MANIPAL
INSTITUTE OF TECHNOLOGY
A Constituent Institute of Manipal University, Manipal

DEPARTMENT OF INFORMATION & COMMUNICATION TECHNOLOGY
Manipal – 576 104

Operating Systems Lab [ICT-2265]

Fourth Semester

B. Tech. (IT/CCE)

CONTENTS

Course Objectives

- To describe the basics of Linux/Unix shell scripting.
- To implement operating system concepts.

Course Outcomes

At the end of this course, students will be able to

- Work on Unix and Unix like Operating Systems.
- Efficiently use the shell commands.
- Implement shell script on Unix and Unix like platforms.
- Write, compile and debug programs in C language on Unix and Unix like platforms.
- Implement of Operating System concepts using C language on Unix and Unix like platforms.

Evaluation plan

- Internal Assessment Marks : 60%
  - ✓ Continuous evaluation component (biweekly):10 marks
  - ✓ The assessment will depend on punctuality, program execution, maintaining the observation note and answering the questions in viva voce
- End semester assessment of 2 hour duration: 40%

INSTRUCTIONS TO THE STUDENTS

Pre- Lab Session Instructions

- Students should carry the Lab Manual Book and the required stationery to every lab session.
- Be in time and follow the institution dress code.
- Must Sign in the log register provided.
- Make sure to occupy the allotted seat and answer the attendance.
- Adhere to the rules and maintain the decorum.

In- Lab Session Instructions

- Follow the instructions on the allotted exercises.
- Show the program and results to the instructors on completion of experiments.
- On receiving approval from the instructor, copy the program and results in the Lab record.
- Prescribed textbooks and class notes can be kept ready for reference if required.
- Implement the lab exercises on UNIX or other Unix like platform. Use C for high level language implementation.

General Instructions for the exercises in Lab

- Implement the given exercise individually and not in a group.
- The programs should meet the following criteria:
    - ✓ Programs should be interactive with appropriate prompt messages, error messages if any, and descriptive messages for outputs.
    - ✓ Programs should perform input validation (Data type, range error, etc.) and give appropriate error messages and suggest corrective actions.
    - ✓ Comments should be used to give the statement of the problem and every function should indicate the purpose of the function, inputs and outputs.
    - ✓ Statements within the program should be properly indented.
    - ✓ Use meaningful names for variables and functions.
    - ✓ Make use of constants and type definitions wherever needed.

- Plagiarism (copying from others) is strictly prohibited and would invite severe penalty in evaluation.
- The exercises for each week are divided under three sets:
  - ✓ Solved exercise
  - ✓ Lab exercises - to be completed during lab hours
  - ✓ Additional Exercises - to be completed outside the lab or in the lab to enhance the skill
- If a student misses a lab class then he/ she must ensure that the experiment is completed during the repetition class in case of genuine reason (medical certificate approved by HOD) with the permission of the faculty concerned
- Questions for lab tests and examination are not necessarily limited to the questions in the manual, but may involve some variations and / or combinations of the questions.
- A sample note preparation is given as a model for observation.

THE STUDENTS SHOULD NOT

- Bring mobile phones or any other electronic gadgets to the lab.
- Go out of the lab without permission.

LAB NO: 1 **Date:**

UNIX SHELL COMMANDS

Objective:

1. To recall the UNIX special characters and commands.
2. To describe basic commands.

1. UNIX shell and special characters

A shell is an environment in which we can run our commands, programs, and scripts. There are different flavors of shells, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

Shell Prompt:

The prompt, $, which is called command prompt, is issued by the shell. While the prompt is displayed, you can type a command. The command is a binary executable. Once the Enter key is pressed, the shell reads the command line arguments and performs accordingly. It determines the command to be executed by looking for input executable name placed in standard location (ex: /usr/bin). Multiple arguments can be provided to the command (executable) separated by spaces.

Following is a simple example of date command which displays current date and time:

$date

Thu Jun 25 08:30:19 MST 2009

Shell Types:

In UNIX there are two major types of shells:

1. The Bourne shell. If you are using a Bourne-type shell, the default prompt is the $ character.
2. The C shell. If you are using a C-type shell, the default prompt is the % character.

There are again various subcategories for Bourne Shell which are listed as follows:

- Bourne shell (sh)
- Korn shell (ksh)
- Bourne again shell (bash)
- POSIX shell (sh)

The different C-type shells follow:

- C shell (csh)
- TENEX/TOPS C shell (tosh)

The original UNIX shell was written in the mid-1970s by Stephen R. Bourne while he was at AT&T Bell Labs in New Jersey. The Bourne shell was the first shell to appear on UNIX systems, thus it is referred to as "the shell". The Bourne shell is usually installed as /bin/sh on most versions of UNIX. For this reason, it is the shell of choice for writing scripts to use on several different versions of UNIX.

Special Characters:

Before we continue to learn about UNIX shell commands, it is important to know that there are many symbols and characters that the shell interprets in special ways. This means that certain type of characters: a) cannot be used in certain situations, b) may be used to perform special operations, or, c) must be “escaped” if you want to use them in a normal way.

Character	Description
\	Escape character. If you want to refer a special character, you must "escape" it with a backslash first. Example*: touch /tamp/filename**
/	Directory separator, used to separate a string of directory names. Example*: /usr/src/unix*
.	Current directory. Can also "hide" files when it is the first character in a filename.
..	Parent directory
~	User's home directory
*	Represents 0 or more characters in a filename, or by itself, all files in a directory. Example: pic*2002 can represent the files pic2002, picJanuary2002, picFeb292002, etc.
?	Represents a single character in a filename. Example: hello?.txt can represent hello1.txt, helloz.txt, but not hello22.txt
[ ]	Can be used to represent a range of values, e.g. [0-9], [A-Z], etc. Example: hello[0-2].txt represents the names hello0.txt, hello1.txt, and hello2.txt
|	"Pipe". Redirect the output of one command into another command. Example*: ls | more*
>	Redirect the output of a command into a new file. If the file already exists, over-write it. Example: *ls > myfiles.txt*
>>	Redirect the output of a command onto the end of an existing file. Example: *echo "Mary 555-1234" >> phonenumbers.txt*
<	Redirect a file as input to a program. Example: more < phonenumbers.txt
<<	Reads from a stream literal (an inline file, passed to the standard input) Example: *tr a-z A-Z << END_TEXT* *This is OS lab manual* *For IT students* *END_TEXT*

<<<	Reads from a string. Example: *bc <<< 9+5*
;	Command separator. Allows you to execute multiple commands on a single line. Example: *cd /var/log ; less* messages
&&	Command separator as above, but only runs the second command if the first one finished without errors. Example*: cd /var/logs && less* messages
&	Execute a command in the background, and immediately get your shell back. Example: *find / -name core > /tmp/corefiles.txt &*

2. Shell commands and getting help

Executing Commands

Most common commands are located in your shell's "PATH", meaning that you can just type the name of the program to execute it. Example: typing *ls* will execute the *ls* command. Your shell's "PATH" variable includes the most common program locations, such as /bin, /usr/bin, /usr/X11R6/bin, and others. To execute commands that are not in your current PATH, you have to give the complete location of the command. [PATH is an environmental variable. To display the value of PATH variable execute *echo $PATH*]

Examples: /home/bob/myprogram

./program (Execute a program in the current directory)

~/bin/program (Execute program from a personal bin directory)

[Before executing the program, the program file has to be granted with execution permission. For granting execute permission the command *chmod +x* has to be executed.]

Command Syntax

When interacting with the UNIX operating system, one of the first things you need to know is that, unlike other computer systems you may be accustomed to, everything in UNIX is case-sensitive. Be careful when you're typing in commands - whether a character is upper or lower case does make a difference. For instance, if you want to list your files with the *ls* command, if you enter LS you will be told "command not found". Commands

can be run by themselves, or you can pass in additional arguments to make them do different things. Each argument to the command should be separated by space. Typical command syntax can look something like this:

command [-argument] [-argument] [--argument] [file]

Examples:	*ls*	#List files in current directory
	ls -l	#Lists files in "long" format
	ls -l --color	#As above, with colorized output
	cat filename	#Show contents of a file
	cat -n filename	#Show contents of a file, with line numbers

Getting Help

When you're stuck and need help with a UNIX command, help is usually only a few keystrokes away! Help on most UNIX commands is typically built right into the commands themselves, available through online help programs ("man pages" and "info pages"), and of course online.

Many commands have simple "help" screens that can be invoked with special command flags. These flags usually look like *-h* or *--help*. Example: grep –help. "Man Pages" are the best source of information for most commands can be found in the online manual pages. To display a command's manual page, type man <commandName>.

Examples:	*man ls*	Get help on the "ls" command.
	man man	A manual about how to use the manual!

To search for a particular word within a man page, type /<*word*>. To quit from a man page, just type the "Q" (or q) key.

