Bansilal Ramnath Agarwal Charitable Trust's

Vishwakarma Institute of Technology, Pune-37

(An autonomous Institute of Savitribai Phule Pune University)

Department Of Information Technology

Lab Manual

Course Code	Course Name	Teaching Scheme (Hrs / Week)	Credits
IT3221	Operating Systems	2 Theory, 2 Lab,1 Tut	4

Course Outcomes:

The student will be able to –

1. Examine the functions of a contemporary Operating system with respect to convenience, efficiency and the ability to evolve.
2. Demonstrate knowledge in applying system software and tools available in modern operating system (such as threads, system calls, semaphores, etc.) for software development.
3. Apply various CPU scheduling algorithms to construct solutions to real world problems.
4. Identify the mechanisms to deal with Deadlock.
5. Understand the organization of memory and memory management hardware.
6. Analyze I/O and file management techniques for better utilization of secondary memory.

Class : - TY	Branch	: - Information Technology
Year : - 2024-25	Prepared By	: - Mrs. Aparna R. Sawant
Required H/W and S/W : C, C++, JAVA Linux /Win XP with P-IV 128 MB RAM		

INDEX

Experiment No:1

Title: Basic Linux Command

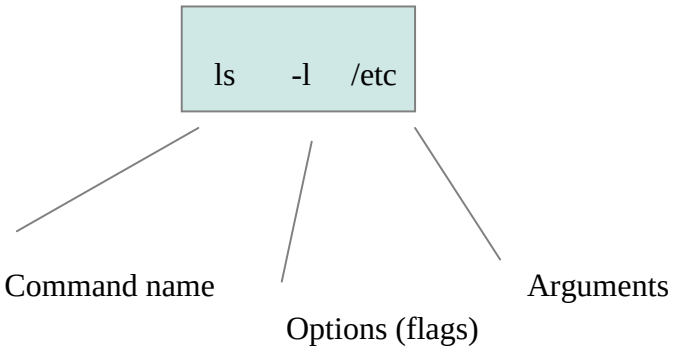
Problem Statement: Execution of basic Linux commands

Objectives:

1. To understand how to use Unix commands.
2. To understand How and Why they are used in Shell Programming

Description:

To execute a command, type its name, options and arguments at the shell prompt.



General Purpose Commands

1. date: To display current date & time of the system.
2. cal :To display calendar of current month.
3. who:List who is currently logged on to the system.
4. Whoami:Report what user you are logged on as.
5. echo :Echo a string (or list of arguments) to the terminal
6. bc:To perform mathematical operations
7. clear:To clear the screen
8. alias : Used to tailor commands

   Ex alias erase=rm Generate an employee report using AWK programming.

   alias grep=”grep -i”

   alias cp=“cp –i”
9. man <cmd name>: To get help for any command

10. passwd: To change the password
11. exit: To logout from the terminal

File & Directory Related Commands

1. cp <fromfile> <tofile>: Copy from the <fromfile> to the <tofile>
2. mv <fromfile> <tofile> : Move/rename the <fromfile> to the <tofile>
3. rm <file>:Remove the file named <file>
4. mkdir <newdir>:Make a new directory called <newdir>
5. rmdir <dir>:Remove an (empty) directory
6. cd <dir> :Change the current working directory to dir
7. pwd : Print (display) the working directory
8. cat > <file> :To create new file n save it by pressing ^d
9. cat >> <file>: To append contents into file
10. cat <file>:To see the contents of existing file
11. more <file>:Paging out the contents of file
12. file <file>:To check the type of file
13. wc <file>:To count lines,words,charaters of file
14. cmp <file1> <file2>:To compare two files
15. comm <file1> <file2>:To display common values between two files
16. diff <file1> <file2>:To convert one file to another
17. gzip <file>:To compress the file
18. gunzip <file>:To unzip the contents of zipped file
19. ls :List the files in the current working directory
20. ls <dir>:List all files & directories in given directory
21. ln <fromfile><tofile>: Creates a symbolic link to a file

Conclusion: we have studied basic Linux command

Experiment No:2

Title: Advance Linux Command

Problem Statement: Execution of advance Linux commands

Objectives:

1. To understand how to use Unix commands.
2. To understand How and Why they are used in Shell Programming

Simple Filters

1. pr <file> :Paginating the file

   Ex pr –h "test" –d –n fname

2. head <file>:Display first 10 lines of file

   Ex head –n -3 fname

3. tail <file> :To display last 10 lines of file

   Ex tail -3 fname ; tail –c 100 fname

4. cut <file> :Splitting file vertically

   Ex cut –c 2-10,12-14 fname

   a. cut –d "|" –f 2,4 fname

5. paste <file1> <file2> :To combine two file vertically rather than horizontally

   Ex paste –d "|" fname1 fname2

6. sort <file>:To sort file in order by field wise

   Ex sort –t"|" –k 2 fname

   sort –r fname

7. uniq <file> :Locate repeated & nonrepeated lines

   Ex uniq fname; uniq –d fname

8. tr ch1 ch2 < <file1>:To translate occurrence of ch1 by ch2

   Ex tr '|' '+' < fname1

9. tee: read from standard input and write to standard output and files

   Ex. ls *.txt | wc -l | tee count.txt

File permission: Use the chmod command to change file permissions

1. **Changing permission relative manne**r

Category	Operation	Perm.
u-user	+ assign	r-read
g-group	- removal	w-write
o-other	= assign abs perm.	x-execute
a-all		

Syntax: chmod category operation perm. <file>

Ex chmod u+x fname

chmod a+x fname

chmod u-x fname

chmod a-x,go+r fname

2. Changing permission absolute manner

Read=4

Write =2

Execute=1 Generate an employee report using AWK programming.

