CS 5223: Operating Systems Project

PHASE I

Deadline: Monday, April 15, 2019

Note: changes from previous version are highlighted in yellow.

Introduction

The description of Phase I is given below in three parts:

1. characteristics of the hardware
2. machine language-assembly language
3. characteristics of the software.

1. Characteristics of the Hardware

The computer to be simulated has a simple load-store architecture.

MEMORY: The main memory consists of 1024 bytes (locations 0 to 1023). A byte is the basic addressing unit. Each byte is 8 bits long.

CPU: The CPU contains 8 16-bit general purpose registers (GPRs), named r0, ..., r7. The program counter (PC) is 16-bit. All arithmetic is done in 2's complement. A system-wide CLOCK (maintained in the CPU) will be used to time the execution of user programs. The CLOCK is to be reset before each user job is loaded and executed. The CLOCK will be incremented by one virtual time unit each time an instruction is executed. If the instruction is an 1/0 instruction, the CLOCK will be incremented an additional 10 virtual time units

INPUT/OUTPUT DEVICES: In the first phase of the project, all I/O for user jobs will be done at the terminal. In other words, the terminal simulates or virtualizes both the input and the output devices. For outputting the information collected and/or generated by your operating system, files will serve as the virtualized I/O devices.

2. Machine Language -Assembly Language

ADDRESSSING MODES

- indirect: the contents of register ri point to memory location.
  Value = Mem[Reg[ri]]
- absolute: only used for branch instruction - branch to address specified.
- displacement : the contents of register ri added to a displacement point to the memory location.
  Value = Mem[Reg[ri] + displacement]

MACHINE LANGUAGE

Each instruction is encoded in exactly one half-word (16 bits). 4-bits are used to specify an opcode and 3-bits to specify a register. There are three encoding formats.

(1) Register Encoding

These instructions receive all their operands in registers.

```
ppppdddssttt000
```

`pppp` is the opcode, which determines which operation is performed. `ddd` encodes the destination register number, between 0 and 7. `sss` encodes the first source register number, between 0 and 7. `ttt` encodes the second source register number, between 0 and 7. A table of instructions and their opcodes is below.

(2) Immediate Encoding

This encoding is used for instructions which require a 6-bit immediate operand. These instructions typically receive one operand in a register, another as an immediate value coded into the instruction itself, and place their results in a register. This encoding is also used for load, store, branch, and other instructions that require an immediate operand.

```
ppppdddsssiiiiii
```

pppp is the opcode, which determines which operation is performed. A table of instructions and their opcodes is below. iiiiii encodes the 6-bit immediate operand. 2's-complement encoding is used to represent a number.

(3) Trap Encoding

This encoding is used for the trap instruction.

```
pppppiiiiiiiiiii
```

pppp is the opcode, which determines which operation is performed. iiiiiiiiiiii encodes the 12-bit 2's-complement encoding to represent an exception or trap number.

Note: Since memory address is specified in terms of bytes, the Program Counter is incremented by 2 for each instruction executed (unless it is a branch instruction)

Instruction Syntax

This is a table of all the different types of instruction as they appear in the assembly listing. Note that each syntax is associated with exactly one encoding which is used to encode all instructions which use that syntax.

Encoding	Syntax	Template	Comments
Register	Operation	opcode rd, rs, rt	add, sub, seq, sgt, sne
	Move	opcode rd, rs	ttt = 0 (only one source register)
Immediate	Load	opcode rd, immed6(rs)	if immed6 = 0, it can be left out (indirect addressing). immed6 – refers to 6 bits. load
	Store	opcode immed6(rd), rs	if immed6 = 0, it can be left out (indirect addressing) store
	Movei	opcode rd, immed6	sss = 0. movei
	Immed-Op	opcode rd, rs, immed6	addi, subi,
	Branch	opcode rs, label	immed6 is calculated as label. ddd = 000. beqz, bnez
Trap	Trap	opcode immed12	immed12 – refers to 12 bits. trap, lock, unlock

Opcode Table

These tables list all of the available operations. For each instruction, the 4-bit opcode is shown. The syntax column indicates which syntax is used to write the instruction in assembly text files. Note that which syntax is used for an instruction also determines which encoding is to be used. Finally the operation column describes what the operation does plus some special notation as follows:

"MEM [a]:n" means the n bytes of memory starting with address a. n will be in decimal.

The address must always be aligned; that is, a must be divisible by n, which must be a power of 2.

