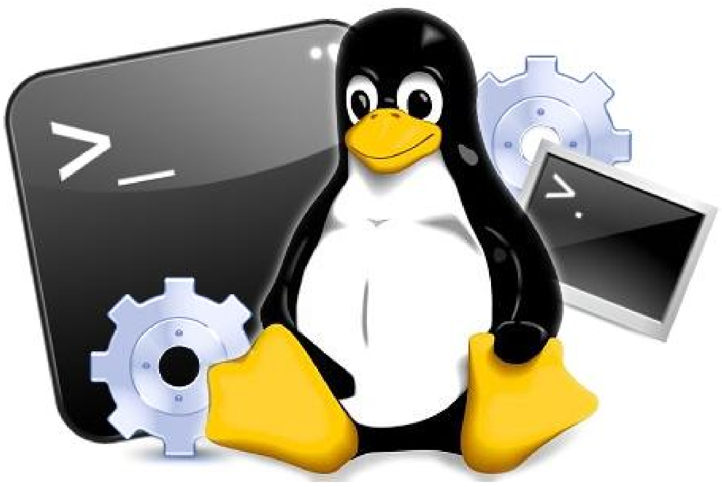
CS 206L: OPERATING SYSTEM LAB

BS(CS)@DSU

Summer 2017



Department of Computer Science
DHA Suffa University

Objective:

To learn LINUX commands, shell functionality and LINUX system calls, shell programming and usage of VI editor.

This manual is used to provide an understanding of the design aspects of operating system.

Recommended Systems/Software Requirements:

Virtual Box 4.x or greater with Fedora/Centos Installed on it.

Contents

About Linux

What is Linux?

Linux is a computer operating system. An operating system is the software that provides the interface between the hardware of a computer system and the applications programs that are used on it. Simply put, the operating system provides the link between the hardware of the computer and the user. Popular operating systems include DOS (used on PCs) and VM/CMS (used on mainframes, now becoming rare).Linux is available on a wide variety of computer systems, including personal computers, workstations, mainframes and supercomputers. It was developed for, and is particularly well-suited to, multi-user systems, but is now also run on 'stand-alone' machines.

What's special about Linux?

Linux has the following advantages: Linux is written in the high level language C. This makes it easy to install on new computing systems. Applications written to run on a Linux system will hopefully run on any Linux system, regardless of the hardware. Popularity Linux is available on many widely-used systems. It is very widely used and it has become the de facto standard for academic users, and for all multi-user applications.

Linux features

The kernel and the shell

The Linux operating system consists basically of the kernel and the shell. The kernel is the part carries out basic operating system functions such as accessing files, allocating memory and handling communications.

A shell provides the user interface to the kernel. A number of shells are available on the Linux operating system including the Bourne shell and the C shell. The shell is basically an Lbtensive program that runs all the time that you are logged on to the computer, and provides an interactive interface between the user and the computer functions. The C shell is the default shell for interactive work on many Linux systems.

File hierarchy

Linux has a hierarchical tree-like file store. The file store contains files and directories, as illustrated in the diagram below.

An illustration of a fragment of the Linux file store hierarchy.

The top-level directory is known as the **root**. Beneath the root are several system directories. The root is designated by the / character.

The directories below the root are designated by the path names:

/bin **/etc** **/usr**

Confusingly, the / character is also used as a separator in path names. So, from the figure above, the directory lnp5jb can be referred to by the path name /bin/home/sunserv1_b/lnp5jb. Historically, user directories were often kept in the directory /usr. However, it is often desirable to organize user directories in a different manner.

Users have their own directory in which they can create and delete files, and create their own sub-directories. For Example:

/user/ei/eib035

Belongs to someone who has the user name eib035.

Some typical system directories below the root directory:

/bin contains many of the programs which will be Executed by users

/etc files used by system administrators

/dev hardware peripheral devices

/lib system libraries

/usr normally contains applications software

/home home directories for different systems

Lb. No: 1 Linux for non-programmers

Aim:

Testing the use of LINUX commands such as **date, clear, chmod, man, mail, passwd, pwd, cat, ls, mv, mkdir, cd, rm, rmdir, wc etc, introduction to Vi editor.**

Synopsis

command [-switches] [arguments]
where:

command: the Linux command being run

[-switches]: optional parameters, usually single letters, providing additional functionality to the command

[arguments]: what the command is to be run against, e.g. a file.

[Anything in square brackets is optional]
Note: Linux commands are case sensitive

General commands

man man - read the *man* manual page

man *command* - read the *command* manual page

date - print the current date and time

cal - print this month's calendar

cal 2004 - print calendar for 2004

finger - display information about other users on the system who are logged on

finger user name - look up information about the named user

who - show who is currently on the system and what they are doing

File manipulation

ls - list the names of all the files in the current directory ls -l - list the names of all the files in the current directory in long format

ls -a - list all the files, including hidden files, in the current directory

cat file1 - print the contents of file1 onto the terminal screen

more file1 - look at file1 one page at a time. Press the space bar to see another page, the return key to see another line. Type q to quit

cp file1 file2 - make a copy of file1 and call it file2

mv file1 file2 - change the name of file1 to be file2

File processing

grep word file1 - look through file1 for case sensitive match on word and print the line that contains word

grep -i word file1 - look through file1 for case insensitive match on word and print the line that contains word

grep -v word file1 - look through file1 for case sensitive match on word and print the line that do not contain word

grep -l word * - look through all the files in the current directory and print the names of those files which contain word

wc file1 - count lines, words, and characters in file1

wc -l file1 - count lines in file1.

wc -l /usr/dict/words - count the number of lines (words) in /usr/dict/words

File system

pwd - print the current working directory

cd dir1 - change directory to dir1. cd with no argument changes back to the user's home directory mkdir myfiles - create a new directory named myfiles

rmdir myfiles - deletes a directory named myfiles (must be empty)

rm file1 - deletes file1

chmod - change protection of files. chmod o-r file1 makes file1 unreadable to people outside of the owner's group. See man chmod for more details

What is vi?

The default editor that comes with the Linux operating system is called vi (**vi**sual editor). [Alternate editors for Linux environments include pico and emacs, a product of GNU.]

The Linux vi editor is a full screen editor and has two modes of operation:

1. *Command mode* commands which cause action to be taken on the file, and
2. *Insert mode* in which entered text is inserted into the file.

In the command mode, every character typed is a command that does something to the text file being edited; a character typed in the command mode may even cause the vi editor to enter the insert mode. In the insert mode, every character typed is added to the text in the file; pressing the <Esc> (*Escape*) key turns off the Insert mode.

While there are a number of vi commands, just a handful of these is usually sufficient for beginning vi users. To assist such users, this Web page contains a sampling of basic vi commands. The most basic and useful commands are marked with an asterisk (* or star) in the tables below. With practice, these commands should become automatic.

NOTE: Both Linux and vi are **case-sensitive**. Be sure not to use a capital letter in place of a lowercase letter; the results will not be what you expect.

To Get Into and Out Of vi

To Start vi

To use vi on a file, type in vi filename. If the file named filename exists, then the first page (or screen) of the file will be displayed; if the file does not exist, then an empty file and screen are created into which you may enter text.

vi filename	*edit filename starting at line 1*

vi -r filename	*recover filename that was being edited when system crashed*

To exit vi

Usually the new or modified file is saved when you leave vi. However, 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*

Moving the Cursor

Unlike many of the PC and MacIntosh editors, **the mouse does not move the cursor** within the vi editor screen (or window). You must use the the key commands listed below. On some Linux platforms, the arrow keys may be used as well; however, since vi was designed with the Qwerty keyboard (containing no arrow keys) in mind, the arrow keys sometimes produce strange effects in vi and should be avoided.