Ex chmod 666 fname

chmod 644 fname

chmod -R 644

Change owner & group

Syntax: chown options owner files

Ex chown “xyz” fname

5

Syntax: chgrp options group files

Ex chgrp “xyz” fname

Redirection: Provide a powerful command line controls

Most Linux commands read input, such as a file or another attribute for the command, and write output. By default, input is being given with the keyboard, and output is displayed on your

screen. Your keyboard is your standard input (stdin) device, and the screen or a particular terminal window is the standard output (stdout) device. There are 3 types of redirection available in linux

1. Standard input redirection: It is used to redirect standard input.

Ex. cat < fname

2. Standard output redirection : It is used to redirect standard output.

Ex cat >fname

3. Standard error redirection: It is used to redirect standard error.

Ex cat fname 2>Errorfile

Pipe : Connects commands so the output of previous command becomes input for the second.

Vertical bar(|) is the pipe operator.

Ex. ls -l | more

cat file1 file2 | sort > file3

Concatenates file1 and file2

Sends the result to the sort command

Store the alphabetized, concatenate result as a new file called file3

Grep: Global Regular Expression Print

Searching and pattern matching tools

Searches files for one or more pattern arguments. It does plain string, basic regular expression, and extended regular expression searching

Following are some of the options for grep

-i ignore case for matching

-v doesn't display lines matching expression

-n display line numbers along of occurrences

-c counting number of occurrences

-l display list of file names

-e exp for matching

-f file take patterns from file

-E treat pattern as an extended reg. exp

-F matches multiple fixed strings (fgrep)

Problems to be solved in the lab:

1. Change your password to a password you would like to use for the remainder of the semester.
2. Display the system's date.
3. Count the number of lines in the /etc/passwd file.
4. Find out who else is on the system.
5. Direct the output of the man pages for the date command to a file named *mydate*.
6. Create a subdirectory called *mydir*.
7. Move the file *mydate* into the new subdirectory.
8. Go to the subdirectory *mydir* and copy the file *mydate* to a new file called *ourdate*
9. List the contents of *mydir*.
10. Do a long listing on the file *ourdate* and note the permissions.
11. Display the name of the current directory starting from the root.
12. Move the files in the directory *mydir* back to the HOME directory.
13. List all the files in your HOME directory.
14. Display the first 5 lines of *mydate*.
15. Display the last 8 lines of *mydate*.
16. Remove the directory *mydir*.
17. Redirect the output of the long listing of files to a file named *list*.
18. Select any 5 capitals of states in India and enter them in a file named *capitals1*. Choose 5 more capitals and enter them in a file named *capitals2*. Choose 5 more capitals and enter them in a file named *capitals3*. Concatenate all 3 files and redirect the output to a file named *capitals*.
19. Concatenate the file *capitals2* at the end of file *capitals*.
20. Redirect the file *capitals* as an input to the command "wc –l".
21. Give read and write permissions to all users for the file *capitals*.
22. Give read permissions only to the owner of the file *capitals*. Open the file, make some changes and try to save it. What happens ?

23. Create an alias to concatenate the 3 files *capitals1*, *capitals2*, *capitals3* and redirect the output to a file named *capitals*. Activate the alias and make it run.
24. What are the environment variables PATH, HOME and TERM set to on your terminal ?
25. Find out the number of times the string "the" appears in the file *mydate*.
26. Find out the line numbers on which the string "date" exists in *mydate*.
27. Print all lines of *mydate* except those that have the letter "i" in them.
28. Create the file *monotonic* as follows:

    ^a?b?b?c?……..x?y?z$

    Run the egrep command for *monotonic* against /usr/dict/words and search for all 4 letter words.
29. List 5 states in north east India in a file *mystates*. List their corresponding capitals in a file *mycapitals*. Use the *paste* command to join the 2 files.
30. Use the *cut* command to print the 1st and 3rd columns of the /etc/passwd file for all students in this class.
31. Count the number of people logged in and also trap the users in a file using the *tee* command.

Conclusion: Thus we have studied Advance Linux command

FAQS:

1. What is a pipe?
2. What is a filter?
3. What is the purpose of the grep command?
4. How does input output redirection take place?
5. What is an alias?

Experiment Number: 02

TITLE: Shell Programming

OBJECTIVES:

1. To understand how to perform Shell programming in Unix/Linux.
2. To explain the purpose of shell programs
3. Design and write shell programs of moderate complexity using variables, special variables, flow control mechanisms, operators, arithmetic and functions.

Description:

Shell

- Interface to the user
- Command interpreter
- Programming features

Shell Scripting

- Automating command execution
- Batch processing of commands
- Repetitive tasks

Shells in Linux

- Many shells available
- Examples -sh, ksh, csh, bash
- Bash is most popular

The Bourne Again Shell

- Abbreviated bash
- Default in most Linux distributions
- Most widely used on all UNIX platforms
- Derives features from ksh, csh, sh, etc.
- Support for many programming features variables, arrays, loops, decision operators, functions, positional parameters
- Pipes, I/O re-direction

- Misc. features - expansions, job control
- Built in Commands - read, echo, alias

Shell variables:

1) **System variables** - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS

   BASH=/bin/bash Our shell name

   HOME=/home/vivek Our home directory

   LOGNAME=studentsstudents Our logging name

   PATH=/usr/bin:/sbin:/bin:/usr/sbin Our path settings

   PS1=[\u@\h \W]\$ Our prompt settings

   PWD=/home/students/Common Our current working directory

   SHELL=/bin/bash Our shell name

   USERNAME=vivek User name who is currently login to this PC

2) **User defined variables (UDV**) - Created and maintained by user. This type of variable defined in lower letters.