Sometimes, you might not remember the name of UNIX command and you need to search for it. For example, if you want to know how to change a file's permissions, you can search the man page descriptions for the word "permission" like this: *man -k permission.* All matched manual page names and short descriptions will be displayed that includes the keyword "permission" as regular expression.

3. Commands for Navigating the UNIX file systems

The first thing you usually want to do when learning about the UNIX file system is take some time to look around and see what's there! These next few commands will: a) Tell you where you are, b) take you somewhere else, and c) show you what's there. The following are the various commands used for UNIX file system navigation. Note

the words enclosed in angular brackets (<>) represents user defined arguments and should be replaced with actual arguments. Example: ls <dirName> should be replaced with actual existing directory name such as ls ABC.

a. **pwd *(*"Print Working Directory"):** Shows the current location in the directory tree.

b. **cd ("Change Directory"):** When typed all by itself, it returns you to your home directory. Few of the arguments to cd are:
   i. **cd <dirName>:** changes current path to the specified directory name. Example: *cd /usr/src/unix*
   ii. **cd ~ :** "~" is an alias for your home directory. It can be used as a shortcut to your "home", or other directories relative to your home
   iii. **cd ..:** Move up one directory. For example, if you are in /home/vic and you type *cd ..*, you will end up in /home. Note: there should be space between *cd* and .. .
   iv. **cd -:** Return to previous directory. An easy way to get back to your previous location!

c. **ls:** List all files in the current directory, in column format. Few of the arguments for ls command are as follows:
   i. **ls <dirName>:** List the files in the specified directory. Example*: ls /var/log*
   ii. **ls -l:** List files in "long" format, one file per line. This also shows you additional info about the file, such as ownership, permissions, date, and size.
   iii. **ls –a:** List all files, including "hidden" files. Hidden files are those files that begin with a ".",
   iv. **ls –ld < dirName >:** A "long" list of "directory", but instead of showing the directory contents, show the directory's detailed information. For example, compare the output of the following two commands: ls -l /usr/bin ls -ld /usr/bin
   v. **ls /usr/bin/d*:** List all files whose names begin with the letter "d" in the /usr/bin directory.

3.1 Filenames, Wildcards, and Pathname Expansion

Sometimes you need to run a command on more than one file at a time. The most common example of such a command is ls, which lists information about files. In its simplest form, without options or arguments, it lists the names of all files in the working directory except special hidden files, whose names begin with a dot (.). If you give ls filename arguments, it will list those files—which is sort of silly: if your current directory has the files duchess and queen in it and you type ls duchess queen, the system will simply print those filenames. But sometimes you want to verify the existence of a certain group of files without having to know all of their names; for example, if you use a text editor, you might want to see which files in your current directory have names that end in .txt. Filenames are so important in UNIX that the shell provides a built-in way to specify the pattern of a set of filenames without having to know all of the names themselves. You can use special characters, called wildcards, in filenames to turn them into patterns. The following provides the list of the basic wildcards.

Wildcard	**Matches**
?	Any single character
*	Any string of characters
[set]	Any character in set
[!set]	Any character not in set

Example:

$ls

bob darlene dave ed frank fred program.log program.o program.c

$ls program.?

program.o program.c

*$ls fr**

frank fred

*$ls *ed*

ed fred

*$ls *r**

darlene frank fred

*$ls g**

ls: cannot access g*: No such file or directory

The remaining wildcard is the set construct. A set is a list of characters (e.g., abc), an inclusive range (e.g., a-z), or some combination of the two. If you want the dash character to be part of a list, just list it first or last.

Using the set construct wildcards are as follows:

Expression	Matches
[abc]	a, b, or c
[.,;]	Period, comma, or semicolon
[-_]	Dash or underscore
[a-c]	a, b, or c
[a-z]	All lowercase letters
[!0-9]	All non-digits
[0-9!]	All digits and exclamation point
[a-zA-Z]	All lower- and uppercase letters
[a-zA-Z0-9_-]	All letters, all digits, underscore, and dash

In the original wildcard example, program.[co] and program.[a-z] both match program.c and program.o, but not program.log. An exclamation point after the left bracket lets you "negate" a set. For example, [!.;] matches any character except period and semicolon; [!a-zA-Z] matches any character that isn't a letter. To match “!” itself, place it after the first character in the set, or precede it with a backslash, as in [\!].

The range notation is handy, but you shouldn't make too many assumptions about what characters are included in a range. It's safe to use a range for uppercase letters, lowercase letters, digits, or any subranges thereof (e.g., [f-q], [2-6]). Don't use ranges on punctuation characters or mixed-case letters: e.g., [a-Z] and [A-z] should not be trusted to include all of the letters and nothing more.

The process of matching expressions containing wildcards to filenames is called wildcard expansion or globbing. This is just one of several steps the shell takes when reading and processing a command line; another that we have already seen is tilde expansion, where tildes are replaced with home directories where applicable.

However, it's important to be aware that the commands that you run only see the results of wildcard expansion. That is, they just see a list of arguments, and they have no

knowledge of how those arguments came into being. For example, if you type ls fr* and your files are as on the previous page, then the shell expands the command line to ls fred frank and invokes the command ls with arguments fred and frank. If you type ls g*, then (because there is no match) ls will be given the literal string g* and will complain with the error message, g*: No such file or directory. This is different from the C shell's wild-card mechanism, which prints an error message and doesn't execute the command at all.

The wildcard examples that we have seen so far are actually part of a more general concept called pathname expansion. Just as it is possible to use wildcards in the current directory, they can also be used as part of a pathname. For example, if you wanted to list all of the files in the directories /usr and /usr2, you could type *ls /usr**. If you were only interested in the files beginning with the letters b and e in these directories, you could type *ls /usr*/[be]** to list them.

4. Working With Files and Directories

These commands can be used to: find out information about files, display files, and manipulate them in other ways (copy, move, delete). The various commands used for working with files and directories are:

a. **touch:** changes the file timestamps, if the file does not exists then this command creates an empty file. Example: *touch abc xyz mno* creates three empty files in the current directory
b. **file:** Find out what kind of file it is. For example, *file /bin/ls* tells us that it is a UNIX executable file.
c. **cat:** Display the contents of a text file on the screen. For example: *cat file.txt* would displays the file content.
d. **head:** Display the first few lines of a text file. Example: *head /etc/services*
e. **tail:** Display the last few lines of a text file. Example: *tail /etc/services*. *tail -f* displays the last few lines of a text file.
f. **cp:** Copies a file from one location to another. Example: *cp mp3files.txt /tmp* (copies the mp3files.txt file to the /tmp directory)
g. **mv:** Moves a file to a new location, or renames it. For example: *mv mp3files.txt /tmp* (copy the file to /tmp, and delete it from the original location)
h. ***rm*:** Delete a file. Example: *rm /tmp/mp3files.txt*

i. ***mkdir*:** Make Directory. Example: *mkdir /tmp/myfiles/* creates a folder named myfiles in /tmp folder.

j. ***rmdir*:** Remove Directory. rmdir will only remove directory when it is empty. Use of *rm -R* will remove the directory as well as any files and subdirectories as long as they are not in use. Be careful though, make sure you specify the correct directory or you can remove a lot of stuff quickly.
Example*: rmdir /tmp/myfiles/*

5. Commands used for Finding Things

The following commands are used to find files. *ls* is good for finding files if you already know approximately where they are, but sometimes you need more powerful tools such as these:

a. **which:** Shows the full path of shell commands found in your path. For example, if you want to know exactly where the *grep* command is located on the file system, you can type *which grep*. The output should be something like: /bin/grep

b. **whereis:** Locates the program, source code, and manual page for a command (if all information is available). For example, to find out where *ls* and its man page are, type: *whereis ls.* The output will look something like: ls: /bin/ls /usr/share/man/man1/ls.1.gz

c. **locate:** A quick way to search for files anywhere on the file system. For example, you can find all files and directories that contain the name *mozilla* by typing: *locate mozilla*

d. **find:** A very powerful command, but sometimes tricky to use. It can be used to search for files matching certain patterns, as well as many other types of searches. A simple example is: *find . -name *.sh*. This example starts searching in the current directory and all subdirectories, looking for files with *sh* at the end of their names.

6. Piping and Re-Direction

Before we move on to learning even more commands, let's side-track to the topics of piping and re-direction. The basic UNIX philosophy, therefore by extension the UNIX philosophy, is to have many small programs and utilities that do a particular job very well. It is the responsibility of the programmer or user to combine these utilities to make more useful command sequences.

