SATHYABAMA UNIVERSITY

(Established Under Section 3 of UGC act 1956)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

LAB MANUAL

B.E COMPUTER SCIENCE & ENGINEERING

SCSX4013 -Operating System Lab

List of Experiments

1. Study of basic LINUX & Vi Editor command
2. Searching a SubString in given text
3. Menu Based Math Calculator
4. Printing pattern using loop statement
5. Converting File names from Uppercase to Lowercase
6. Manipulate Date/Time/Calendar
7. Showing various system information
8. Implementation of process scheduling mechanism – FCFS, SJF, Priority Queue.
9. Reader – Writer Problem.
10. Dinner's Philosopher Problem.
11. First Fit, Worst Fit, Best Fit allocation strategy.
12. Bankers Algorithm
13. Implement the producer consumer problem using Semaphore
14. Implement some memory management Scheme

Study of basic LINUX & Vi Editor command

Basic Linux Commands

File Handling

- Text Processing
- System Administration
- Process Management
- Archival
- Network
- File Systems
- Advanced Commands

Primary – man(manual) pages.

Ⓟ **man** <command> shows all information about the command

Ⓟ <command> help shows the available options for that command

✦ **Secondary** – Books and Internet

File Handling commands

- **mkdir** – make directories

Usage: mkdir [OPTION] DIRECTORY...

eg. mkdir prabhat

- **ls** – list directory contents

Usage: ls [OPTION]... [FILE]...

eg. ls, ls l,

ls prabhat

- **cd** – changes directories

Usage: cd [DIRECTORY]

eg. cd prabhat

- **pwd** -print name of current working directory

Usage: pwd

- **vim** – Vi Improved, a programmers text editor

Usage: vim [OPTION] [file]...

eg. vim file1.txt

cp – copy files and directories
Usage: cp [OPTION]... SOURCE DEST

eg. cp sample.txt sample_copy.txt

cp sample_copy.txt target_dir

✦ **mv** – move (rename) files

Usage: mv [OPTION]... SOURCE DEST

eg. mv source.txt target_dir

mv old.txt new.txt

rm remove

files or directories

Usage: rm [OPTION]... FILE...

eg. rm file1.txt , rm rf some_dir

• **find** – search for files in a directory hierarchy

Usage: find [OPTION] [path] [pattern]

eg. find file1.txt, find name file1.txt

• **history** – prints recently used commands

Usage: history

Pattern

A Pattern is an expression that describes a set of strings which is used to give a concise description

of a set, without having to list all elements.

eg. ab*cd matches anything that starts with ab and ends with cd etc.

ls *.txt – prints all text files

Text Processing

• **cat** – concatenate files and print on the standard output

Usage: cat [OPTION] [FILE]...

eg. cat file1.txt file2.txt

cat n file1.txt

• **echo** – display a line of text

Usage: echo [OPTION] [string] ...

eg. echo I love India

echo $HOME

• **wc** print

the number of newlines, words, and bytes in files

Usage: wc [OPTION]... [FILE]...

eg. wc file1.txt

wc L file1.txt

sort – sort lines of text files

Usage: sort [OPTION]... [FILE]...

eg. sort file1.txt

sort r file1.txt

System Administration

• **chmod** – change file access permissions

Usage: chmod [OPTION] [MODE] [FILE]

eg. chmod 744 calculate.sh

• **chown** – change file owner and group

Usage: chown [OPTION]... OWNER[:[GROUP]] FILE...

eg. chown remo myfile.txt

su – change user ID

Usage: su [OPTION] [LOGIN]

eg. su remo, su

• **passwd** – update a user's authentication tokens(s)

Usage: passwd [OPTION]

eg. Passwd

• **who** – show who is logged on

Usage: who [OPTION]

eg. who , who b, who q

Process management

ps – report a snapshot of the current processes

Usage: ps [OPTION]

eg. ps, ps el

- kill – to kill a process(using signal mechanism)

Usage: kill [OPTION] pid

eg. kill 9

archival

tar – to archive a file

Usage: tar [OPTION] DEST SOURCE

eg. tar cvf

/home/archive.tar /home/original tar xvf/home/archive.tar

- **zip** – package and compress (archive) files

Usage: zip [OPTION] DEST SOURSE

eg. zip original.zip original

- **unzip** – list, test and extract compressed files in a ZIP archive

Usage: unzip filename

eg. unzip original.zip

- **du** – estimate file space usage

Usage: du [OPTION]... [FILE]...

eg. du

- **df** – report filesystem disk space usage

Usage: df [OPTION]... [FILE]...

eg. df

- **quota** – display disk usage and limits

Usage: quota [OPTION]

eg. quota v

Advanced Commands

- **reboot** – reboot the system

Usage: reboot [OPTION]

eg. reboot

- **poweroff** – power off the system

Usage: poweroff [OPTION]

eg. Poweroff

SHELL PROGRAMMING COMMANDS

Shell is a user program or it's environment provided for user interaction. Shell is an command language interpreter that executes commands read from the standard input device (keyboard) or from a file.
Shell is not part of system kernel, but uses the system kernel to execute programs, create files etc.