   To define UDV use following syntax

   Syntax: variable name=value

   'value' is assigned to given 'variable name' and Value must be on right side = sign.

   Example:

   $ no=10# this is ok

   $ 10=no# Error, NOT Ok, Value must be on right side of = sign.

   To define variable called 'vech' having value Bus

   $ vech=Bus

   To define variable called n having value 10

   $ n=10

To print or access UDV use following syntax

   Syntax: $variablename

   echo Command: Use echo command to display text or value of variable.

   Syntax: echo [options] [string, variables...]

Options

-n Do not output the trailing new line.

-e Enable interpretation of the following backslash escaped characters in the strings:

\a alert (bell)

\b backspace

\c suppress trailing new line

\n new line

\r carriage return

\t horizontal tab

\\ backslash

Shell Arithmetic: Use to perform arithmetic operations.

Syntax: expr op1 math-operator op2

Examples:

$ expr 1 + 3

$ expr 2 - 1

$ expr 10 / 2

$ expr 20 % 3

$ expr 10 * 3

$ echo `expr 6 + 3`

The read Statement: Use to get input (data from user) from keyboard and store (data) to variable.

Syntax: read variable1, variable2,...variableN

Ex. echo "Your first name please:"

read fname

Command line arguments:

$1,$2...$9 –positional parameter representing cnd line args

$# -total no. of args

$0 –name of executed cmd

$* -complete set of positional parameters

$? –exit status of last cmd

$$ - pid of current shell

$! -pid of last background process

Control structures in shell:

1) Decision control structure: The syntax is as follows:

```
if <condition>
then
        <do something>
else
        <do something else>
fi
```

Can use the 'test' or [ ] command for condition

```
Ex. if test $a = $b                    if [ $a = $b ]
    then                                then
                echo $a                         echo $a
    fi                                  fi
```

Test & [ ] operator:

1) Numeric comparison

- -eq equal to
- -ne not equal to
- -gt greater than
- -ge greater than equal to
- -lt less than
- -le less than equal to

2) Compares two strings or a single one for a null value

- s1=s2 string s1 =s2
- s1!=s2 string s1 not equal to s2
- -n str string str not a null string
- -z str string str as a null string
- Stg string str is assigned & not null

- s1==s2 string s1 = s2

3) Checks files attributes

- -f file file exists & is regular
- -r file file exists & is readable
- -w file file exists & is writable
- -x file file exists & is executable
- -d file file exists & is directory
- -s file file exists & has size greater than zero
- f1 –nt f2 f1 is newer than f2
- f1 –ot f2 f1 is older than f2
- f1 –ef f2 f1 is linked to f2

2) Loops

1. for: The syntax is as follows:

for VAR in LIST

do

<something>

done

2. While: The syntax is as follows:

while <condition>

do

<something>

done

3. until : The syntax is as follows:

until <condition>

do

<something>

done

3) The case Statement : The syntax is as follows:

case $variable-name in

pattern1) command ... .. command;;

pattern2) command ... .. command;;

patternN) command ... .. command;;

*) command ... .. command;;

esac

How to create file?

Syntax: vi filename.sh

i-insert mode

:w save

:wq Save and quit the file

How to execute the shell script?

1) Method1

$ chmod +x filename.sh

$ ./filename.sh

2) Method 2

$ bash filename.sh

How to de-bug the shell script?

While programming shell sometimes you need to find the errors (bugs) in shell script and correct the errors (remove errors - debug). For this purpose you can use -v and -x option with sh or bash command to debug the shell script.

General syntax is as follows:

Syntax:

sh option { shell-script-name }

OR

bash option { shell-script-name }

Option can be

-v Print shell input lines as they are read.

-x After expanding each simple-command, bash displays the expanded value of PS4 system variable, followed by the command and its expanded arguments

Functions: Functions enable you to break down the overall functionality of a script into smaller, logical subsections, which can then be called upon to perform their individual task when it is needed.

Using functions to perform repetitive tasks is an excellent way to create code reuse. Code reuse is an important part of modern object-oriented programming principles. Shell functions are similar to subroutines, procedures, and functions in other programming languages.

Creating Functions: To declare a function, simply use the following syntax:

```
function_name () {
  list of commands
}
```

The name of your function is function_name, and that's what you will use to call it from elsewhere in your scripts. The function name must be followed by parentheses, which are followed by a list of commands enclosed within braces.

Example:

Following is the simple example of using function:

```
#!/bin/sh
# Define your function here
Hello () {
  echo "Hello World"
}
# Invoke your function
Hello
```

When you would execute above script it would produce following result:

$./test.sh

Hello World

Passing argument to function in shell scripts:

Passing argument to the functions in shell script is very easy. Just use $1, $2, .. $n variables that represent arguments in the function. Following is the example of function that takes two arguments and prints them.

```
 #!/bin/sh
# Define your function here
Hello () {
   echo "Hello World $1 $2"
}
# Invoke your function
Hello Zara Ali
```

This would produce following result:

```
$./test.sh
Hello World Zara Ali
$
```

Thus if your function will use three arguments you can just use $1, $2 and $3 in your code.

You may want to use **$#** which gives you number of arguments passed to the function if you want to implement a function with variable arguments.

Returning Values from Functions:

If you execute an exit command from inside a function, its effect is not only to terminate execution of the function but also of the shell program that called the function.

