

MULTI PROGRAMMING OPERATING SYSTEM (MOS) PROJECT

Contents

Sr. No.	Title of Experiments	Page No.
1	INTRODUCTION	3
2	FIRST PHASE	14
3	SECOND PHASE	18
4	THIRD PHASE	24
5	FINAL PHASE	34

1. INTORODUCTION OF PROJECT

Outcomes:

- Understand all general concept of Multiprogramming Operating System.
- Understand How to write assembly program to execute on multiprogramming operating system.

A1. INTRODUCTION

The appendix describes a tractable project involving the design and implementation of a multiprogramming operating system (MOS) for a hypothetical computer configuration that can be easily simulated (Shaw and Weideman, 1971). The purpose is to consolidate and apply, in an almost realistic setting, some of the concepts and techniques discussed in this book. In particular, the MOS designer/implementer must deal directly with problems of input-output, interrupt handling, process synchronization, scheduling, main and auxiliary storage management, process and resource data structures, and systems organization.

We assume that the project will be coded for a large central computer facility (The “host” system) which, on the one hand, does not allow users to tamper with the operating system or the machine resources but on the other hand, does provide a complete set of services, including filing services, debugging aids, and a good higher-level language. The global strategy is to simulate the hypothetical computer on the host and writes the MOS for this simulated machine. The MOS and simulator will consist of approximately 1000 to 1200 cards of program, with most of the code representing the MOS. The project can be completed over a period of about two months by students concurrently taking a normal academic load.

The characteristics and components of the MOS computer are specified in the next section. Section A3 outlines the format of user jobs. The path of a user job through the system, and the functions and main components of the MOS are described in Section A4. The following section (A5) then lists the detailed requirements for the project. In the final Sec. A6, some limitations of the project are described.

A2. MACHINE SPECIFICATIONS

The MOS computer is described from two points of view: the “virtual” machine seen by the typical user and the “real” machine used by the MOS designer/implementer.

1. The Virtual Machine:

The virtual machine viewed by a normal user is illustrated in *Fig. A-I*. Storage consists of a maximum of 100 words, addressed from 00 to 99; each word is divided into four one-byte units, where a byte may contain any character acceptable by the host machine. The CPU has three registers of interest: a four-byte general register **R**, a one-byte Boolean toggle **C**, which may contain either ‘*T*’ (true) or ‘*F*’ (false), and a two-byte instruction counter **IC**.

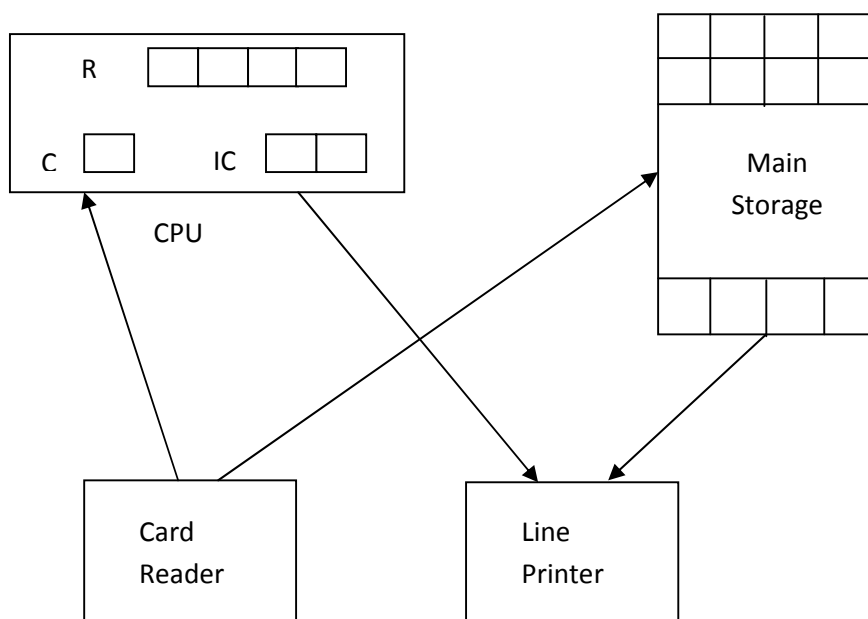


Fig A-1 Virtual user machine.

A storage word may be interpreted as an instruction or data word. The operation code of an instruction occupies the two high-order bytes of the word, and the operand address appears in the two low-order bytes. Table A-I gives the format and interpretation of each instruction. Note that the input instruction (**GD**) reads only the first 40 columns of a card and that the output instruction (**PD**) prints a new line of 40 characters. The first instruction of a program

must *always* appear in location 00. With this simple machine, a batch of compute-bound, IO-bound, and balanced programs can be quickly written. The usual kinds of programming errors are also almost guaranteed to be made. (Both these characteristics are desirable, since the MOS should be able to handle a variety of jobs and user errors.)

Table A-1 Instruction Set of Virtual Machine

Instruction		Interpretation
Operator	Operand	
<i>LR</i>	X1, X2	$R := [\quad];$
<i>SR</i>	X1, X2	$:= R;$
<i>CR</i>	X1, X2	If R = [] then C := 'T' else C := 'F';
<i>BT</i>	X1, X2	If C = 'T' then IC := ;
<i>GD</i>	X1, X2	Read([+i], i = 0... 9);
<i>PD</i>	X1, X2	Write([+i], i = 0... 9);
<i>H</i>		halt

2. The Real Machine

(a) Components

Figure A-2 contains a schematic of the real machine. The CPU may operate in either a *master* or a *slave* mode. In master mode, instructions from supervisor storage are directly processed by the higher-level language processor (HLP); in slave mode, the HLP interprets a 'micro program' in the read-only memory which simulates (emulates) the CPU of the virtual machine and accesses virtual machine programs in user storage via a paging mechanism. The HLP is any convenient and available higher-level language. (This organization allows the virtual machine emulator and the MOS to be coded in a higher-level language available on the host system, while maintaining some correspondence with real computers.)