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 nextword*
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*

Adding, Changing, and Deleting text

Unlike PC editors, you cannot replace or delete text by highlighting it with the mouse. Instead use the commands in the following tables.

Perhaps the most important command is the one that allows you to back up and *undo* your last action. Unfortunately, this command acts like a toggle, undoing and redoing your most recent action. You cannot go back more than one step.

u	*UNDO WHATEVER YOU JUST DID; a simple toggle*

The main purpose of an editor is to create, add, or modify text for a file.

Inserting or Adding text

The following commands allow you 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*

Changing text

The following commands allow you 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* c Nc	*change (replace) the nextN lines, starting with the current line,* *stopping when <Esc> is hit*

Deleting text

The following commands allow you 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* d Nd	*delete N lines, beginning with the current line;* *e.g., 5dd deletes 5 lines*

Cutting and Pasting text

The following commands allow you to copy and paste text.

Yy	*copy (yank, cut) the current line into the buffer*
Nyy *or* y Ny	*copy (yank, cut) the nextN lines, including the current line, into the buffer*
P	*put (paste) the line(s) in the buffer into the text after the current line*

Saving and Reading Files

These commands permit you to input and output files other than the named file with which you are currently working.

:r filename<Return>	*read file named filename and insert after current line (the line with cursor)*
:w<Return>	*write current contents to file named in original vi call*
:w newfile<Return>	*write current contents to a new file named newfile*
:w! prevfile<Return>	*write current contents over a pre-existing file named prevfile*

Lb. No: 2 Shell programming

Aim:

a) Illustration of shell function such as wild cards, redirection, pipes, sequencing, grouping, background processing, command substitution, sub shells.

b) Write shell scripts with the help of variables, loops (for, while), and conditional statements (if else, case). Shell variables, arguments to shell procedure, test command, arithmetic with EXPR command, interactive shell procedures with read.

Wild card

a wildcard is a character that may be substituted for any of a defined subset of all possible characters.

*	The * wildcard character substitutes for one or more characters in a filename. For instance, to list all the files in your directory that end with .c, enter the command
	ls *.c
?	? (question mark) serves as wildcard character for any one character in a filename. For instance, if you have files namedprog1, prog2, prog3 , and prog3 in your directory, the Linux command:
	ls prog?
[ ]	defines a class of characters (- for range, ! to exclude). For instance [abc]?? 3 character filename beginning with "a", "b", or "c" in your directory, the Linux command:
	ls [abc]??

Pipes

The way of connecting two or more commands together so that the output from one program becomes the input of the next program form a pipe.

To make a pipe, put a vertical bar (|) on the command line between two commands.

When a program takes its input from another program, performs some operation on that input, and writes the result to the standard output, it is referred to as a filter.

The grep Command:

The grep program searches a file or files for lines that have a certain pattern. The syntax is:

```
[XYZ] $grep pattern file(s)
```

The name "grep" derives from the ed (a Linux line editor) command g/re/p which means "globally search for a regular expression and print all lines containing it."

A regular expression is either some plain text (a word, for example) and/or special characters used for pattern matching.

The simplest use of grep is to look for a pattern consisting of a single word. It can be used in a pipe so that only those lines of the input files containing a given string are sent to the standard output. If you don't give grep a filename to read, it reads its standard input; that's the way all filter programs work:

```
[XYZ]$ ls -l | grep "Aug"
-rw-rw-rw-   1 john  doc     11008 Aug  6 14:10 ch02
-rw-rw-rw-   1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-r--   1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-r--   1 carol doc      1605 Aug 23 07:35 macros
[XYZ]$
```

There are various options which you can use along with grep command:

Option	Description
-v	Print all lines that do not match pattern.
-n	Print the matched line and its line number.
-l	Print only the names of files with matching lines (letter "l")
-c	Print only the count of matching lines.
-i	Match either upper- or lowercase.

Next, let's use a regular expression that tells grep to find lines with "carol", followed by zero or more other characters abbreviated in a regular expression as ".*"), then followed by "Aug".

Here we are using -i option to have case insensitive search:

```
[XYZ]$ls -l | grep -i "carol.*aug"
-rw-rw-r--   1 carol doc      1605 Aug 23 07:35 macros
[XYZ]$
```

The sort Command:

The sort command arranges lines of text alphabetically or numerically. The example below sorts the lines in the food file:

```
[XYZ]$sort food
Afghani Cuisine
Bangkok Wok
Big Apple Deli
Isle of Java
Mandalay
Sushi and Sashimi
Sweet Tooth
Tio Pepe's Peppers
[XYZ]$
```

The sort command arranges lines of text alphabetically by default. There are many options that control the sorting:

Option	Description
-n	Sort numerically (example: 10 will sort after 2), ignore blanks and tabs.
-r	Reverse the order of sort.
-f	Sort upper- and lowercase together.
+x	Ignore first x fields when sorting.

More than two commands may be linked up into a pipe. Taking a previous pipe example using grep, we can further sort the files modified in August by order of size.

The following pipe consists of the commands ls, grep, and sort:

```
[XYZ]$ls -l | grep "Aug" | sort +4n
-rw-rw-r--  1 carol doc      1605 Aug 23 07:35 macros
-rw-rw-r--  1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-rw-  1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-rw-  1 john  doc     11008 Aug  6 14:10 ch02
```

```
[amrood]$
```

This pipe sorts all files in your directory modified in August by order of size, and prints them to the terminal screen. The sort option +4n skips four fields (fields are separated by blanks) then sorts the lines in numeric order.

Command Substitution:

It is another way of combining two or more commands in which output of one command becomes the argument of another command. Command substitution is basically another way to do a pipe.

To consider a simple example, suppose you need to display today's date with a statement like this:

The date today is Thu Jan 1 17:01:18 IST 2013

For this use the expression`date` as an argument to echo:

```
[XYZ]$ echo The date today is `date`
      The date today is Thu Jan 1 17:01:18 IST 2013
[XYZ]$ echo "There are `ls|wc -l` files in the current directory"
There are 50 files in the current directory
```

Starting a Process:

When you start a process (run a command), there are two ways you can run it:

Foreground Processes

Background Processes

Foreground Processes:

By default, every process that you start runs in the foreground. It gets its input from the keyboard and sends its output to the screen.

You can see this happen with the ls command. If I want to list all the files in my current directory, I can use the following command:

```
[XYZ]$ls ch*.doc
```

This would display all the files whose name start with ch and ends with .doc:

```
ch01-1.doc   ch010.doc  ch02.doc    ch03-2.doc
```

```
ch04-1.doc   ch040.doc  ch05.doc    ch06-2.doc
ch01-2.doc ch02-1.doc
```

The process runs in the foreground, the output is directed to my screen, and if the ls command wants any input (which it does not), it waits for it from the keyboard.

While a program is running in foreground and taking much time, we cannot run any other commands (start any other processes) because prompt would not be available until program finishes its processing and comes out.

Background Processes:

A background process runs without being connected to your keyboard. If the background process requires any keyboard input, it waits.

The advantage of running a process in the background is that you can run other commands; you do not have to wait until it completes to start another!