Data Manipulation: Arithmetic

Instruction	Opcode	Syntax	Operation
add	0000	Reg	rd ← rs + rt Example: add r3, r4, r5 r3 ← r4 + r5
addi	0001	Immed	rd ← rs + immed6 Example: addi r3, r3. 45 r3 ←r3 + 45
sub	0010	Reg	rd ← rs - rt Example: sub r3, r4, r5 r3 ← r4 - r5
subi	0011	Immed	rd ← rs - immed6 Example: subi r3, r2, 45 r3 ← r2 - 45

Data Movement

Instruction	Opcode	Syntax	Operation
load	1000	Immed	rd ← MEM [Reg[rs] + immed6]:2. Memory_address = contents of rs + immed6 Load into rd contents of Memory_address Example: load r3, 28(r4) r3 ← MEM[Reg[r4] + 28] if contents of r4 = 32, load into r3 contents of memory location 60
store	1001	Immed	rs → MEM [Reg[rd] + immed6]:2 Memory_address = contents of rd + immed6 Store into Memory_address contents of rs Example: store 28(r3), r4 r4 → MEM[Reg[r3] + 28] if contents of r3 = 32, store into memory location 60 contents of r4. If r4 stores 451, then memory location 60 will store 451
move	1010	Reg	rd ← rs, ttt = 0; move contents of rs to rd Example: move r3, r4 r3 ← r4
movei	1011	Immed	rd ← immed6, rs = 000; stores a value in rd Example: movei r3, 45 r3 ← 45

Data Manipulation: Conditional

Instruction	Opcode	Syntax	Operation
seq	1101	Reg	rd ← (rs == rt); If (rs == rt) is true, then rd ← 1, else rd ← 0 Example: seq r3, r4, r5 If (r4 == r5) is true, then r3 ← 1, else rd3 ← 0
sgt	1110	Reg	rd ← (rs > rt); If (rs > rt) is true, then rd ← 1, else rd ← 0 Example: sgt r3, r4, r5 If (r4 > r5) is true, then r3 ← 1, else rd3 ← 0
sne	1111	Reg	rd ← (rs != rt); If (rs != rt) is true, then rd ← 1, else rd ← 0 Example: sne r3, r4, r5 If (r4 != r5) is true, then r3 ← 1, else rd3 ← 0

Flow of Control

Instruction	Opcode	Syntax	Operation

beqz	0111	Immed	if (rs == 0) then pc += immed6, dest 000
			Example: beqz r3, 32
			If (r3== 0) then increment pc by 32

bnez	1100	Immed	if (rs != 0) then pc += immed6, dest 000
			Example: bnez r3, 32
			If (r3!= 0) then increment pc by 32

trap	0100	Trap	- if immed12=0, halt Example: trap 0; program halts - if immed12=1, then print the value in r1 in decimal format to stdout Example: trap 1; prints contents of r1 to stdout - if immed12=2, then read the integer value from stdin into r1

Example: trap 2; read integer value from stdin into r1

Instruction	*Resource Lock* Opcode	Syntax	Operation
lock	0101	Trap	lock immed12 Example: trap32; locks a word of memory starting at decimal location 32
unlock	0110	Trap	unlock immed12 Example: trap32; unlocks a word of memory starting at decimal location 32

Note: For this project we will only be considering memory resources (that is, memory addresses will be locked/unlocked)

An Example of encoding

Give an instruction:

```
addi  r2, r0, 5
```

From Opcode Table - opcode is 0001 and its syntax is Immed. From Instruction Syntax Table - template for Immed is opcode rd, rs, immed6. So d=2, s=0, immed6=5. Immed belongs to Immediate Encoding format `ppppddddsssiiiiii`. So its encoding is 0001001000000101, or 1205.

Allocating memory space

Allocating memory space is given the encoding `0000000vvvvvvvvv`. `vvvvvvvvv` is the value to be stored in memory. The instruction is `alloc`.

For example:

```
alloc 5;    // value of 5 is stored in memory in first two bytes
alloc 7;    // value of 7 is stored in memory in next two bytes
```

The program starts with memory allocation. After memory has been allocated, instructions are specified.

The encoding for the above two allocations is:

0000000000000101
0000000000000111

3. Characteristics of the software

This routine handles errors and special conditions. In case of any kind of error The SYSTEM (Phase I) will be the driver for this simulation. The SYSTEM is to contain four subroutines: LOADER, CPU, MEMORY, and ERROR-HANDLER.

LOADER

The LOADER will be responsible for loading each user's program into main memory. In Phase I, each user program is loaded into the main memory from location 0.