The CPU registers of interest are:

C: a one-byte “Boolean&” toggle,
R: a four-byte general register,
IC: a two-byte virtual machine location counter,
PI, SI, IOI, TI: four interrupt registers,
PTR: a four-byte page table register,
CHST[i], i = 1, 2, 3: three channel status registers, and
MODE: mode of CPU, master’ or ‘slave’.

User storage contains 300 four-byte words, addressed from 000 to 299. It is divided into 30 ten-word blocks for paging purposes. Supervisor storage is loosely defined as that amount of storage required for the MOS.

Auxiliary storage is a high-speed drum of 100 tracks, with 10 four-byte words per track. A transfer of 10 words to or from a track takes one time unit. (Rotational delay time is ignored.) The card reader and line printer both operate at the rate of three time units for the 10 of one record. These devices have the same characteristics as the virtual machine devices; i.e., 40 bytes (10 words) of information are transferred from the first 40 card columns or to the first 40 print positions on a read or write operation, respectively.

Channels 1 and 2 are connected from peripheral devices to supervisor storage, while channel 3 is connected between auxiliary storage and both supervisor and user memory.

(b) Slave Mode Operation

User storage addressing while in slave mode is accomplished through paging hardware. The *PTR* register contains the length and page table base location for the user process currently running. The four bytes $a_0 a_1 a_2 a_3$, in the *PTR* have this interpretation: a_1 is the page table length minus 1, and $10a_2 + a_3$, is the number of the user storage block in which the page table resides, where a_1, a_2 , and a_3 are digits.

A two-digit instruction or operand address, $x_1 x_2$, in virtual space is mapped by the relocation hardware into the real user storage address:

$$10 [10 (10a_2 + a_3) + x_1] + x_2$$

Where (\cdot) means “the contents of address” and it is assumed that $x_1 \leq a_1$.

All pages of a process are required to be loaded into user storage prior to execution. It is assumed that each virtual machine instruction is emulated in one time unit. All interrupts occurring during slave mode operation are honored at the end of instruction cycles and cause a switch to master mode. The operations GD, PD, and H result in supervisor-type interrupt that is, “supervisor calls.” A program-type interrupt is triggered if the emulator receives an invalid operation code or if $x_1 > a_1$, during the relocation map (invalid virtual space address).

(c) *Master Mode Operation*

Master mode programs residing in supervisor storage have access to user storage and the CPU registers. The CPU is *not* interruptible in master mode however; an appropriate interrupt register is set when an interrupt-causing event (timer or IO) occurs. The interrupt registers may be interrogated and reset by the instruction *Test(x)*, which returns a value and has the effect:

```
if x = I then begin Test := IOI; IOI := 0 end else
if x = 2 then begin Test := PI; PI := 0 end else
if x = 3 then begin Test := SI; SI := 0 end else
if x = 4 then begin Test := TI; TI := 0 end else
if (IOI + PI + SI + TI) > 0 then Test := 1 else Test := 0;
```

All users IO is performed in master mode. An IO operation is initiated by the instruction

StartIO(Ch, S, D, n);

Where *Ch* is the channel number, *S* is an array of source blocks (IO word units), *D* is an array of destination blocks, and *n* is the number of blocks to be transmitted. If a *StartIO* is issued on a busy channel, the CPU idles in a *wait* state until the channel is free, whereupon the *StartIO* is accepted. (Issuing a *StartIO* on a busy channel is generally not advisable.) The status of any channel may be determined by examining the channel status registers *CHST*; *CHST* [*i*] = 1 if channel *i* is busy and *CHST* [*i*] = 0 when channel *i* is free (*i*=1, 2, 3).

To switch back to slave mode, the instruction

Slave(ptr, c, r, Ic)

is issued. *Slave* sets *PTR* to *ptr*, *C* to *c*, *R* to *r*, *IC* to *ic*, and then switches to slave mode, at the start of the emulator execution cycle.

Master mode instructions are normally executed in *zero* time units. However, it is occasionally necessary to force the CPU to wait for some specified time interval before continuing. This occurs implicitly when a *StartIO* on a busy channel is issued. An explicit wait is affected by the instruction

Superwait(t);

This causes the CPU to idle in a wait state for *t* unit of time.

(d) Channels

When a *StartIO* is accepted by the addressed channel *I*, *CHST[i]* is set to 1 (busy), and the IO transmission occurs completely in parallel with continued CPU activity, at the completion of the IO, *CFIST[i]* is set to 0 and an *IO Interrupt* signal is raised.

(e) Timer

The timer hardware decrements supervisor storage location *TM* by 1 at the end of every 10 time units of CPU operation. A *timer interrupt* occurs Whenever *TM* decremented to zero; the time continues decrementing at the same rate so that *TM* may also have negative values. *TM* may be set and interrogated in master mode.

(f) Interrupts

Four types of interrupts are possible:

- (1) Program: protection (page table length), invalid operation code
- (2) Supervisor: *GD*, *PD*, *H*.
- (3) IO: completion interrupts
- (4) Timer: decrement to zero

The events causing interrupts of types (1) and (2) can happen only in slave mode; events of type (3) and (4) can occur in both master and slave mode, and several of these events may happen simultaneously. The interrupt causing event is recorded in the interrupt registers regardless of whether the interrupt are inhibited (master mode) or enabled slave mode.

The interrupt register are set by an interrupt event to the following values:

(1) $PI = 1$: protection; $PI = 2$: invalid operation code

(2) $SI = 1$: GD ; $SJ = 2$: PD ; $SI = 3$: H

(3) $IOI = 1$: channel 1; $IOI = 2$: channel 2; $IOI = 4$: channel 3; if several IO completion interrupts are raised simultaneously, the values are summed; for example. $IOI = 6$ indicates that both channel 2 and channel 3 completion interrupts are raised.