The simplest way to start a background process is to add an ampersand (&) at the end of the command.

```
[XYZ]$ls ch*.doc &
```

This would also display all the files whose name start with ch and ends with .doc:

```
ch01-1.doc   ch010.doc  ch02.doc    ch03-2.doc
ch04-1.doc   ch040.doc  ch05.doc    ch06-2.doc
ch01-2.doc ch02-1.doc
```

Here if the ls command wants any input (which it does not), it goes into a stop state until I move it into the foreground and give it the data from the keyboard.

That first line contains information about the background process - the job number and process ID. You need to know the job number to manipulate it between background and foreground.

If you press the Enter key now, you see the following:

```
[1]   +   Done               ls ch*.doc &
[XYZ]$
```

The first line tells you that the ls command background process finishes successfully. The second is a prompt for another command.

Listing Running Processes:

It is easy to see your own processes by running the ps (process status) command as follows:

```
[XYZ]$ps
PID      TTY     TIME       CMD
18358    ttyp3   00:00:00   sh
18361    ttyp3   00:01:31   abiword
18789    ttyp3   00:00:00   ps
```

One of the most commonly used flags for ps is the -f (f for full) option, which provides more information as shown in the following example:

```
[XYZ]$ps -f
UID      PID  PPID C STIME    TTY   TIME CMD
XYZ   6738 3662 0 10:23:03 pts/6 0:00 first_one
XYZ  6739 3662 0 10:22:54 pts/6 0:00 second_one
XYZ  3662 3657 0 08:10:53 pts/6 0:00 -ksh
XYZ   6892 3662 4 10:51:50 pts/6 0:00 ps -f
```

Here is the description of all the fileds displayed by ps -f command:

Column	Description
UID	User ID that this process belongs to (the person running it).
PID	Process ID.
PPID	Parent process ID (the ID of the process that started it).
C	CPU utilization of process.
STIME	Process start time.
TTY	Terminal type associated with the process
TIME	CPU time taken by the process.
CMD	The command that started this process.

There are other options which can be used along with ps command:

Option	Description
-a	Shows information about all users
-x	Shows information about processes without terminals.
-u	Shows additional information like -f option.
-e	Display Lbtended information.

Stopping Processes:

Ending a process can be done in several different ways. Often, from a console-based command, sending a CTRL + C keystroke (the default interrupt character) will exit the command. This works when process is running in foreground mode.

If a process is running in background mode then first you would need to get its Job ID using pscommand and after that you can use kill command to kill the process as follows:

```
[XYZ]$ps -f
UID      PID  PPID C STIME    TTY   TIME CMD
XYZ   6738 3662 0 10:23:03 pts/6 0:00 first_one
XYZ   6739 3662 0 10:22:54 pts/6 0:00 second_one
XYZ   3662 3657 0 08:10:53 pts/6 0:00 -ksh
XYZ   6892 3662 4 10:51:50 pts/6 0:00 ps -f
[XYZ]$kill 6738
Terminated
```

Here kill command would terminate first_one process. If a process ignores a regular kill command, you can use kill -9 followed by the process ID as follows:

```
[XYZ]$kill -9 6738
Terminated
```

Parent and Child Processes:

Each Linux process has two ID numbers assigned to it: Process ID (pid) and Parent process ID (ppid). Each user process in the system has a parent process.

Most of the commands that you run have the shell as their parent. Check ps -f example where this command listed both process ID and parent process ID.

Zombie and Orphan Processes:

Normally, when a child process is killed, the parent process is told via a SIGCHLD signal. Then the parent can do some other task or restart a new child as needed. However, sometimes the parent process is killed before its child is killed. In this case, the "parent of all processes," initprocess, becomes the new PPID (parent process ID). Sometime these processes are called orphan process.

When a process is killed, a ps listing may still show the process with a Z state. This is a zombie, or defunct, process. The process is dead and not being used. These processes are different from orphan processes. They are the processes that has completed Lbecution but still has an entry in the process table.

Daemon Processes:

Daemons are system-related background processes that often run with the permissions of root and services requests from other processes.

A daemon process has no controlling terminal. It cannot open /dev/tty. If you do a "ps -ef" and look at the tty field, all daemons will have a ? for the tty.

More clearly, a daemon is just a process that runs in the background, usually waiting for something to happen that it is capable of working with, like a printer daemon is waiting for print commands.

If you have a program which needs to do long processing then its worth to make it a daemon and run it in background.

What is Shell Script?

Normally shells are interactive. It means shell accept command from you (via keyboard) and execute them. But if you use command one by one (sequence of 'n' number of commands) , the you can store this sequence of command to text file and tell the shell to execute this text file instead of entering the commands. This is known as shell script.

Shell script defined as:

"Shell Script is series of command written in plain text file. Shell script is just like batch file is MS-DOS but have more power than the MS-DOS batch file."

Why to Write Shell Script?

Shell script can take input from user, file and output them on screen.

Useful to create our own commands.

Save lots of time.

To automate some task of day today life.

System Administration part can be also automated.

How to write shell script?

Following steps are required to write shell script:

1. Use any editor like vi or mcedit to write shell script.
2. After writing shell script set execute permission for your script as follows

syntax:

```
chmod permission your-script-name
```

examples:

```
$ chmod +x your-script-name
$ chmod 755 your-script-name
```

Note: This will set read write execute(7) permission for owner, for group and other permission is read and execute only(5).

1. execute your script as

Syntax:

```
bash your-script-name
sh your-script-name
./your-script-name
```

examples:

```
$ bash bar
$ sh bar
$ ./bar
```

Variables in Shell

To process our data/information, data 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 (Its a named storage location that may take different values, but only one at a time).

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.

You can see system variables by giving command like $ set, some of the important System variables are:

System Variable	**Meaning**
BASH=/bin/bash	Our shell name
BASH_VERSION=1.14.7(1)	Our shell version name
COLUMNS=80	No. of columns for our screen
HOME=/home/vivek	Our home directory
LINES=25	No. of columns for our screen
LOGNAME=students	students Our logging name
OSTYPE=Linux	Our Os type
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

NOTE: that Some of the above settings can be different in your PC/Linux environment. You can print any of the above variables contains as follows:

```
$ echo $USERNAME

$ echo $HOME
```

How to define User defined variables (UDV)

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.

Rules for Naming variable name (Both UDV and System Variable)

Variable name must begin with Alphanumeric character or underscore character (_), followed by one or more Alphanumeric character.

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.

You can define NULL variable as follows (NULL variable is variable which has no value at the time of definition)

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

How to print or access value of UDV (User defined variables)

To print or access UDV use following syntax
Syntax:

$variable name

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
```

Shell Arithmetic

Use to perform arithmetic operations.

Syntax:

exprop1 math-operator op2

examples:

```
$ expr1 + 3
$ expr2 – 1
$ expr10 / 2
$ expr20 % 3
$ expr10 \* 3
$ echo `expr6 + 3`
```

Note:

expr20 %3 - Remainder read as 20 mod 3 and remainder is 2.
expr10 * 3 - Multiplication use * and not * since its wild card.

For the last statement not the following points

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.

Second, expr is also end with ` i.e. back quote.

Here expr6 + 3 is evaluated to 9, then echo command prints 9 as sum

Here if you use double quote or single quote, it will NOT work

The read Statement

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

Syntax:

read variable1, variable2,...variableN

Following script first ask user, name and then waits to enter name from the user via keyboard. Then user enters name from keyboard (after giving name you have to press ENTER key) and entered name through keyboard is stored (assigned) to variable fname.

Decision making
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
```

Condition is defined as:

"*Condition is nothing but comparison between two values.*"

For compression you can use test or [expr] statements or even exist status can be also used.