6.1 Piping Commands Together

The pipe character, | is used to chain two or more commands together. The output of the first command is *piped* into the next program, and if there is a second pipe, the output is sent to the third program, etc. For example: *ls -la /usr/bin* | less lists the files one screen at a time

6.2 Redirecting Program Output to Files

There are times when it is useful to save the output of a command to a file, instead of displaying it to the screen. For example, if we want to create a file that lists all of the MP3 files in a directory, we can do something like this, using the > redirection character. Example: *ls -l /home/vic/MP3/*.mp3 > mp3files.txt* creates a new file and copies the output of the listing. A similar command can be written so that instead of creating a new file called mp3files.txt, we can append to the end of the original file*:ls -l /home/vic/extraMP3s/*.mp3 >> mp3files.txt*

7. Shortcuts

a. *ctrl+c* Halts the current command
b. *ctrl+z* Stops the current command,
c. *ctrl+d* Logout the current session, similar to exit
d. *ctrl+w* Erases one word in the current line
e. *ctrl+u* Erases the whole line
f. *!!* Repeats the last command
g. *exit* Logout the current session

Lab Exercises

1. Execute and write output of all the commands explained so far in this manual.
2. Explore the following commands along with their various options. (Some of the options are specified in the bracket)
    a. *cat* (variation used to create a new file and append to existing file)
    b. *head* and *tail* (-n, -c)
    c. *cp* (-n, -i, -f)
    d. *mv* (-f, -i) [try (i) mv dir1 dir2 (ii) mv file1 file2 file3 ... directory]
    e. *rm* (-r, -i, -f)
    f. *rmdir* (-r, -f)
    g. *find* (-name, -type)

3. List all the file names satisfying following criteria
    a. has the extension .txt.
    b. containing atleast one digit.
    c. having minimum length of 4.
    d. does not contain any of the vowels as the start letter.

LAB NO: 2 **Date:**

ADVANCED UNIX SHELL COMMANDS

Objectives:

1. To know the UNIX shell, special characters and commands.
2. To describe Unix commands.

1. Commands used to extract, sort, filter and process data

a. grep

grep is a command-line utility for searching plain-text data sets for lines matching a regular expression. grep was originally developed for the UNIX operating system, but is available today for all UNIX-like systems. Its name comes from the ed command g/re/p (globally search a regular expression and print), which has the same effect: doing a global search with the regular expression and printing all matching lines. Use grep –help to know the possible arguments for grep.

grep <someText> <fileName> #search, case sensitive, for <someText> in <file-name>, use -i for case insensitive search

grep -r <text> <folderName>/ # search for file names with occurrence of the text

With regular expressions:

grep -E ^<text> <fileName> #search start of lines with the word text

grep -E <0-4> <fileName> #shows lines containing numbers 0-4

grep –E <a-zA-Z> <fileName> # retrieve all lines with alphabetical letters.

Usage

Grep <word> <filename>

grep <word> file1 file2 file3

grep < word1> <word2> filename

cat <otherfile> | grep <word>

command | *grep* <something>

grep --color < word> <filename>

grep text * # * stands for all files in current directory

Examples:

$ *cat fruitlist.txt*

apple

apples
pineapple
fruit-apple
banana
pear
peach
orange

$ grep apple fruitlist.txt
apple
apples
pineapple
fruit-apple

$ grep -x apple fruitlist.txt # match whole line
apple

$ grep ^p fruitlist.txt
pineapple
pear
peach
$ grep -v apple fruitlist.txt #print unmatched lines
banana
pear
peach
orange

b. sort

sort is a program that prints the lines of its input or concatenation of all files listed in its argument list in sorted order. Sorting is done based on one or more sort keys extracted from each line of input. By default, the entire input is taken as sort key. Blank space is the default field separator. Note: Sort doesn't modify the input file content.

sort <any number of filenames> #sort the content of file(s)
sort <fileName> #sort alphabetically
sort -o <file> <outputFile> #write result to a file

sort -r <fileName> #sort in reverse order
sort -n <fileName> #sort numbers

c. wc (word count)

This shell command can be used to print the number of lines words in the input file/s.

$wc <fileName> #Number of lines, number of words, byte size of <fileName>.

Other arguments includes: -l (lines), -w (words), -c (byte size), -m

$ wc * : counts for all files in the current directory.

d. cut

cut is a data filter: it extracts columns from tabular data. If you supply the numbers of columns you want to extract from the input, cut prints only those columns on the standard output. Columns can be character positions or—relevant in this example—fields that are separated by TAB characters (default delimiter) or other delimiters.

Examples

ls –l | cut –d "-" -f 2 #-d specifies the delimiter and default delimiter is tab. The permission for the group for the files can be obtained using this statement

cut –c 1-3 record.txt # -c specifies characters to be extracted

cut -c1,4,7 record.txt #characters 1, 4, and 7

cut -c1-3,8 record.txt #characters 1 thru 3, and 8

cut -c3- record.txt #characters 3 thru last

cut –f 1,4,7 record.txt #tab-separated fields 1, 4, and 7 #-f specifies fields to be extracted.

e.sed (stream editor)

sed performs basic text transformations on an input stream (a file or input from a pipeline) in a single pass through the stream, so it is very efficient. However, it is sed's ability to filter text in a pipeline which particularly distinguishes it from other types of editor.

***sed* basics**

sed can be used at the command-line, or within a shell script, to edit a file non-interactively. Perhaps the most useful feature is to do a 'search-and-replace' for one string to another. You can embed your sed commands into the command-line that invokes sed using the '-e' option, or put them in a separate file e.g. 'sed.in' and invoke sed using

the '-f sed.in' option. This latter option is most used if the sed commands are complex and involve lots of regexps!

For instance*: sed -e 's/input/output/' my_file*

will echo every line from my_file to standard output, changing the first occurrence of 'input' on each line into 'output'. sed is line-oriented, so if you wish to change every occurrence on each line, then you need to make it a 'greedy' search & replace like so:

sed -e 's/input/output/g' my_file # g stands for global

By default the output is written to stdout. You may redirect this to a new file, or if you want to edit the existing file in place you should use the *-i* flag:

sed -e 's/input/output/' my_file > new_file

sed -i -e 's/input/output/' my_file

sed and regexps

What if one of the characters you wish to use in the search command is a special symbol, like / (e.g. in a filename) or * etc? Then you must escape the symbol just as for grep (and awk). Say you want to edit a shell scripts to refer to /usr/local/bin and not /bin any more, then you could do this

sed -e 's/\/bin/\/usr\/local\/bin/' my_script > new_script

What if you want to use a wildcard as part of your search – how do you write the output string? You need to use the special symbol '&' which corresponds to the pattern found. So say you want to take every line that starts with a number in your file and surround that number by parentheses:

sed -e 's/[0-9]/(&)/' my_file*

where [0-9] is a regexp range for all single digit numbers, and the '*' is a repeat count, means any number of digits.

Other _sed_ commands

The general form is *sed -e '/pattern/ command' my_file***,** where 'pattern' is a regexp and 'command' can be one of 's' = search & replace, or 'p' = print, or 'd' =delete, or 'i'=insert, or 'a'=append, etc. Note that the default action is to print all lines that do not match anyway, so if you want to suppress this you need to invoke sed with the '-n' flag and then you can use the 'p' command to control what is printed. So if you want to do a listing of all the subdirectories you could use below statement, as ls –l includes “d” at the start while listing the subdirectories.

ls -l | sed -n -e '/^d/ p'

Below statement, deletes all lines that start with the comment symbol '#' in my_file.

sed -e '/^#/ d' my_file

To insert a new line after a matching pattern is found the option “a” is used

sed –i ‘/word/a “xyz”’ filename

You can also use the range form

sed -e '1,100 command' my_file

to execute 'command' on lines 1,100. You can also use the special line number '$' to mean 'end of file'. For example below statement deletes all but the first 10 lines of a file.

sed -e '11,$ d' my_file

f. **tr (translate)**

The *tr* filter is used to translate one set of characters from the standard inputs to another.

Examples:

$tr “[a-z]” “[A-Z]” < filename #maps all lowercase characters in filename to uppercase. Content of the file is not changed.

tr 'abcd' 'jkmn' #maps all characters a to j, b to k, c to m, and d to n.

The character set may be abbreviated by using character ranges. The previous example could be written: *tr 'a-d' 'jkmn'*

The s flag (suppress) causes tr to compress sequences of identical adjacent characters in its output to a single token. For example,

tr -s '\n' #replaces sequences of one or more newline characters with a single newline.

The d flag causes tr to delete all tokens of the specified set of characters from its input. The *tr -d '\r'* statement removes carriage return characters.

The c flag indicates the complement of the first set of characters. The invocation *tr -cd '[:alnum:]'* therefore removes all non-alphanumeric characters.

2. Process management commands

a. ps

The *ps* command (short for "process status") displays the currently-running processes. *ps* command displays process id (PID), TTY (Terminal associated with the process), time The amount of CPU time used by the process and command name (CMD). For example:

$ *ps*

PID TTY TIME CMD

7431	pts/0	00:00:00	su
7434	pts/0	00:00:00	bash
18585	pts/0	00:00:00	ps

b. kill

kill is a command that is used in several popular operating systems to send signals to running processes in order to request the termination of the process. The signals in which users are generally most interested are SIGTERM and SIGKILL. The SIGTERM signal is sent to a process to request its termination. Unlike the SIGKILL signal, it can be caught and interpreted or ignored by the process. This allows the process to perform nice termination releasing resources and saving state if appropriate. The SIGKILL signal is sent to a process to cause it to terminate immediately (kill). This signal cannot be caught or ignored, and the receiving process cannot perform any clean-up upon receiving this signal.

