**INTORODUCTION OF PROJECT**

**Outcomes:**

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

**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 Weiderman, 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.

**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.

**a) The Virtual Machine:**

The virtual machine viewed by a normal user is illustrated in *Fig. A-1.* 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.*

R

000

01

03

C

IC

MAIN STORAGE

CPU

LINE PRINTER

CARD READER

98

99

**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* appeal 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.)

|  |  |  |
| --- | --- | --- |
| Instruction | | Interpretation |
| Operator | Operand |
| LR | x1,x2 | R:=[α] |
| SR | x1,x2 | α: =R |
| CR | x1,x2 | If R: =[α] then C:=’T’ else C:=’F’ |
| BT | x1,x2 | If C=’T’ then IC:= α |
| GD | x1,x2 | Read([β+i],i=0….9) |
| PD | x1,x2 | Write([β+i],i=0….9) |
| H |  | halt |

**Table A-1 Instruction Set of Virtual Machine**

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, TI****:* 3 interrupt registers,

***PTR:*** a four-byte page table register,

***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.

***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 a0 a1 a2, a3, in the *PTR* have this interpretation: a1 is the page table length minus 1, and l0a2, + a3, is the number of the user storage block in which the page table resides, where a1, a2, and a3 are digits.

A two-digit instruction or operand address, x1 x2, in virtual space is mapped by the relocation hardware into the real user storage address:

10 [1O ( lOa2, + a3 ) + x1 ] + x2

Where (α] means “the contents of address” and it is assumed that x1<=a1.

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 x1, > a1, during the relocation map (invalid virtual space address).

***Master Mode Operation***

Add Description of Handling of PI,SI and TI interrupt

***Interrupts***

Three types of interrupts are possible:

(1) Program: Protection (page table length), invalid operation code

(2) Supervisor: *GD, PD, H.*

(4) Timer: Decrement to zero

The events causing interrupts of types (1) and (2) can happen only in slave mode; events of type 3 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 regard less 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: Opcode; P1= 2: invalid operation code,PI=3: Page fault

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

(3) *TI* = 2: Timer

**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 cord>* contains four entries:

(1) $*AMJ* 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) *<line estimate>* cc. 13—16, 4-digit maximum output estimate.

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

l0(i - 1), 10(I - 1) + 1,. .., 10(I - 1) + 9, i = 1, 2…. *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 X *n* words, *n* 10.

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

**PROJECT REQUIREMENTS**

Three sets of program modules must be designed and implemented:

1. Major simulators for hardware, including the interrupt system, timer, reader, printer and the slave mode paging 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 10 jobs (a “run”) should be prepared for testing purposes.

Phase: 1

Title: Implementation of a multiprogramming operating system

Problem Statement: Stage I:

i. CPU/ Machine Simulation

ii. Supervisor Call through interrupt

Description:

Assumption:

* 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 interrupt 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)

: Loaded/stored/placed into

MOS (MASTER MODE)

SI = 3 (Initialization)

Case SI of

1: Read

2: Write

3: Terminate

Endcase

READ:

IR [4] ← 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)

EXECUTEUSERPROGRAM

WRITE:

IR [4] ← 0

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

EXECUTEUSERPROGRAM

TERMINATE:

Write 2 blank lines in output file

MOS/LOAD

LOAD:

m ← 0

While not e-o-f

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

Control card: $AMJ, end-while

$DTA, MOS/STARTEXECUTION

$END, end-while

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

Store buffer in memory locations m through m + 9

m ← m + 10

End-While

STOP

MOS/STARTEXECUTION

IC ← 00

EXECUTEUSERPROGRAM

EXECUTEUSERPROGRAM (SLAVE MODE)

Loop

IR ← M [IC]

IC ← IC+1

Examine IR[1,2]

LR: R ← M [IR[3,4]]

SR: R → M [IR[3,4]]

CR: Compare R and M [IR[3,4]]

If equal C ← T else C ← F

BT: If C = T then IC ← IR [3,4]

GD: SI = 1

PD: SI = 2

H: SI = 3

End-Examine

End-Loop