Variables in Shell

In Linux (Shell), there are two types of variable:
(1) **System variables** - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.
(2) **User defined variables (UDV)** - Created and maintained by user. This type of variable defined in lower letters.

USER-DEFIEND VARIABLES

Syntax:

variable name=value

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

Example:

To define variable called 'vech' having value Bus

```
$ vech=Bus
```

To define variable called n having value 10

```
$ n=10
```

RULES FOR NAMING THE VARIABLES

(1) Variable name must begin with Alphanumeric character or underscore character (_), followed by one or more Alphanumeric character. For e.g. Valid shell variable are as follows
HOME
SYSTEM_VERSION
vech
no
(2) Don't put spaces on either side of the equal sign when assigning value to variable. For e.g. In following variable declaration there will be no error

```
$ no=10
```

But there will be problem for any of the following variable declaration:

```
$ no =10
$ no= 10
$ no = 10
```

(3) Variables are case-sensitive, just like filename in Linux. For e.g.

```
$ no=10
$ No=11
$ NO=20
$ nO=2
```

Above all are different variable name, so to print value 20 we have to use $ echo $NO and not any of the following

```
$ echo $no
```
will print 10 but not 20
```
$ echo $No
```
will print 11 but not 20
```
$ echo $nO
```
will print 2 but not 20

(4) You can define NULL variable as follows (NULL variable is variable which has no value at the time of definition) For e.g.
$ vech=
$ vech=""
Try to print it's value by issuing following command
```
$ echo $vech
```
Nothing will be shown because variable has no value i.e. NULL variable.

(5) Do not use **?,*** etc, to name your variable names.

To print or access UDV use following syntax
Syntax:
$variablename

Define variable vech and n as follows:
```
$ vech=Bus
$ n=10
```
To print contains of variable 'vech' type
```
$ echo $vech
```
It will print 'Bus',To print contains of variable 'n' type command as follows
```
$ echo $n
```

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

echo [options] [string, variables...]
Displays text or variables value on screen.
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

For e.g. **$ echo -e "An apple a day keeps away \a\t\tdoctor\n"**

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`
```

There are three types of quotes

Quotes	Name	Meaning
"	Double Quotes	"Double Quotes" - Anything enclose in double quotes removed meaning of that characters (except \ and $).
'	Single quotes	'Single quotes' - Enclosed in single quotes remains unchanged.
`	Back quote	`Back quote` - To execute command

Example:
$ echo "Today is date"
Can't print message with today's date.
$ echo "Today is `date`".
It will print today's date as, Today is Tue Jan.... ,Can you see that the `date` statement uses back quote?

The read Statement

Use to get input (data from user) from keyboard and store (data) to variable.
Syntax:
read variable1, variable2,. .variableN

```
$ vi sayH
#
#Script to read your name from key-board
#
echo "Your first name please:"
read fname
echo "Hello $fname, Lets be friend!"
```

Run it as follows:
$ *Your first name please:* ***vivek***
Hello vivek, Lets be friend!

More command on one command line

Syntax:
command1;command2
To run two command with one command line.

Examples:
$ date;who
Will print today's date followed by users who are currently login. Note that You can't use

$ date who

for same purpose, you must put semicolon in between date and who command.

CONTROL STATEMENTS

- Decision making
- Loops

Is there any difference making decision in Real life and with Computers? Well real life decision are quit complicated to all of us and computers even don't have that much power to understand our real life decisions. What computer know is 0 (zero) and 1 that is Yes or No. To make this idea clear, lets play some game (WOW!) with bc - Linux calculator program.

$ bc

See what happened if you type 5 > 2 as follows

5 > 2

1

1 (One?) is response of bc, How? bc compare 5 with 2 as, Is 5 is greater then 2, (If I ask same question to you, your answer will be YES), bc gives this 'YES' answer by showing 1 value. Now try

5 < 2

0

Try following in bc to clear your Idea and not down bc's response

5 > 12
5 == 10
5 != 2
5 == 5
12 < 2

Expression	Meaning to us	Your Answer	BC's Response
5 > 12	Is 5 greater than 12	NO	0
5 == 10	Is 5 is equal to 10	NO	0
5 != 2	Is 5 is NOT equal to 2	YES	1
5 == 5	Is 5 is equal to 5	YES	1
1 < 2	Is 1 is less than 2	Yes	1

It means when ever there is any type of comparison in Linux Shell It gives only two answer one is YES and NO is other.

if condition

if condition which is used for decision making in shell script, If given condition is true then command1 is executed.

Syntax:

```
if condition
then
        command1 if condition is true or if exit status
        of condition is 0 (zero)
        ...
        ...
fi
```

EXAMPLE

```
if [ -e . ]
then
        echo "Yes."
else
        echo "No."
fi
```

test command or [expr]

test command or [expr] is used to see if an expression is true, and if it is true it return zero(0), otherwise returns nonzero for false.

Syntax:

test expression OR [expression]

Example:

Following script determine whether given argument number is positive.

```
if test $1 -gt 0
then
echo "$1 number is positive"
fi
```

For Mathematics, use following operator in Shell Script

Mathematical Operator in Shell Script	Meaning	Normal Arithmetical/ Mathematical Statements	But in Shell	
			For test statement with if command	For [expr] statement with if command
-eq	is equal to	5 == 6	if test 5 -eq 6	if [5 -eq 6]
-ne	is not equal to	5 != 6	if test 5 -ne 6	if [5 -ne 6]

-lt	is less than	5 < 6	if test 5 -lt 6	if [5 -lt 6]
-le	is less than or equal to	5 <= 6	if test 5 -le 6	if [5 -le 6]
-gt	is greater than	5 > 6	if test 5 -gt 6	if [5 -gt 6]
-ge	is greater than or equal to	5 >= 6	if test 5 -ge 6	if [5 -ge 6]

if...else...fi
If given condition is true then command1 is executed otherwise command2 is executed.
Syntax:

```
if condition

then
            condition is zero (true - 0)

            execute all commands up to else statement

else
            if condition is not true then

            execute all commands up to fi
fi
```

For e.g. Write Script as follows:

```
$ vi isnump_n
# Script to see whether argument is positive or negative
#
if [ $# -eq 0 ]
then
echo "$0 : You must give/supply one integers"
exit 1
fi
if test $1 -gt 0
then
echo "$1 number is positive"
else
echo "$1 number is negative"
fi
```

Multilevel if-then-else

Syntax:

```
           if condition
           then
                      condition is zero (true - 0)
                      execute all commands up to elif statement
           elif condition1
           then
                      condition1 is zero (true - 0)
                      execute all commands up to elif statement
           elif condition2
           then
                      condition2 is zero (true - 0)
                      execute all commands up to elif statement
           else
                      None of the above condtion,condtion1,condtion2 are true
(i.e.
                      all of the above nonzero or false)
                      execute all commands up to fi
           fi
```

Loops in Shell Scripts

Loop defined as:
"*Computer can repeat particular instruction again and again, until particular condition satisfies. A group of instruction that is executed repeatedly is called a loop.*"

Bash supports:

- for loop
- while loop
- **Note** that in each and every loop,
- (a) First, the variable used in loop condition must be initialized, then execution of the loop begins.
- (b) A test (condition) is made at the beginning of each iteration.
- (c) The body of loop ends with a statement that modifies the value of the test (condition) variable.

for Loop

Syntax:

```
            for { variable name } in { list }
            do
                      execute one for each item in the list until the list is
                      not finished (And repeat all statement between do and
done)
            done
```

Before try to understand above syntax try the following script:

```
$ cat > testfor
for i in 1 2 3 4 5
do
```

```
echo "Welcome $i times"
done
```

Even you can use following syntax:

Syntax:

```
for (( expr1; expr2; expr3 ))
do
      .....
                  ...
      repeat all statements between do and
      done until expr2 is TRUE
Done
```

EXAMPLE

```
$ cat > for2
for ((  i = 0 ;  i <= 5;  i++  ))
do
  echo "Welcome $i times"
done
```

while loop

Syntax:

```
while [ condition ]
do
      command1
      command2
      command3
      ..
      ....
 done
```

EXAMPLE