(4) $TI = 1$: Timer

The following code describes the *hardware* actions on an interrupt in slave mode:

Comment Save state of slave process in supervisor storage locations c , r , and ic ;

$c := C$; $r := R$; $ic := IC$;

Comment Switch to master mode;

MODE := 'master'

comment Determine cause of interrupt and transfer control;

if IOI != 0 then go to IOInt else

if P1 != 0 then go to PROGInt else

if SI != 0 then go to SUPInt else

go to T1Mint;

Comment IOInt, PROGInt, SUPInt, and TIMint are supervisor storage locations; Note that the order of interrupt register testing implies a hardware priority scheme; this can be easily changed by master mode software.

A3. JOB, PROGRAM AND DATA CARD FORMATS

A user job is submitted as a deck of control, program, and data cards in the order:

<JOB card>, *<Program>*, *<DATA card>*, *<Data>*, *<ENDJOB card>*.

1. The *<JOB card>* contains four entries:

- (1) *SAMJ* cc. 1-4, A Multiprogramming Job
- (2) *<job Id>* cc. 5—8, a unique 4-character job identifier.
- (3) *<time estimate>* cc. 9—12, 4-digit maximum time estimate.
- (4) *<time estimate>* cc. 13—16, 4-digit maximum output estimate.

2. Each card of the *<Program>* deck contains information in card columns 1-40. The i_{th} card contains the initial contents of user virtual memory locations.

$$10(i - 1), 10(I - 1) + 1, \dots, 10(I - 1) + 9, \quad i = 1, 2, \dots, n,$$

Where n is the number of cards in the *<Program>* deck. Each word may contain a VM instruction or four bytes of data. The number of cards n in the program deck defines the size of the user space; i.e., n cards define $10 \times n$ words, $n \geq 10$.

3. The *<DATA card>* has the format:

4. The *(Data)* deck contains information in cc. 1—40 and is the user data retrieved by the VM *GD* instructions.

5. The *(JOB card)* has the format:

SEND cc. 1-4

<job Id> cc. 5—8, same *<job Id>* as *<JOB card>*

The *<DATA card>* is omitted if there are no *<Data>* cards in a job.

A4. THE OPERATING SYSTEM

The primary purpose of the MOS is to process a batched stream of user jobs efficiently. This is accomplished by multiprogramming systems and user processes.

A job *J* will pass sequentially through the following phases:

1. *Input Spooling.* *J* enters from the card reader and is transferred to the drum.
2. *Main Processing.* The program part of *J* is loaded from the drum into user storage. *J* is then ready to run and becomes a process *j*. Until *j* terminates, either normally or as a result of an error, its status will generally switch many times among:
 - (a) ready—waiting for the CPU.
 - (b) running—executing on the CPU.
 - (c) blocked—waiting for completion of an input-output request. Input-output requests are translated by the MOS into drum input-output operations.
3. *Output Spooling.* *J*'s Output, including charges, systems messages, and his original program, is printed from the drum.

In general, many jobs will simultaneously be in the main processing phase. The MOS is to be documented and programmed as a set of interacting processes. A typical design might have the following major processes:

Reading – Cards: Read cards into supervisor storage.

Job to Drum: Create a job descriptor and transfer a job to the drum

Loader: Load job into user storage

Get – Put-Data: Process VM input-output instructions.

Line -from- Drum: Read output lines from drum into supervisor storage.

Print - Lines: Write output lines on the printer.

The operating system is normally activated by slave mode operation. The interrupt handling routines will typically call the process scheduler (CPU allocator) after they service an interrupt.

A major task of the MOS is the management of hardware and software resources. These include user storage, drum storage, channel 3, software *buffers*, job descriptors, and the *CPU*.

The MOS is also responsible for maintaining statistics on hardware utilization and job characteristics. The following statistics are computed from software measurements:

1. *Resource Utilization*. Fraction of total time that each channel is busy, fraction of total time that the CPU is busy (in slave mode), mean user storage utilization and mean drum utilization.
2. *Job Characteristics*. Mean run time (**on** VM), mean time in system, mean user storage required, mean input length, and mean output length.

These statistics are to be printed at the end of a run.

A5. PROJECT REQUIREMENTS

Three sets of program modules must be designed and implemented:

1. Major simulators for hardware, including the interrupt system, timer, channels, reader, printer, auxiliary storage, user storage, and the slave mode paging system. (The HLP and supervisor storage is assumed available directly from the host system.)
2. The “micro-program” that emulates the VM.
3. The MOS.

These three parts should be clearly and cleanly separated. It should not be difficult to change the size and time parameters of the hardware, specifically drum and user storage size, 10 times, instruction times, and the timer “frequency.”

Students should work in small teams of two or three, each team doing the complete project. Several weeks after the project is assigned, a complete design of the MOS as a set of interacting processes is submitted. The design includes a description of the major processes *in*

the system and how they interact, the methods to be used for the allocation and administration of each resource, and the identification and contents of the main data structures. A batch stream of about 60 jobs (a “run”) should be prepared for testing purposes.

A6. SOME LIMITATIONS

The MOS and machine deviate from reality in simplifying some features of real systems and omitting others. Significant features that are lacking include: a more general virtual machine that would permit multistep jobs and the use of language translators, a system to organize and handle a loader variety of data files, an operator communication facility, and master mode operation of the CPU in nonzero time. The project specifications could be expanded in some of the above directions, but there appears to be an unacceptable overhead in doing so. Instead, similar tractable case studies emphasizing other aspects of operating systems, such as file systems or time-sharing, should be designed.

2. MULTIPROGRAMMING OPERATING SYSTEM (M1OS) PROJECT

FIRST PHASE

Outcomes:

- Students Should able to write their own Operating System Code to execute own assembly program
- Understand Single Programming Concept.