After a user program has been loaded, it has to be executed. Subroutine CPU will be responsible for the execution. Whenever a user program terminates, control is transferred back to the SYSTEM.

ERROR-HANDLER PROCEDURE

ERROR-HANDLER (N) with N -> error number

This routine handles errors and special conditions. In case of any kind of error (illegal op code, memory range fault, illegal input, program size too large, etc.), the CPU or LOADER will trap to the ERROR-HANDLER by a number (error number). This routine will print out the appropriate error message (using the error number and based on the nature of the error) and dump the memory (only the first 64 bytes) by calling the MEMORY procedure with the DUMP option.

Error–handler must be able to detect:

- Illegal instruction
- Memory range fault
- Program size too large
- Alloc large data
- Infinite loop in the program

MEMORY PROCEDURE

MEMORY (X, Y, Z) with

X -> READ, WRIT, or DUMP	(the control signal)
Y -> memory address (EA)	(address of current instruction or the memory address register)
Z -> variable	(current instruction or the memory buffer register)

READ operation: The contents of memory location EA will be read into variable z.
(EA) -> Z

WRIT operation: The contents of variable Z will be written into memory location Y.
(Z) -> EA

Variable Z may be a register used by the CPU, or it may be a buffer used by the LOADER to help load each user program into main memory.

DUMP operation: The contents of physical memory will be printed as follows

```
0000 byte0   byte1     byte2   byte3    byte4    byte5   byte6   byte7
0008 byte8   byte9     ...       ...    ...
  :    :       :         :       :        :       :       :       :
  :    :       :         :       :        :       :       :       :
```

For Phase I of the project, only the first 64 bytes of memory will have to be printed out with the above format.

LOADER PROCEDURE

LOADER (X) with X showing the starting address

The LOADER loads the information, i.e., each user job, from the input device (i.e., a file).
All the information on the input records is in HEX.

Format of a set of records:

- length (three-digit HEX number) starting address of program (three-digit HEX number)
- user program records (each record is sixteen HEX digits specifying four half-words (16 bits) of information).
- trace switch (1 for on and 0 for off)
  where:
  - "trace switch" consists of one bit in three-digit HEX number; if the value of the "trace switch" is 1, a trace has been requested and it must be generated

If the trace switch parameter of the procedure LOADER is set, the CPU will provide the following information for each instruction.

- PC
- Instruction
- Register and effective address EA (if applicable)
- contents of Register and EA (if applicable) before execution
- contents of Register and EA (if applicable) after execution

CPU PROCEDURE

call: CPU (X, Y) with X showing the PC value and Y showing if the trace switch is on or off

The CPU will loop indefinitely executing instructions until a TRAP instruction or an 1/0 instruction is interpreted

At the time when a TRAP instruction is encountered, control is returned to the SYSTEM and the next job in the batch (if there is one) will be run. The program terminates when a TRAP 0 is encountered. When there are no more jobs, the simulation terminates

When an 1/0 instruction is encountered, control is returned to the SYSTEM and the CLOCK is incremented by 10. The SYSTEM routine reads data from the input device (i.e., the terminal in Phase I) and stores it in a register. After this is done, the SYSTEM recalls the CPU. Notice that when an 1/0 operation is executed, the CLOCK will first be incremented by one virtual time unit in the CPU (execution time of one instruction) and then by another ten virtual time units in the SYSTEM (wait time for 1/0). The value of the CLOCK in HEX is printed out at the termination of each user program.

User Test Job

This program stores 5 and 7 into memory. It reserves space for the result. It fetches 5 and 7 from memory, adds them stores them in the space reserved and also outputs the result to standard output. Labels `start` and `main` are to make the program more readable.

```
start:      alloc 5;    //store 5 at location 0 in memory
            alloc 0;    //store 0 at location 2 in memory

main:
            movei r3,0;         //put 0 in r3
            load r2,(r3);       //store 5 in r2
            trap 2;             //get input from stdin. Value stored in r1
            add r4,r2,r1;       //r4=r1+r2
            store 2(r3),r4;     //store contents of r4 in memory location 2
            move r1,r4;         //move contents of r4 to r1
            trap 1;             //output contents of r4 to stdout
            trap 0;             //program halts
```

Loader Format:

```
00A 004
00050000B60084C0
400208889702A300
40014000
000
```

NOTES:

DUMP output note: The memory dump, which is output as a result of the DUMP operation, should be all in HEX and should only contain the relevant part of memory (i.e., the part actually occupied and used by the user program)