Expression is defined as:

"*An expression is nothing but combination of values, relational operator (such as >,<, <> etc) and mathematical operators (such as +, -, / etc).*"

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
```

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 (( Lbpr1; Lbpr2; Lbpr3 ))
do
          .....
          .....
          repeat all statements between do and
          done until Lbpr2 is TRUE
```

```
  Done
```

In above syntax BEFORE the first iteration, ***Lbpr1*** is evaluated. This is usually used to initialize variables for the loop. All the statements between do and done is executed repeatedly UNTIL the value of ***Lbpr2*** is TRUE. AFTER each iteration of the loop, ***Lbpr3*** is evaluated. This is usually use to increment a loop counter.

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

while loop

Syntax:

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

Loop is executed as long as given condition is true. For e.g.. Above for loop program (shown in last section of for loop) can be written using while loop as:

```
$cat > nt1
#!/bin/sh
#
#Script to test while statement
#
#
if [ $# -eq 0 ]
Then
        echo "Error - Number missing form command line argument"
        echo "Syntax : $0 number"
        echo " Use to print multiplication table for given number"
        exit 1
fi
n=$1
i=1
while [ $i -le 10 ]
do
        echo "$n * $i = `expr$i \* $n`"
        i=`expr$i + 1`
done
```

Common Test Cases

test returns true in the following cases:

test s	s is not a null string
test -n string	string is nonzero
test sting1 = string2	the two strings are equal
test sting1 != string2	the two strings are not equal
test integer1 -eq integer2	the integers are equal
test integer1 -ge integer2	integer1 is greater or equal to integer2
test integer1 -gt integer2	integer1 is greater than integer2
test integer1 -le integer2	integer1 is less than or equal to integer2
test integer1 -lt integer2	integer1 less than integer2
test integer1 -ne integer2	integer1 is not equal to integer2
test -f file	if the file exists and is not a directory
test -r file	the file is readable

test -w file	the file is writable
test -d dir	dir is a directory

An example of Test

Here, we simply check to see if java exists in /usr/bin using the "test" command.

```
if test -f /usr/bin/java
then
        echo "Java exists in /usr/bin/"
fi
```

Using Square brackets

Square brackets can be used to replace the test command in most command shells. Here is an example of this. Note that the spaces between the brackets and the test operation are vital here.

```
if [ -f /usr/bin/java ]
then
        echo "Java exists in /usr/bin/"
fi
```

Testing strings

When testing strings in shell scripts be aware that a null string may cause the shell interpreter to error.

For example:

```
[string1 == string2]
```

will be fine as long as neither string is null, but if one or both evaluate to null, then the shell command interpreter will throw an error. Put quotes around your string to avoid this error condition:

```
[ "string1" == "string2" ]
```

Using Double Square brackets

With some shell command interpreters such as the Korn Shell and Bash you can also use double square brackets. The test operators used for single brackets are the most standard, whereas the double square bracket test operators behave in a slightly different way.

For example:

```
[[ string1 == string2 ]]
```

is valid in in Korn shell, and checks to see if the value of string1 equals string2.

If you were to write these using single brackets:

[string1 ==	This is wrong and some command shells would

string2]	complain
[string1 = string2]	Better. This would work unless one string was null
["string1" = "string2"]	Even better! This should work with all Bourne derived command shells

Command Line Arguments

Command line arguments are treated as special variables within the script, the reason I am calling them variables is because they can be changed with the shift command. The command line arguments are enumerated in the following manner *$0*, *$1*, *$2*, *$3*, *$4*, *$5*, *$6*, *$7*, *$8* and *$9*. *$0* is special in that it corresponds to the name of the script itself. *$1* is the first argument, *$2* is the second argument and so on. To reference after the ninth argument you must enclose the number in brackets like this *${nn}*. You can use the **shift** command to shift the arguments 1 variable to the left so that *$2* becomes *$1*, *$1* becomes *$0* and so on, *$0* gets scrapped because it has nowhere to go, this can be useful to process all the arguments using a loop, using one variable to reference the first argument and **shifting** until you have Lbhausted the arguments list.

As well as the command line arguments there are some special built-in variables:

$# represents the parameter count. Useful for controlling loop constructs that need to process each parameter.

$@ Lbpands to all the parameters separated by spaces. Useful for passing all the parameters to some other function or program.

$- Lbpands to the flags(options) the shell was invoked with. Useful for controlling program flow based on the flags set.

$$ Lbpands to the process id of the shell innovated to run the script. Useful for creating unique temporary filenames relative to this instantiation of the script.

Note:

The command line arguments will be referred to as parameters from now on, this is because SH also allows the definition of functions which can take parameters and when called the *$n* family will be redefined, hence these variables are always parameters, its just that in the case of the parent script the parameters are passed via the command line. One Lbception is *$0* which is always set to the name of the parent script regardless of whether it is inside a function or not.

Lb. No: 3 LINUX system calls

Aim:

C programs to illustrate the use of various LINUX system calls.

File system calls: -

Open( ), close( ), creat( ), read( ), write( ), dup( ), lseek( ).

Process Management System calls: -

Fork( ), exit( ), wait( ), signal( ), kill( ), alarm( ).

System calls for File Processing

The following table briefly describes the function of each.

System calls	**Function**
open	open an existing file or create a new file
read	Read data from a file
write	Write data to a file
lseek	Move the read/write pointer to the specified location
close	Close an open file
unlink	Delete a file
chmod	Change the file protection attributes
stat	Read file information from inodes

Files to be included for file-related system calls.

#include	*<unistd.h>*
#include	*<fcntl.h>*
#include	*<sys/types.h>*
#include	*<sys/uio.h>*
#include	*<sys/stat.h>*

Open files

The ***open*** system call can be used to open an existing file or to create a new file if it does not exit already. The syntax of *open* has two forms:

*int open(const char *path, int flags); and*

*int open(const char *path, int flags, mode_t modes);*

The first form is normally used to open an existing file, and the second form to open a file and to create a file if it does not exit already. Both forms returns an integer called the *file descriptor*. The file descriptor will be used for reading from and writing to the file. If the file cannot be opened or created, it returns -1. The first parameter *path* in both forms sPecifies the file name to be opened or created. The second parameter (*flags*) specifies how the file may be used. The following list some commonly used flag values.