Examples:

A process can be sent a SIGTERM signal in four ways (the process ID is '1234' in this case):

```
kill 1234
kill -s TERM 1234
kill -TERM 1234
kill -15 1234
```

The process can be sent a SIGKILL signal in three ways:

```
kill -s KILL 1234
kill -KILL 1234
kill -9 1234
```

3. <u>File permission commands</u>

a. **chmod (Change mode)**

The *chmod* numerical format accepts up to four octal digits. The three rightmost digits refer to permissions for the file owner, the group, and other users.

Numerical permissions

#	Permission	rwx
7	read, write and execute	rwx
6	read and write	rw-
5	read and execute	r-x

4	read only	r--
3	write and execute	-wx
2	write only	-w-
1	execute only	--x
0	none	---

The *chmod* command also accepts a finer-grained symbolic notation, which allows modifying specific modes while leaving other modes untouched. The symbolic mode is composed of three components, which are combined to form a single string of text:

$ chmod [references][operator][modes] file ...

The references (or classes) are used to distinguish the users to whom the permissions apply. If no references are specified it defaults to "all" but modifies only the permissions allowed by the umask. The references are represented by one or more of the following letters:

Reference	**Class**	**Description**
u	user	the owner of the file
g	group	users who are members of the file's group
o	others	users who are neither the owner of the file nor members of the file's group
a	all	all three of the above, same as ugo

The chmod program uses an operator to specify how the modes of a file should be adjusted. The following operators are accepted:

Operator	Description
+	adds the specified modes to the specified classes
-	removes the specified modes from the specified classes
=	the modes specified are to be made the exact modes for the specified classes

The modes indicate which permissions are to be granted or removed from the specified classes. There are three basic modes which correspond to the basic permissions:

Mode	Name	Description
r	read	read a file or list a directory's contents
w	write	write to a file or directory
x	execute	execute a file or recurse a directory tree

Command	Explanation
chmod a+r Comments.txt	read is added for all classes (i.e. User, Group and Others)
chmod +r Comments.txt	omitting the class defaults to all classes, but the resultant permissions are dependent on umask.
chmod a-x Comments.txt	execute permission is removed for all classes.
chmod a+rx viewer.sh	add read and execute for all classes.
chmod u=rw,g=r,o= Plan.txt	user(i.e. owner) can read and write, group can read, others cannot access.
chmod -R u+w,go-w docs	add write permissions to the directory docs and all its contents (i.e. recursively) for user and deny write access for everybody else.
chmod ug=rw groupAgreements.txt	user and group members can read and write (update the file).
chmod 664 global.txt	sets read and write and no execution access for the user and group, and read, no write, no execute for all others.

chmod 0744 myCV.txt	equivalent to u=rwx (400+200+100),go=r (40+ 4). The 0 specifies no special modes.

4. Other useful commands

a. **echo**

This is one of the most commonly and widely used built-in command, that typically used in scripting language and batch files to display a line of text/string on standard output or a file. This command writes its arguments to standard output. Example: echo this is OS lab manual, prints the input string on the terminal. It is not necessary to surround the strings with quotes, as it does not affect what is written on the screen. If quotes (either single or double) are used, they are not repeated on the screen.

b. bc (Basic Calculator)

After this command *bc* is started and it waits for your commands, example:

$bc (hit *enter* key)

5 + 2

7 #7 is response of bc i.e. addition of 5 + 2 you can even try

5 / 2

2 # to perform floating point operations use *bc -l*

5 > 2

1. #0 (Zero) is response of bc, How? Here it compare 5 with 2 as, Is 5 is greater than 2, (If I ask same-question to you, your answer will be YES) In UNIX (bc) gives this 'YES' answer by showing 0 (Zero) value.

2.

The vi editor

The *vi* editor is a visual editor used to create and edit text, files, documents and programs. It displays the content of files on the screen and allows a user to add, delete or change part of text. There are three modes available in the *vi* editor, they are

i. Command mode
ii. Input (or) insert mode.

Starting vi :

The *vi* editor is invoked by giving the following commands in UNIX prompt.

Syntax : *$vi <filename> (or) $vi*

This command would open a display screen with 25 lines and with tilt (~) symbol at the start of each line. The first syntax would save the file in the filename mentioned and for the next the filename must be mentioned at the end.

Options :

1.*vi +n <filename>* - this would point at the nth line (cursor pos).

2.*vi –n <filename>* - This command is to make the file to read only to change from one mode to another press escape key.

Saving and Quitting from vi

To move editor from command node to edit mode, you have to press the <ESC> key.

<ESC> w Command	To save the given text present in the file.
<ESC> q! Command	To quit the given text without saving.
<ESC> wq Command	This command quits the vi editor after saving the text in the mentioned file.
<ESC> x Command	This command is same as "wq" command it saves and quit.
<ESC> q Command	This command would quit the window but it would ask for again to save the file.

Lab Exercises

1. Execute all the commands explained in this section and write the output.
2. Write grep commands to do the following activities:
   - To select the lines from a file that have exactly two characters.
   - To select the lines from a file that start with the upper case letter.
   - To select the lines from a file that end with a period.
   - To select the lines in a file that has one or more blank spaces.
   - To select the lines in a file and direct them to another file which has digits as one of the characters in that line.
3. Create file studentInformation.txt using *vi* editor which contains details in the following format.

RegistrationNo:Name:Department:Branch:Section:Sub1:Sub2:Sub3

1234:XYZ:ICT:CCE:A:80:60:70 … (add atleast 10 rows)

   i) Display the number students(only count) belonging to ICT department.
   ii) Replace all occurrences of IT branch with "Information Technology" and save the output to ITStudents.txt
   iii) Display the average marks of student with the given registration number "1234" (or any specific existing number).

iv) Display the title row in uppercase. The remaining lines should be unchanged.
Example:
REGISTRATIONNO:NAME:DEPARTMENT:BRANCH:SECTION:…
1234:XYZ:ICT:CCE:A:10:30:50 … (Hint: use ; for running multiple commands)

3. List all the files containing "MIT" in the current folder. Also display the lines containing MIT being replaced with Manipal Institute of Technology. (Hint: use grep, cut & sed)
4. Write a shell command to display the number of lines, characters, words of files containing a digit in its name.
5. Run *wc* command in the background many times using *wc &*. Kill all the processes named *wc*.

Additional Exercises

1. Write a *sed* command that deletes the character before the last character in each line in a file.
2. Write a shell command to count the number lines containing digits present in a file.

LAB NO: 3 **Date:**

UNIX SHELL PROGRAMMING (SHELL SCRIPTING)

Objectives:

1. To recall the Unix shell programming.
2. To identify System variables.

1. The UNIX shell programming

Shell programming is a group of commands grouped together under single filename. After logging onto the system a prompt for input appears which is generated by a Command String Interpreter program called the shell. The shell interprets the input, takes appropriate action, and finally prompts for more input. The shell can be used either interactively - enter commands at the command prompt, or as an interpreter to execute a shell script. Shell scripts are dynamically interpreted, NOT compiled.

2. shbang line, comments, wildcards and keywords

The shbang line "shbang" line is the very first line of the script and lets the kernel know what shell will be interpreting the lines in the script. The shbang line consists of a #!, followed by the full pathname to the shell, and can be followed by options to control the behavior of the shell.

E*XAMPLE* #!/bin/sh

Comments Comments are descriptive material preceded by a # sign. They are in effect until the end of a line and can be started anywhere on the line.

E*XAMPLE* # this text is not interpreted by the shell

Wildcards There are some characters that are evaluated by the shell in a special way. They are called shell meta characters or "wildcards". These characters are neither numbers nor letters. For example, the *, ?, and [ ] are used for filename expansion.

The <, >, 2>, >>, and | symbols are used for standard I/O redirection and pipes. To prevent these characters from being interpreted by the shell they must be quoted.

E*XAMPLE*

Filename expansion:

rm *; ls ??; cat file[1-3];

Quotes protect metacharacters:

echo "How are you?"

Shell keywords :

Some of the shell keywords are echo, read, if fi, else, case, esac, for, while, do, done, until, set, unset, readonly, shift, export, break, continue, exit, return, trap , wait, eval ,exec, ulimit , umask.

3. shell variables, expressions and statements

Shell variables change during the execution of the program.

Variable naming rules:

- ➢ A variable name is any combination of alphabets, digits and an underscore („-„);
- ➢ No commas or blanks are allowed within a variable name.
- ➢ The first character of a variable name must either be an alphabet or an underscore.
- ➢ Variables names should be of any reasonable length.
- ➢ Variables name are case sensitive. That is, Name, NAME, name, Name, are all different variables.