TRACE information output note: When the trace bit is on, the trace information generated should be put in a separate file (the trace-file) in columns with these column headers:

PC	instruction	R	EA	before execution (R)	before execution (EA)	after execution (R)	after execution (EA)

Example

PC	instruction	R	EA	(R)	(EA)	(R)	(EA)
04	add r4, r3, r2	r4		2		15	
		r3		7		7	
		r2		8		8	
06	store 4(r0), r4	r0		0		0	
		r4		15		15	
			4		0		15

To keep the length of the trace-file manageable, you are to append the trace information to the trace-file according to the following virtual time line: 0 to 10 <gap> 20 to 30 <gap> 50 to 60 <gap> 100 to 110 <gap> 190 to 200, etc.

NAMING STANDARDS: The driver of the first step of the project is to be named SYSTEM. Other descriptive names such as MEMORY, CPU, LOADER, ERROR-HANDLER, and trace-file must also be used as specified in this handout.

IMPORTANT Design Note: Conventional architectural and operating system component functionalities and restrictions should be adhered to. There is ample opportunity for tailoring/customizing or personalizing/individualizing during the implementation. Nonetheless, any significant deviation from the specification must be approved by the instructor.

DOCUMENTATION GUIDELINES: Your simulation program (i.e., the instruction set simulation of the underlying architecture and the batch operating system) must include external and internal documentation. External documentation appears in the form of header blocks and is an explanation of the functionality of the program/subprogram/function/module, a description of the global variables, and a discussion of the implementation approach. Internal documentation is the documentation that is mixed with the program code and is used to clarify potentially obscure segments of program code, e.g., case statements, loops, conditions, etc. Use meaningful names and blank lines to enhance the understandability of your code. DO NOT pollute the user's environment with unnecessary echoes and prompts. Note that the assembly version of the test jobs are programs too, and thus must be documented and commented.

DUE DATE note: You are to sign up for a demonstration of your project before proceeding top phase II. The demonstrations will be on **CSX**

INPUT: The inputs to your SYSTEM are files of Assembler code, each of which is a batch consisting of a single user job in Loader Format.

OUTPUT: The following information is to be output to a file by your operating system for each job. Note that this is the "output" of your operating system as distinct from the output " of each user job that will appear on the screen.

1. Cumulative job identification number. This number is reset each time that you start your operating system, thus it should be one for each job in this phase (since each job in this phase is a batch of size one)
2. A message indicating the nature of termination (normal or abnormal, plus a descriptive error message if abnormal).
3. CLOCK value in Hex at termination

4. Run time for the job (in DECIMAL) probably subdivided into execution time, I/O time, etc. Note that this is in the Decimal form. For clarity, append the word Decimal/Hex, as the case may be, after each number that is output.

Note that the output of the current user job as well as its input will not appear in the above-mentioned file, instead they will appear on the screen and will be included in the typescript of your interactive session.

YOU ARE TO TURN IN THE FOLLOWING:

1. SYSTEM simulation program listing (modular, readable, and well-documented).
2. The image of your interactive session containing the I/O of one test job plus a record of the appropriate commands issued.
3. Your own non-trivial test job in the given assembly language with explanations, i.e., instructions with comments, followed by its Loader Format.
4. Sample compilation and executions of a test job to be given later with the trace switch on and with the trace switch off, along with the display of the trace-file generated.
5. Sample executions of several small test jobs containing injected errors as listed above in order to exercise your ERROR-HANDLER routine (the trace switch should be off for these sample executions)
6. A two to three page typed write-up of the software engineering issues involved in the design and development of STEP 1 including the following:
    a. Your general approach to the problem (i.e., did you use a design language, pseudo code, a flow chart, etc.?)
    b. A list of the utilities used (makefile, various debuggers, SCCS, RCS, etc.)
    c. Bulk complexities of your simulation including number of lines of code (divided into declarations, comment lines, executable statements, etc.), number of procedures, number of decisions, number of blank lines, etc.
    d. An approximate break-down of the time spent in the design, coding, and testing of your program.
    e. A brief justification of your choice for the implementation language.