If you instead want to just terminate execution of the function, then there is way to come out of a defined function. Based on the situation you can return any value from your function using the **return** command whose syntax is as follows:

return code

Here *code* can be anything you choose here, but obviously you should choose something that is meaningful or useful in the context of your script as a whole.

```
#!/bin/sh
# Define your function here
Hello () {
  echo "Hello World $1 $2"
  return 10
}
# Invoke your function
Hello Zara Ali
# Capture value returned by last command
ret=$?
echo "Return value is $ret"
```

This would produce following result:

```
$./test.sh
Hello World Zara Ali
Return value is 10
```

Nested Functions:

One of the more interesting features of functions is that they can call themselves as well as call other functions. A function that calls itself is known as a *recursive function*.

Following simple example demonstrates a nesting of two functions:

```
#!/bin/sh
# Calling one function from another
number_one () {
  echo "This is the first function speaking..."
```

```
  number_two
}
number_two () {
  echo "This is now the second function speaking..."
}
# Calling function one.
number_one
```

This would produce following result:

```
This is the first function speaking...
This is now the second function speaking...
```

ALGORITHMS

For Palindrome checking:

1. Start.
2. Accept the string from user.
3. Find the actual length of string as len.
4. Initialize a pointer to character in string to 1 and also flag to true.
5. Take the character pointed to by the pointer and the character pointed to by len
6. If the characters are not found, make the flag as false and go to step 9.
7. Decrement variable len by 1 and increment pointer by 1.
8. If the value of len is less than or equal to 1 then repeat from step 5.
9. If the flag is true, display the message that 'String is a palindrome' else display 'String is not palindrome.'
10. Stop.

Test Condition:

- String should not be NULL.

For Bubble sort:

1. Start.
2. Accept how many numbers to be sort say n .
3. Accept the numbers in array say num [].
4. Initialize i and j to 0.
5. While i<n .
6. do

```
        assign j=0
        while j<n
         do
           j=i
           if num[j]<num[i] then
                swap num[i] and num[j]
           end if
        j=j+1
        done
        i=i+1
done
```

7. Display the sorted list.
8. Stop.

Test Conditions:

- A certain maximum number of elements can be entered.
- Negative numbers are allowed.

For Substring checking:

1. Start.
2. Accept the two strings.
3. Compare two strings character by character.
4. If match is found then store the pointer of first string in an array.
5. Bring counter for second string to the first position.
6. Repeat the steps 3 to 5 until first string gets over.
7. Display the position array if substring exists else display substring does not exist.
8. Stop.

Test condition:

- First string should not be NULL.

Conclusion: Thus we have studies how to use shell script. It is Useful for integrating our existing applications. And useful for System Administrator.

FAQs:

1. Define shell. What are types of shell?
2. What are different control structures used in shell programming?
3. What do you mean by positional parameters & enlist them?
4. How to specify default statement in case statement?
5. What are types of variables?
6. How functions are used in Shell scripting?

Experiment Number: 04

TITLE: Classical problems using Threads and Mutex

PROBLEM STATEMENT: Implementation of Classical problem Reader Writer using Threads and Mutex

OBJECTIVES:

1. To understand the solution to the problems of mutual exclusion.
2. To grasp techniques and to develop the skills in the use of the tools: semaphores, threads, mutex in concurrent programming.

DESCRIPTION: The problem consists of readers and writers that share a data resource. The readers only want to read from the resource, the writers want to write to it. Obviously, there is no problem if two or more readers access the resource simultaneously. However, if a writer and a reader or two writers access the resource simultaneously, the result becomes indeterminable. Therefore the writers must have exclusive access to the resource.

A data object is to be shared among several concurrent processes. Some of these processes may only want to read content of the shared object, whereas others may want to update the shared object.

We distinguish between these two types of processes by referring to those processes that are interested in only reading as readers and to rest as writers. Obviously, if two readers access the shared data object simultaneously, no adverse effect will result.

1. The **first** readers-writers problem: Requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object.
2. The **second** readers – writers problem: Requires that, once a writer is ready, that writer performs its write as soon as possible. At given time, there is only one writer and any number of readers. When a writer is writing, the readers cannot enter into the database .The readers need to wait until the writer finishes to write to the database. Once a reader succeeds in reading the database, subsequent readers can enter into the critical section without waiting for the precedent reader finish to read. On the other hand, a writer who

arrives later than the reader who is reading currently is required to wait till the last reader finishes read. Only when the last reader finishes reading, can the writer enter into the critical section and is able to write to the database.

MUTEX:

A mutex is mutual exclusion lock. Only one thread can hold the lock. Mutexes are used to protect data or other resources from concurrent access. A mutex has attributes, which specify the characteristics of the mutex.

ALGORITHM:

Using mutex and threads:

1. Start.
2. Create two threads associated with processes update and display.
3. Declare mutex variable timer_lock and initialize to 0 using pthread_mutex_init function.
4. Join two threads. Initiate processes update and display.

Update ( )

1. Lock access to critical sections using mutex timer_lock.
2. Update the values of variables seconds, minutes, and hours.
3. Unlock the access to critical sections using mutex.

Display( )

1. Lock the access to critical sections using mutex timer_lock.
2. Display the values of variables hours, minutes, seconds.
3. Unlock the access to critical sections using mutex timer_lock.

CONCLUSION: Thus learned how Mutex is Applicable in all applications where synchronization is used such as all Operating systems.

FAQS:

1. What is critical section?
2. Differentiate between thread and process?

3. Can a mutex be locked more than once?
4. Can a thread acquire more than one lock (Mutex)?
5. How will you change the solution for n readers and n writer?

Experiment Number: 05

TITLE: Classical problems using Threads and Semaphore

PROBLEM STATEMENT: Implementation of Classical problems Producer Consumer using Threads and Semaphore.

OBJECTIVES:

1. To understand problems of mutual exclusion and Producer Consumer Problem.
2. To grasp techniques and to develop the skills in the use of the tools: semaphores, threads, and mutex in concurrent programming.