ASSUMPTIONS:

- Jobs entered without error in input file
- No physical separation between jobs
- Job outputs separated in output file by 2 blank lines
- Program loaded in memory starting at location 00
- No multiprogramming, load and run one program at a time
- SI interrupts for service request

NOTATION

M: memory; IR: Instruction Register (4 bytes)

IR [1, 2]: Bytes 1, 2 of IR/Operation Code

IR [3, 4]: Bytes 3,4 of IR/Operand Address

M [&]: Content of memory location &

IC: Instruction Counter Register (2 bytes)

R: General Purpose Register (4 bytes)

C: Toggle (1 byte)

MOS (MASTER MODE)

SI = 3 (Initialization)

Case SI of

1: Read

2: Write

3: Terminate

Endcase

READ

IR [4] \leftarrow 0

Read next (data) card from input file in memory locations IR [3,4] through IR [3,4] +9

If M [IR [3,4]] = \$END, abort (out-of-data)

EXEC UTEUSERPROGRAM

WRITE

IR [4] \leftarrow 0

Write one block (10 words of memory) from memory locations IR [3,4] through IR [3,4] + 9
to

output file

EXEC UTEUSERPROGRAM

TERMINATE

Write 2 blank lines in output file

MOS/LOAD

LOAD

$m \leftarrow 0$

While not e-o-f

Read next (program or control) card from input tile in a buffer

Control card: \$EOF, end-while

\$DTA, MOS/STARTEXECUTION

\$EOJ, end-while

Program Card: If $m = 100$, abort (memory exceeded)

Store buffer in memory locations m through $m + 9$

$m \leftarrow m + 10$

End-While

STOP

MOS/STARTEXECUTION

$IC \leftarrow 00$

EXECUTEUSERPROGRAM

EXECUTEUSERPROGRAM (SLAVE MODE)

Loop

IR4 \leftarrow M[IC]

IC \leftarrow IC+1

Examine IR [1, 2]

LR: R \leftarrow M [IR [3, 4]]

SR: R \rightarrow M [IR [3, 4]]

CR: Compare R and M [IR3, 4]]

if equal C T else C 4—F

BT: If C=T then IC \leftarrow R [3, 4]

GD: SI=1

PD: SI=2

H: SI=3

End-Examine

End-Loop

3. MULTIPROGRAMMING OPERATING SYSTEM (MOS) PROJECT

SECOND PHASE

Outcomes:

- Understand all Memory Management techniques in Operating System.
- Understand How to manage main memory using paging mechanism and how to execute the assembly program.

ASSUMPTIONS:

- Jobs may have program errors
- PI interrupt for program errors introduced
- No physical separation between jobs
- Job outputs separated in output file by 2 blank lines
- Paging introduced, page table stored in real memory
- Program pages allocated one of 30 memory block using random number generator
- Load and run one program at a time
- Time limit, line limit, out-of-data errors introduced
- TI interrupt for time-out error introduced
- 2-line messages printed at termination

NOTATION

M:	memory
IR:	Instruction Register (4 bytes)
IR [1,2]:	Bytes 1,2 of IR/Operation Code
IR [3, 4]:	Bytes 3, 4 of IR/Operand Address
M[&]:	Content of memory location &
IC:	Instruction Counter Register (2 bytes)
R:	General Purpose Register (4 bytes)
C:	Toggle (1 byte)
PTR:	Page Table Register (4 bytes)
PCB:	Process Control Block (data structure)
VA:	Virtual Address
RA:	Real Address

TTC: Total Time Counter
 LLC: Line Limit Counter
 TTL: Total Time Limit
 TLL: Total Line Limit
 EM: Error Message
 ← Loaded/stored/placed into

INTERRUPT VALUES

SI = 1 on GD
 = 2 on PD
 = 3 on H
 TI = 2 on Time Limit Exceeded
 PI = 1 Operation Error
 = 2 Operand Error
 = 3 Page Fault

EM **Error**

0 No Error
 1 Out of Data
 2 Line Limit Exceeded
 3 Time Limit Exceeded
 4 Operation Code Error
 5 Operand Error
 6 Invalid Page Fault

BEGIN

INITIALIZATION

SI=3, TI=0

MOS (MASTER MODE)

Case TI and SI of

<u>TI</u>	<u>SI</u>	<u>ACTION</u>
0	1	READ
0	2	WRITE
0	3	TERMINATE (0)
2	1	TERMINATE (3)
2	2	WRITE, THEN TERMINATE (3)
2	3	TERMINATE (0)

Case TI and PI of

<u>TI</u>	<u>PI</u>	<u>ACTION</u>
0	1	TERMINATE (4)
0	2	TERMINATE (5)
0	3	If Page Fault Valid, ALLOCATE, update page Table, Adjust IC if necessary, EXECUTE USER PROGRAM OTHERWISE TERMINATE (6)
2	1	TERMINATE (3, 4)
2	2	TERMINATE (3, 5)
2	3	TERMINATE (3)

READ

If next data card is \$END, TERMINATE (1)

Read next (data) card from input file in memory locations RA through RA + 9

EXECUTEUSERPROGRAM

WRITE

LLC \leftarrow LLC +1

If LLC > TLL, TELMINATE (2)

Write one block of memory from locations RA through RA + 9 to output file

EXECUTEUSERPROGRAM

TERMINATE (EM)

Write two blank line in output file

Write two lines of appropriate terminate message as indicated by EM

LOAD

While not e-o-f

Read next (program or control) card from input file in a buffer

Control card: \$AMJ, create and initialize PCB,

ALLOCATE (Get Frame for Page Table)

Initialize Page Table and PTR

Endwhile

\$DTA, STARTEXECUTION

\$END, end-while

Program Card; ALLOCATE (Get Frame for Program Page)

Update Page Table.

