LAB 1

Introduction to Linux Operating System

OBJECTIVE

To discover the features of Linux operating system and understand its file system hierarchy.

Linux Operating System

* Operating system for several types of computer platforms, primarily INTEL based PC’s.
* Goal has been to create a UNIX clone.
* Free of any commercially copy righted software.
* It has all the features of a modern fully fledged UNIX, like;
  1. Multitasking
  2. Multi-user
  3. Programmable Shells
  4. Device independence
  5. Open system portability
  6. Virtual memory
  7. Shared libraries
  8. Demand loading
  9. Proper memory management
  10. TCP/IP networking

Hardware Support

1. For Ubuntu Linux

* **Processor:** 2 Ghz dual core processor
* **Hard Disk:** 25GB hard drive space ( or USB stick, memory card or external drive)
* **Memory:** 4 GB RAM
* **VGA: capable of 1024\*768 screen resolution**
* **CD/DVD/USB for the installer media**

1. For Fedora ([A Red Hat Community Project](http://redhat.com)).

* **Processor:** 2 GHz dual core processor of faster
* **Hard Disk:** 15 GB unallocated drive space (20 GB recommended)
* **Memory:** 2 GB (minimum), 4 GB (recommended).
* **VGA: capable of 1024\*768 screen resolution**
* **CD/DVD/USB for the installer media**

Software Support

* Every Linux based OS has GCC, Emacs, X—Windows all standards Unix Utilities, TCP/IP, IPX/SPX, including SLIP and PPP.
* Copyright
* Linux Kernel Copyright belongs to Linux Torvalds
* Placed under GNU(General Public License) means that you may freely, copy, change and distribute it.

Linux File System

* Linux is committed to the *Filesystem Hierarchy Standard* (*FHS*) a collaborative document that defines the names and locations of many files and directories.
* Inverted tree structure, with the root at the top. The top directory**, /,** known as Root Directory.
* Linux operating system resides on two file system: the root file system known as **/**, and a file system mounted under **/usr.**

/

**/var /bin /sbin /usr /etc /lib**

**adam gwenl**

**mail news text data temp**

*Inverted Tree structure*

**/var (variable data files)**

* Any program that write log files or need spool or lock directories probably should write them to the /var directory."…. This includes spool directories and files, administrative and logging data, and transient and temporary files."
* The /var/spool directory hold data that is of transitory nature, such as mail and news that is recently received from or queued for transmission to another site.

**/bin**

* The /bin directory contains executable programs, know as binaries. Many Linux commands such as ls, arch, cat, cpio etc are actually programs found in this directory.

**/sbin**

* The /sbin directory is also used to store system binary files. Most files (fdisk, halt, hdparm, shutdown etc.) These files are used for system administration purpose.

**/usr**

* The /usr directory and its sub-directories are very important to the operation of your Linux system contains several directories with some of the most important programs.
* Sub-directories of /usr contains large software packages.
* The /usr directory usually has its own partition, and it should be mountable read-only.

**/etc**

* The /etc directory is reserved for configuration files that are local to your machine.
* The password file, startup scripts for Linux, list of IP addresses and many other types of configuration information stored in this directory.

**/lib**

* The /lib directory should contain only those libraries that are needed to execute the binaries in /bin and /sbin.

**/home**

* The /home directory is the base for user home directories.

**/dev**

* The /dev directory contains filesystem entries which represent devices that are attached to the system. These files are essential for the system to function properly. The dev/mouse file is for reading input for mouse.

Logging on to Linux

* To use the Linux operating system, you must first gain access to it by logging in. when you are log in; you are placed in your home directory.
* login [name | option]
* Enter your **Login** Name and press ENTER.

LOGIN:

* Enter your **PASSWORD** and press ENTER.

Login:

Password:

* Once you have entered all the correct information, the “prompt character” appears on the screen. This is a dollar sign($) for Bourne shell users and a percent sign(%) for C-shell users.
* The **man** command

man [option] [section] [title]

Displays information from the online reference manuals. man locates and prints the named title from the designated reference section.

Example

man date (display the usage and syntax of date command)

## Creating Directories

## A directory is a collection of files. It can be thought of as a “folder” that contains many different files. Directories are given names, with which you can identify them. Furthermore, directories are maintained in a tree-like structure; that is, directories may contain other directories.

* **mkdir**

Use mkdir command to create new directories and thereby organize files in a filing cabanet. This is an incredibly simple command. You specify the name of the directory to create.

Example:

**$ mkdir lab1**

**$ ls ----** Command for listing

## Deleting Directories

Directories are deleted with **rmdir** command. This command deletes only empty directories. If the directory contains any file or directory, first that file or directory must be deleted.

* **rmdir** <Directory Name>
* **cd** (Change Directory)

Use the cd command to open your new created Directory.

Example:

**$ cd ssuet**

**/ssuet/ $ ls**

**/ssuet/ $**

**Note:** If you want to go one step back use **cd ..** Command you are now at your previous location.

* **pwd**

You need a way to determine your current directory location. The pwd command, for print working directory, can do this for you.

Example:

**$ pwd**

**/ssuet /Labs**

Lab Task:

1. Create multiple directories and subdirectories.
2. Search different options of ***rmdir*** command using man command.

# LAB 2

**Basic Linux Commands**

OBJECTIVE

To use basic Linux commands related to file creation, editing and permissions

## File Naming Rules

1. A file name can be a combination of letters, numbers and special characters.
2. All letters, both upper (A-Z) and lower case (a-z) can be used.
3. Numbers from 0 – 9 can be used.
4. Special characters like plus (+), minus (-), underscore (\_), or dot (.) can be used.
5. LINUX is case sensitive and uppercase and lowercase letters are treated separately. So myfile, Myfile, MyFile, and myFilE are different names.

### Creating a File with *cat* command

The cat command is a basic command used for creating new files containing text.

#### Example:

$ Cat > newfile

this is first line. <Enter>

this is the second line. <Enter> this is third and last line. <Enter>

<CTRL+d> or <CTRL+Z>

### Listing Files

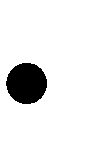
Now that you have created a file, you can verify it by listing it using the *ls* command.

#### $ ls

**newfile abcd.c 12bdfg.c history.c**

or simple, **$ ls newfile newfile**

**The ‘*ls’* Command and options**

 ***ls***

***ls*** *command* shows the contents of current directory.

#### $ ls

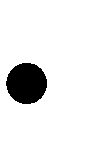
**newfile abcd.c 12bdfg.c history.c**

or simple, **$ ls newfile newfile**

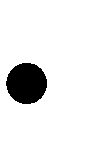
* ***ls –l***

To list all information about the contents of directory you would use the –l option to ls as shown in the following example.

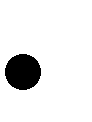
-rwxr-x--x 1 denie users 99 dec 6 09:18 story

 *ls -a*

To list all of the entries (Hidden Files) of a directory you would use the –a option. Files that begin with a “.” and “..” are called Parent and Childs simultaneously

 *ls –p, ls -F*

Use the *-p* option to *ls,* which puts a “/” in after directory names.

 *ls -R*

The -R option to ls shown in the following example will perform this recursive listing.

### Example:

#### $ ls -R

.elm

.exrc clean 2456abcd

### Displaying the contents of a file

To see the contents of a file use the cat command

#### $ cat newfile

This is first line.