4/9/2019

Loader Samples

```
//2 + 2 +  + ... + 2  (10 times)


movei r2, 1             //move 1 to r2 - counter
movei r3, 2             //move 2 to r3 - result
movei r4, 10            //move 10 to r4 - loop counter
addi r3, 2        //add 2 to result
addi r2, 1        //add 1 to counter
seq r5, r4, r2          //check if counter has reached 10
beqz r5, -8             //if not go back up
move r1, r3             //move result to r1
trap 1                  //output result
trap 0                  //program halts

Result
Decimal: 20
Hex: 14



Loader format:

00A 000
b401b602b80a16c2
1481db107178a2c0
40014000
000


00A
b401b602b80a16c2
1481db107178a2c0
40014000
001
```

```
//1 + 2 + 3 +  + ... + 10


movei r2, 1             //move 1 to r2 - counter
movei r3, 10            //move 10 to r3 - loop counter
movei r4, 1             //move 1 to r4 - result
addi r2, r2, 1          //add 1 to counter
add r4, r4, r2          //add
seq r5, r3, r2          //check if counter has reached 10
bnez r5, -8             //if not go back up
move r1, r4             //move result to r1
trap 1                  //output result
trap 0                  //program halts

Result
Decimal: 55
Hex: 37



Loader format:

00A 000
b401b60ab8011481
0910dad07178a300
40014000
001
```

```
alloc 5           //store 5 at location 0 in memory
alloc 0            //store 0 at location 2 in memory

movei r3,0        //put 0 in r3
load r2,(r3)      //store 5 in r2
trap 2            //get input from stdin. Value stored in r1
add r4,r2,r1      //r4=r1+r2
move r1,r4        //move contents of r4 to r1
trap 1            //output contents of r4 to stdout
trap 0            //halt
```

RESULT
5 + value entered at keyboard

Loader Format:

```
009 004
00050000b60084c0
40020888a3004001
4000
000



009 004
00050000b60084c0
40020888a3004001
4000
001
```

```
alloc 0

movei r7,0        //r7=0
trap 2            //input
move r6,r1        //r1 -> r6
store (r7),r6     //store contents of r6 in memory location 0
trap 2            //input
move r6,r1        //r1 -> r6
load r6,(r7)      //load contents of memory location 0 into r6
move r1,r6        //r6 -> r1
trap 1            //output
trap 0            //halt


INPUT: 2 inputs
OUTPUT: First input

00B 002
0000be004002ac40
9f804002ac408dc0
a38040014000
001
```

CORRECTED

```
//memory gets locked


alloc 0                 //store 0 at location 0 in memory

movei r2, 0             //store 0 in r2
movei r3, 17            //store 17 in r3
store 0(r2), r3         //store contents of r3 in memory location 0
movei r3, 25            //store 25 in r3
lock 0                  //lock memory location 0
load r3, (r2)           //load what is in memory location 0 to r3
unlock 0                //unlock memory location 0
move r1 r3              //move contents of r3 to r1
trap 1                  //output contents of r1
load r3, (r2)           //load what is in memory location 0 to r3
move r1 r3              //move contents of r3 to r1
trap 1                  //move contents of r3 to r1
trap 0                  //halt


RESULT
first 25 is output
next 17 is output


Loader format:

00E 002
0000b400b61194c0
b619500086806000
a2c040018680a2c0
40014000
001
```

//keeps adding 1 - PROGRAM TOO LARGE

```
movei r2, 0             //move 0 to r2
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
```

```
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
```

```
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
```

```
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
```

```
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
```

```
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
```

```
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
```

```
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
```

```
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
addi r2, r2, 1          //add 1
move r1, r2             //move counter result to r1
trap 1                  //output result
trap 0                  //halt
```

ERROR - Program size too large

Loader format:

```
209 000
b400148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
```

1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481

1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
1481148114811481
14811481a2904001
4000
001

```
//reduce from upper value 10 to lower value -83.
//Error - Immed value too large


movei r2, 0             //move 0 to r2 - counter
movei r3, 10            //move 10 to r3 - upper value
movei r4, -83           //move -83 to r4 - lower value - data allocated is too
                        //large
subi r3, r3, 1          //subtract 1 from upper value 10
addi r2, r2, 1          //add 1 to counter
sne r5, r4, r3          //is upper value == lower value
beqz r5, -8             //if not go back up
move r1, r2             //move counter result to r1
trap 1                  //output result
trap 0



Loader format:

00A 000
b400b60ab9ad36c1
1481fb187178a280
40014000
001
```

```
alloc 5                 //store 5 at location 0 in memory
alloc 0                 //store 0 at location 2 in memory

movei r3,0              //put 0 in r3
load r2,(r3)            //store 5 in r2
trap 2                  //get input from stdin. Value stored in r1
add r4,r2,r1            //r4=r1+r2
move r1,r4,r5           //move contents of r4 to r1 - ILLEGAL INSTRUCTION
trap 1                  //output contents of r4 to stdout
trap 0
```

