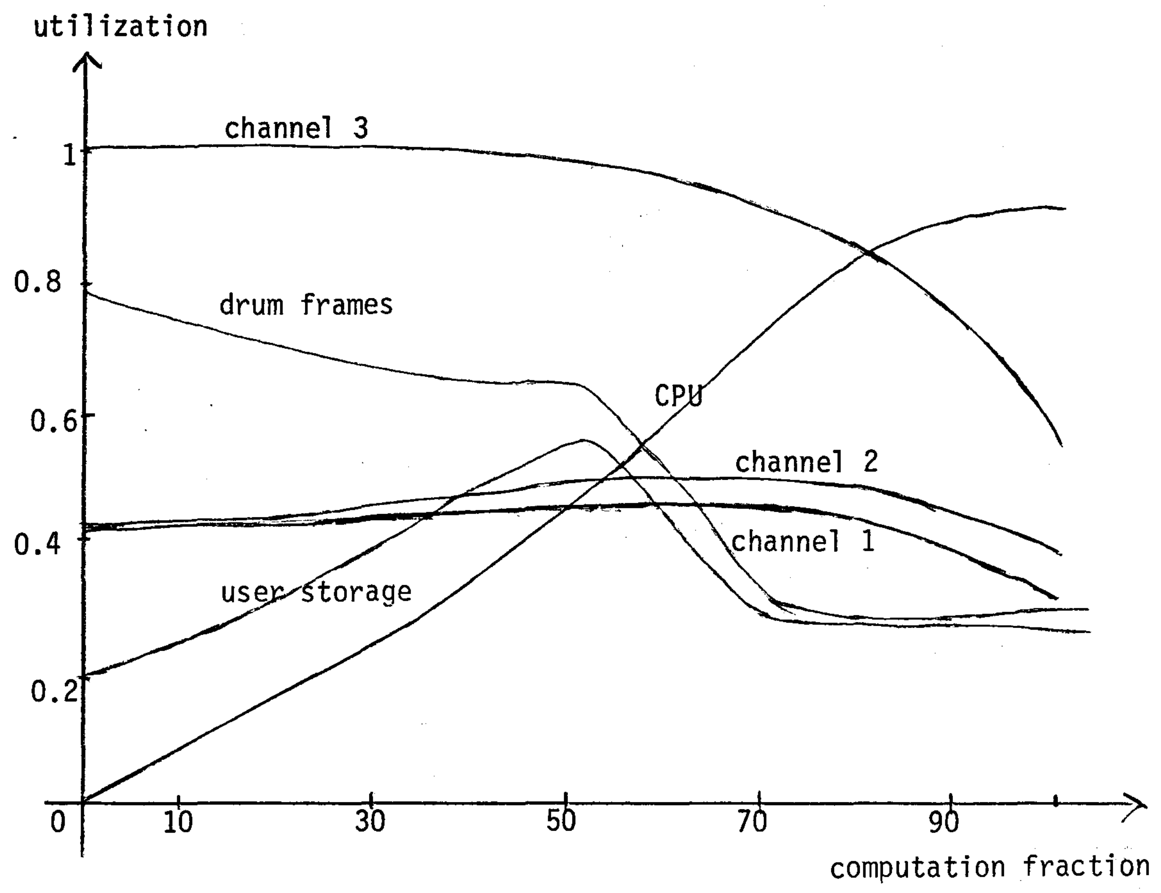


Fig 5.1 CPU and channels utilization

utilized for about 40% to 50% of the total system run time.

3. Channel 3 was utilized about 74% of the total system run time for a batch of user jobs with computation fraction of 0.9 and its utilization increased to 1 rapidly as the computation fraction of the user jobs decreased. This is due to the fact that a batch of heavily I/O-bound user jobs produces a lot of virtual reads and writes on the drum, and hence channel 3 will be busy for most of the time in transferring data between the main and backing stores.

4. The user storage has the highest utilization for a batch of balanced user jobs, and its utilization is lower for a batch of highly I/O-bound or compute-bound user jobs. This is due to the fact that for a batch of highly I/O-bound user jobs, most of the user jobs cannot be loaded into the user storage because the requesting free drum frames for storing output data are not available at loading time. For a batch of highly compute-bound user jobs, once a user job is terminated, the free user storage page frames are not utilized until new user jobs have been read in by the reader.

5. For the same reason as stated for channel 3, the drum frames are highly utilized for a batch of highly

I/O-bound jobs and its utilization decreases as the computation fraction of user jobs increases.

6. The average run time and turnaround time for a user job in the system increase as the computation fraction of the user jobs decreases. This is because the execution time of the I/O instructions is longer than those of the compute-type instructions.

To study the MOS behavior under different hardware assumptions, the five batches of sample jobs were run on the simulated MOS by changing the size of the user storage from 300 pages to 100 pages and 500 pages. We find that the utilization of the CPU and the three channels are almost the same as before, while the user storage has a higher utilization in the case of small capacity (100 pages) and a lower utilization in the case of large capacity (500 pages) relative to that with the original user storage size (300 pages). The turnaround times of user jobs are slightly longer for a small capacity of user storage, and slightly shorter for a large capacity of user storage.