Load Program Page in Allocated Frame

End-While

End-While

STOP

STARTEXECUTION

$IC \leftarrow 00$

EXECUTEUSERPROGRAM

END (MOS)

EXECUTEUSERPROGRAM (SLAVE MODE)

ADDRESS MAP (VA, RA)

Accepts VA, either computes & returns RA or sets $P1 \leftarrow 2$ (Operand Error) or $P1 \leftarrow 3$ (Page Fault)

LOOP

ADDRESSMAP (IC, RA)

If $PI \neq 0$, End-LOOP (F)

$IR \leftarrow M[RA]$

$IC \leftarrow IC + 1$

ADDRESSMAP (IR[3,4], RA)

If $PI \neq 0$, End-LOOP (E)

Examine IR[1,2]

LR: $R \leftarrow M[RA]$

SR: $R \rightarrow M[RA]$

CR: Compare R and M [RA]

If equal $C = T$ else $C \leftarrow F$

BT: if $C = T$ then $IC \leftarrow IR[3,4]$

GD: $SI = 1$ (Input Request)

PD: $SI = 2$ (Output Request)

H: $SI = 3$ (Terminate Request)

Otherwise $PI \leftarrow$ (Operation Error)

End-Examine

End-LOOP (X) $X = F$ (Fetch) or E (Execute)

SIMULATION

Increment TTC

If $TTC = TTL$ then $TI \leftarrow 2$

If SI or P1 or $TI \neq 0$ then Master Mode, Else Slave Mode

4. MULTIPROGRAMMING OPERATING SYSTEM (MOS) PROJECT

THIRD PHASE

Outcomes:

- Understand all Scheduling of processes for the execution with Operating System.
- Understand and implement any CPU scheduling algorithm for the execution of user assembly program

ASSUMPTIONS (Added):

- Multiprogramming and virtual memory added
- TI “time slice out” interrupt introduced
- Paging retained without even odd restrictions
- I/O Processing through 3 channels introduced
- Spooling and buffering for i/o through channels introduced
- Drum (secondary storage) introduced
- I/O interrupt introduced

NOTATION S (Added):

TS: Time Slice

TSC: Time Slice Counter

CH_i: Channel i $i=1,2,3$

RD: Read

WT: Write

IS: Input Spool

OS: Output Spool

LD: Load

SWP: Swap

eb(q): Empty buffer (queue)

ifb(q): Inputful buffer (queue)

ofb(q): Outputful buffer (queue)

LQ: Load queue

RQ: Ready queue

SQ: Swap queue

IOQ: Input-Output (read/write) queue

TQ: Terminate (output spool) queue

IRi: Interrupt Routine for channel i i = 1, 2, or 3

SPOOLING AND BUFFERING INFO

- *Buffer Pool: 3 Types: Empty, Inputful, Outputful*

- *Channels: 3*

 - Channel 1: Cardreader to Supervisor Memory*

 - Channel 3: Supervisor Memory and Drum (either way)*

 - Channel 2: Supervisor Memory to Printer*

- *Spooling: Input and Output*

(a) Input (Before Execution): Program and data cards transferred from Card Reader to Drum Performed by Channels 1 and 3

Channel 1:

- Started with an Empty buffer*

- Fills it with the next card from card reader*

- Returns Inputful buffer*

Channel 3:

- Started with the next Inputful buffer, and an available drum track*

- Writes the buffer to the drum track*

- Returns an Empty buffer*

OUTPUT *(After the program has terminated)*

- Output lines stored on drum tracks during execution sent to printer*

- Performed by channels 3 and 2*

Channel 3:

- Started with an Empty buffer, and the next output drum track*

- Fills the buffer with the next output line from the drum track*

- Returns an Outputful buffer*

Channel 2:

- Started with the next Outputful buffer*

Sends it to the printer

Returns an Empty buffer

Note that a channel cannot be started if appropriate type of buffer is not available.

INTERRUPT VALUES (Added):

TI = 1 on Time Slice Out.

IOI: 1 channel 1 done

2 channel 2 done

4 channel 3 done

Error Message Coding: (No Change)

BEGIN

INITIALIZATION

IOI= 1

MOS (MASTER MODE)

Case TI and SI of

<u>TI</u>	<u>SI</u>	<u>ACTION</u>
0 or 1	1	Move PCB, RQ → IOQ (Read)
0 or 1	2	Move PCB, RQ → IOQ (Write)
0 or 1	3	Move PCB, RQ → TQ (Terminate [0])
2	1	Move PCB, RQ → TQ (Terminate [5])
2	2	Move PCB, RQ → IOQ (Write) then TQ (Terminate [3])
2	3	Move PCB, RQ → TQ (Terminate [0])

Case TI and PI of

<u>TI</u>	<u>PI</u>	<u>ACTION</u>
0 or 1	1	Move PCB, RQ → TQ (Terminate [4])
0 or 1	2	Move PCB, RQ → TQ (Terminate [5])
0 or 1	3	Page Fault

If Valid

If Frame Available

Allocate

Update Page Table

Adjust IC, if necessary

Else

Move PCB, RQ \rightarrow SQ

Else

Move PCB, RQ \rightarrow TQ (Terminate [6])

2	1	Move PCB, RQ \rightarrow TQ (TERMINATE [3, 4])
2	1	Move PCB, RQ \rightarrow TQ (TERMINATE [3, 5])
2	1	Move PCB, RQ \rightarrow TQ (TERMINATE [3])

Case IOI of

0	No Action
1	IR1
2	IR2
3	IR2, IR1
4	IR3
5	IR1, 1R3
6	IR3, 1R2
7	IR2, IR1, 1R3

IR1

Read next card in given eb, change status to ifb, place on ifb (q)

If not e-o-f and eb(q) not empty

Get next eb

Start Channel 1

Examine ifb

\$AMJ: Create and initialize PCB
Allocate frame for Page Table
Initialize Page Table and PTR
Set $F \leftarrow P$ (Program cards to follow)

Change Status from ifb to eb

Return buffer to eh(q)

\$DTA: Set $F \leftarrow D$ (data cards to follow)

Change status from ifb to eb

Return buffer to eb(q)

\$END: Place PCB on LQ, change status from jib to eb, return buffer to eb(q)

Otherwise place ifb on ifb(q), save F information (program or data card for channel 3)

IR2

Print given ofb, change status from ofb to eb

Return buffer to eb(q)

If ofb(q) not empty,

Get next ofb

Start Channel 2

IR3

(First, complete the assigned task and the follow up action for channel 3 for each possible task, and then assign new task to it in priority order.)