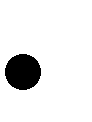
This is the second line. This is third and last line.

### Redirecting Standard Output

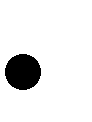
The Redirection of stdout is controlled by “>” the greater-than symbol.

$ cat newfile > newfile2

### Joining Two or More Files

 Two or more files can be joined into a single file by the use of the *cat* command and redirecting output to a file.

##### $ cat file1 file2 file3 > file4

 The following command creates a new file, containing all files in the directory.

##### $ cat \* > file5

**Appending to a File**

Use the double redirection symbol “>>” to keep the previous contents of a file.

##### $ cat file1 >> file2

### Redirecting Standard Error

Use “2>” for stderr redirection to tell the shell that we want to redirect the error messages instead of stdout (file descriptor value is 0). Consider the following commands;

##### $ ls xyz 2> abc

##### $ cat abc

##### Standard File Descriptors

|  |  |
| --- | --- |
| ***File Descriptor Number*** | ***Description*** |
| *0* | *Standard Input* |
| *1* | *Standard Output* |
| *2* | *Standard Error* |

**File Name Expansion and wild cards**

An overview of file name expansion is useful to insure you’re comfortable with this topic.

1. (\*) To list all files in a directory that end in **“.c”**, do the following:

#### $ ls \*.c

**conf.c SAM.c newfile.c**

1. (?) To find all of the files in a directory named “conf” with an extension of one character, you could do the following:

#### $ ls conf.?

**conf.c conf.o conf.1**

1. [list] To list all of the files in a directory named “conf” with only the extension “c” or “o”, do the following:

#### $ ls conf.{co}

**conf.c conf.o**

1. str{str1,str2,str3,….} To list files that start with “ux”, and have the extension “300” or “700”, do the following:

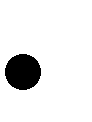
#### $ ls ux\*.{700,300}

**uxbootlf.700 uxinstfs.300 unistker.300 unistker.700**

1. (~) To list the files in home directory you could use **~** :

#### $ ls -a ~

|  |  |  |
| --- | --- | --- |
| **.** | **.cshrc.org** | **.login .shrc.org** |
| **..** | **.exrc** | **.abcd .defg.c** |
| **.chrcs** | **.history** | **.profile** |

 To see the contents of the directory /abc without opening it do the following command:

#### $ ls /abc

images pictures games …..

## “Who” and “whoami”

use the *whoami* command to identify who is logged in.

##### $ whoami

username

The who command, tells login names of all users logged into the system, along with their time and the terminal line they are using.

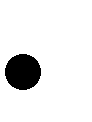
##### $ who

*operator pts/ta Jan 23 14:56*

## Some Magical Differences

The following command creates a file, containing all list of files and directories. First list your current directory with *ls* command. Then enter this;

##### Ls > filename

 Consider the *who* command. We redirected its output to a file with the name

##### whofile.

1. ***who > whofile***

**Sorting**

***$ cat unsorted***

This is number 3

This is number 5

This is number 1

Use the *sort* command to arrange(sort) these lines. Use the sort command with input redirection to this file, result will be appeared.

##### $ Sort < unsorted

This is number 1

This is number 3

This is number 5

**Searching**

We can find a particular word with *grep* command like;

**$ grep abc <*filename*>**

For multiple words use it like;

**$ grep “sir syed” <*filename*>**

Note: For case-insensitive use *–i* with grep command.

## Copying and Moving Files

## Copying Files

The files are copied with *cp* command.

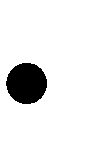
#### *$ cp file1 file2*

## Moving and Renaming Files

The *mv* command is used for renaming files and moving files from one place to another in the directory structure.

### For Renaming a File;

#### *$ mv myfile newfile*

 For Moving a File;

**$ mv file1 ./lab**

## Miscellaneous File Handling Commands

**The Head and Tail**

*Head* command lists the first ten lines of a text file, and the *tail* command lists the last ten lines of a file.

**$ head -2 <*filename*>**

**$ tail -5 <*filename*>**

## Counting Characters, Words, and Lines in a Text File

Use *wc* command for *word count*.

**$ wc <*filename*>**

It displays the number of lines, words, and characters, respectively. Use *wc – c* for characters, *wc –l* for lines, and *wc – w* for words.

## Command Aliases

Aliases can also be used to remember commands by assigning them more meaningful names.

##### $ alias dir=ls

**Disk Usage by Directory**

The *du* command shows a breakdown of disk usage by directory. For example, *cd* to your home directory and type "*du -k*" (The *-k* option displays output in Kilobytes)

**$ cd ~**

**$ du –k**

1 ./News

3 ./mail

713 ./public\_html/photo

1363 .

The output lists the number of Kilobytes used by each subdirectory. In this example, the user is using a total of 1363 Kilobytes or about 1.5 Megabytes of disk space.

The *-s* option causes *du* to work in summary mode.

#### $ du -k -s

1363.

## Lab task

1. Create multiple files and perform all commands.
2. Remove non empty directory.
3. date (for Current Date).

#### ps

1. finger
2. cal (For Calender).

#### free

1. Create a file name *<yearfile>* which displays the contents of current year.
2. Display calendar and date simultaneously.
3. df ./
4. df –k ./
5. du –s \*

## Home Assignment

1. Using ls – List the contents of a directory shown, from these contents explain any ten switches with example.
2. Write in detail about “*w”* command and also the difference between *who* and *w*

command.

1. Write the difference of the following: >, <, >>.

Note: Write in your hand writing.

LAB 3

Process Creation using fork() system call

OBJECTIVE

To use gcc compiler and demonstrate process creation using fork() system call

THEORY

First compile the above program using command ‘gcc program name.cc’ , then execute it through the command ‘./a.out &’. The & will make it a background process, which is terminated only if the process ends, or is interrupted with a DEL key. While the said program is running in the background, open a file with any editor and type something, save it and execute it. For reference consider Program 1.

**Process Identification**

There is a function – *getpid( ) –* that enables us to get the identification number of a process. For reference consider Program 2 and Program 3.

The *pid* value will depend on the number of processes already running. It will always be unique. This *pid*  cannot be changed although it may be reused once the process no longer exists.

Run the program 3 twice. Both times as a background process, i.e suffix it with an ampersand (&). Once both are running as a background processes key the ps-a command and you will see that memory contains these two processes.

**Parent And Child**

A process in UNIX is not a standalone. In the same way as a human being or animal is born from another, a process to has to come from the womb of another process. This results in a parent-child relationship existing between processes. For reference consider Program 4.

Program 4 will give the ID number of its parents.

When we boot the system, a special process called the swapper or scheduler is created with a PID of 0. The swapper manages memory allocation for processes and influences CPU allocation. The swapper in turn creates three: the process dispatcher, vhand, and bdflush with ID numbers 1,2 and 3 respectively.

This is done by executing the file init which exists in the etc sub-directory. The process dispatcher now gives birth to the shell. From now on all processes initiated by us are children of the shell and in turn descendents of the process dispatcher. This gives rise to a tree-like structure, with ADAM as the swapper.

UNIX keeps track of all processes in an internal data structure called the process table. A listing of the process table can be got using the ps-el command.