One conclusion from the above analyses is that the capacity of the user storage in the range of 200 to 300 pages is sufficient for running a batch of small or medium size user jobs, and the overall CPU and channels can better be utilized for a batch of balanced or compute-bound user jobs.

By changing the I/O transfer rates of the three I/O devices, the utilizations of the CPU and the three channels are altered significantly for the running of the same five batches of sample jobs. The resource utilization statistics collected from running these sample jobs under the following new hardware assumptions are tabulated in Table 5.3:

Devices	I/O transfer rate	
	New assumptions	Previous assumptions
Reader	10	3
Printer	10	3
Drum	5	2

By comparing Table 5.3 with Table 5.1, we find that the CPU has a relatively lower utilization when compared to the results obtained with previous MOS software assumptions, and channels 1 and 2 have a relatively higher utilization than before, while the channel 3 utilization is almost the same as before, except for a batch of highly compute-bound user jobs. It appears that the performance of the MOS under the new hardware assumptions may be improved by attaching an additional reader and printer to the system.

Computation fraction / Utilization / Resource	0.9	0.7	0.5	0.3	0.1
CPU	0.43	0.27	0.17	0.09	0.03
Channel 1	0.66	0.65	0.60	0.58	0.55
Channel 2	0.78	0.71	0.64	0.60	0.56
Channel 3	0.94	0.98	0.98	0.99	0.99
User Storage	0.15	0.16	0.32	0.39	0.26
Drum frames	0.21	0.20	0.45	0.72	0.75

Table 5.3 Resource Utilization Statistics under the New I/O Transfer rates.

6. CONCLUSION OF THE PROJECT

The primary goal of this project was to demonstrate how an operating system for a hypothetical machine can be constructed. The end-product of the project, namely MOS, can be used as a teaching tool in a course on operating systems, with parts of the simulated operating system being used for demonstrations to the students. Products similar to MOS can be used as research tools for testing new ideas in operating systems primitives and design methodologies.

Even though the MOS and machine presented in this project deviate from a real system, it has the major characteristics of a small computer system which can support multiprogramming. Since the MOS has been designed and implemented in a modular and structured manner, it is easy to expand it to include some of the following omitted features:

(1) demand paging,

(2) an expanded virtual instruction set,

(3) a more general virtual machine that would permit multistep jobs and the use of language translators,

(4) a system to organize and handle files,

and

(5) an operator communication facility.

By simulating some more I/O devices in the system, this MOS can be modified and expanded to include the timesharing as a subsystem.

Unlike some other software projects, operating systems are better implemented in a language which allows coroutines or parallel processing [B4,D1]; such as PL/I with multitasking or concurrent PASCAL. Since this facility is not available at McMaster University, we have to implement the MOS with a sequential language, namely, PASCAL. It has some advantages over most other sequential languages, e.g. its powerful data structures as compared with those in FORTRAN. One disadvantage of implementing an operating system program in PASCAL is the inflexibility of changing the size of the simulated machine. The whole operating system program has to be recompiled if we wish to alter the size of the simulated machine.

Finally, we give the timing breakdown of this project:

(1) The preparatory step: reference collecting and studying took one month.

(2) The design stage: took one month.

(3) The implementation stage: coding, testing and debugging took two months.

(4) The documentation stage: writing up for the project took one and half month.

APPENDIX A: JOB, PROGRAM, AND DATA CARD FORMATS

A user job is submitted as a deck of control, program, and data cards in the following order;
<JOB card>, <Program>, <DATA card>, <Data>, <ENDJOB card>.

1. The <JOB card> contains six entries which appear in the following order:
   $AMJ, user A/CNO, time estimate, line estimate, user storage estimate, job identifier.
   These entries are explained as follows:
   (1) $AMJ: stands for A Multiprogramming Job, it must be punched at column 1 - 4.
   (2) User A/C NO: a four-character user account number (column 6 - 10). All the user account number are stored in the user account file (ACFILE). A user job can only be processed if the A/C NO appearing on the <JOB card> is one of the account numbers in the ACFILE.
   (3) Time estimate; estimated time required to run the job. If it is omitted, the default value will be taken.
   (4) Line estimate; estimated number of output lines to be printed. If it is omitted, default value will be taken.
   (5) User storage estimate; estimated number of user storage (in ten-word block unit) required to run the

job. If it is omitted, the number of source cards in the user job deck will be assumed.

(6) Job identifier: user job identifier (maximum up to ten characters) to be printed on the user's job listing.