THEORY: The producer-consumer problem illustrates the need for synchronization in systems where many processes share a resource. In this problem, two processes share a fixed size buffer. One process produces information and puts it in the buffer, while the other process consumes information from the buffer. These processes do not take turns accessing the buffer, they both work concurrently.

SEMAPHORE:

- It can be used to restrict access to the database under certain conditions.
- Semaphore allows a sleep and wakeup mutual exclusion policy to be implemented.
- Basically, a semaphore is a new type of variable. A semaphore can have a value of 0(meaning no wakeups are saved) or a positive integer value, indicating the number of sleeping processes.
- Two different operations can be performed on a semaphore, down and up, corresponding to Sleep and Wakeup.

ALGORITHM

Using semaphore and thread:

1. Declare two thread variables of pthread_t structure.
2. Create two threads associated with producer and consumer processes using pthread_create function.
3. Declare two semaphore variables of sem_t structure bfull and bempty.

4. Initialize the semaphore variable empty to 0 and full to 1 using sem_init function.
5. Join two threads using pthread_join function.
6. Initiate two processes.
7. Destroy both the semaphore variables using sem_destroy function.
8. Stop.

Producer ( )

1. If semaphore variable bfull is 1, then the producer will wait on empty using the sem_wait function.
2. Else the producer will produce the string and execute the critical section. Then signal the bfull semaphore variable using the sem_post function on full and unblock the consumer thread.

Consumer ( )

1. If there is no data to consume, then the thread waits on the semaphore variable bfull using the sem_wait function.
2. Else the consumer executes the critical section and consumes the data. Then the consumer signals the bempty variable using sem_post function and unblocks the producer thread.

Test Conditions: Maximum string length should not be exceeded**.**

CONCLUSION: Thus learned how Mutex is Applicable in all applications where synchronization is used such as all Operating systems.

FAQS

1. How will you change the solution for *n* producers and *n* consumers?
2. What do you mean by mutual Exclusion condition?
3. Are binary semaphore and mutex same?
4. What is difference between thread and process?
5. What mean by is semaphore?
6. What is mean by critical section?

Experiment Number: 06

TITLE: CPU Scheduling

PROBLEM STATEMENT: Write a program to compute the finish time, turnaround time and waiting time by using different CPU scheduling algorithms

OBJECTIVES:

To study different process scheduling algorithms.

THEORY:

Scheduling decisions may take place when a process switches from:

1.Running to waiting.

2.Running to ready.

3.Waiting to ready.

4.Running to terminate.

Non-preemptive scheduling:

The current process keeps the CPU until it releases the CPU by either terminating or by switching to a waiting state.

Non-preemptive scheduling occurs only under situations 1 and 4, it does not require special hardware (timer).

Preemptive scheduling:

The currently running process may be interrupted and moved to the ready state by the operating system. It requires special hardware (timer), mechanisms to coordinate access to shared data.

Definitions:

- Throughput Number of processes/time unit.
- Turnaround Time it take to execute a process from start to finish.
- Waiting Time Total time spent in the ready queue.
- Response time Amount of time it takes to start responding (average, variance).

ALGORITHMS

First Come First Serve (FCFS):

1. Accept burst time and arrival time for every process entered by user.
2. Compare the arrival time of all processes.
3. Sort processes in ascending order with respect to their arrival time.
4. Execute all processes in sorted order.
5. Calculate the finish time of each process using the formula.
   FT = start time + burst time.
6. Calculate the turnaround and waiting time for all processes.
7. Display finish time, turnaround time, waiting time, Gantt chart.
8. Stop.

Shortest Job First (SJF):

1. Accept the number of processes from the user.
2. Accept arrival time and burst times for each process.
3. Start with arrival time.
4. For every arrival time, check which jobs are available.
5. Select job with shortest burst time.
6. Complete the selected process.
7. Continue steps 4 & 6 until all processes are complete.
8. Display finish time, turnaround time, waiting time with Gantt chart.
9. Stop.

Round Robin:

1. Accept the number of processes from the user.
2. Accept arrival time and burst times for each process.
3. Start with arrival time
4. Execute all processes present at arrival time for one time slot.
5. Increment arrival time.

6. Continue steps 3 to 4 until all the processes are complete.
7. Display finish time, turnaround time, waiting time, Gantt chart.
8. Stop.

Preemptive SJF:

1. Accept the number of processes from the user.
2. Accept arrival time and burst times for each process.
3. Sort all the processes according to the arrival time.
4. Start with the first process
5. After the first time slice of the process, if any other process has less arrival time then execute that.
6. Continue this process till all the processes are completed.
7. Display finish time, turnaround time, waiting time with Gantt chart.
8. Stop.

CONCLUSION: Operating system uses scheduling algorithms in order to provide good response to users.

FAQS:

1. Why is scheduling used?
2. What is the difference between preemptive & non- preemptive scheduling?
3. Which algorithm is more useful & why?
4. What do you mean by time quantum?

Experiment Number: 07

TITLE: Banker's Deadlock Avoidance Algorithm

PROBLEM STATEMENT: Write a program to check whether given system is in safe state or not using Banker's Deadlock Avoidance algorithm

OBJECTIVES:

1. To simulate deadlock avoidance.
2. To check if state is safe or not.

THEORY:

Deadlock: In operating systems or databases, a situation in which two or more processes are prevented from continuing while each waits for resources to be freed by the continuation of the other.