Case Task of

IS: Write given ifb on given track

Place track number in P or D part of PCB

Change status from ifb to eb

Return buffer to eb(q)

OS: Read information (Output line) from given track into given eb

Change status from eb to ofb

Return buffer to ofb(q)

Release track

Decrement line count in PCB

If last line, fill two other ebs (if available) with blanks, change status from eb to ofb and place the buffers on ofb(q)

Release PCB, all remaining drum tracks and all memory blocks.

Prepare 2 lines of messages from next PCB (if available) on TQ, move them into ebs (if available), change status from eb to olb, and place these buffers also on ofb(q)

LD: Load program card from given track into indicated memory block

Decrement count in PCB

If zero, place PCB on RQ after all the initializations

RD: Read data card from given track into indicated memory block

Decrement count in PCB

Move PCB to RQ after setting $TSC \leftarrow 0$

WT: Write information from the indicated memory block to the given track

Increment line count (TLC) in PCB

If $TI = 2$ or 3 , move PCB to TQ

Else move PCB to RQ after setting $TSC \leftarrow 0$

SQ(W): Write the information from the victim frame to the given track.

Locate drum track with faulted page

Task $\leftarrow SQ(R)$

Start Channel 3

SQ(R): Read drum track with faulted page in newly allocated frame Move PCB, $SQ \rightarrow RQ$ after setting $TSC \leftarrow 0$

End-Case

(Now Assign New Task in Priority Order)

If a PCB on TQ (output spool first)

 If eb(q) not empty

 Get next buffer from eb(q)

 Find track number of next output line

 Task \leftarrow OS

 Start Channel 3

Else (input spool next)

 If ifb(q) not empty and a drum track available

 Get next buffer from ifb(q)

 Get a drum track

 Task \leftarrow IS

 Start Channel 3

Else (load next)

 If a PCB on LQ (load next) and a memory frame available

 Find track number of next program card

 Allocate a frame

 Update Page Table

 Task \leftarrow LD

 Start Channel 3

Else (now i/o)

 If a PCB on IOQ

 If Read (GD)

 If no more data card

 Move PCB, IOQ \rightarrow TQ (Terminate [3])

 Else

 Find track number of next data card

 Get memory RA

 Task \leftarrow GD

 Start Channel 3

 Else If Write (PD)

 If TLC > TLL, Move PCB IOQ \rightarrow TQ (Terminate[2])

```

Else
    Get a drum track, if available
    Update PCB
    Find memory RA
    Task  $\leftarrow$  PD
    Start Channel 3
Else (allocate memory)
    If a PCB on SQ
        If a memory frame now available
            Allocate
            Update page Table
            Adjust IC, if necessary
            Move PCB SQ  $\rightarrow$  RQ with TSC  $\leftarrow$  0
        Else
            Run page replacement algorithm
            Find a victim frame
            Allocate and De allocate this frame
                by updating both page tables
            If victim frame not written into,
            locate drum track for faulted page
            Task  $\leftarrow$  SQ (R)
            Start Channel 3
    Else
        Task  $\leftarrow$  SQ(W)
        Start Channel 3
(END OF 1R3)

```

START CH_i

```

Adjust IOI (Subtract 1,2, or4)
Reset Ch timer to zero
Set Ch flag to busy.

```

STARTEXECUTION

```

IC  $\leftarrow$  00

```

EXECUTEUSERPROGRAM

END (MOS)

EXECUTEUSERPROGRAM (SLAVE MODE)

ADDRESS MAP (VA, RA)

Accepts VA, either computes & returns RA or sets $PT \leftarrow 2$ (Operand Error) or

$PI \leftarrow 3$ (Page Fault)

LOOP

ADDRESSMAP (IC, RA)

If $PI \neq 0$, End-LOOP (F)

$IR \leftarrow M[RA]$

$IC \leftarrow IC+1$

ADDRESSMAP (IR[3,4], RA)

If $PI \neq 0$, End-LOOP (E)

Examine IR[1,2]

LR: $R \leftarrow M[RA]$

SR: $R \rightarrow M[RA]$

CR: Compare R and M [RA]

If equal $C \leftarrow T$ else $C \leftarrow F$

BT: If $C=T$ then $IC \leftarrow IR[3,4]$

GD: $SI = 1$ (Input Request)

PD: $SI = 2$ (Output Request)

H: $SI = 3$ (Terminate Request)

Otherwise $PI \leftarrow 1$ (Operation Error)

End-Examine

End-LOOP (X) $X = F$ (Fetch) or E (Execute)

SIMULATION

Increment TTC

If $TTC = TTL$ then $TI \leftarrow 2$

Increment TSC

If $TSC = TS$, then $TI \leftarrow 1$

For all $CH_i, i = 1, 2, 3$

 If CH_i flag busy,

 Increment CH_i timer

 If CH_i timer = CH_i total time

 Increment IOI accordingly

 (Set channel completion interrupt)

End - For

If SI or PI or TI or IOI $\neq 0$ then Master Mode, Else Slave Mode

5. OPERATING SYSTEM PROJECT

FINAL VERSION

Outcomes:

- Understand how to load multiple programs into main memory.
 - Understand how to achieve multiprogramming on single processor system..
- i) There are 4 processors: the CPU, and the 3 channels. CPU is master, channels are slaves. Channels are started with fix task by CPU. CPU then becomes free, and thus CPU and all the channels could be on at the same time. When channels complete the assigned task, they inform the CPU (the master) of task completion, and thus of readiness to accept the next task, by sending interrupt signals called Input-Output Interrupt (IOI). Recall that Channel 1 and 2 takes five time units each, and Channel 3 takes two time units to complete an assigned task.