Flag	**Description**
O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_NONBLOCK	do not block on open
O_APPEND	append on each write
O_CREAT	create file if it does not exit
O_TRUNC	truncate size to 0
O_EXCL	error if create and file exists
O_SHLOCK	atomically obtain a shared lock
O_EXLOCK	atomically obtain an Exclusive lock
O_DIRECT	eliminate or reduce cache effects
O_FSYNC	synchronous writes
O_NOFOLLOW	do not follow symlinks

The flag (*O_CREAT*) may be used to create the file if it does not exit. When this flag is used, the third parameter (*modes*) must be used to specify the file access permissions for the new file. Commonly used modes (or access permissions) include

Constant Name	**Octal Value**	**Description**
S_IRWXU	700	/* RWX mask for owner */
S_IRUSR	400	/* R for owner */
S_IWUSR	200	/* W for owner */
S_IXUSR	100	/* X for owner */
S_IRWXO	7	/* RWX mask for other */
S_IROTH	4	/* R for other */
S_IWOTH	2	/* W for other */
S_IXOTH	1	/* X for other */
R: read, W: write, and X: executable		

For example, to open file "tmp.txt" in the current working directory for reading and writing:

fd = open("tmp.txt", O_RDWR);

To open "sample.txt" in the current working directory for appending or create it, if it does not exit, with read, write and execute permissions for owner only:

fd = open("tmp.txt", O_WRONLY|O_APPEND|O_CREAT, S_IRWXU);

A file may be opened or created outside the current working directory. In this case, an absolute path and relative path may prefix the file name. For example, to create a file in /tmp directory:

open("/tmp/tmp.txt", O_RDWR);

Read from files

The system call for reading from a file is ***read***. Its syntax is

*ssize_t read(int fd, void *buf, size_t nbytes);*

The first parameter *fd* is the file descriptor of the file you want to read from, it is normally returned from *open*. The second parameter *buf* is a pointer pointing the memory location where the input data should be stored. The last parameter *nbytes* specifies the maximum number of bytes you want to read. The system call returns the number of bytes it actually read, and normally this number is either smaller or equal to *nbytes*. The following segment of code reads up to 1024 bytes from file tmp.txt:

```
int actual_count = 0;
int fd = open("tmp.txt", O_RDONLY);
void *buf = (char*) malloc(1024);


actual_count = read(fd, buf, 1024);
```

Each file has a pointer, normally called read/write offset, indicating where nLbt *read* will start from. This pointer is incremented by the number of bytes actually read by the *read* call. For the above example, if the offset was zero before the *read* and it actually read 1024 bytes, the offset will be 1024 when the *read* returns. This offset may be changed by the system call *lseek*, which will be covered shortly.

Write to files

The system call ***write*** is to write data to a file. Its syntax is

*ssize_t write(int fd, const void *buf, size_t nbytes);*

It writes *nbytes* of data to the file referenced by file descriptor *fd* from the buffer pointed by *buf*. The *write* starts at the position pointed by the offset of the file. Upon returning from *write*, the offset is advanced by the number of bytes which were successfully written. The function returns the number of bytes that were actually written, or it returns the value -1 if failed.

Reposition the R/W offset

The ***lseek*** system call allows random access to a file by reposition the offset for nextread or write. The syntax of the system call is

off_t lseek(int fd, off_t offset, int reference);

It repositions the offset of the file descriptor *fd* to the argument *offset* according to the directive *reference*. The *reference* indicate whether *offset* should be considered from the beginning of the file (with *reference* 0), from the current position of the read/write offset (with *reference* 1), or from the end of the file (with *reference* 2). The call returns the byte offset where the next *read/write* will start.

Close files

The ***close*** system call closes a file. Its syntax is

int close(int fd);
It returns the value 0 if successful; otherwise the value -1 is returned.

Delete files

The ***unlink*** may be used to delete a file (A file may have multiple names (also called links), here we assume that a file in this context has only one name or link.). Its syntax is:

*int unlink(const char *path);*

path is the file name to be deleted. The *unlink* system call returns the value 0 if successful, otherwise it returns the value -1.

Change file access permissions

File access permissions may be set using the ***chmod*** system call (note that there is a command with the same name for setting access permissions.). It has two forms:

*int chmod(const char *path, mode_t mode);* or

int fchmod(int fd, mode_t mode);

Both forms set the access permission of a file to *mode*. In the first form the file is identified by its name, and in the second it is identified by a file descriptor returned from the *open* system call. For *mode*, you may use any of the constants defined in the *Open files* section of this tutorial.

The system call returns the value 0 if successful, otherwise it returns the value -1.

Accessing file information from Inodes

The ***stat*** system call can be used to access file information of a file from its inode. It can appear in two forms:

*int stat(const char *path, struct stat *sb);* or *int fstat(int fd, struct stat *sb);*

Both forms return the information through the *stat* structure pointed by *sb*. In the first form, the file is identified by its name and in the second form, it is identified by its file descriptor returned from a call to *open*. The *stat* structure includes at least the following elements:

Element	**Description**
st_mode	file protection mode
st_uid	user ID of the file owner
st_size	file size in bytes

No permission is needed to *stat* a file. However since the second form requires a file descriptor of the file and a file descriptor may be only obtained by *open*, *fstat* can only be applied to files that has proper access permissions.

The call returns the value 0 if successful, otherwise it returns the value -1. The call fails if the specified path or the file does not exit.

Process Management System calls

NAME

fork - create a child process

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

DESCRIPTION

fork() creates a child process that differs from the parent process only in its PID and PPID, and in the fact that resource utilizations are set to 0. File locks and pending signals are not inherited.

Under Linux, fork() is implemented using copy-on-write pages, so the only penalty that it incurs is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child.

RETURN VALUE

On success, the PID of the child process is returned in the parent's thread of execution, and a 0 is returned in the child's thread of execution. On failure, a -1 will be returned in the parent's context, no child process will be created, and err no will be set appropriately.

NAME

exit - cause normal process termination

SYNOPSIS

```
#include <stdlib.h>

void exit(int status);
```

DESCRIPTION

The exit() function causes normal process termination and the value of status & 0377 is returned to the parent.

All functions registered with at exit() and on_exit() are called, in the reverse order of their registration. (It is possible for one of these functions to use at exit() or on_exit() to register an additional function to be executed during exit processing; the new registration is added to the front of the list of functions that remain to be called.)

All open streams are flushed and closed. Files created by tmpfile() are removed.

The C standard specifies two constants, exit_SUCCESS and exit_FAILURE, that may be passed to exit() to indicate successful or unsuccessful termination, respectively.

RETURN VALUE

The exit() function does not return.

NAME

wait, waitpid, waitid - wait for process to change state

SYNOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);

pid_t waitpid(pid_t pid, int *status, int options);

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

DESCRIPTION

All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state.

If a child has already changed state, then these calls return immediately. Otherwise they block until either a child changes state or a signal handler interrupts the call (assuming that system calls are not automatically restarted using the SA_RESTART flag of sigaction(2)). In the remainder of this page, a child whose state has changed and which has not yet been waited upon by one of these system calls is termed waitable.

NAME

signal - ANSI C signal handling

SYNOPSIS

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

DESCRIPTION

The signal() system call installs a new signal handler for the signal with number signum. The signal handler is set to sighandler which may be a user specified function, or either SIG_IGN or SIG_DFL.

Upon arrival of a signal with number signum the following happens. If the corresponding handler is set to SIG_IGN, then the signal is ignored. If the handler is set to SIG_DFL, then the default action associated with the signal occurs. Finally, if the handler is set to a function sighandler then first either the handler is reset to SIG_DFL or an implementation-dependent blocking of the signal is performed and nextsighandler is called with argument signum.

Using a signal handler function for a signal is called "catching the signal". The signals SIGKILL and SIGSTOP cannot be caught or ignored.

RETURN VALUE

The signal() function returns the previous value of the signal handler, or SIG_ERR on error.

NAME

kill - send signal to a process

SYNOPSIS

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

DESCRIPTION

The kill() system call can be used to send any signal to any process group or process.

If pid is positive, then signal sig is sent to pid.

If pid equals 0, then sig is sent to every process in the process group of the current process.

If pid equals -1, then sig is sent to every process for which the calling process has permission to send signals, Lbcept for process 1 (init), but see below.

If pid is less than -1, then sig is sent to every process in the process group -pid.