```
$cat > nt1
#Script to test while statement
#
#   echo " Use to print multiplication table for given number"
n=$1
i=1
while [ $i -le 10 ]
do
  echo "$n * $i = `expr $i \* $n`"
  i=`expr $i + 1`
done
```

The case Statement

The case statement is good alternative to Multilevel if-then-else-fi statement. It enable you to match several values against one variable. Its easier to read and write.

Syntax:

```
case $variable-name in
   pattern1)   command
                    ...
                    ..
                    command;;
   pattern2)   command
                    ...
                    ..
                    command;;
   patternN)   command
                    ...
                    ..
                    command;;
   *)             command
                    ...
                    ..
                    command;;
esac
```

The *$variable-name* is compared against the patterns until a match is found. The shell then executes all the statements up to the two semicolons that are next to each other. The default is *) and its executed if no match is found. For e.g. write script as follows:

```
$ cat > car
#
# case $rental in
   "car") echo "For $rental Rs.20 per k/m";;
   "van") echo "For $rental Rs.10 per k/m";;
   "jeep") echo "For $rental Rs.5 per k/m";;
   "bicycle") echo "For $rental 20 paisa per k/m";;
   *) echo "Sorry, I can not gat a $rental for you";;
esac
```

Exercises no.2 to 7 has to be implemented using the shell commands.

Ex. No:8 SCHEDULING MECHANISMS

Aim: Write a C program to implement the various process scheduling mechanisms such as FCFS, SJF, Priority .

Algorithm for FCFS scheduling:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Set the waiting of the first process as '0' and its burst time as its turn around time

Step 5: for each process in the Ready Q calculate

(a) Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)

(b) Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

Step 6: Calculate

(a) Average waiting time = Total waiting Time / Number of process

(b) Average Turnaround time = Total Turnaround Time / Number of process

Step 7: Stop the process

Algorithm for SJF

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Start the Ready Q according the shortest Burst time by sorting according to lowest to highest burst time.

Step 5: Set the waiting time of the first process as '0' and its turnaround time as its burst time.

Step 6: For each process in the ready queue, calculate

(a) Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)

(b) Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

Step 6: Calculate

(a) Average waiting time = Total waiting Time / Number of process

(b) Average Turnaround time = Total Turnaround Time / Number of process

Step 7: Stop the process

Algorithm for Priority Scheduling:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Sort the ready queue according to the priority number.

Step 5: Set the waiting of the first process as '0' and its burst time as its turn around time

Step 6: For each process in the Ready Q calculate

(a) Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)

(b) Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

Step 7: Calculate

(a) Average waiting time = Total waiting Time / Number of process

(b) Average Turnaround time = Total Turnaround Time / Number of process

Step 8: Stop the process

Ex. No:9 READERS AND WRITERS PROBLEM

Aim:

To write a program to implement readers and writers problem

Algorithm:

Start ;

/* Initialize semaphore variables*/

integer mutex=1; // **Controls access to RC**

integer DB=1; **// controls access to data base**

integer RC=0; **// Number of process reading the database currently**

1.Reader( ) // The algorithm for readers process

Repeat continuously

DOWN(mutex); **// Lock the counter RC**

RC=RC+1; **// one more reader**

If(RC=1)DOWN(DB); **// This is the first reader.Lock the database for reading**

UP(mutex); **// Release exclusive access to RC**

Read database(); **// Read the database**

DOWN(mutex); **// Lock the counter RC**

RC=RC-1; **// Reader count less by one now**

If(RC=0)UP(DB); **// This is the last reader .Unlock the database.**

UP(mutex); **// Release exclusive access to RC**

End

2.Writer( ) // The algorithm for Writers process

Reepeat continuously

DOWN(DB); **// Lock the database**

Write_Database(); **// Read the database**

UP(DB); **// Release exclusive access to the database**

End

Step a: initialize two semaphore mutex=1 and db=1 and rc, (Mutex controls the access to read count rc)
Step b: create two threads one as Reader() another as Writer()
Reader Process:

Step 1: Get exclusive access to rc(lock Mutex)
Step 2: Increment rc by 1
Step 3: Get the exclusive access bd(lock bd)
Step 4: Release exclusive access to rc(unlock Mutex)
Step 5: Release exclusive access to rc(unlock Mutex)
Step 6: Read the data from database
Step 7: Get the exclusive access to rc(lock mutex)
Step 8: Decrement rc by 1, if rc =0 this is the last reader.
Step 9: Release exclusive access to database(unlock mutex)