Local variables are in scope for the current shell. When a script ends, they are no longer available; i.e., they go out of scope. Local variables are set and assigned values.

EXAMPLE

variable_name=value
name="John Doe"
x=5

Global variables are called environment variables. They are set for the currently running shell and any process spawned from that shell. They go out of scope when the script ends.
EXAMPLE

```
VARIABLE_NAME=value
export VARIABLE_NAME
PATH=/bin:/usr/bin:.
export PATH
```

Extracting values from variables: To extract the value from variables, a dollar sign is used.

EXAMPLE [here, echo command is used display the variable value]

```
echo $variable_name
echo $name
echo $PATH
```

4. Shell input and output

Input:

To get the input from the user *read* is used.

Syntax : *read* x y #no need of commas between variables

The read command takes a line of input from the user and assigns it to a variable(s) on the right-hand side. The read command can accept multiple variable names. Each variable will be assigned a word. No need to declare the variables to be read from user

Output :

echo can be used to display the results. Wildcards must be escaped with either a backslash or matching quotes.

Syntax :

echo “Enter the value of b” (or) echo Value of b is $b(for variable).

5. Basic Arithmetic operations

The shell does not support arithmetic operations directly (ex: a=b+c). UNIX/UNIX commands must be used to perform calculations.

Command	**Syntax**	**Example**
expr	*expr* expression operators: + , - , /, %,=,==,!=	*a=$(expr $a + 1)* *a=`expr $a + 1`* space should not be present between = and *expr*. Space should be present between operator and operands. To access values $ has to be used for operands. Performs only integer arithmetic operations.
test [ ]	[condition/expression] Note one space should be present after [and before]. Also operand and operator must be separated by a space. *operators*: **Integers**: -eq, -ne, -gt,-lt,-ge, -le, ==, != **Boolean**: !, -o(or), -a(and) **String**: =, !=,-z(zero length), -n(non-zero length), [$str] (true if $str is not empty) **File**: -f (ordinary file), -d (directory), -r (readable), -w, -x, -s (size is zero), -e (exists)	*echo "Enter Two Values"* *read a b* *result=$[a == b]* *echo "Check for Equality $result"* **O/P:** Enter Two Values 4 4 Check for Equality 1 *test* works in combination with control structures refer *section 6*.

test (( ))	Performs integer arithmetic. Here spacing does not matter also we need not include $ for the variables. Useful in performing increment or detriment operations.	*echo "Enter the two values"* *read a b* *echo "enter operator(+, -, /, % *)"* *read op* *((a++))* *result=$((a $op b))* *echo "Result of performing $a $op $b is $result"* **O/P:**Enter the two values 4 6 enter operator(+, -, /, % *) * Result of performing 5 * 6 is 30
bc	refer section 4 of Lab 2. *bc* can be used to perform floating point operations.	*echo "Enter the two values"* *read a b* *echo "Enter operator(+, -, /, % *)"* *read op* *result=`bc -l <<<$a\$op$b`* # or use result=`*echo "$a $op $b" | bc -l*` *echo " Result of performing $a $op $b is $result"* **O/P:**Enter the two values 4 5 Enter operator(+, -, /, % *) * Result of performing 4 * 5 is 20

6. Control statements

The shell control structure is similar to C syntax, but instead of brackets {} statements like *then- fi* or *do-done* are used. The *then, do* has to be used in next line, otherwise *;* has to be used to mark the next line.

Control Structure	**Syntax**	**Example**
if	*if* condition ; *then* command(s) *fi* *OR* *if* condition *then* command(s) *fi*	*read character* *if ["$character" = "2"]; then* *echo " You entered two."* *fi* **O/P:** 2 You entered two
if else	*if* condition ; *then* command(s) *else* command(s) *fi*	*read fileName* *if [-e $fileName]; then* *echo " File $fileName exists"* *else* *echo " File $fileName does not exist"* *fi* **O/P:** LAB3.sh File LAB3.sh exists
else if ladder	*if* condition ; *then* command(s) *elif* condition ; *then* command(s) *fi*	*read a b* *if [$a == $b]; then* *echo "$a is equal to $b"* *elif [$a -gt $b]; then* *echo "$a is greater than $b"* *elif ((a<b)) ; then* *echo "$a is less than $b"* *else* *echo "None of the condition met"* fi **O/P:** 4 5 4 is less than 5

switch case	*case* word *in* pattern1) command(s) ;; pattern2) command(s) ;; ... *) command(s) ;; *esac*	*echo -n "Enter a number 1 or string Hello or character A"* *read character* *case $character in* *1) echo "You entered one.";;* *"Hello") echo -n "You entered two."* *echo "Just to show multiple commands";;* *'A') echo "You entered three.";;* **) echo "You did not enter a number"* *echo "between 1 and 3."* *esac* **O/P:** Enter a number 1 or string Hello or character A: Hello You entered two. Just to show multiple commands
for	*for* ((initialization; condition; expo)); *do* command(s) *done*	*read n* *for ((i=1; i<=n; i++));do* *echo -n $i* *done* **O/P:** 5 12345

for each	*for* variable in list *do* command(s) *done*	*IFS=$'\n'* #field separator is \n instead of default space *x=`ls -l | cut -c 1`* *for i in $x;do* *if [$i = "d"] ; then* *echo "This is the directory"* *fi* *done* **O/P:** $ls -l -rw-r--r-- 1 … script.sh -rw-r--r-- 1 … file2.txt drwxr-xr-x 2 … test $bash script.sh This is the directory
while	while condition do command(s) to be executed while the condition is true done	read n i=1; while ((i <= n)); do echo -n $i " " ((i++)) done echo “" O/P: 5 1 2 3 4 5
until	until condition do command(s) to be executed until condition is true i.e while the condition is false. Done	read n i=1 until ((i > n)); do echo -n $i " " ((i++)) done O/P: 5 1 2 3 4 5

exit	exit num command may be used to deliver an num exit status to the shell (num must be an integer in the 0 - 255 range).	echohi echo "last error status $?" exit $? #exit the script with las error status echo "HI" # never printed O/P: echohi: command not found last error status 127

7. Execution of a shell script

Prepare the shell script using either *text editor* or *vi*. After preparing the script file in use *sh* or *bash* command to execute a shell script. Example: $*bash test.sh* [Here the test.sh is the file to be executed]. OR give executable permission to the script and run ./script-Name. Example: *$chmod +x test.sh*
$./test.sh

Lab Exercises

1. Write a shell script to find whether a given file is the directory or regular file.
2. Write a shell script to list all files (only file names) containing the input pattern (string) in the folder entered by the user.
3. Write a shell script to replace all files with .txt extension with .text in the current directory. This has to be done recursively i.e if the current folder contains a folder "OS" with abc.txt then it has to be changed to abc.text (Hint: use find, mv)
4. Write a shell script to calculate the gross salary. GS=Basics + TA + 10% of Basics. Floating point calculations has to be performed.
5. Write a program to copy all the files (having file extension input by the user) in the current folder to the new folder input by the user. ex: user enter .text TEXT then all files with .text should be moved to TEXT folder. This should be done only at single level. i.e if the current folder contains a folder name ABC which has .txt files then these files should not be copied to TEXT.
6. Write a shell script to modify all occurrences of "ex:" with "Example:" in all the files present in current folder only if "ex:" occurs at the start of the line or after a period

(.). Example: if a file contains a line: “ex: this is first occurrence so should be replaced” and “second ex: should not be replaced as it occurs in the middle of the sentence.”

7. Write a shell script which deletes all the even numbered lines in a text file.

Additional Exercises

1. Write a shell script to check whether the user entered number is prime or not.
2. Write a shell script to find the factorial of number.
3. Write a shell script that, given a file name as the argument will write the even numbered line to a file with name evenfile and odd numbered lines to a file called oddfile.

LAB NO: 4 **Date:**

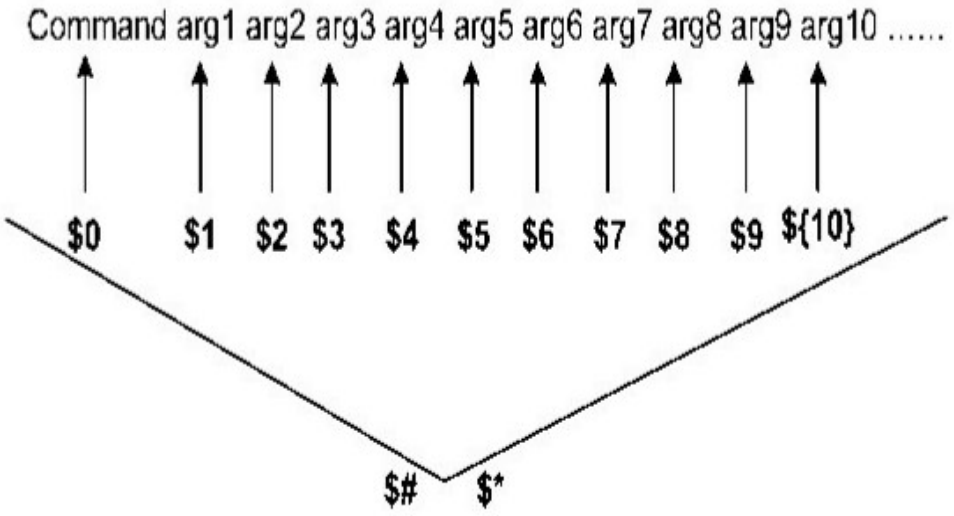
ADVANCED SHELL SCRIPTING

Objectives:

1. To learn the command line arguments in shell scripting.
2. To know system variables.
3. To understand basics of arrays and functions.