RESULT
Illegal instruction

Loader Format:

```
009 004
00050000b60084c0
40020888a3284001
4000
001
```

```
//infinite loop

movei r1,0        //put 0 in r2
addi r1,r1,0      //r1 =0
beqz (r1),-4      //points back to main
trap 1
trap 0
```

Loader Format:

```
005 000
b2001240707c4001
4000
001
```

```
alloc 0

movei r7,0        //r7=0
trap 2            //input
move r6,r1        //r1 -> r6
store 4(r7),r6    //store contents of r6 in memory location 4 - MEMORY RANGE
                  //FAULT
trap 2            //input
move r6,r1        //r1 -> r6
load r6,(r7)      //load contents of memory location 0 into r6
move r1,r6        //r6 -> r1
trap 1            //output
trap 0            //halt


INPUT: 1 input
OUTPUT: error

00B 002
0000be004002ac40
9f844002ac408dc0
a38040014000
001
```

```
alloc 5           //store 5 at location 0 in memory
alloc 0                 //store 0 at location 2 in memory
movei r3,0              //put 0 in r3
load r2,(r3)            //store 5 in r2
trap 2                  //get input from stdin. Value stored in r1
add r4,r2,r1            //r4=r1+r2
store 2(r3),r4          //store contents of r4 in memory location 2
move r1,r4        //move contents of r4 to r1
trap 1                  //output contents of r4 to stdout
trap 0                  //program halts
```

Loader Format:

```
00A 004
00050000B60084C0
400208889702A300
40014000
000
```

Operating Systems Project

Phase II

Deadline: Demonstration and submission of code/report
April 18th – April 19th

You are to design and implement a multiprogramming batch operating system with the following characteristics:

Memory Manager: Dynamic Partitioning scheme
Process Manager: Round Robin with inverse of remainder of quantum

This simulation is an extension of your simulation developed in Phase I. The purpose is to overlap the I/O and computation of user jobs. The new features are introduced in the subsequent sections. A batch file of user jobs will be input to your system.

LOADER EXTENSION

There are three extensions to the loader format.

- Job ID or JOB # in HEX
- There will be a three-digit priority associated with a program. 000 is low priority whereas 001 is high priority.
- A arrival time is associated with each program indicating the time it is available. A program can enter the system (that is, be allocated memory space) anytime after the arrival time.

The extended loader format is:

- *Job ID* or *JOB #* <three-digit HEX number> *arrival time* <four-digit HEX number> *length* <three-digit HEX> *start address*<three-digit HEX> – (length and start address as in phase I)
- user program records – as in phase I
- last record: *trace switch* *priority*

For example

```
015 0100 00C 006
000500070000CF00
C400D502A200AF04
E400C40003010300
000 001
```

Program Job ID - 015 hex, arrival time - 0100 hex, length of job - 00C hex, start of program - 006 hex. High priority job. This job can be executed at time unit 0100 (hex) at the earliest.

`trap 1` and `trap 2` I/O instructions. Everytime an I/O instruction is executed, the SYSTEM displays the Job ID or JOB #.

MEMORY MANAGER

Changes are to be made to the various modules, such as the MEMORY and the LOADER modules, in order to add a dynamic partition memory allocation scheme to your operating system (i.e., to your Phase I).

The MEMORY-MANAGER will be responsible for maintenance of the data structures used to implement this memory management scheme; that is, it will be responsible for their initialization and updating.

In Phase II, main memory will consist of 256 bytes.

Allocation Strategy: You are to implement a MAXIMUM-USAGE (or BEST-FIT) allocation strategy when allocating memory to a user-job. In other words, the minimum amount of memory will be left unused.

Deallocation Strategy: Whenever a user job leaves the SYSTEM, its corresponding memory block is deallocated.

Compaction: Fragmented bits of memory into which no user job will fit, must be compacted to provide a larger memory space into which one or more user jobs may fit.

Memory Management: All the jobs in the job list will start at logical memory address 0. The memory management unit will be responsible for keeping track of the logical to physical memory mapping.

The degree of multiprogramming will vary with time.

PROCESS MANAGER

Every user job in the SYSTEM will now be represented by a Process Control Block or PCB. The PCB corresponding to a job will be created whenever a job enters the SYSTEM, and it will be destroyed when that user job terminates, either normally or abnormally.