**The 'fork( )'**

Processes initiated by us can also create children in the same manner as the swapper and the process dispatcher did. These children processes are created using the fork ( ) function. It is by forking processes that we can exploit the multitasking capability of UNIX. For reference consider Program 5.

The child process begins from the fork( ). All the statements after the call to fork( ) will be executed twice. Once by the parent process and once by the child process. But had there been any statements before the fork( ) they would have been only executed by the parent process. For reference consider Program 6.

# Example programs

## Program 1

main()

{

int pid;

pid = getpid( );

printf(“Process ID is %d \n”,pid);

}

## Program 2

main()

{

int ppid;

ppid = getppid( );

printf(“Parent Process ID is %d\n”,ppid);

}

## Program 3

main()

{

fork( );

printf(“Hello World\n”);

}

## Program 4

main()

{

int pid;

pid = fork( );

if(pid>0)

printf(“Parent process PID is %d\n”,pid);

}

## Program 5

main()

{

int pid

pid = fork( );

if (pid == 0)

printf(“Child Process\n”);

}

## Program 6

## main( )

{

int pid;

pid=fork( );

if(pid==0)

{

printf(“I am the child, my process ID is %d\n”,getpid( ));

printf(The child’s parent process ID is %d\n”,getppid( ));

}

else

{

printf(“I am the parent, my process ID is %d\n”,getpid());

printf(“The parents parent process ID is %d\n,getppid( ));

}

}

# Exercise

1. Write few lines about given Program and what is the output of same program?

#include<stdio.h>

main(int arc,char\*ar[])

{

int pid; char s[100];

pid=fork();

if(pid<0)

printf("error");

else if(pid>0)

{

wait(NULL);

printf("\n Parent Process:\n");

printf("\n\tParent Process id:%d\t\n",getpid());

execlp("cat","cat",ar[1],(char\*)0);

error("can’t execute cat %s,",ar[1]);

}

else

{

printf("\nChild process:");

printf("\n\tChildprocess parent id:\t %d",getppid());

sprintf(s,"\n\tChild process id :\t%d",getpid());

write(1,s,strlen(s));

printf(" ");

printf(" ");

printf(" ");

execvp(ar[2],&ar[2]);

error("can’t execute %s",ar[2]);

}

}

LAB # 4

**Program execution using Exec System Calls**

**OBJECTIVE**

To demonstrate program execution in Linux using various exec() system calls

**THEORY**

**The 'exec( )'**

Forking provides a way for an existing process to start a new one, but what about the case where the new process is not part of the same program as parent process? This is the case in the shell; when a user starts a command it needs to run in a new process, but it is unrelated to the shell.

This is where the exec system call comes into play. exec will replace the contents of the currently running process with the information from a program binary. Thus the process the shell follows when launching a new program is to firstly fork, creating a new process, and then exec (i.e. load into memory and execute) the program binary it is supposed to run.

**The versions of exec are:**

|  |  |
| --- | --- |
| execl | (const char \*path, const char \*arg, ...); |
| execlp | (const char \*file, const char \*arg, ...); |
| execle | (const char \*path, const char \*arg , ..., char \* const envp[]); |
| execv | (const char \*path, char \*const argv[]); |
| execvp | (const char \*file, char \*const argv[]); |

**Example programs**

**Program 1**

main ( ) {

printf(“Before exec my ID is %d \n”,getpid( ));

printf(“My parent process’s id is %d\n”,getppid( ));

printf(“exec starts\n”);

execl(“/home/guest/ex2”,”ex2”,(char\*)0);

printf(“this will not print\n”);

}

**Program 2**

main( ) {

printf(“After the exec my process id is %d\n”,getpid( ));

printf(“My parent process’s id is %d\n”,getppid( ));

printf(“exec ends\n”);}

**Program 3**

main()

{

pid\_tpid;

pid=fork();

if(pid<0)

{

fprintf(stderr,”Fork Failed”);

return 1; }

else if(pid==0){

execlp(“/bin/ls”,”ls”,NULL);

}

else{

wait(NULL);

printf(“Child Complete”);

}

return 0;

}

**Exercise**

1. Describe all the versions of exec system call.
2. Write a program that will call another program from child process.

LAB # 5

InTER PROCESS cOMMUNICATION using signal

OBJECTIVE

To show Inter-process communication using signals

# THEORY

Communication takes place if there is something to share. Ideas, thoughts, feelings, beliefs, culture and just about every other facet that goes to make a human being. Through computers so far we have shared data – data linked to business, science and now as satellite communications make their mark news, views and what have you.

Processes in computer memory are said to be communicating when one process passes data to another or vice-versa. And for these processes in memory to communicate with each other they need to be able to share data effectively. UNIX provides a number of ways to share or communicate data: signals and pipes.

**Signals**

Processes can send each other signals. For example say process A has finished processing some data, it may send signal to process B. Process B will also receive the output of the processed data from process A. Based on this output it has to decide what response to give.

There are many different types of signals, each with some mnemonic name: SIGINT, for example. These names, which are defined in the signal.h header file are symbolic for some positive integer.

Suppose we run a process that takes inordinately long, thus leading us to believe that something is wrong. To terminate it, we either press the DEL or CTRL \ keys. This result in brought back to the shell.

What happens is that the part of the kernel looking after keyboard interrupts realizes that the DEL has been pressed. The kernel then sends a signal called SIGINT to all processes associated with the terminal. And they terminate. For reference consider Program 1.

Compile and run Program 1. Then press the DEL key. The program terminates because the pressing of the DEL key forces the kernel to send a SIGINT signal which is interpreted as “terminate a program”.

**About ‘SIGALRM’**

The alarm( ) function sets up a process alarm clock. Signals are used to tell the process that the clock’s timer has expired. After the call, the process carries on. Whenever we want to place a time limit on some activity we can use the alarm system call. If the process takes longer than the alarm set, it is interrupted by the SIGALRM signal and the process terminates. For reference consider Program 2.

If the above Program 1 is run in the foreground, 5seconds later a message Alarm Call is flashed on screen.

The sleep( ) function which we use so often is written using alarm( ) and one more function pause( ). The pause( ) basically waits for a signal to be sent before it terminates. And the alarm( ) waits for the time period specified to elapse. The moment the time period elapses, a signal, the SIGALRM, is sent. This terminates the pause( ). For reference Program 3.

# Example programs

## Program 1

#include<signal.h>

main( )

{

printf(“Use Ctrl-Z key for exiting\n”);

for(;;);

}

## Program 2

main( )

{

alarm(5);

for(;;);

}

## Program 3

#include<signal.h>

main( )

{

void abc( );

signal(SIGALRM,abc);

alarm(3\*6);

pause( );

}

void abc( )

{

printf(“Time to ring the Sec.Gen. my boy \n”);

}

# Exercise

1. What did you learn after running Program 3? Write few lines.
2. Write a program which creates a child process and then wait for the child to terminate. Send SIGCHLD signal from child to parent and show that parent resumes.