1. Command Line arguments:

Command line arguments (also known as positional parameters) are the arguments specified at the command prompt with a command or script to be executed. The locations at the command prompt of the arguments as well as the location of the command, or the script itself, are stored in corresponding variables. These variables are special shell variables.



Positional parameter	Description
$0	The command or script name
$#	Total number of arguments.
$1 to $9	Arguments 1 through 9
${10} and so on	Arguments 10 and further
$*	All the arguments
$$	PID of the running script
$@	Returns a sequence of strings (``$1", ``$2'' ... ``$n''). Same as $* when unquoted. $@ interprets each quoted argument as a separate argument. for i in "$@"; do echo $i # loop $# times done for i in "$*";do echo $i # loop 1 times done

EXAMPLE:

At the command line:

$scriptname arg1 arg2 arg3 ...

Inside script:

```
echo $1 $2 $3 ${10}   #Positional parameters
echo $*               #All the positional parameters
echo $#               #The number of positional parameters
shift
```

2. System variables

When you log in on UNIX, your current shell (login shell) sets a unique working environment for you which is maintained until you log out. You can see system variables by giving command like $ set, few of the important system variables are

System Variable	value	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	Our logging name
OSTYPE	UNIX	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
USER	vivek	User name who is currently login to this PC

NOTE that some of the above settings can be different in your PC. You can print any of the above variables contain as follows

$ echo $USER

$ echo $HOME

[Caution: Do not modify System variable this can some time create problems.]

3. Arrays and Functions

3.1 Arrays: An array variable that can hold multiple values at the same time. Arrays provide a method of grouping a set of variables. Instead of creating a new name for each variable that is required, you can use a single array variable that stores all the other variables.

Array Declaration

If you are using **ksh** shell then here is the syntax of array initialization:

set -A array_name value1 value2 ... valuen

If you are using **bash** shell the here is the syntax of array initialization:
array_name=(value1 ... valuen) or
declare -a array_name

Accessing Array values

After you have set any array variable, you access it as follows:
${array_name[index]}
Here array_name is the name of the array, and index is the index of the value to be accessed.

Example:

```
read -a inputArrayOfNumbers  # input separated by spaces and not by carriage return
echo -n "Entered input is…"
for i in ${inputArrayOfNumbers[@]} ; do
echo -n $i " "
done
```

O/P:

5 4 45 3
Entered input is…5 4 45 3

Example 2:

```
declare –a arrayOfNumber
j=0
for i in $@
do
arrayOfNumber[j]=$i
((j++))
done
echo “${arrayOfNumber[@]}”
```

3.2 Functions:

Functions enable you to break down the overall functionality of a script into smaller, logical subsections, which can then be called upon to perform their individual task when it is needed. Using functions to perform repetitive tasks is an excellent way to create code reuse.

Function Definition

To define a function, simply use the following syntax:

function_name () { # can also use function function_name

list of command(s)

to use parameters passed to the function use $1, $2…

}

Calling a Function

To call the function in the script use the following syntax:

function_name Arg1 arg2… # without spaces

Example :

```
function_add() {
a=$1
b=$2
c=`echo $a+$b | bc`
echo $c
}
function_add 3 5
```

Returning values from function

Exit status can be returned from the function using *return* statements in the function definition.

Example:

```
function_name () {
  list of command(s)
   retval=0
  return "$retval"
```

In the main routine the values can be retrieved using *$?*.

Example:

echo The previous function has a return value of $?

retval=$? # to get the exit status of the function.

Variables can be defined outside the scope of the any function, so that they can be shared among the functions and main routine. To return the string *echo* can be used in the function definition and use retval=$(*function_name*) in the main routine.

Arrays as parameter to function

An array can be passed as a parameter to the function as normal variable, while in the definition it can be accessed using *$@*.

Example:

```
myFunction() {
param1=("${!1}")
param2=("${!2}")
for i in ${param1[@]}; do
for j in ${param2[@]}; do
if [ "${i}" == "${j}" ]; then
echo ${i}
echo ${j}
fi
done
done
}
a=(foo bar baz)
b=(foo bar qux)
myFunction a[@] b[@] # would display foo foo bar bar.
```

Lab Exercises

1. Write a shell script to make a duplicate copy of a specified file through command line.
2. Write a shell script to remove all files that are passed as command line arguments interactively.
3. Write a program to sort the strings that are passed as a command line arguments. (ex: ./script.sh "OS Lab" "Quoted strings" "Command Line" "Sort It". The output should be "Command Line" "OS Lab" "Quoted strings" "Sort It". (make use of usrdefined sort function)
4. Implement *wordcount* script that takes *-linecount*, *-wordcount*, *-charcount* options and performs accordingly, on the input file that is passed as command line argument (use case statement)
5. Write a menu driven shell script to read list of patterns as command line arguments and perform following operations.
    a. Search the patterns in the given input file. Display all lines containing the patten in the given input file.

b. Delete all occurances of the pattern in the given input file.
c. Exit from the shell script.

Additional Exercises

1. Write a shell script to input a file and display permissions of the owner group and others.
2. Write a shell script to display all files that are created between the input years range.(ex with 2014-2015)
3. Write a shell script that accepts a file name starting and ending line numbers as arguments and displays all the lines between the given line numbers.

LAB NO: 5 **Date:**

SYSTEM CALLS FOR PROCESS CONTROL

Objectives:

1. To implement the C program on UNIX platform.
2. To demonstrate the uses of system calls in C programming.

1. Executing a C program on UNIX platform

Compile/Link a Simple C Program - hello.c

Below is the Hello-world C program hello.c:

```
// hello.c
#include <stdio.h>
int main() {
   printf("Hello, world!\n");
   return 0;
}
```

To compile the hello.c:

> gcc hello.c // Compile and link source file hello.c into executable a.out

The default output executable is called "a.out".

To run the program:

$ ./a.out

In Bash or Bourne shell, the default PATH does not include the current working directory. Hence, you may need to include the current path (./) in the command. (Windows include the current directory in the PATH automatically; whereas UNIXes do not - you need to include the current directory explicitly in the PATH.)

To specify the output filename, use -o option:

> gcc -o hello hello.c // Compile and link source file hello.c into executable hello

$ ./hello // Execute hello specifying the current path (./)

2. Use of System Calls in C programming

The main system calls that will be needed for this lab are:

- fork()
- execl(), execlp(), execv(), execvp()
- wait()
- getpid(), getppid()
- getpgrp()

fork():

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

The fork system call does not take any argument. The process that invokes the fork() is known as the parent and the new process is called the child. If the fork system call fails, it will return a -1. If the fork system call is successful, the process ID of the child process is returned in the parent process and a 0 is returned in the child process. When a fork() system call is made, the operating system generates a copy of the parent process which becomes the child process. Both parent and child resume execution of the instruction after the fork statement. *vfork()* is a variant of fork. It creates the child process and blocks the parent, also both parent and child share the same address space. Refer the manual pages for fork and vfork.