Deadlock Characterization: Deadlock can arise if four conditions hold simultaneously. (All four must hold)

1. Mutual exclusion: Only one process at a time can use a resource
2. Hold and Wait: A process holding at least one resource is waiting to acquire additional resources held by other processes.
3. No preemption: A resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. Circular Wait: There exists a set $\{P_0, P_1, \ldots P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2..P_{n-1} is waiting for a resource that is held by P_n ,and P_n is waiting for a resource that is held by P_0.

Deadlock Prevention: Restrain the ways that processes can make resource requests:

Mutual Exclusion- not required for sharable resources; must hold for non-sharable resources

Hold and wait-must guarantee that whenever a process requests a resource, it does not hold any other resources.

- Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
- Low resource utilization; starvation possible

No Preemption-

- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
- Preempted resources are added to the list of resources for which the process is waiting.
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Circular wait-Impose a total ordering of all resources types and require that each process request resources in a increasing order of enumeration.

Deadlock Avoidance:

Requires that the system has additional information in advance.

- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need
- The deadlock –avoidance algorithm dynamically examines the resource allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

ALGORITHMS

1. Start.
2. Calculate the current need for each process, for each resource from the data entered by user.
3. For a process, check if current available resources satisfy all current needs.

4. If all are satisfied completes the process and adds all its current allocations to available resources.
5. If all are not satisfied, check for next process.
6. Repeat from step 3, for as many times as there are processes.
7. If all processes have been completed, system is in safe state and display safe sequence. Else system is not in safe state.
8. Stop.

Test Condition: Safe sequence is not unique. A different safe sequence may also be possible.

APPLICATIONS

Can be used in a system where prior information regarding usage of resources for different processes is known in advance.

FAQS:

1. What do you mean by deadlock?
2. What are 4 conditions necessary for deadlock existence?
3. What are time complexities of deadlock avoidance and deadlock detection algorithm?

Experiment Number: 08

TITLE: Page Replacement Algorithm

PROBLEM STATEMENT: Write a program for different placement algorithms.

OBJECTIVES:

To study different Page Replacement algorithms

THEORY:

Page: One of numerous equally sized chunks of memory.

Page Replacement: This policy determines which page should be removed from main memory so that page from secondary memory replaces it.

ALGORITHMS

First In First Out (FIFO):

1. Start.
2. Accept the sequence of requirement of pages.
3. Initialize a pointer to the first page.
4. Check if the page is already present in the main memory.
5. If present, repeat from step 4, for next page.
6. If the page is not present, remove the page, which has entered the main memory first and put this page.
7. Increment the pointer to page to be replaced to next page.
8. Repeat from step 4 till all pages in sequence are over.
9. Stop.

Least Recently Used (LRU):

1. Start.
2. Accept the sequence of requirement of pages.
3. Initialize a count for each page in main memory to zero.
4. Check if the page is already present in the main memory.

5. If the page is not present, find the page in memory with maximum count, which is the least recently used and replace it.
6. Set it's count to zero and increment others count.
7. If found, set its count to zero and increment others count.
8. Repeat from step 4 till all pages in sequence are over.
9. Stop.

Optimal:

1. Start.
2. Accept the sequence of requirement of pages.
3. Initialize a count for each page in main memory to zero.
4. Check if the page is already present in the main memory.
5. If the page is not present, find the page which will not be used for longest period.
6. Repeat steps 4 till all pages in sequence are over.
7. Stop.

CONCLUSION: Used in memory management in Operating Systems.

FAQS:

1. What do you mean by virtual memory?
2. What is segmentation?
3. What is a translation look aside buffer?
4. What is thrashing?
5. What is Belady's anomaly?

Experiment Number: 09

TITLE: Disk Scheduling Algorithm

PROBLEM STATEMENT: Write a program to implement different disk scheduling algorithms.

OBJECTIVES:

To study different disk scheduling algorithms

THEORY:

In multiprogramming systems several different processes may want to use the system's resources simultaneously. For example, processes will contend to access an auxiliary storage device such as a disk. The disk drive needs some mechanism to resolve this contention, sharing the resource between the processes fairly and efficiently.

First Come First Serve (FCFS)

The disk controller processes the I/O requests in the order in which they arrive, thus moving backwards and forwards across the surface of the disk to get to the next requested location each time. Since no reordering of request takes place the head may move almost randomly across the surface of the disk. This policy aims to minimize response time with little regard for throughput.

Shortest Seek Time First (SSTF)

Each time an I/O request has been completed the disk controller selects the waiting request whose sector location is closest to the current position of the head. The movement across the surface of the disk is still apparently random but the time spent in movement is minimized. This policy will have better throughput than FCFS but a request may be delayed for a long period if many closely located requests arrive just after it.

SCAN

The drive head sweeps across the entire surface of the disk, visiting the outermost cylinders before changing direction and sweeping back to the innermost cylinders. It selects the next waiting requests whose location it will reach on its path backwards and forwards across the disk. Thus, the movement time should be less than FCFS but the policy is clearly fairer than SSTF.

Circular SCAN (C-SCAN)

C-SCAN is similar to SCAN but I/O requests are only satisfied when the drive head is traveling in one direction across the surface of the disk. The head sweeps from the innermost cylinder to the outermost cylinder satisfying the waiting requests in order of their locations. When it reaches the outermost cylinder, it sweeps back to the innermost cylinder without satisfying any requests and then starts again

ALGORITHMS

First Come First Serve (FCFS):

1. Start.
2. Accept the number of tracks *n.*
3. Accept the requested tracks and store it in the track [ ].
4. Consider the first value of track[ ] as starting track.
5. Process the track values in given order to calculate difference as
   trackdiff [i] = track[i] - track[i+1] .
6. Calculate totaldiff = totaldiff + trackdiff[i].
7. Calculate average seek time.
8. Display the value of average seek time.
9. Stop.