If sig is 0, then no signal is sent, but error checking is still performed.

For a process to have permission to send a signal it must either be privileged (under Linux: have the CAP_KILL capability), or the real or effective user ID of the sending process must equal the real or saved set-user-ID of the target process. In the case of SIGCONT it suffices when the sending and receiving processes belong to the same session.

RETURN VALUE

On success (at least one signal was sent), zero is returned. On error, -1 is returned, and errno is set appropriately.

NAME

alarm - set an alarm clock for delivery of a signal

SYNOPSIS

```
#include <unistd.h>


unsigned int alarm(unsigned int seconds);
```

DESCRIPTION

alarm() arranges for a SIGALRM signal to be delivered to the process in seconds seconds.

If seconds is zero, no new alarm() is scheduled.

In any event any previously set alarm() is cancelled.

RETURN VALUE

alarm() returns the number of seconds remaining until any previously scheduled alarm was due to be delivered, or zero if there was no previously scheduled alarm.

GRUB bootloader

Introduction

GNU GRUB is a bootloader capable of loading a variety of free and proprietary operating systems. GRUB will work well with Linux, DOS, Windows, or BSD. GRUB stands for GRand Unified Bootloader.

GRUB is dynamically configurable. This means that the user can make changes during the boot time, which include altering existing boot entries, adding new, custom entries, selecting different kernels, or modifying initrd. GRUB also supports Logical Block Addressmode. This means that if your computer has a fairly modern BIOS that can access more than 8GB (first 1024 cylinders) of hard disk space, GRUB will automatically be able to access all of it.

GRUB can be run from or be installed to any device (floppy disk, hard disk, CD-ROM, USB drive, network drive) and can load operating systems from just as many locations, including network drives. It can also decompress operating system images before booting them.

How does GRUB work?

When a computer boots, the BIOS transfers control to the first boot device, which can be a hard disk, a floppy disk, a CD-ROM, or any other BIOS-recognized device. We'll concentrate on hard disks, for the sake of simplicity.

The first sector on a hard is called the Master Boot Record (MBR). This sector is only 512 bytes long and contains a small piece of code (446 bytes) called the primary boot loader and the partition table (64 bytes) describing the primary and Lbtended partitions.

By default, MBR code looks for the partition marked as active and once such a partition is found, it loads its boot sector into memory and passes control to it.

GRUB replaces the default MBR with its own code.

Furthermore, GRUB works in stages.

Stage 1 is located in the MBR and mainly points to Stage 2, since the MBR is too small to contain all of the needed data.

Stage 2 points to its configuration file, which contains all of the complLb user interface and options we are normally familiar with when talking about GRUB. Stage 2 can be located anywhere on the disk. If Stage 2 cannot find its configuration table, GRUB will cease the boot sequence and present the user with a command line for manual configuration.

Stage 1.5 also exists and might be used if the boot information is small enough to fit in the area immediately after MBR.

The Stage architecture allows GRUB to be large (~20-30K) and therefore fairly complLb and highly configurable, compared to most bootloaders, which are sparse and simple to fit within the limitations of the Partition Table.

Naming convention

The device syntax used in GRUB is a wee bit different from what you may have seen before in your operating system(s), and you need to know it so that you can specify a drive/partition.

Look at the following examples and explanations:

```
(fd0)
```

First of all, GRUB requires that the device name be enclosed with `(' and `)'. The `fd' part means that it is a floppy disk. The number `0' is the drive number, which is counted from zero. This expression means that GRUB will use the whole floppy disk.

```
(hd0,1)
```

Here, `hd' means it is a hard disk drive. The first integer `0' indicates the drive number, that is, the first hard disk, while the second integer, `1', indicates the partition number (or the pc slice number in the BSD terminology). Once again, please note that the partition numbers are counted from zero, not from one. This expression means the second partition of the first hard disk drive. In this case, GRUB uses one partition of the disk, instead of the whole disk.

```
(hd0,4)
```

This specifies the first extended partition of the first hard disk drive. Note that the partition numbers for extended partitions are counted from `4', regardless of the actual number of primary partitions on your hard disk.

```
(hd1,a)
```

This means the BSD `a' partition of the second hard disk. If you need to specify which pc slice number should be used, use something like this: `(hd1,0,a)'. If the pc slice number is omitted, GRUB searches for the first pc slice which has a BSD `a' partition.

Of course, to actually access the disks or partitions with GRUB, you need to use the device specification in a command, like `root (fd0)' or `unhide (hd0,2)'. To help you find out which number specifies a partition you want, the GRUB command-line options have argument completion. This means that, for example, you only need to type

```
root (
```

followed by a <TAB>, and GRUB will display the list of drives, partitions, or file names. So it should be quite easy to determine the name of your target partition, even with minimal knowledge of the syntax.

Note that GRUB does not distinguish IDE from SCSI - it simply counts the drive numbers from zero, regardless of their type. Normally, any IDE drive number is less than any SCSI drive number, although that is not true if you change the boot sequence by swapping IDE and SCSI drives in your BIOS.

Now the question is, how to specify a file? Again, consider an example:

```
(hd0,0)/vmlinuz
```

This specifies the file named `vmlinuz', found on the first partition of the first hard disk drive. Note that the argument completion works with file names, too.

That was easy, admit it. Now read the nextchapter, to find out how to actually install GRUB on your drive.

Lb. No: 4(a) FCFS SCHEDULING

Aim:

Write a C program to implement the various process scheduling mechanisms such as FCFS.

Algorithm:

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

Lb. No: 4(b) SJF SCHEDULING

Aim:

Write a C program to implement the various process scheduling mechanisms such as SJF Scheduling.

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

Lb. No: 5(a) PRIORITY SCHEDULING

Aim:

Write a C program to implement the various process scheduling mechanisms such as Priority Scheduling.

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

Lb. No: 5(b) ROUND ROBIN SCHEDULING

Aim:

Write a C program to implement the various process scheduling mechanisms such as Round Robin Scheduling.

Algorithm for RR

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue and time quantum (or) time slice

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

Step 4: Calculate the no. of time slices for each process where

No. of time slice for process(n) = burst time process(n)/time slice

Step 5: If the burst time is less than the time slice then the no. of time slices =1.

Step 6: Consider the ready queue is a circular Q, calculate

a) Waiting time for process(n) = waiting time of process(n-1)+ burst time of process(n-1) + the time difference in getting the CPU from process(n-1)

b) Turn around time for process(n) = waiting time of process(n) + burst time of process(n)+ the time difference in getting CPU from 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

Lb. No: 6(a) IMPLEMENTATION OF DEKKER'S ALGORITHM

AIM:

To implement Mutual exclusion Using Dekker's Algorithm.

ALGORITHM:

1. Start the program.
2. Declare and initialise variable.
3. Define process 1 and process 2 as below.
   a) Check Pi wants to enter as 1.
   b) Check if Pj wants to enter, if so then check whether favored process is j or i.
   c) If it is j, Pj wants to enter as 0 and wait until favored process equals i.
4. Else process I enters critical region.
5. On exit from critical region, set favored process as j and Pi wants to enter as 0.
6. If only one process wants to enter then that particular function is invoked.
7. If both process want to enter critical section then both process invoked. It allows favored process to run first.
8. Stop the program.

Lb. No: 6(b) PRODUCER – CONSUMER PROBLEM

AIM:

To implement producer – consumer problem using semaphore.