LAB #6

Multithreading using pthread Library

OBJECTIVE

To apply multithreading in Linux using pthread library

**EXAMPLES**

**Program 1**

#include<stdio.h>

#include<pthread.h>

#include<stdlib.h>

int sum;

void \*runner(void \*param);

main(int argc , char \*argv[])

{

pthread\_t tid;

pthread\_attr\_t attr;

if(argc !=2)

{

printf(“usage :a.out <integer value> \n”);

exit(0);

}

if(atoi (argv[1])<0)

{

printf(“%d must be >=0\n” , atoi(argv[1]));

exit(0);

}

pthread\_attr\_init(&attr);

pthread\_create(&tid,&attr,runner,argv[1]);

pthread\_join(tid,NULL);

printf(“sum=%d\n},sum);

}

void \*runner(void \*param)

{

int upper=atoi(param);

int i;

sum = 0;

if(upper>0)

{

for(i=1;i<upper;i++)

{

sum+=i;}

pthread\_exit(0);}}

**COMPILE AND RUN**

gcc -pthread thread.c -o a1

./a1

**Exercise**

1. Write few lines about what you learn after running the above Program and show the output.

# LAB 7

Open Ended Lab

OBJECTIVE

INTRODUCTION

CODE

OUTPUT

CONCLUSION

# LAB #8

CPU Scheduling Algorithms

OBJECTIVE

To demonstrate the working of FCFS and SFJ CPU scheduling algorithms

## Introduction

Scheduling is the task of operating system in which processes are given access to system resources. Scheduling is a fundamental operating system function that determines which process run, when there are multiple processes in ready queue. CPU scheduling is important because it impacts resource utilization and other performance parameters.

## Scheduling Criteria

Different CPU scheduling algorithms have different properties and may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the different properties of the various algorithms.

***CPU Utilization*:** We want to keep the CPU as busy as possible. CPU utilization may range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).

***Throughput:*** If the CPU is busy executing process, then work is being done. One calculate of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, throughput might be 10 processes per second.

***Turnaround time:***From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

***Waiting Time:***The CPU scheduling algorithm does not affect the amount of time during which a procedure executes or does I/O; it affects only the amount of time that a process spends waiting in the prepared queue. Waiting time is the sum of the periods spent waiting in the ready queue.

*Response time:* In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly easy, and can continue computing new results while earlier results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time that it takes to start responding, but not the time that it takes to output that response. The turnaround time is generally incomplete by the speed of output device.

## Scheduling Algorithms

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms which we are implementing in this lab:

1. First-Come, First-Served (FCFS) Scheduling
2. Shortest-Job-First (SJF) Scheduling
3. Priority Scheduling
4. Round Robin(RR) Scheduling

### First Come First Serve (FCFS) Algorithm:

1. Start the process.
2. Declare the array size.
3. Get the number of elements to be inserted.
4. Select the process that first arrived in the ready queue
5. Make the average waiting the length of next process.
6. Start with the first process from its selection as above and let other process to be in queue.
7. Calculate the total number of burst time.
8. Display the values.
9. Stop the process.

### Shortest Job First (SJF) Algorithm:

1. Start the process.
2. Declare the array size.
3. Get the number of elements to be inserted.
4. Select the process which have shortest burst will execute first.
5. If two process have same burst length, then FCFS scheduling algorithm used.
6. Make the average waiting the length of next process.
7. Start with the first process from its selection as above and let other process to be in queue.
8. Calculate the total number of burst time.
9. Display the values.
10. Stop the process.

## Lab task

1. Perform the implementation and show the output of FCFS and SJF(Non-Preemptive), algorithms.

## Home Assignment

1. Write the algorithm of SJF (Preemptive) algorithm and also implement it and show the output.

# LAB 9

CPU Scheduling Algorithms

OBJECTIVE

To demonstrate the working of Round Robin and Priority CPU scheduling algorithms

## Scheduling Algorithms

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms which we are implementing in this lab:

1. First-Come, First-Served (FCFS) Scheduling
2. Shortest-Job-First (SJF) Scheduling
3. Priority Scheduling
4. Round Robin(RR) Scheduling

### Priority Scheduling Algorithm:

1. Start the process.
2. Declare the array size.
3. Get the number of elements to be inserted.
4. Get the priority for each process and value
5. start with the higher priority process from it’s initial position let other process to be queue.
6. calculate the total number of burst time.
7. Display the values
8. Stop the process.

### Round Robin (RR) Algorithm:

1. Start the process.
2. Declare the array size.
3. Get the number of elements to be inserted.
4. Get the value.
5. Set the time sharing system with preemption.
6. Define quantum is defined from 10 to 100ms.
7. Declare the queue as a circular.
8. Make the CPU scheduler goes around the ready queue allocating CPU to each process for the time interval specified.
9. Make the CPU scheduler picks the first process and sets time to interrupt after quantum expired dispatches the process.
10. If the process has burst less than the time quantum than the process releases the CPU.

## Lab task

1. Perform the implementation and show the output Priority(Non-Preemptive) and Round Robin algorithms.

## Home Assignment

1. Write the algorithm of Priority (Preemptive) algorithm and also implement it and show the output.

LAB 10

NAMED PIPES

OBJECTIVE

To use named pipes to communicate between processes

# THEORY

While pipes are powerful inter-process communication they are not without drawbacks.

Firstly, pipes can only be used on processes that have a common ancestry like a parent and child process. And, secondly, they are not permanent. A process creates them and the termination of the process leads to their destruction.

To overcome these deficiencies a variation on these pipes has been implemented known as NAMED PIPES. These named pipes basically function in the same way as pipes: acting as one way communication channels. But unlike pipes, a named pipe is a permanent fixture. UNIX treats it just likes a file, giving it a size, owner and access permission. It can be opened, closed or deleted like any other file.

Now let’s see how we can use this named pipe.

Do a cat < testpipe &. This results in the cat program being made into a back ground process. Why we are making it a background process? Well, had we not, the cat would have kept waiting to read something. But since the pipe is empty, the program would hang. By making it a background process we ensure that the program does not hang (try it).

But this process is unable to read from the named pipe testpipe. Why? Because in the first place this pipe has not even been opened. A named pipe can only be opened if there is someone waiting at both its ends. In this case cat tries to read from it, but there is no corresponding process trying to write to it. As a result pipe is not opened at all.

So what we need here is some way of writing to the pipe. And that is what a ls –l>testpipe at the prompt, will do. The moment we give this command at the prompt, the pipe is opened in the background process for a read and in this (the ls) process for a write. The output of the ls –l is now outputted to the pipe instead of to the screen. But since, at the read end of the pipe, a cat is being performed we see the output on the screen.

UNIX, we know, is based on a client-server relationship. On the server machine we have the OS and all other files. This is the main machine on which all processing is done. All other machines connected to this main machine are known as nodes or clients. They are merely a screen and keyboard connection. All commands or programs specified at these nodes are actually run on the server.

Probably one of the most common uses of named pipes is for communicating between a client and server. We keep a process that has opened a named pipe running in the background. This process waits for some other process to write to the named pipe. Once something is written, the message is printed on screen.

Basically, there will be two programs that run hand in hand. One that waits for something to be written to a name pipe and one that can be used to write something to the named pipe. Of course, the write program has to be aware of named pipe it can write to. And the program that keeps the named pipe ready and waiting to receive information has to be active. The read program will have to be run on the server while the write program can be run by any client. This is what client-server communication is all about. For reference consider Program 1.