Shortest Service Time First (SSTF):

1. Start.
2. Accept the number of tracks *n.*
3. Accept the requested tracks and store it in track[ ].
4. Consider the first value of track[ ] as starting track .
5. First process all the tracks which are having value less than starting track in decreasing order of tracks as trackdiff[i] = track[i]-track[i+1] .
6. Then process all the tracks which are having value greater than starting in increasing order of track as trackdiff[i] = track[i] - track[i+1].
7. Calculate totaldiff = totaldiff + trackdiff[i]
8. Calculate average seek time.
9. Display the value of average seeks time.
10. Stop.

SCAN:

1. Start.
2. Accept the number of tracks *n.*
3. Accept the requested tracks and store it in track[ ]
4. Consider the first value of track[ ] as starting track .
5. First process all the tracks which are having value greater than starting track in increasing order of track as trackdiff[i] = track[i] - track[i+1]
6. then process all the tracks which are having value less than starting track in decreasing order of track as trackdiff[i] = track[i] - track[i+1]
7. Calculate totaldiff = totaldiff + trackdiff[i]
8. Calculate average seek time.
9. Display the value of average seek time.
10. Stop

C-SCAN:

1. Start.
2. Accept the number of tracks *n.*
3. Accept the requested tracks and store it in track[ ]
4. Consider the first value of track[ ] as starting track .
5. First process all the tracks which are having value greater than starting track in increasing order of track as trackdiff[i]= track[i] - track[i+1]
6. Then process all the tracks which are having value less than starting track in increasing order of track(Wrap around to starting track) as trackdiff[i] = track[i] - track[i+1]
7. Calculate totaldiff = totaldiff + trackdiff[i]
8. Calculate average seek time.
9. Display the value of average seek time.
10. Stop

CONCLUSION:

1. Applicable in Multimedia application to reduce disk access.

FAQS:

1. What do you mean by seek time?
2. What is difference between SCAN & C-SCAN method?
3. Give an analysis of comparison of 4 algorithms.

INTORODUCTION OF PROJECT

Outcomes:

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

Al. 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.

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-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.*

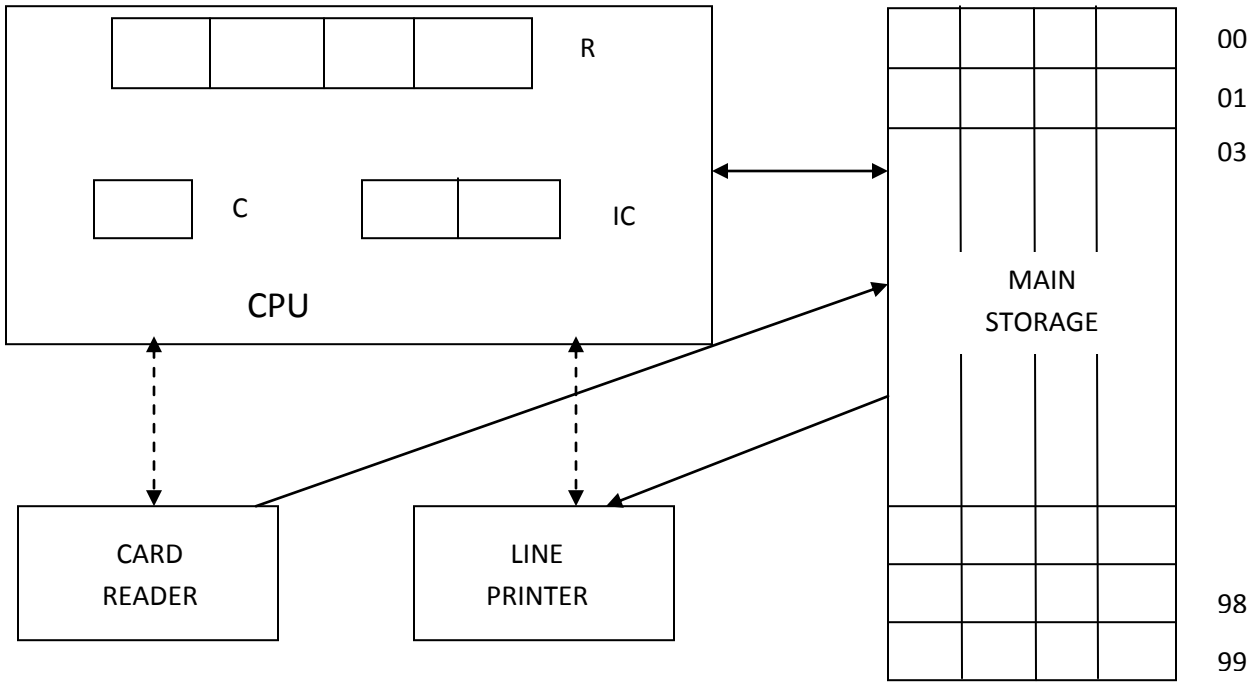


Fig A-1: Virtual user machine.

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

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.)