ALGORITHM:

1. Start the program.
2. Declare three semaphore variables.

   Mutex initialised to 0 which allows only one process to execute at any time.

   Two variables to indicate the limit of buffer.
3. Wait and signal are two functions to implement the semaphore.

   Wait-waits until semaphore variable reach 1 and then decrements it.

   Signal – increments the semaphore variable by 1.
4. The reader process, checks if any process is writing. If so it waits else it reads the content of shared variable and then signals.
5. The Writer process checks if any other process is accessing the shared variable. If not it changes the value of shared variable and then signals.
6. End he program.

Lb. No: 6(c) READER WRITER PROBLEM USING SEMAPHORE

AIM:

To implement reader – writer problem using semaphore.

ALGORITHM:

1. Start the program.
2. Declare three semaphore variable mutex and a shared variable which is used by reader and writer processes.
3. Wait and signal are two functions to implement the semaphore.

   Wait-waits until semaphore variable reach 1 and then decrements it.

   Signal – increments the semaphore variable by 1.
4. The reader process, checks if any process is writing. If so it waits else it reads the content of shared variable and then signals.
5. The Writer process checks if any other process is accessing the shared variable. If not it changes the value of shared variable and then signals.
6. Pthreads is used to execute two processes simultaneously.
7. End he program.

Lb. No: 7 and 8 BANKER'S ALGORITHM

AIM:

To implement deadlock avoidance & Prevention by using Banker's Algorithm.

Deadlock avoidance & Dead Lock Prevention

Banker's Algorithm:

When a new process enters a system, it must declare the maximum number of instances of each resource type it needed. This number may exceed the total number of resources in the system. When the user request a set of resources, the system must determine whether the allocation of each resources will leave the system in safe state. If it will the resources are allocation; otherwise the process must wait until some other process release the resources.

Data structures

n-Number of process, m-number of resource types.

Available: Available[j]=k, k – instance of resource type Rj is available.

Max: If max[i, j]=k, Pi may request at most k instances resource Rj.

Allocation: If Allocation [i, j]=k, Pi allocated to k instances of resource Rj

Need: If Need[I, j]=k, Pi may need k more instances of resource type Rj,

Need[I, j]=Max[I, j]-Allocation[I, j];

Safety Algorithm

1. Work and Finish be the vector of length m and n respectively, Work=Available and Finish[i] =False.

2. Find an i such that both

Finish[i] =False

Need<=Work

If no such I exists go to step 4.

3. work=work+Allocation, Finish[i] =True;

4. if Finish[1]=True for all I, then the system is in safe state.

Resource request algorithm

Let Request i be request vector for the process Pi, If request i=[j]=k, then process Pi wants k instances of resource type Rj.

1. if Request<=Need I go to step 2. Otherwise raise an error condition.

2. if Request<=Available go to step 3. Otherwise Pi must since the resources are available.

3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows;

   Available=Available-Request I;

   Allocation I =Allocation+Request I;

   Need i=Need i-Request I;

If the resulting resource allocation state is safe, the transaction is completed and process Pi is allocated its resources. However if the state is unsafe, the Pi must wait for Request i and the old resource-allocation state is restored.

ALGORITHM:

4. Start the program.

5. Get the values of resources and processes.

6. Get the avail value.

7. After allocation find the need value.

8. Check whether its possible to allocate.

9. If it is possible then the system is in safe state.

10. Else system is not in safety state.

11. If the new request comes then check that the system is in safety.

12. or not if we allow the request.

13. stop the program.

Lb. No: 9(a) LRU PAGE REPLACEMENT ALGORITHM

AIM:

To implement page replacement algorithm LRU (Least Recently Used) Here we select the page that has not been used for the longest period of time.

ALGORITHM:

Step 1: Create a queue to hold all pages in memory

Step 2: When the page is required replace the page at the head of the queue

Step 3: Now the new page is inserted at the tail of the queue

Step 4: Create a stack

Step 5: When the page fault occurs replace page present at the bottom of the stack

Lb. No: 9(b) FIFO PAGE REPLACEMENT ALGORITHM

AIM:

To implement page replacement algorithm FIFO (First In First Out)

ALGORITHM:

Step 1: Create a queue to hold all pages in memory

Step 2: When the page is required replace the page at the head of the queue

Step 3: Now the new page is inserted at the tail of the queue

Lb. No: 9(c) OPTIMAL(LFU) PAGE REPLACEMENT ALGORITHM

AIM:

To implement page replacement algorithms Optimal (The page which is not used for longest time)

ALGORITHM:

Optimal algorithm

Here we select the page that will not be used for the longest period of time.

Step 1: Create a array

Step 2: When the page fault occurs replace page that will not be used for the longest

period of time

Lb. No: 10(a) Best fit Algorithm

AIM:

To implement Best fit Algorithm for Memory Management.

ALGORITHM:

Step 1: Start the program.

Step 2: Get the number of holes and size of holes.

Step 3: Enter the number of processes and their sizes for process creation.

Step 4: Compare the process and size then the process is successfully to allocate given hole.

Step 5: In best-fit memory management scheme allocates the smallest hole that is big enough.

Lb. No: 10(b) Worst fit Algorithm

AIM:

To implement Worst Fit Algorithm for Memory Management.

ALGORITHM:

Step 1: Start the program.

Step 2: Get the number of holes and size of holes.

Step 3: Enter the number of processes and their sizes for process creation.

Step 4: Compare the process and size then the process is successfully to allocate given hole.

Step 5: In worst fit memory management scheme the biggest hole is allocated.

Lb. No: 10(c) first fit Algorithm

AIM:

To implement First Fit Algorithm for Memory Management.

ALGORITHM:

Step 1: Start the program.

Step 3: Get the number of holes and size of holes.

Step 4: Enter the number of processes and their sizes for process creation.

Step 5: Compare the process and size then the process is successfully to allocate given hole.

Step 6: In first fit memory management scheme the first biggest hole is allocated first.

Lb. No: 11 DISK SCHEDULING ALGORITHMS

INTRODUCTION

In operating systems, seek time is very important. Since all device requests are linked in queues, the seek time is increased causing the system to slow down. Disk Scheduling Algorithms are used to reduce the total seek time of any request.

TYPES OF DISK SCHEDULING ALGORITHMS

Although there are other algorithms that reduce the seek time of all requests, I will only concentrate on the following disk scheduling algorithms:

a) First Come-First Serve (FCFS)

b) Shortest Seek Time First (SSTF)

c) Elevator (SCAN)

d) Circular SCAN (C-SCAN)

e) LOOK

f) C-LOOK

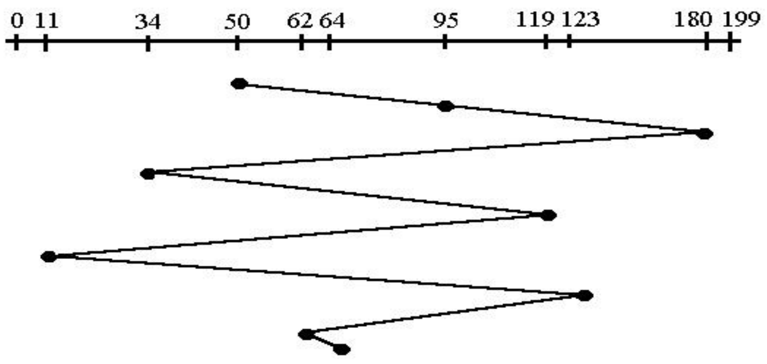
These algorithms are not hard to understand, but they can confuse someone because they are so similar. What we are striving for by using these algorithms is keeping Head Movements (# tracks) to the least amount as possible. The less the head has to move the faster the seek time will be. I will show you and Lbplain to you why C-LOOK is the best algorithm to use in trying to establish less seek time.