In Program 1, we opened the same named pipe we had created at the command line. The file is being opened in the read write mode. In the never ending for loop, a read( ) is activated. The moment a message is read from the pipe it is printed by the printf( ) function.

(Run the Program 1 and verify it what it is doing)

Now we need a program that will write messages to the pipe. For reference consider Program 2.

Program 2, when it is run, has to be passed a message or messages. It is this message that will be read and printed by the receiving program.

(Run the Program 2 and verify it what it is doing)

# Example programs

## Program 1

#include<fcntl.h>

#include<stdio.h>

#define MSGSIZ 63

main( )

{

int fd;

char msgbuf[MSGSIZ+1];

if((fd=open(“testfile”,O\_RDWR))<0)

perror(“pipe open failed”);

for(;;)

{

if(read(fd,msgbuf,MSGSIZ+1)>0)

printf(“message received:%s \n”,msgbuf);

}

}

## Program 2

#include<fcntl.h>

#include<stdio.h>

#include<errno.h>

#define MSGSIZ 63

main(argc,argv)

int argc;

char \*argv[ ];

{

int fd,j,nwrite;

char msgbuf[MSGSIZ];

if(argc<2)

{

printf(“Usage:<filename><message><message>…\n”);

}

if((fd=open(“testfile”,O\_WRONLY))<0)

perror(“fifo open failed”);

for(j=1;j<argc;j++)

{

strcpy(msgbuf,argv[j]);

if((nwrite=write(fd,msgbuf,MSGSIZ+1))<=0)

perror(“message write failed”);

}

}

## Program 3

#include<fcntl.h>

#include<stdio.h>

#define MSGSIZ 63

main( )

{

int fd;

char msgbuf[MSGSIZ+1];

if(mknod(“myfifo”,010666,0)<0)

perror(“myfifo failed”);

if((fd=open(“myfifo”,O\_RDWR))<0)

perror(“fifo open failed”);

for(;;)

{

if(read(fd,msgbuf,MSGSIZ+1)>0)

printf(“message received:%s \n”,msgbuf);

}

}

# Exercises

1. What did you learn after running the above Program 1 and Program 2?
2. What did you learn after running the Program 3?

LAB 11

INTERPROCESS COMMUNICATION

OBJECTIVE

To use pipes and implement Inter Process communication

# THEORY

**pipe**() creates a pipe, a unidirectional data channel that can be used for inter-process communication.  The array *pipefd* is used to return two file descriptors referring to the ends of the pipe.  *pipefd[0]* refers to the read end of the pipe.  *pipefd[1]* refers to the write end of the pipe.  Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe. On success, zero is returned.  On error, -1 is returned, and [*errno*](http://man7.org/linux/man-pages/man3/errno.3.html) is set

**Program 1**

#include<stdio.h>

#include<unistd.h>

#include<sys/ipc.h>

#include<sys/uio.h>

#include<sys/types.h>

#include<fcntl.h>

main()

{

int pid,pfd[2],n,a,b,c;

if(pipe(pfd)==-1)

{

printf("\nError in pipe connection\n");

exit(1);

}

pid=fork();

if(pid>0)

{

printf("\nParent Process");

printf("\n\n\tFibonacci Series");

printf("\nEnter the limit for the series:");

scanf("%d",&n);

close(pfd[0]);

write(pfd[1],&n,sizeof(n));

close(pfd[1]);

exit(0);

}

else

{

close(pfd[1]);

read(pfd[0],&n,sizeof(n));

printf("\nChild Process");

a=0;

b=1;

close(pfd[0]);

printf("\nFibonacci Series is:");

printf("\n\n%d\n%d",a,b);

while(n>2)

{

c=a+b;

printf("\n%d",c);

a=b;

b=c;

n--;

}

}

}

# Exercise

1. What did you learn after running the Program 1?

LAB 11

SEMAPHORES

# OBJECTIVE

To use semaphore to implement process synchronization

# THEORY

Assume that two processes were accessing the same file. The one that got the time slice first would initialize a variable before working on the file. When the other process is given the time slice it would first check the value of this variable realize it has been initialized by some other process and refrain from accessing the file. When the first process finished it would reinitialize this variable. As a result the second process which has been waiting for just this to happen will get access to the file.

This variable in UNIX terms is known as semaphore and is basically an integer which acts as a counter. Its value depends on the number of resources there are to share. For example if we had one file which is to be shared then the semaphore can have a value 0 or 1. A 0 initialized semaphore signifies that the resource in this case the file is in use and therefore all other processes would have to wait. The moment the process that has access to the file finishes it sets this semaphore value to 1. Thereby allowing one other process access.

**Creating a Semaphore**

Now that we have a basic idea of semaphores and why they are used lets get to the hows, i.e. how to create and implement them.

To start with do a ipcs –s at the prompt. This will tell us if there are any semaphores existing. Details about any semaphore will be under the following column headings – key, semid, owner, perms, nsms and status. For reference consider Program 1 and Program 2.

Run the Program 1 that creates a semaphore:

semget( ) returns an value which is the identifier of the semaphore. If this value is a –1 it means an error.

And that’s what this program returned. So where’s the problem? Why because we have not specified how many semaphores we want.

UNIX does not just create one semaphore, what it allows us instead is the ability to create a set of semaphores. And in this set there should be a minimum of at least 1 sub-semaphore. The second parameter we pass the semget( ) function defines how many sub-semaphores there will be in this set of semaphores. Unfortunately we have assigned a 0 to this variable nsem which is below the minimum required. As a result an error is returned.

Do a ipcs –s to verify that this semaphore has not been created.

**Who is using the Resource**

A semaphore is basically used by different processes to synchronize access to a resource. Of course what value the semaphore should have to determine access or not has to be specified by us. For example may be we have designed our system in such a way that when a semaphore is deciphered as having a value of 1 it means that a certain resource is in use by another process. And a value of 2 means that a resource is free and that our process can now access to it.

Using the semctl( ) function we can find out which process has set the value of semaphore. Avalue of GETPID passed to the semctl( ) will result in it passing the PID of the process that has set the value of the semaphore. For reference consider Program 3 and Program 4.

Note: kill the semaphore first and then run Program 3 twice.

# Example programs

## Program 1

#include<sys/types.h>

#include<sys/ipc.h>

main( )

{

int semid,key,nsem;

key=(key\_t) 0\*20;

nsem=0;

semid=semget(key,nsem,IPC\_CREAT | 0666);

printf(“Created semaphore with ID:%d \n”,semid);

}

## Program 2

#include<sys/types.h>

#include<sys/ipc.h>

main ( )

{

int semid,key,flag,nsem;

key=(key\_t) 0\*20;

flag=IPC\_CREAT | 0666;

nsem=1;

semid=semget(key,nsem,flag);

printf(“Created Semaphore with ID:%d \n”,semid);

}

## Program 3

#include <sys/types.h>

#include<sys/ipc.h>

#include<sys/sem.h>

#include<errno.h>

main( )

{

int semid,retval;

semid=semget(0\*20,1,0666 | IPC\_CREAT);

retval = semctl(semid,0,GETPID,0);

printf(“PID returned by semctl is %d and actual PID is %d\n”, retval,getpid( ));

}

## Program 4

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/sem.h>

#include<errno.h>

main( )

{

int semid,retval;

semid=semget(0\*20,1,0666| IPC\_CREAT);

retval=semctl(semid,0,GETPID,0);

printf(“PID returned by semctl is %d and actual PID is %d\n”,retval,getpid( ));

retval=semctl(semid,0,SETVAL,1);

printf(“PID returned by semctl is %d and set value is %d\n”,retval,SETVAL);

}

# Exercises

1. What was the mistake in the Program 1, and how it is rectified in Program 2?
2. Do ipcs –s at the prompt to see a listing of the semaphores and an entry for the semaphore will be displayed.
3. Run the Programs 3 and 4, and find out what is the problem in 3 and how it is rectified in 4.