In order to distinguish which channel is sending interrupt, different interrupt values are sent by channels:

Ch 1 Set IOI to 1

Ch 2 Sets IOI to 2

Ch 3 Sets IOI to 4

Of course, different channels may send (raise) interrupt signals at the same time, thus IOI can take a value between 0 (no interrupt) and 7.

In previous implementations, it was implicitly assumed that only one of the 4 processors could be active at a time.

We will now explicitly simulate concurrent operation by the CPU, and the 3 channels by keeping timers for each channel:

Ch1T, Ch2T, Ch3T

We will also keep flag for each channel to indicate whether channel is busy or free. The CPU should interrogate this flag to ensure that it assigns a task to only a free channel:

Ch1F, Ch2F, Ch3F.

- ii) In earlier versions, we had only one user program in memory. We will now implement multiprogramming — several user programs in memory at the same time
- iii) The only program in memory had complete control of the CPU, when the CPU was assigned to it by the operating system. Now that there programs in memory at the same time, operating system will allocate the CPU to each process by turn for a fixed amount of time called TIME QUANTUM or TIME INTERVAL or TIME SLICE. This is simulated by keeping a timer for each process:

TS (for Time Slice)

and when TS reaches time quantum, a timer interrupt (TI) is raised. Another timer

TT (for Total Time)

is also kept for each process, and when this timer reaches the time estimate indicated on the job card, then also a timer interrupt (TI) is raised. In order to distinguish between these two types of interrupts, TS expiry sets TI to 1, and IT expiry sets TI to 2. Since both interrupts may occur at the same time, value of TI will vary between 0 (no interrupt) and 3.

- iv) In earlier version, when Operating Systems Services were needed, service calls for operating systems were directly made by user programs through routines such as MOS/READ. Now user program will communicate with Operating System only through interrupts.

Two types of interrupts are provided for this purpose.

Interrupts

PI and SI are set in slave mode when user program runs on CPU.

IOI and TI are external interrupts, which will be simulated outside of slave mode.

MOS will be interrupt-driven, i.e., which functions in MOS will execute will be completely determined by the interrupts which have occurred.

Since the project simulates interrupt, and interrupt may occur any time, MOS should check and service interrupt after the execution of every user instruction. Thus the control of CPU will switch to MOS after CPU executes one user instruction in slave mode. When MOS has serviced all the interrupts raised during the execution of this instruction, control of CPU is given back to user program for execution of the next instruction.

Concurrent Operation

The CPU and the channels work concurrently. Thus the operations of

Input spooling (Ch 1 and Ch3)

Loading (Ch3)

Execution (CPU)

Input-Output, i.e., GD & PD (Ch3)

and Output Spooling (Ch3 and Ch2)

could, and perhaps most of the time will, be taking place simultaneously except for the fact that Ch3 can do only one thing at a time.

Channel 3

Channel 3 performs four different tasks:

Input Spooling

Loading

Input-Output (GD & PD)

Output Spooling

It is possible that any of these four tasks could be waiting to be assigned to Channel 3. The operating system will lay down a priority (most likely static rather than dynamic) system to decide as to which task should be assigned to Channel 3.

An Outline of Project Design

Begin

Definitions and Initializations

While there is still any job in the system

 Slave Mode (user program, P1 & SI set here)

 Simulate 101 and TI

 Master Mode (MOS, service interrupts)

End-while

Wrap-up

End

Slave Mode

Same as before except:

- i. assume error may occur, check for it, and set P1 to 1 or 2
- ii. when page fault occurs, don't call MOS/ALLOCATE MEM, only set P1 to 3
- iii. replace service calls by interrupts as shown below

SPOOLING AND BUFFERING INFO

- *Buffer Pool: 3 Types: Empty, Inputful, Outputful*

- *Channels: 3*

Channel 1: Cardreader to Supervisor Memory

Channel 3: Supervisor Memory and Drum (either way)

Channel 2: Supervisor Memory to Printer

- *Spooling: Input and utpu*

(a) input (Before Execution): Program and data cards transferred from Card Reader to Drum

Performed by Channels 1 and 3

Channel 1:

Started with an Empty buffer

Fills It with the next card from card reader

Returns inputful buffer

Channel 3:

Started with the next inputful buffer, and an available drum track

Writes the buffer to the drum track

Returns an Empty buffer

OUTPUT (After the program has terminated)

Output lines stored on drum tracks during execution sent to printer

Performed by channels 3 and 2

Channel 3:

Started with an Empty buffer, and the next output drum track

Fills the buffer with the next output line from the drum track

Returns an Outputful buffer

Channel 2:

Started with the next Outputful buffer

Sends it to the printer

Returns an Empty buffer

Note that a channel cannot be started if appropriate type of buffer is not available.

Service Calls

Interrupts

MOS/READ (RA) SI ---1

MOS/WRITE (RA) SI ---2

MOS/TERMINATE SI ---3

Simulate IOI and TI

 Simulate IOI

 For all Chi (i = 1,2, or 3)

 If Chi F = I (Chi is busy)

 Then ChiT --- ChiT + 1

If $ChiT = (5 \text{ for } i = 1 \text{ and } 2, 2 \text{ for } i = 3)$

Then increment 101 (by 1 for $i = 1$, 2 for $i = 2$, and 4 for $i = 3$)

End — For

Simulate TI

For the process which is running

$TS \leftarrow TS + 1$

If $TS = 10$ (assuming 10 units for time quantum)

Then $TI \leftarrow TI + 1$;

$TT \leftarrow TT + i$

If $TT = \text{time estimate in } \$ \text{ AMJ card}$

Then $TI \leftarrow TI + 2$;