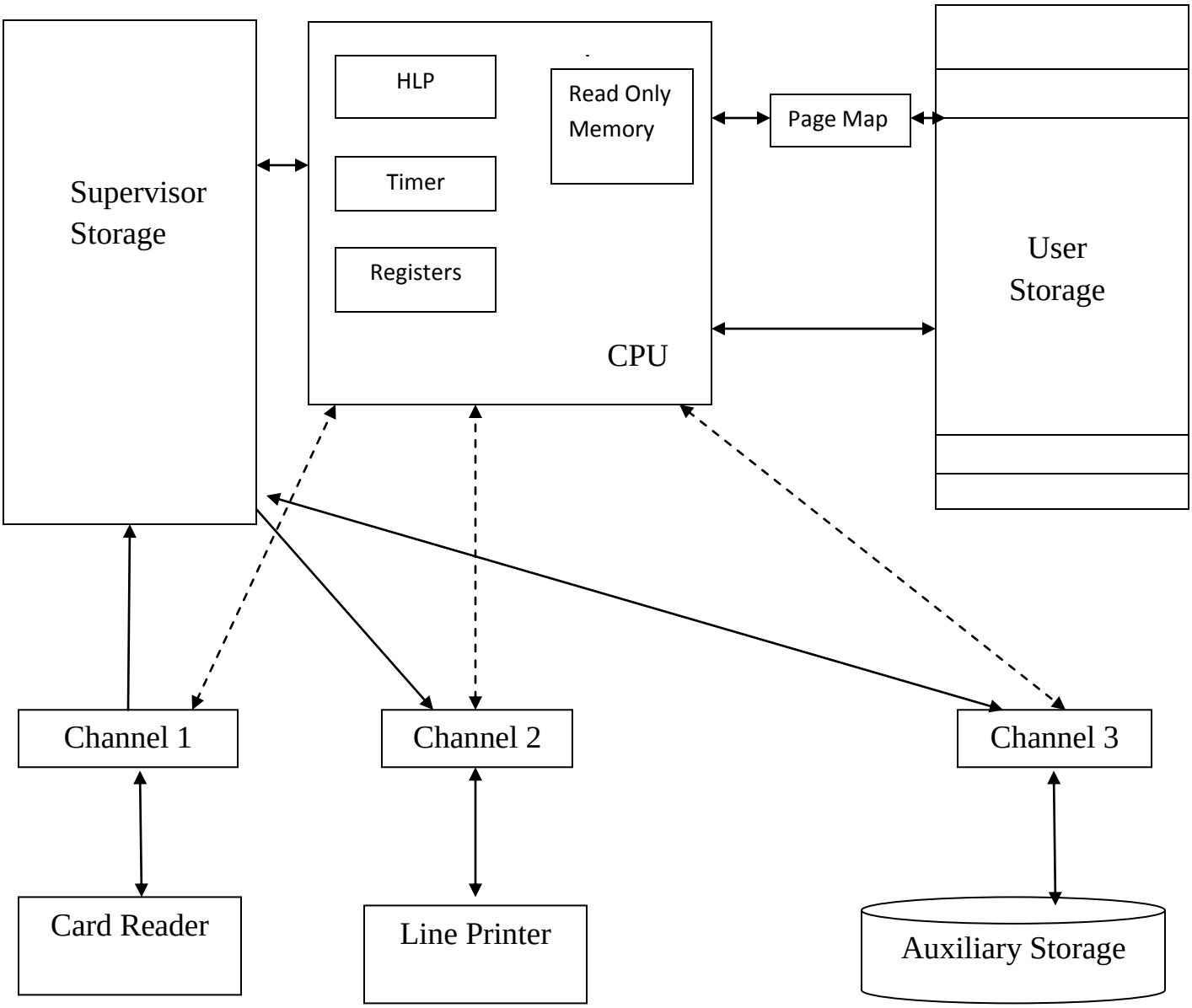


Fig A-2 : Real Machine

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, 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 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).

(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 = 1 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 + P1 + 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 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: Protection; P1= 2: invalid operation code

(2) SI = 1: GD; SI= 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 IOIint else

if P1 != 0 then go to PROGint else

if SI != 0 then go to SUPint else

go to T1Mint;

Comment IOIint, 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 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 i^{th} 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.

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 o 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.

Project

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 = 0 (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

Question Bank:

1. What is use of different registers?
2. What is significance of SI?
3. When SI would set to 1/2/3?
4. What is Interrupt?
5. What is system call?

Project

Title: Implementation of a multiprogramming operating system

Problem Statement: Stage II:

i. Paging
ii. Error Handling
iii. Interrupt Generation and Servicing
iv. Process Data Structure

Description:

Assumption:

- 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

Error Message Coding

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

TI	SI	Action
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

TI	PI	Action
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 ← LLC + 1

If LLC > TLL, TERMINATE (2)

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

EXECUTEUSERPROGRAM

TERMINATE (EM):

Write 2 blank lines in output file

Write 2 lines of appropriate Terminating Message as indicated by EM

LOAD

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 ← 00

EXECUTEUSERPROGRAM

END (MOS)

EXECUTEUSERPROGRAM (SLAVE MODE):

ADDRESS MAP (VA, RA)

Accepts VA, either computes & returns RA or sets PI ← 2 (Operand Error) or PI ← 3 (Page Fault)

LOOP

ADDRESSMAP (IC, RA)

If PI ≠ 0, End-LOOP (F)

IR ← M[RA]
IC ← IC+1
ADDRESSMAP (IR[3,4], RA)
If PI ≠ 0, End-LOOP (E)
Examine IR[1,2]
LR: R ← M [RA]
SR: R → M [RA]
CR: Compare R and M [RA]
If equal C ← T else C ← F
BT: If C = T then IC ← IR [3,4]
GD: SI = 1 (Input Request)
PD: SI = 2 (Output Request)
H: SI = 3 (Terminate Request)
Otherwise PI ← 1 (Operation Error)
End-Examine
End-LOOP (X) X = F (Fetch) or E (Execute)

SIMULATION:
Increment TTC
If TTC = TTL then TI ← 2

If SI or PI or TI ≠ 0 then Master Mode, Else Slave Mode

Question Bank:

1. What is significance of PI?
2. What is use of PTR?
3. What PCB contains?
4. What different errors may occur in a job?
5. When TI would set to 1 / 2?
6. What is a difference between relative ,absolute and logical address?