A PCB will include at least the following information:

a. job ID or JOB # (starting from one and counting up in HEX)
b. a copy of Registers (PC, etc.)
c. arrival time
d. time the job entered the SYSTEM
e. CPU time used by the job
f. time of completion of the current I/O operation
g. remainder of the last quantum
h. priority

Once in the SYSTEM, each user job may be in one of the following states:

running: A job is running when it is in the CPU. Each process will be given a time quantum of 30 virtual time units to run.

ready: A job is ready when it is waiting to be dispatched. Ready processes are kept in the ready queue.

blocked: A process is blocked when it is waiting for I/O completion. Blocked processes are kept on the blocked list, which is organized in ascending I/O completion time order. Obviously, when a job terminates either normally or abnormally, checking the blocked queue takes precedence over loading new user jobs.

SCHEDULER

To introduce multiprogramming to your operating system, you will add a new module: the SCHEDULER. The SCHEDULER will be responsible for dispatching jobs (the dispatcher) and maintaining the ready queue and the blocked list. The ready list is organized as a Round Robin fashion. When jobs finish their I/O and return to the ready queue, they will be inserted back in the ready queue based on a priority scheme that favors I/O-bound jobs and uses the remainder of their last quantum.

There will be two ready queues. The jobs in the high priority queue will execute for 300 time units, after which jobs in the low priority queue will execute for 100 time units. At the end of the execution period of a queue, jobs in the other queue will start executing.

For both queues, returning jobs will join back to the ready queue based on the following simple rule: The position in the queue is determined by the remaining quantum time. The higher the quantum time remaining, the higher the position in the queue. A time quantum or time slice is for 30 secs. After 30 secs, the job goes to the bottom of queue. For example, assume the ready queue contains the following

JOB	REMAINDER OF QUANTUM	POSITION IN READY QUEUE
P_{10}	28	as the first PCB
P_5	24	as the second PCB
P_2	21	as the third PCB
P_4	12	as the fourth PCB
P_8	06	as the fifth PCB

A job P_1 with a quantum time of 22 remaining, will re-join the ready queue at the 3rd position. Jobs P_2, P_4 and P_8will move down to positions 4, 5 and 6 respectively. If a number of jobs are blocked due to 1/0, the returning jobs are simply appended to the appropriate slot in the ready list. A job that has used all its quantum time will rejoin the list at the bottom of the queue.

An executing job stops executing before the end of its time slice only if one of the following occur

- it completes execution (trap 0)
- an execution error occurs
- an I/O instruction needs to be executed.

Whenever the SCHEDULER tries to find a job on the ready queue unsuccessfully (i.e., there are no ready processes), and the block list is empty or the waiting time for the first job on the blocked list is

not over, the CPU will have to idle.

Job initiation is done after one of the memory resident jobs is completed or terminated. To initiate a new job, the SCHEDULER will transfer control to the SYSTEM, which will call the LOADER, and then the MEMORY-MANAGER to do the appropriate memory allocation for the next job in the batch (if possible). A job can only be loaded if its arrival time is less than or equal to the current value of the system clock.

Once a job is residing in memory, the SCHEDULER creates a PCB for it.

Each job (high and low priority) gets a time quantum of 30 virtual time units (execution of at most 30 instructions) each time the CPU is allocated to it. Once a job is running, one of three things may happen:

1. If the job uses all 30 virtual time units and still needs more CPU time, the job loses the CPU and control is transferred to the SCHEDULER. The SCHEDULER will append the job (actually its PCB) to the bottom of the ready queue.

2. If the job terminates before its time quantum expires, the CPU is released and control is transferred to the SCHEDULER, which has to destroy the process (i.e., its PCB) and deallocate memory. Also, the job must be spooled out from the memory. As a simplifying assumption, the output spooling takes zero time; so the CLOCK is not incremented for spooling. Upon termination of each job, the following information will be output (time instances in HEX and time intervals in DECIMAL):

    a. job ID or JOB #
    b. all the start and end addresses occupied by this job
    c. arrival time
    d. time the job entered the SYSTEM
    e. time the job is leaving the SYSTEM
    f. execution time
    g. time spent doing I/O
    h. priority

    For abnormal terminations, output the above information partially (as much as possible), a message as to the nature of the problem to the extent known to the SYSTEM, and a dump of all the relevant partition (even if only partially filled).

3. If the job requests I/O (trap instruction) in the middle of a time quantum, it releases the CPU and control is transferred to the SCHEDULER. The SCHEDULER inserts the given PCB in the appropriate place on the blocked list. Each job has to remain on the blocked list for 10 time units (time for I/O operation completion). Note: Time spent for I/O is not part of the quantum time, that is, quantum time does not get decremented.