Step 10 Release exclusive access to rc(unlock mutex)

Ex. No:10 DINING PHILOSOPHER PROBLEM

Aim: Write a program to solve the Dining Philosophers problem.

Algorithm:

1. Initialize the state array S as 0, S_i =0 if the philosopher i is thinking or 1 if hungry.
2. Associate two functions getfork(i) and putfork(i) for each philosopher i.
3. For each philiosopher I call getfork(i) , test(i) and putfork(i) if i is 0
4. Stop

Algorithm for getfork(i):

Step 1: set S[i]= 1 i.e. the philosopher i is hungry

Step 2: call test(i)

Algorithm for putfork(i)

Step 1: set S[i]=0 I.e. the philosopher i is thinking

Step 2: test(LEFT) and test(RIGHT)

Algorithm for test(i)

Step 1: check if (state[i]==HUNGRY && state[LEFT]!=EATING && state[RIGHT]!=EATING)

Step 2: give the i philosopher a chance to eat.

Ex. No:11 MEMORY ALLOCATION TECHNIQUES

AIM:

To implement memory allocation techniques using

a) First fit
b) Best fit
c) Worst fit &
d) To make comparative study

THEORY:

Memory Management Algorithm

In an environment that supports dynamic memory allocation, a number of strategies are used to allocate a memory space of size n (unused memory partition) from the list free holes to the processes that are competing for memory.

First Fit: Allocation the first hole which is big enough.

Best Fit: Allocation the smallest hole which is big enough

Worst Fit: Allocation the largest hole which is big enough

ALGORITHM:

Step 1: Start the program.

Step 2: Get the number of memory partition and their sizes.

Step 3: Get the number of processes and values of block size for each process.

Step 4: First fit algorithm searches all the entire memory block until a hole which is big enough is encountered. It allocates that memory block for the requesting process.

Step 5: Best-fit algorithm searches the memory blocks for the smallest hole which can be allocated to requesting process and allocates if.

Step 6: Worst fit algorithm searches the memory blocks for the largest hole and allocates it to the process.

Step 7: Analyses all the three memory management techniques and display the best algorithm which utilizes the memory resources effectively and efficiently.

Step 8: Stop the program.

Ex. No:12 BANNKER'S ALGORITHM

Aim: Write a program to implement Banker's Algorithm

Banker's Algorithm is used for avoidance:

This algorithm was suggested by Dijkstar, the name banker is used here to indicate that it uses a banker's activity for providing loans and receiving payment against the given loan. This algorithm places very few restrictions on the processes competing for resources. Every request for the resource made by a process is thoroughly analyzed to check, whether it may lead to a deadlock situation. If the result is yes then the process is blocked on this request. At some future time, its request is considered once again for resource allocation. So this indicated that, the processes are free to request for the allocation, as well as de-allocation of resources without any constraints. So this generally reduces the idling of resources.

Suppose there are (P) number of Processes and (r) number of resources then its time complexity is proportional to P x r^2

At any given stage the OS imposes certain constraints on any process trying to use the resource. At a given moment during the operation of the system, processes P, would have been allocated some resources. Let these allocations total up to S.

Let (K=r-1) be the number of remaining resources available with the system. Then k>=0 is true, when allocation is considered.

Let maxk be the maximum resource requirement of a given process Pi.

Actk be the actual resource allocation to Pi at any given moment.

Then we have the following condition.

Maxk<=p for all k and

To

Disadvantages of Banker's algorithm:

1. The maximum number of resources needed by the processes must be known in advance
2. The no of processes should be fixed.

Ex. No:13 PRODUCER CONSUMER PROBLEM

Aim:

To write a program to implement producer consumer problem.

Algorithm:

Step 1: Start.

Step 2: Let n be the size of the buffer

Step 3: check if there are any producer

Step 4: if yes check whether the buffer is full

Step 5: If no the producer item is stored in the buffer

Step 6: If the buffer is full the producer has to wait

Step 7: Check there is any cosumer.If yes check whether the buffer is empty

Step 8: If no the consumer consumes them from the buffer

Step 9: If the buffer is empty, the consumer has to wait.

Step 10: Repeat checking for the producer and consumer till required

Step 11: Terminate the process.

Ex. No:14 **PAGING**

Aim: To implement the Memory management policy- Paging.

Algorithm:

Step 1: Read all the necessary input from the keyboard.

Step 2: Pages - Logical memory is broken into fixed - sized blocks.

Step 3: Frames – Physical memory is broken into fixed – sized blocks.

Step 4: Calculate the physical address using the following

Physical address = (Frame number * Frame size) + offset

Step 5: Display the physical address.

Step 6: Stop the process.