The operating system will pass to the child process most of the parent's process information. However, some information is unique to the child process:

- The child has its own process ID (PID)
- The child will have a different PPID than its parent
- System imposed process limits are reset to zero
- All recorded locks on files are reset
- The action to be taken when receiving signals is different

The following is a simple example of fork()

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void){
  printf("Hello \n");
  fork();
  printf("bye\n");
  return 0;
}
```

Hello is printed once by parent process. bye is printed twice, once by the parent and once by the child. If the fork system call is successful a child process continues execution at the point where it was called by the parent process.
A summary of fork() return values:

- fork_return > 0: this is the parent
- fork_return == 0: this is the child
- fork_return == -1: fork() failed and there is no child. See code snippet below to see how to check errors.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#define BUFLEN 10
int main(void)
{
  int i;
  char  buffer[BUFLEN+1];
  pid_t pid1;
  pid1  = fork( );
  if (pid1 == 0)
  {       strncpy(buffer, "CHILD\n", BUFLEN); /*in the child process*/
    buffer[BUFLEN] = '\0';
  }
```

```
  else if(fork_return > 0)  // for parent process
  {
    strncpy(buffer, "PARENT\n", BUFLEN); /*in the parent process*/
    buffer[BUFLEN] = '\0';
  }
  else if(fork_return == -1)
  {
    printf("ERROR:\n");
    switch (errno)
    {
      case EAGAIN:
       printf("Cannot fork process: System Process Limit Reached\n");
     case ENOMEM:
       printf("Cannot fork process: Out of memory\n");
    }
    return 1;
  }

  for (i=0; i<5; ++i) /*both processes do this*/
  {
    sleep(1); /*5 times each*/
    write(1, buffer, strlen(buffer));
  }
  return 0;
}
```

A few notes on this program:

- The function call sleep will result in a process "sleeping" a specified number of seconds. It can be used to prevent the process from running to completion within one time slice.
- One process will always end before the other. If there is enough intervening time before the second process ends, the system call will redisplay the prompt, producing the last line of output where the output from the child process is appended to the end of the prompt (i.e.. %child)

A few additional notes about fork():

- an orphan is a child process that continues to execute after its parent has finished execution (or died)
- to avoid this problem, the parent should execute: wait(&return_code);

wait():

```
#include <sys/types.h>
#include <sys/wait.h>
  int *status;
pid_t pidOfLastTerminatedChild= wait(&status);
```

A parent process usually needs to synchronize its actions by waiting until the child process which has either stopped or terminated its actions. The wait() system call allows the parent process to suspend its activities until one of these actions has occurred. The wait() system call accepts a single argument, which is a pointer to an integer and returns a value defined as type pid_t. If the calling process does not have any child associated with it, wait will return immediately with a value of -1. If any child processes are still active, the calling process will suspend its activity until a child process terminates.

Example of wait():

```
int status;
pid_t fork_return;
fork_return = fork();
if (fork_return == 0) /* child process */ {
  printf("\n I'm the child!");
  exit(0); }
else { /* parent process */
  wait(&status);
  printf("\n I'm the parent!");
  if (WIFEXITED(status)) //    #include<sys/wait.h>
printf("\n Child returned: %d\n", WEXITSTATUS(status));   //#include<sys/wait.h>
}
```

A few notes on this program:

 wait(&status) causes the parent to suspend until the child process finishes execution

 details of how the child stopped are returned via the status variable to the parent. Several macros are available to interpret the information. Two useful ones are:

- ➢ WIFEXITED evaluates as true, or 0, if the process ended normally with an exit or return call.
- ➢ WEXITSTATUS if a process ended normally you can get the value that was returned with this macro.

exec*():

```
#include <unistd.h>
extern char **environ;
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg , ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

"The *exec* family of functions replaces the current process image with a new process image." (man pages)

Commonly a process generates a child process because it would like to transform the child process by changing the program code the child process is executing. The text, data and stack segment of the process are replaced and only the u (user) area of the process remains the same. If successful, the exec system calls do not return to the invoking program as the calling image is lost.

It is possible for a user at the command line to issue an exec system call, but it takes over the current shell and terminates the shell.

% exec command [arguments]

The versions of *exec* are:

- execl
- execv
- execle
- execve
- execlp
- execvp

The naming convention: exec*

- 'l' indicates a list arrangement (a series of null terminated arguments)
- 'v' indicate the array or vector arrangement (like the argv structure).
- 'e' indicates the programmer will construct (in the array/vector format) and pass their own environment variable list
- 'p' indicates the current PATH string should be used when the system searches for executable files.

NOTE:

- In the four system calls where the PATH string is not used (execl, execv, execle, and execve) the path to the program to be executed must be fully specified.

Library Call Name	**Argument Type**	**Pass Current Environment Variables**	**Search PATH automatic?**
execl	list	yes	no
execv	array	yes	no
execle	list	no	no
execve	array	no	no
execlp	list	yes	yes
execvp	array	yes	yes

execlp

- this system call is used when the number of arguments to be passed to the program to be executed is known in advance

execvp

- this system call is used when the numbers of arguments for the program to be executed is dynamic

```
/* using execvp to execute the contents of argv */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
  execvp(argv[1], &argv[1]);
  perror("exec failure");
  exit(1);
}
```

Things to remember about *exec**:

- this system call simply replaces the current process with a new program -- the pid does not change.
- the exec() is issued by the calling process and what is exec'ed is referred to as the new program -- not the new process since no new process is created.
- it is important to realize that control is not passed back to the calling process unless an error occurred with the exec() call.
- in the case of an error, the exec() returns a value back to the calling process
- if no error occurs, the calling process is lost.

A few more Examples of valid exec commands:

```
execl("/bin/date","",NULL); // since the second argument is the program name,
                            // it may be null
execl("/bin/date","date",NULL);

execlp("date","date", NULL); //uses the PATH to find date, try: %echo $PATH
```

getpid():

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

pid_t getpid(void);
pid_t getppid(void);
```

getpid() returns the process id of the current process. The process ID is a unique positive integer identification number given to the process when it begins executing.
getppid() returns the process id of the parent of the current process. The parent process forked the current child process.

getpgrp():

```
#include <unistd.h>
pid_t getpgrp(void);
```

Every process belongs to a process group that is identified by an integer process group ID value. When a process generates a child process, the operating system will automatically create a process group.
The initial parent process is known as the process leader. getpgrp() will obtain the process group id.

Lab Exercises

1. Write a C program to create a child process. Display different messages in parent process and child process. Display PID and PPID of both parent and child process.
2. Write a C program to accept a set of strings as command line arguments. Sort the strings and display them in a child process. Parent process should display the unsorted strings only after the child displays the sorted list.
3. Write a C program to read N strings. Create two child processes, each of this should perform sorting using two different methods (bubble, selection, quicksort etc). The parent should wait until one of the child process terminates.

Additional Exercises

1. Write a C program to simulate the unix commands: ls -l, cp and wc commands. [**NOTE:** DON'T DIRECTLY USE THE BUILT-IN COMMANDS]

LAB NO:6 **Date:**

PROCESS SCHEDULING

Objectives:

1. To implement process scheduling algorithms.

1. Basic Concepts :

CPU scheduling is the basis of multi programmed operating systems. By switching the CPU among processes, the operating system can make the computer more productive. In a single-processor system, only one process can run at a time; others(if any) must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

The idea is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU.

CPU Scheduler:

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the CPU scheduler. The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

2. Problem Description and Algorithm :

2.1 Round Robin Scheduling (RR):

The **round-robin (RR) scheduling algorithm** is designed especially for timesharing systems. It is similar to (First Come First Serve) FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a **time quantum** or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds in length.

To implement RR scheduling, we keep the ready queue as a FIFO queue o£ processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the **tail** of the ready queue. The CPU scheduler will then select the next process in the ready queue.

Example: Time quantum=3

Process	Arrival Time	Execution Time
P1	0	8
P2	5	4
P3	3	9
P4	7	16

P1	P3	P1	P2	P3	P4	P1	P2	P3	P4

0 3 6 9 12 15 18 20 21 24 37

Average waiting time: (12+12+12+14)/4 = 12.5.

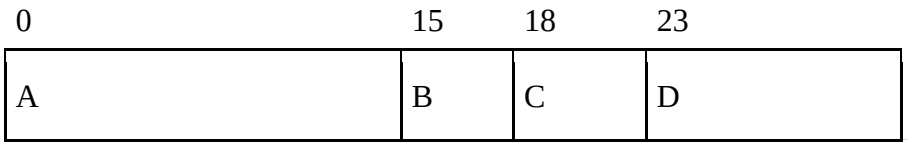
2.2 Shortest Job First (SJF):

This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

Example:Non-Preemptive

Process	Arrival time	Burst time
A	0	15
B	5	3
C	8	5
D	10	7

SJF Waiting time A= 0, B=10, C=10, D=13. TT A=15, B=13, C=15, D=20.

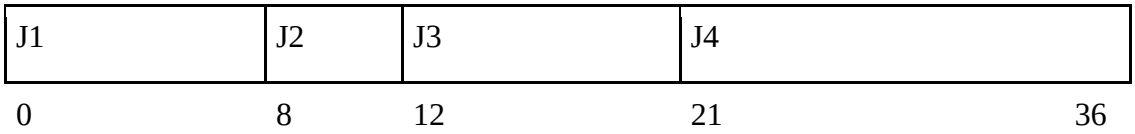


2.3 Priority Scheduling:

A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

Example: Non-Preemptive

Process	Arrival Time	Execution Time	Priority
J1	0	8	1
J2	2	4	2
J3	9	9	2
J4	4	15	3



Average waiting time: (0+3+17)/3 = 6.67.

Lab Exercise

1. Develop a menu driven C program to implement the following process scheduling algorithms: preemptive-SJF, RR and non-preemptive priority scheduling algorithms.