When a job relinquishes the CPU, and the appropriate action has been taken (as described above), the SCHEDULER will check the blocked list to see if a job has completed its I/O and is ready to run again. If such a job is found, it is taken from the blocked list and placed on the ready queue

based on the remainder of its last time slice (see above). Subsequently, the SCHEDULER will look for the next job to run at the top of the ready queue.

Only if no other job can be found to run will the CPU have to sit idle while waiting for the completion of the I/O of a job (for our simulation, idling means incrementing the CLOCK by the amount of time the CPU has to idle). This is the only occasion the CLOCK is incremented other than during program execution in the CPU.

LOAD MODULE: JOB # (number)
<Loader format>
END

Your control record interpreter (part of the LOADER?) should detect and handler errors by trapping to the ERROR-HANDLER. Any fatal error results in the termination and output spooling of that job from memory (the part that has been loaded, if any). Error messages that will be used, in addition to the ones already handled by the ERROR-HANDLER in phase I, include:

MISSING arrival time
MISSING job (i.e., missing loader format)
MISSING the END record
MISSING/UNRECOGNIZABLE TRACE BIT
MISSING/UNRECOGNIZABLE PRIORITY
UNRECOGNIZABLE CHARACTER ENCOUNTERED WHILE LOADING
PROGRAM TOO LONG (i.e., > 256)

Note: a job that contains errors will be sandwiched between two jobs that do not have errors.

STATISTICS:

When the processing of all user jobs is completed, output the following information for all the jobs that completed normally in the test batch. Note that the Arrival time is the time when the job becomes available. The Start time or the time a user job enters the system, is the load time which is the time a user job is allocated a memory partition.

- current value of the CLOCK
- mean user job run time
- mean user job I/O time
- mean user job time in the SYSTEM
- total CPU idle time
- total time lost due to abnormally terminated jobs
- number of jobs that terminated normally
- number of jobs that terminated abnormally
- highest degree of multiprogramming
- lowest degree of multiprogramming
- mean degree of multiprogramming
- mean difference between arrival time and start time

(These performance evaluation metrics must be calculated by your operating system code and not after the fact.)

INPUT:
You will be given an input file that contains multiple jobs in the following format:

JOB # (number)
<Loader format>
END

All jobs will be sorted on JOB # and arrival time.
Example:

```
JOB 1
001 0100 00C 006
000500070000CF00
C400D502A200AF04
E400C40003010300
000 001
END
JOB 2
002 0200 00A 002
…
END
JOB 3
003 …
…
END
JOB 4
004 …
…
END
JOB 5
005 …
…
END
```

Note: Job id is the same as JOB # and are used interchangeably. There is no line space between jobs.

OUTPUT:

Your simulation must produce one progress file per batch. This file monitors the dynamic changes of your operating system's execution and can be used to debug your system. This file records the significant events that your operating system undergoes, and also checkpoints your system at those events outputting periodic snapshots of the system including snapshots of various queues.

Events that trigger the information collection and the making of an entry in the file include:

- job initiation – once

- job termination
- I/O request
- context switch

The file must include:

- warnings and error messages indexed by user job ids
- current degree of multiprogramming
- JOB# of user job currently executing
- partition dump for all jobs
- total external fragmentation and memory allocation
- queue list for the various queues
- degree of multiprogramming

For each test batch and while processing each test batch, your system is to generate the following output. Note that there is a certain amount of intentional overlap among the output files.

- A progress file that contains the information mentioned above, plus the final per batch statistics, also as mentioned above.

- One separate trace file per each job whose trace switch is on; these trace files must be created and incrementally updated dynamically.

- An output file that essentially simulates the user screens, in that it contains the individual user output (as specified in Phase I, including warnings and error messages) in the order of user job termination either normally of abnormally.

Other requirements and deliverables, including the write-up on software engineering issues, are as mentioned in Phase I.

One test batch will be made available to you to run on your SYSTEM. In the mean time, you can use the test jobs given in Phase I and your own test job to create your own batch of jobs.

The specifics of various project phases may change slightly as a result of class discussions.

All deliverables are to be file submissions.

The naming standards, documentation guidelines, and the late penalty policy as stated in the specifications for Phase I apply to Phase II as well.

Summary of the deliverables for Phase II:

- progress file
- trace files.
- dump files.
- output file that simulates the user screen

- Statistics file
- Softcopy of code and Software engineering issues report.

Dr. J P Thomas
3/17/2019