Master Mode

Operating system maintains several queues:

Buffer Pool

Queue for empty buffers (EBQ)

Queue for inputful buffers (IFBQ)

Queue for outputful buffers (OFBQ)

PCB

Load queue (LQ): Input spool completed, PCB waiting for loading or being loaded by Channel 3

Ready queue (RQ): Loading completed, PCB ready for execution by CPU

Memory queue (MQ): PCB was executing, now waiting to get memory

Input-Output Queue (IOQ): PCB was executing, now waiting for I/O (GD/PD) to be completed by Channel 3

Terminate queue (TQ): Execution completed, PCB waiting for output spooling or being output-spooled by Channel 3 — Channel 2 combine

MOS

Service all interrupts *if set*:

Case PI of:

- 0 return
- 1 op-code error message
 move PCB: RQ --- TQ
- 2 operand error message
 move PCB: RQ --- TQ
- 3 check page fault
 Invalid: move PCB: RQ --- TQ
 Valid: memory available?
 - Yes: allocate
 - No: (Graduate Group) Execute Page Replacement Algorithm (Undergraduate Group) Move PCB: RQ --- MQ

End-case

Case TI of

- o return
- 1 move PCB :RQ---RQ
- 2, 3 time-out error message
 move PCB: RQ --- TQ

Case SI of

- o return
- 1 (GD) if no more data card
 out-of-data error message
 move PCB: RQ --- TQ
 else move PCB: RQ --- IOQ
- 2 (PD) if print line — limit to exceed,
 print — line — limit exceeded error message
 move PCB: RQ---TQ
 else move PCB: RQ --- IOQ
- 3 (H) normal termination message
 move PCB: RQ --- TQ

IOI Interrupt

Coding of 101 interrupt is central to MOS Design, and will probably account for more than half of MOS code.

There should be three separate interrupt Routines (IR) coded for each channel:

Ch1IR, Ch2IR, Ch3IR

An interrupt is raised by a channel when it completes a task, and hence it is now free to accept a new task. Thus, 1-0 interrupt routines are written to perform two tasks in the order shown.

- Housecleaning after last task
- Assigning new task, if possible.

In the project, since channels are being simulated, an additional task of simulating channel operation (moving information appropriately from source to destination) must be done before anything else is done. Thus interrupt routines will be written to perform three tasks in the order shown.

- Move information for the last task
- Housecleaning after the last task
- Assigning new task, if possible

Following notations will be used:

EB Empty Buffer
EBQ Empty Buffer Queue
IFB Inputful Buffer
IFBQ Inputful Buffer Queue
OFB Outputful Buffer
OFBQ Outputful Buffer Queue

Coding will follow the following pattern:

Case IOI of

- o return
- I CH1IR
- 2 Ch2IR
- 3 Ch2IR, ChIIR
- 4 Ch3IR
- 5 Ch3IR, Ch1IR
- 6 Ch3IR, Ch2IR
- 7 Ch3IR, Ch2IR, ChIIR

Ch1IR

Move next card into the assigned EB

Place buffer on IFBQ

Start Ch 1

Start Ch1

If $Ch1F = 0$, and a card on card reader, and a buffer on EBQ

Then

Take an EB

$Ch1F \leftarrow 1$

$Ch1T \leftarrow 0$

Ch2IR

Move the assigned OFB to printer

Place buffer on EBQ

StartCh2

StartCh2

If $Ch2F = 0$, and a buffer on OFBQ,

Then

Take an OFB

$Ch2F \leftarrow 1$

$Ch2T \leftarrow 0$

Ch3IR

(Note: A defined variable TASK indicates the current Ch3 function:

IS (Input Spool)

LD (Load)

IO (GD/PD)

OS (Output Spool)

First perform simulation of last (one of four mentioned above) assigned task and related housecleaning functions, and then StartCh3:

Case TASK of:

IS: Move assigned IFB to assigned drum truck

Place buffer on EBQ

LD: Move next program card from drum truck (location in PCB on LQ) to assigned memory block.

Check if this was last program card. If so, move PCB: LQ --- RQ

[undergrad project only]

GD: Move next data card from drum truck (location in PCB on IOQ) to assigned memory block.

Move PCB: IOQ --- RQ

PD: Move a line image from assigned memory block to assigned drum track. Move

PCB: IOQ --- RQ

OS: Move next output line from drum track (location in PCB on TQ) to assigned ED.

Place buffer on OFBQ.

If this was the last output line, kill PCB.

End-case

StartCh3

(NOTE: There are 4 tasks, prioritize them, and assign in that order. Assume OS, IS, GD/PD, LD priority to be the priority order.)

If TQ and EBQ not empty

 If starting a new PCB, take buffer(s) from EBQ.

 Move message(s) from PCB to buffer(s)

 Take a buffer from EBQ

 Pick up pointer to next output line stored on the drum track from PCB

 TASK---OS

 Ch3F---I

 Ch3T---O

Else if IFBQ not empty,

 Take a buffer from IFBQ

 If \$AMJ

 then create and update PCB, F --- P

 Place buffer on EBQ

 Else if \$DTA

 Then F---D

 Place buffer on EBQ

 Else if \$END

 Then place PCB on LQ (undergrad project), RQ (graduate project)

 Place buffer in EBQ

 Else

 Allocate drum track

 TASK---IS

 Ch3F--- 1

 Ch3T---O

Else if IOQ not empty

If PCB indicates GD

 Find drum track with next data card and locate memory RA, both from PCB

 TASK---GD

Ch3F--- 1

Ch3T---O

Else

Locate memory RA of the next output line from PCB

Allocate drum track, update PCB

TASK---PD

Ch3F--- 1

Ch3T---O

Elseif LDQ not empty

Allocate memory, update PCB, find drum track of next program card from PCB

TASK---LD

Ch3F--- 1

Ch3T---O

NOTE: Graduate groups will have SWAP queue rather than LOAD queue.