LAB 12

OPERATIONS ON FILES

# OBJECTIVE

To study and use file operations in Linux

# THEORY

To start with, let’s see how running two processes which have access to the same file could result in a mess up of data. Run Program 1 and Program 2 one after the other, both times in the background. To run both the following programs do this at the prompt:

$<first>&

$<second>&

Once both these processes are in memory the time slice will be alternated between them. As a result, since both processes have read write access to the same file, the data they will write be all jumbled up.

(Run the Program 2 and verify the result)

# Example programs

## Program 1

#include<fcntl.h>

#include<unistd.h>

main( )

{

int fd, i;

fd = open(“locktest”,O\_APPEND | O\_CREAT| O\_RDWR,0777);

for(i=0;i<=2000;i++)

write(fd, “A”,1);

}

## Program 2

#include<fcntl.h>

#include<unistd.h>

main( )

{

int fd,i;

fd = open(“locktest”,O\_APPEND | O\_CREAT | O\_RDWR,0777);

for(i=0 ;i<=2000;i++)

write(fd, “B”,1);

}

## Program 3

#include<fcntl.h>

#include<unistd.h>

main( )

{

int fd,i;

fd = open(“locktest”,O\_APPEND | O\_CREAT | O\_RDWR,0777);

lockf(fd,F\_LOCK,0);

for(i=0;i<=2000++)

write(fd, “A”,1);

}

## Program 4

#include<fcntl.h>

#include<unistd.h>

main( )

{

int fd,i;

fd = open(“locktest”,O\_APPEND | O\_CREAT | O\_RDWR,0777);

lockf(fd,F\_LOCK,0);

for(i=0;i<=2000++)

write(fd, “B”,1);

}

# Exercises

1. Write few lines about what did you learn after running Programs 3 and Program 4.

LAB 13

Linux Shell Scripting

OBJECTIVE

To demonstrate the use of different control structures in Shell scripting

The **Vi** Editor

A **text editor** is a program used to edit files that are composed of text: a letter, C program, or a system configuration file. While there are many such editors available for Linux, the only editor that you are guaranteed to find on any UNIX or Linux system is **vi-**- the “***visual editor”.***Vi is not the easiest editor to use, nor is it very self-explanatory. However, because vi is so common in the UNIX/Linux world, and sometimes necessary, it deserves discussion here.

    Your choice of an editor is mostly a question of personal taste and style. Many users prefer the baroque, self-explanatory and powerful emacs--an editor with more features than any other single program in the UNIX world. For example, Emacs has its own built-in dialect of the LISP programming language, and has many extensions (one of which is an Eliza-like artificial intelligence program). However, because Emacs and its support files are relatively large, it may not be installed on some systems. vi, on the other hand, is small and powerful but more difficult to use. However, once you know your way around vi, it's actually very easy.

## *Concepts*

While using vi, at any one time you are in one of three modes of operation. These modes are called **command mode**, **insert mode**, and **last line mode**.

***Command mode*** − This mode enables to perform administrative tasks such as saving

files, executing commands, moving the cursor, cutting *yanking* and pasting lines or words, and finding and replacing.

***Insert mode*** − This mode enables to insert text into the file. Everything that's typed in

this mode is interpreted as input and finally it is put in the file The vi always starts in command mode.

**Last line mode** is a special mode used to give certain extended commands to vi. For example, when you type **“:”** in command mode, user jumps into last line mode

Starting the “vi” Editor

The syntax for vi is:

|  |  |  |
| --- | --- | --- |
| **\*** | **vi filename** | *edit filename starting at line 1* |
|  | **vi -r filename** | *recover filename that was being edited when system crashed* |

The column of **“~”** characters indicates you are at the end of the file. The current cursor position.

* Inserting text.

The following commands allows to insert and add text. Each of these commands puts the vi editor into insert mode; thus, the <Esc> key must be pressed to terminate the entry of text and to put the vi editor back into command mode.

|  |  |  |
| --- | --- | --- |
| **\*** | **i** | *insert text before cursor, until <Esc> hit* |
|  | **I** | *insert text at beginning of current line, until <Esc> hit* |
| **\*** | **a** | *append text after cursor, until <Esc> hit* |
|  | **A** | *append text to end of current line, until <Esc> hit* |
| **\*** | **o** | *open and put text in a new line below current line, until <Esc> hit* |
| **\*** | **O** | *open and put text in a new line above current line, until <Esc> hit* |

* Deleting text.

The following commands allows to delete text.

|  |  |  |
| --- | --- | --- |
| **\*** | **x** | *delete single character under cursor* |
|  | **Nx** | *delete N characters, starting with character under cursor* |
|  | **dw** | *delete the single word beginning with character under cursor* |
|  | **dNw** | *delete N words beginning with character under cursor;    e.g., d5w deletes 5 words* |
|  | **D** | *delete the remainder of the line, starting with current cursor position* |
| **\*** | **dd** | *delete entire current line* |
|  | **Ndd *or* dNd** | *delete N lines, beginning with the current line;    e.g., 5dd deletes 5 lines* |

* Changing text.

The following commands allows to modify text.

|  |  |  |
| --- | --- | --- |
| **\*** | **r** | *replace single character under cursor (no <Esc> needed)* |
|  | **R** | *replace characters, starting with current cursor position, until <Esc> hit* |
|  | **cw** | *change the current word with new text,  starting with the character under cursor, until <Esc> hit* |
|  | **cNw** | *change N words beginning with character under cursor, until <Esc> hit;    e.g., c5w changes 5 words* |
|  | **C** | *change (replace) the characters in the current line, until <Esc> hit* |
|  | **cc** | *change (replace) the entire current line, stopping when <Esc> is hit* |
|  | **Ncc *or* cNc** | *change (replace) the next N lines, starting with the current line, stopping when <Esc> is hit* |

* Commands for moving the cursor.

The mouse does not move the cursor within the vi editor screen (or window). In the table below, the symbol ^ before a letter means that the <Ctrl> key should be held down while the letter key is pressed.

|  |  |  |
| --- | --- | --- |
| **\*** | **j *or* <Return>    [*or* down-arrow]** | *move cursor down one line* |
| **\*** | **k [*or* up-arrow]** | *move cursor up one line* |
| **\*** | **h *or* <Backspace>    [*or* left-arrow]** | *move cursor left one character* |
| **\*** | **l *or* <Space>    [*or* right-arrow]** | *move cursor right one character* |
| **\*** | **0 (zero)** | *move cursor to start of current line (the one with the cursor)* |
| **\*** | **$** | *move cursor to end of current line* |
|  | **w** | *move cursor to beginning of next word* |
|  | **b** | *move cursor back to beginning of preceding word* |
|  | **:0<Return> *or* 1G** | *move cursor to first line in file* |
|  | **:n<Return> *or* nG** | *move cursor to line n* |
|  | **:$<Return> *or* G** | *move cursor to last line in file* |

### Screen Manipulation

The following commands allow the vi editor screen (or window) to move up or down several lines and to be refreshed.

|  |  |  |
| --- | --- | --- |
|  | **^f** | *move forward one screen* |
|  | **^b** | *move backward one screen* |
|  | **^d** | *move down (forward) one half screen* |
|  | **^u** | *move up (back) one half screen* |
|  | **^l** | *redraws the screen* |
|  | **^r** | *redraws the screen, removing deleted lines* |

* Saving files and quitting vi.