Given the following queue -- 95, 180, 34, 119, 11, 123, 62, 64 with the Read-write head initially at the track 50 and the tail track being at 199 let us now discuss the different algorithms.

1. First Come -First Serve (FCFS)

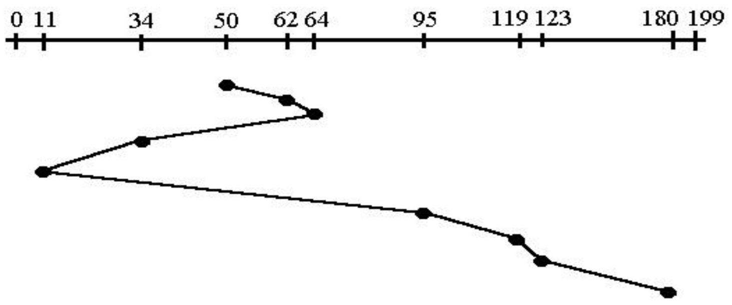
All incoming requests are placed at the end of the queue. Whatever number that is nextin the queue will be the nextnumber served.

Using this algorithm doesn't provide the best results. To determine the number of head movements you would simply find the number of tracks it took to move from one request to the nLbt. For this case it went from 50 to 95 to 180 and so on. From 50 to 95 it moved 45 tracks. If you tally up the total number of tracks you will find how many tracks it had to go through before finishing the entire request. In this example, it had a total head movement of 640 tracks. The disadvantage of this algorithm is noted by the oscillation from track 50 to track 180 and then back to track 11 to 123 then to 64. As you will soon see, this is the worse algorithm that one can use.



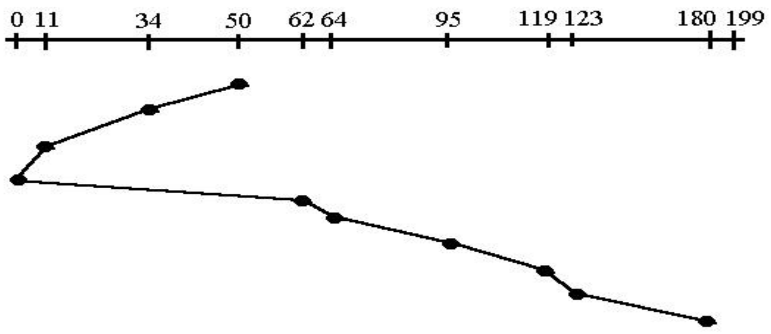
2. Shortest Seek Time First (SSTF)

In this case request is serviced according to next shortest distance. Starting at 50, the next shortest distance would be 62 instead of 34 since it is only 12 tracks away from 62 and 16 tracks away from 34. The process would continue until all the process are taken care of. For example the next case would be to move from 62 to 64 instead of 34 since there are only 2 tracks between them and not 18 if it were to go the other way. Although this seems to be a better service being that it moved a total of 236 tracks, this is not an optimal one. There is a great chance that starvation would take place. The reason for this is if there were a lot of requests close to each other the other requests will never be handled since the distance will always be greater.



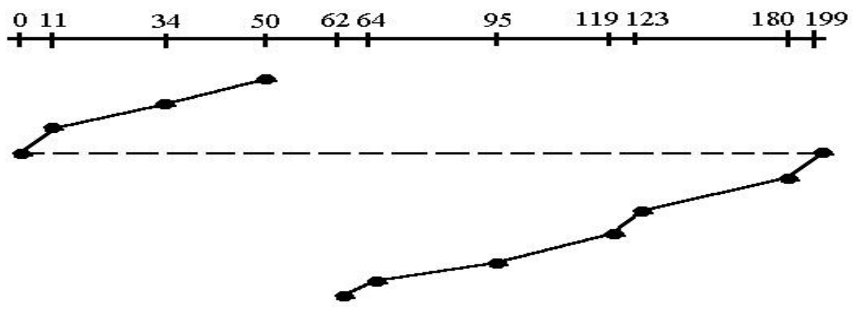
3. Elevator (SCAN)

This approach works like an elevator does. It scans down towards the nearest end and then when it hits the bottom it scans up servicing the requests that it didn't get going down. If a request comes in after it has been scanned it will not be serviced until the process comes back down or moves back up. This process moved a total of 230 tracks. Once again this is more optimal than the previous algorithm, but it is not the best.



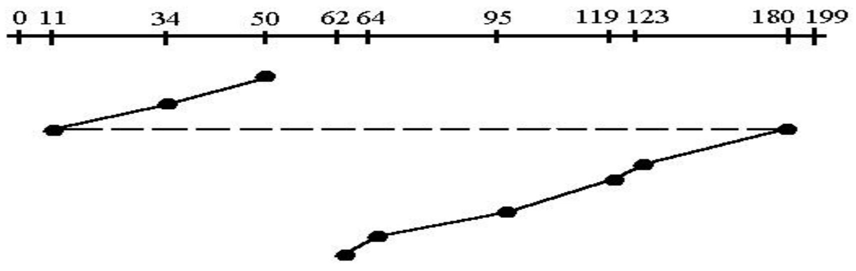
4. Circular Scan (C-SCAN)

Circular scanning works just like the elevator to some Lbtent. It begins its scan toward the nearest end and works it way all the way to the end of the system. Once it hits the bottom or top it jumps to the other end and moves in the same direction. Keep in mind that the huge jump doesn't count as a head movement. The total head movement for this algorithm is only 187 track, but still this isn't the mose sufficient.



5. C-LOOK

This is just an enhanced version of C-SCAN. In this the scanning doesn't go past the last request in the direction that it is moving. It too jumps to the other end but not all the way to the end. Just to the furthest request. C-SCAN had a total movement of 187 but this scan (C-LOOK) reduced it down to 157 tracks.



AIM:

To write a ‘C’ program to implement the Disk Scheduling algorithm for First Come First Served (FCFS), Shortest Seek Time First (SSTF), and SCAN.

PROBLEM DESCRIPTION:

Disk Scheduling is the process of deciding which of the cylinder request is in the ready queue is to be accessed next.

The access time and the bandwidth can be improved by scheduling the servicing of disk I/O requests in good order.

Access Time:

The access time has two major components: Seek time and Rotational Latency.

Seek Time:

Seek time is the time for disk arm to move the heads to the cylinder containing the desired sector.

Rotational Latency:

Rotational latency is the additional time waiting for the disk to rotate the desired sector to the disk head.

Bandwidth:

The disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

ALGORITHM:

Input the maximum number of cylinders and work queue and its head starting position.

First Come First Serve Scheduling (FCFS) algorithm – The operations are performed in order requested.

There is no reordering of work queue.

Every request is serviced, so there is no starvation.

The seek time is calculated.

Shortest Seek Time First Scheduling (SSTF) algorithm – This algorithm selects the request with the minimum seek time from the current head position.

Since seek time increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest to the current head position.

The seek time is calculated.

SCAN Scheduling algorithm – The disk arm starts at one end of the disk, and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk.

At the other end, the direction of head movement is reversed, and servicing continues.

The head continuously scans back and forth across the disk.

The seek time is calculated.

Display the seek time and terminate the program