All the entries in the <JOB card> are separated by a comma. The number of leading or trailing blanks for a numeric parameter in the <JOB card> are not significant, but all the entries of the <JOB card> must be specified within the first fifty columns and in the correct order. Two <JOB card> examples are illustrated as follows:

```
Example 1:  $AMJ,AM70,50,10,20,H.L. ONG
Example 2:  $AMJ,AM50,,0,,PAUL
```

In the first example, the user job specifies that the job may run for a maximum period of fifty time units, a maximum number of ten output lines may be printed, and twenty blocks of user storage should be allocated to the job.

The second example only specifies that there is no output for this job. Thus, default values for the job running time and user storage required will be assumed by the MOS.

2. Each card of the <Program> deck contains information in columns 1 - 50. The i^{th} card contains the initial contents of user virtual memory locations

$10(i-1),\ 10(i-1)+1,\ \ldots,\ 10(i-1)+9$

3. The <DATA card> has the format;

   $DATA (in cc 1 - 5)

4. The <Data> deck contains the user input data (in cc 1 - 50) to be retrieved by the virtual machine RD instructions.

5. The <ENDJOB card> has the format:

   $END (in cc 1 - 4)

   The <DATA card> may be omitted if there are no input data in a job.

   A complete deck of user job is illustrat-d by the following example:

```
Colomn 1                                            Colomn 50
↓                                                   ↓
$AMJ,AM10,50,5,3,JOHN
RD030LD010CR030BT007WR020CR010BT000H
*****
$DATA
1 THIS IS AN I/O BOUND PROGRAM. THE PROGRAM READS
  INPUT DATA CARDS INTO MEMORY AND PRINTS THEM ON
  THE PRINTER UNTIL FIVE ASTERISKS APPEARING ON
  THE FIRST FIVE COLUMNS HAVE BEEN READ IN.
*****
$END
```

The above program reads a data card into the memory. If the contents of the first five columns are not all equal to the character '*', the information in the data card is printed on the printer; otherwise, the job is terminated. The process is repeated until either it halts normally or is aborted by the system.

APPENDIX B: PROGRAM LISTING

A copy of the program listing is kept in the Department of Applied Mathematics, McMaster University.

BIBLIOGRAPHY

[A1] Atwood, J.W., Clark, B.L., Grushcow, M.S., Holt, R.C., Horning, J.J., Sevcik, K.C., and Tsichritzis, D., Project SUE Status Rep., Tech. Rep. CSRG-11. Comput. Systems Res. Group, University of Toronto, Toronto, 1972.

[B1] Brinch Hansen, P., The nucleus of a multiprogramming system, Comm. ACM 13,4, pp. 238-250, April 1970.

[B2] Brinch Hansen, P., Structured multiprogramming, Comm. ACM 15, 7, pp. 574-578, July 1972.

[B3] Brinch Hansen, P., Operating System Principles, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1973.

[B4] Brinch Hansen, P., The Architecture of Concurrent Programs, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1977.

[C1] Control Program -67/Cambridge Monitor System User's Guide, IBM Publication, 1969.

[D1] Dennis, J.B., Coroutines and parallel computation, Proc. Fifth Annu. Princeton Conf. Information Sci. and Systems, pp. 293-294, 1971.

[D2] Dijkstra, E.W., Cooperating Sequential Processes, Technological U., Eindhoven, The Netherlands.

[D3] Dijkstra, E.W., The structure of the T.H.E. multiprogramming system, Comm. ACM 11, 5, pp. 341-346, May 1968.

[H1] Habermann, A.N., Prevention of system deadlocks, Comm. ACM 12,7, pp. 373-377, July 1969.

[H2] Habermann, A.N., Introduction to Operating System Design, Science Research Associates, Inc., Chicago, Palo Alto, Toronto, 1976.

[J1] Jensen, K., and Wirth, N., PASCAL User Manual and Report, Springer-Verlag, New York, 1974.

[K1] Kwong, Y.S., On reduction of asynchronous systems, Theoretical Computer Science, 5, pp. 25-50, 1977.

[K2] Kwong, Y.S., Livelocks In Parallel Programs, Computer Science Technical Reports 78-CS-15 and 78-CS-16, McMaster University, Hamilton, Ontario, 1978.

[L1] Lehman, M.M., and Rosenfeld, J.L., Performance of a simulated multiprogramming system, Proc. AFIPS 1968 Fall Joint Comput. Conf., Vol. 33, pp. 1431-1442.

[L2] Liskov, B.H., The design of the Venus operating systems, Academic Press Inc., London, New York, 1976.

[S1] Shaw, A.C., and Weiderman, N.H., A Multiprogramming System for Education and Research, Proc. IFIP Congress 71, North-Holland Publishing Co, Amsterdam, The Netherlands pp. 1505-1509, 1971.

[S2] Shaw, A.C., The Logical Design of Operating Systems, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1974.

[T1] Tsichritzis, C.D., and Berstein, A.P., Operating Systems, Academic Press, New York, 1974.

[W1] Watson, R.W., Timesharing System Design Concepts, McGraw-Hill Book Co., New York, 1970.