The new or modified file is saved when user leaves vi. It is also possible to quit vi without saving the file.

**Note:** The cursor moves to bottom of screen whenever a colon (:) is typed. This type of command is completed by hitting the <Return> (or <Enter>) key.

|  |  |  |
| --- | --- | --- |
| **\*** | **:x<Return>** | *quit vi, writing out modified file to file named in original invocation* |
|  | **:wq<Return>** | *quit vi, writing out modified file to file named in original invocation* |
|  | **:q<Return>** | *quit (or exit) vi* |
| **\*** | **:q!<Return>** | *quit vi even though latest changes have not been saved for this vi call* |

* Editing another file.

To edit another file, use the **:e** command.

|  |  |  |
| --- | --- | --- |
| **\*** | **:e abc** | to stop *editing* current file and *edit* the file **“abc”** instead |
| **\*** | **:e!** | returns to the last saved version on the file |

* Including other files.

Use the **:r** command, user can include the contents of another file in the current file.

|  |  |  |
| --- | --- | --- |
| **\*** | :r abc.txt | To insert a file immediately after the line containing the cursor: |

Home Assignment:

* Define different flavours of Linux Shells like (Bash, Bourne, Korn, C-Shell, etc).
* Define with some examples.
* Please write in your own hand writing. Maximum of two pages.

Linux Shell Scripting

## *Introduction to Shell Programming*

A shell script is a text file containing a list of commands to be executed by the Linux shell. Running a shell script is equivalent to typing the commands within it at the shell prompt.

Shell program is a series of Linux commands. Shell script is just like batch file in MS-DOS but have more power than the MS-DOS batch file. Useful to create our own commands that can save our lots of time and to automate some tasks.

Linux shells have sophisticated programming capabilities that make shell scripts powerful Linux tools.

## *Types of Shells*

1. **Bourne**
2. **Bash**
3. **Korn**
4. **C-Shell**
5. **Etc....**

Each different Linux shell provides different scripting capabilities.

## *Why Use the (BASH) Bourne Again Shell?*

The (BASH) Bourne Again shell supports a wide range of control structures that it suitable as a high-level programming tool. BASH is not the interactive shell for most users; there are many reasons to write BASH shell scripts.

**Portability** - The BASH shell is available on all Linux systems so your scripts will run almost anywhere. You can take them to your home or workplace or share them with other Linux users.

**Speed** - The BASH shell requires minimal system resources and shell scripts execute quickly.

**Easy to Learn** - BASH shell scripting is simpler and easier to learn than some other shells.

## *The “Echo” Command*

The ***echo*** command prints arguments followed by a newline character.

**Syntax:** ***echo*** [*arguments*]

We will use the ***echo*** command throughout the shell scripting.

**Tip:** You can also use the ***echo*** command in your terminal window

#### Example:

**$ echo Hello World**

Hello World

## *Your First Shell Script*

**Step 1: Creating a script file**

$ vi first.sh or vi first

### Step 2: Specify the shell path

The first line of a script can be used to specify which shell interprets the script. This is done with a **#!** **(Pound sign followed by an exclamation point)** followed by the absolute pathname to the shells executable name. The BASH shell's executable name is ***Bash*** and the absolute pathname on most Linux systems is ***/bin/bash***. The first line of the script should be

**#! /bin/bash**

### Step 3: Write the script.

Any text following a pound sign **(#)** is considered a comment and is ignored. Using the #! Sign as in Step 2 is an exception to this rule. For example, add the following lines to your script.

|  |
| --- |
| **#! /bin/bash** |
| **#** |
| **# My first script** |
| **#** |
| **Clear** |
| **echo "Knowledge is Power"** |

### Step 4: Make the script file executable.

Use the ***chmod*** command to make the script file executable. For example,

**$ chmod 755 first.sh**

This tells Linux that the script file should be treated as a Linux command.

### Step 6: Run the script.

To run script type;

**$ ./first.sh**

## *Variables in Linux*

Variables are one of the programming features that make shell scripting so powerful. Sometimes to process our data/information, it must be kept in computers RAM memory. RAM memory is divided into small locations, and each location had unique number called memory location/address, which is used to hold our data. Programmer can give a unique name to this memory location/address called memory variable or variable (It’s a named storage location that may take different values, but only one at a time). In Linux, 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.

Example:

**$ echo $USERNAME   
$ echo $HOME**

## *How to define User defined variables (UDV)*

To define UDV use following syntax

**Syntax:** *variablename=value*

**NOTE:** Here 'value' is assigned to given ***'variablename'*** and Value must be on right side = sign. For Example

**$ no=10**

**$ 10=no**     # Error

To define variable called **“vech”** having value **Bus**   
**$ vech=Bus**   
To define variable called n having value 10   
**$ n=20**

**Note:** Don't put spaces on either side of the equal sign when assigning value to variable. Variables are case-sensitive, just like filename in Linux.

**$ echo $no**        # will print 10   
**$ echo $vech**    # will print Bus

**$ echo $n**         # will print 20   
   
You can define **NULL** variable as follows (NULL variable is variable which has no value at the time of definition).

**$vech=   
$vech=""**

Try to print it's value **$ echo $vech** , Here nothing will be shown because variable has no value i.e. NULL variable.

*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 # remainder read as 20 mod 3 and remainder is 2   
$ expr 10 \\* 3 # Multiplication use \\* not \* since its wild card  
$ echo `expr 6 + 3`**

For the last statement not the following points

1. First, before ***expr*** keyword we used ***` (back quote)*** sign not the (single quote i.e. ') sign. Back quote is generally found on the key under tilde **(~)** on PC keyboard OR to the above of TAB key.
2. Second, ***expr*** is also end with ***`*** i.e. back quote.
3. Here expr 6 + 3 is evaluated to 9, then echo command prints 9 as sum.
4. Here if you use double quote or single quote, it will NOT work

.   
**$ echo "expr 6 + 3"**    # It will print expr 6 + 3   
**$ echo 'expr 6 + 3'** # It will work

## *More about Quotes*

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`".**

Now it will print today's date as, Today is Tue Jan ...., Note that the `date` statement uses back quote.

## *Exit Status*

By default in Linux if particular command is executed, it return two type of values which is  used to see whether command is successful or not.

1. If return **value is zero (0),** command is ***successful****.*
2. If return **value is nonzero,** command is ***not successful*** or some sort of error executing command/shell script.

This value is known as Exit Status of that command. To determine this exit Status we use **$?** Variable of shell

**Example:** 

**$ rm unknow1file**   
It will show error as follows   
rm: cannot remove `unkowm1file': No such file or directory   
and after that if you give command   **$ echo $?**   
it will print nonzero value to indicate error. Now give command   
**$ ls   
$ echo $?**   
It will print 0 to indicate command is successful.

## *The “read” Statement*

Use to get input (data from user) from keyboard and store (data) to variable.

**Syntax:** *read variable1, variable2,...variableN*

|  |
| --- |
| **$ vi  getin #! /bin/bash #Script to read your Name from key-board # echo "Enter Your First Name: " read name echo "Hello $name”** |

**Tip:** To get input in same line use ***–n*** option with ***echo*** command. Like;

**echo –n " Enter Your First Name : "**

## *The case Statement*

The ***case*** statement checks the value of a variable against a large number of possibilities. Unlike, ***case*** does not check expressions it only matches strings.

**case *value* in**

***pattern1*)**

***commands***

**;;**

***pattern2*)**

***commands***

**;;**

**.**

**.**

**esac**

Some things to note about the above syntax

* The ***case*** statement begins with the word **case** followed by a ***value*** and then the word **in**. ***Value*** is typically a script argument or other shell variable.
* The next line contains a ***pattern*** followed by a closing parenthesis. There is no opening parenthesis and there should not be a space between the pattern and parenthesis. The pattern can be any of the following.
* The word **esac** (**case** spelled backwards) ends a *case* statement.

### Example:

### #! /bin/bash # clear echo "This script tells the use of case command"

### echo "1 : Print the Date" echo "2 : Print the current working directory" echo "3 : Exit"

### echo -n "Enter your choice : "

### read reply

### case $reply in "1") date;; "2") pwd;; "3") exit;; \*)

### echo "Illegal Choice";;

### esac

**Lab Work:** Perform all programs and commands carefully.

LAB 13

Shell Control Structures and Programs

OBJECTIVE

To use control structures in shell scripting

## *Loop Control Structures*

1. *for... do... done*

Computer can repeat particular instruction again and again, until particular condition satisfies. A group of instruction that is executed repeatedly is called a loop.

A ***for* loop** sets a variable equal to every value in a list then executes commands once for each value.

**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***

**For example,**

**for *i* in 1 2 3 4 5  
do  
      echo "Welcome *$i* times"  
done**

The ***for*** loop first creates ***i*** variable and assigned a number to ***i*** from the list of number from ***1 to 5***, The shell execute echo statement for each assignment of ***i***. (This is usually know as iteration) This process will continue until all the items in the list were not finished, because of this it will repeat ***5 echo*** statements.

**For example,**

**for name in Harry Susan Bob Jane**

**do**

**echo Hello $name**

**done**

The variable is ***name***. The list is *Harry Susan Bob Jane*. The output from this *for* statement is

Hello Harry

Hello Susan

Hello Bob

Hello Jane

## *Another Method*

**#! /bin/bash  
for ((i=0 ; i<=5 ; i++))  
do  
echo "Welcome $i times"  
done**

## *Operators used in shell scripting*

|  |  |  |  |
| --- | --- | --- | --- |
| **Mathematical Operator in  Shell Script** | **Meaning** | **Normal Arithmetical/ Mathematical Statements** | **Used in Shell** |
| **-eq** | **is equal to** | **5 == 6** | **if expr [ 5 -eq 6 ]** |
| **-ne** | **is not equal to** | **5 != 6** | **if expr [ 5 -ne 6 ]** |
| **-lt** | **is less than** | **5 < 6** | **if expr [ 5 -lt 6 ]** |
| **-le** | **is less than or equal to** | **5 <= 6** | **if expr [ 5 -le 6 ]** |
| **-gt** | **is greater than** | **5 > 6** | **if expr [ 5 -gt 6 ]** |
| **-ge** | **is greater than or equal to** | **5 >= 6** | **if expr [ 5 -ge 6 ]** |

1. *while... do... done*

A ***while*** loop executes commands as long as an expression is true.

Syntax:

***while*** *[ condition ]*

***do***

*command1*

*command2*

*command3*

*..*

*....*

***done***

**For example,**

**#! /bin/bash**

**# This program generates the multiplication table using** *while* **loop**

**#**

**n=2  
i=1**

**while [ $i -le 10 ] ; do # -le = is less than or equal to**

**echo "$n \* $i = `expr $i \\* $n`"**

**i=`expr $i + 1`  
done**

Above loop can be explained as follows

|  |  |
| --- | --- |
| n=2 | Set the value of ***variable n***. (Here it's set to 2) |
| I=1 | Set ***variable i*** to 1 |
| while [ $i -le 10 ] | This is our loop condition, here if value of ***i*** is less than 10 then, shell execute all statements between do and done |
| do | Start loop |
| echo "$n  \*  $i = `expr $i  \\*  $n`" | Print multiplication table as  2 \* 1 = 2 2 \* 2 = 4 .... 2 \* 10 = 20, Here each time value of variable n is multiply be ***i.*** |
| I=`expr $i + 1` | Increment ***i*** by 1 and store result to ***i***.  ( i.e. i=i+1) |
| done | Loop stops here if ***i*** is not less than 10 i.e. condition of loop is not true. Hence loop is terminated. |

## *Control Structures*

1. *The if Statement*

*if... then.. else.... fi*

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)*

*...*

*...*

***else***

*command2 if condition is false or if exit status*

*of condition is (nonzero)*

*...*

*...*

***fi***

**Example:**

**#! /bin/bash**

**#**

**echo -n "Enter Number"**

**read a**

**if [ $a -lt 0 ] # -lt = is less than**

**then**

**echo "-ve"**

**else**

**echo "+ve"**

**fi**

*Multilevel if... then... elif... else... fi*  
**Syntax:**

***if*** *condition*

***then***

*condition is zero (true - 0)*

*execute all commands up to elif statement*

***elif*** *condition1*

*condition1 is zero (true - 0)*

*execute all commands up to elif statement*

***elif*** *condition2*

*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***

**Example:**

**#! /bin/bash**

**#**

**clear**

**echo -n "Enter First Number : "**

**read num1**

**echo -n "Enter Second Number : "**

**read num2**

**if [ $num1 -gt $num2 ]; then**

**echo "Number1 is greater than Number2"**

**elif [ $num1 -lt $num2 ]; then**

**echo "Number1 is less than Number2"**

**else**

**echo "Both are Equal"**

**fi**

**Lab Work:** Execute all programs and write in the space provided below.

**Exercise**:

1. Write a script which displays Good Morning if hour is less than 12.00, Good Afternoon if hour is less than 5.00p.m, and Good Evening if hour is greater than 5.00p.m.
2. Write a script that determines whether number is odd or even.
3. Write a script which takes two numbers as input and asks user choice for multiplication, division, addition, and subtraction. Like ***(calculator)***, and then calculate the output. Apply suitable checks; if user enters wrong input the program again ask for correct input.
4. Write a script to print no’s as 5, 4, 3, 2, 1 using while loop.
5. Write script to print given number in reverse order, for e.g. If no is 123 it must print as 321.
6. Write script to print given numbers sum of all digits, for e.g. If no is 123 it's sum of all digit will be 1+2+3 = 6.
7. Write Script to find out biggest number from given three no’s. No’s are supplies as command line argument. Print error if sufficient arguments are not supplied.

LAB 14

INTRODUCTION

CODE

OUTPUT

CONCLUSION