Additional Exercises

1. Write C program to implement FCFS. (Assuming all the processes arrive at the same time)
2. Write a C program to implement non-preemptive SJF, where the arrival time is different for the processes.

LAB NO: 7 **Date:**

CLASSICAL PROBLEMS OF SYNCHRONIZATION

Objectives:

1. To solve synchronization problems using threads.

1. Semaphores

A semaphore is a synchronization tool to solve critical section problem. A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: *wait* () and *signal* ().

The definition of wait () is as follows:

```
Wait(S) {
        while S <= 0
                ; // no-op
        S--;
}
```

The definition of signal() is as follows:

```
signal(S) {     S++;    }
```

2. Classical Problems :

2.1 The Bounded – Buffer Problem

Here the pool consists of *n* buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0. The classical example is the production line.

2.2 The Producer Consumer Problem:

A producer process produces information that is consumed by a consumer process. For example, a compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader. The producer-consumer problem also provides a useful metaphor for the client-server paradigm. We generally think of a server as a producer and a client as a consumer. For

example, a Web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client Web browser requesting the resource.

One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

2.3 The Readers Writers Problem:

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as readers and to the latter as writers. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the *readers-writers problem.*

3. Algorithm

3.1. Solution to bounded buffer Producers_Consumers Problem using semaphores

```
process producers_consumers
{parent process}
        const  int capacity =5;
        int   item ;
        buffer: array[1..capacity] of item;
        empty, full : semaphore; {general} //Initialize empty to capacity and full to 0
        pmutex, cmutex :semaphore; {binary} //Initialize to 1
        int in =1, out=1;
 // initiate  processes producers, consumers
```

```
}end { producers_consumers}

process producer;
do
{
     //Produce the data to be put into buffer in anyway
      wait(empty);
      wait(pmutex);
      buffer[in] = producedItem;
      in = (in mod capacity) + 1;
      signal(pmutex);
      signal(full);
      other_X_processing
}while (TRUE);
end;  {producer}

process consumer;
 do
{     wait(full);
      wait(cmutex);
      citem =  buffer[out];
      out = (out mod capacity) + 1;
      signal(cmutex);
      signal(empty);
   // Use the consumed data
} while(TRUE);
 end;  {consumer}
```

3.2. Solution to Readers Writers problem using semaphores

```
program readers_writers;
var
  readercount : integer;
  mutex, write : semaphore; {binary}
```

```
process readerX;
  begin
    while true do
      begin
        {obtain permission to enter}
         wait(mutex);
         readercount := readercount +1;
         if readercount = 1 then wait(write);
         signal(mutex);
              …
           {reads}
              …
         wait(mutex);
          readercount = readercount – 1;
          if readercount = 0 then signal(write);
          signal(mutex);
          other_X_processing
        end {while}
     end; {reader}
process writerZ;
 begin
  while true do
   begin
     wait(write);
       …
      signal(write);
      Other_Z_processing
    end {while}
  end; {writerZ}

{parent process}
begin {readers_writers}
 readercount : = 0;
 signal(mutex);
 signal (write);
```

initiate readers, writers
end {readers_writers}

Lab Exercise

1. Write a C program to solve producer consumer problem with bounded buffer using semaphores.
2. Write a C program to solve the readers and writers Problem.

Additional Exercise

1. Write a C program to solve the Dining-Philosophers problem.

LAB NO: 8 **Date:**

BANKERS ALGORITHM

Objectives:

1. To implement deadlock avoidance and Safe State in a set of concurrent processes.

2. To solve a Banker's Algorithm using Safety Algorithm and Resource Request Algorithm for avoiding deadlocks in a computer system.

1. Deadlocks

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock. Under the normal mode of operation, a process may utilize a resource in only the following sequence:

Request: The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

Use: The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

Release: The process releases the resource.

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

Mutual exclusion: At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

Hold and wait: A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

No preemption: Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

Circular wait: A set { *P0* , Pl, ... , P11 } of waiting processes must exist such that *Po* is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ... , Pn-1 is waiting for a resource held by P_n and Pn is waiting for a resource held by *Po.*

2. Deadlock Avoidance

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

Safe Sate

When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state. System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources plus resources held by all the P_j, with $j < i$. That is:

- If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
- When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
- When P_i terminates, P_{i+1} can obtain its needed resources, and so on

If a system is in safe state then no deadlocks. If a system is in unsafe state then there is a possibility of deadlock

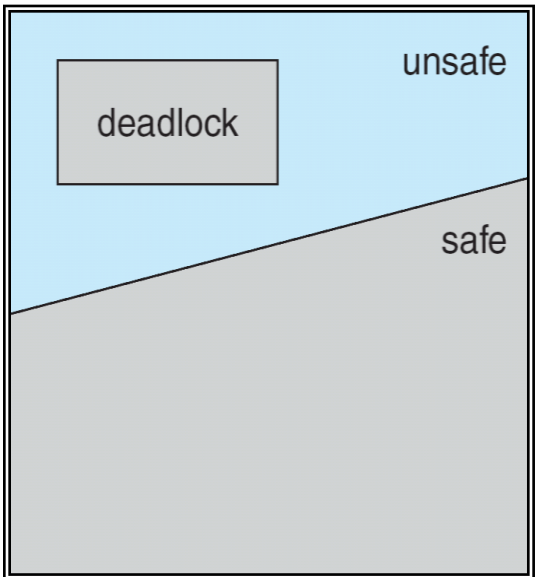


Fig 7.1 Safe, Unsafe and Deadlock State

3. Bankers Algorithm

It is a deadlock avoidance algorithm. The name was chosen because the bank never allocates more than the available cash.
Available: A vector of length *m* indicates the number of available resources of each type. If *Available[j]* equals *k,* then *k* instances of resource type *Ri* are available.

Max: An *n* x *m* matrix defines the maximum demand of each process. If *Max[i] [j]* equals *k,* then process *P;* may request at most *k* instances of resource type Ri.

Allocation: An *n* x *m* matrix defines the number of resources of each type currently allocated to each process. If *Allocation[i][j]* equals lc, then process *Pi* is currently allocated lc instances of resource type *Rj.*

Need: An *n* x *m* matrix indicates the remaining resource need of each process. If *Need[i][j]* equals *k,* then process *Pi* may need *k* more instances of resource type *Ri* to complete its task. Note that *Need[i][j]* equals *Max[i][j] - Allocation [i][j].*

3.1 Safety Algorithm:

- Let *Work* and *Finish* be vectors of length *m* and *n,* respectively. Initialize *Work= Available* and *Finish[i] =false* for *i* = 0, 1, ... , *n* - 1.
- Find an index *i* such that both

a. *Finish[i] ==false*
b. *Needi <= Work*
If no such *i* exists, go to step 4.

- *Work = Work + Allocation;*
  Finish[i] = true
  Go to step 2.
- If *Finish[i] ==true* for all *i,* then the system is in a safe state.

3.2 Resource-Request Algorithm :
This algorithm is used for determining whether requests can be safely granted.

Let $Request_i$*;* be the request vector for process *Pi.* If $Request_i$ *[j] == k,* then process P_i wants *k* instances of resource type *Rj.* When a request for resources is made by process P_i, the following actions are taken:

1. If $Request_i <= Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i <= Available$, go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:
   $Available = Available – Request_i;$
   $Allocation_i = Allocation_i + Request_i;$
   $Need_i = Need_i - Request_i;$

If the resulting resource-allocation state is safe, the transaction is completed, and process P_i is allocated its resources. However, if the new state is unsafe, then P_i must wait for $Request_i$, and the old resource-allocation state is restored.

Lab Exercise
1. Develop a program to simulate banker's algorithm. (Consider safety and resource-request algorithms)

Additional exercises
1. Write a C program to implement the deadlock detection algorithm.

LAB NO: 9 **Date:**

DYNAMIC STORAGE ALLOCATION STRATEGY FOR FIRST FIT AND BEST FIT

Objectives:

1. To learn the algorithm for first and best fit strategies.

2. To write C program which allocates memory requirement for processes using first fit and best fit strategies.

1. Description

One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided. When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate.
The first fit and best fit strategies are used to select a free hole (available block of memory) from the set of available holes.

First fit: Allocate the *first* hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

Best fit: Allocate the *smallest* hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

2. Algorithm

First Fit Allocation

1. Declare structures *hole* and *process* to hold information about set of holes and processes respectively.
2. Get number of holes, say *nh*.
3. Get the size of each hole
4. Get number of processes, say *np*.

5. Get the memory requirements for each process.
6. Allocate processes to holes, by examining each hole as follows:
    a. If hole size > process size then
        i. Mark process as allocated to that hole.
            i. ii. Decrement hole size by process size.
    b. Otherwise check the next from the set of hole
7. Print the list of process and their allocated holes or unallocated status.
8. Print the list of holes, their actual and current availability.
9. Stop

Best Fit Allocation

1. Declare structures *hole* and *process* to hold information about set of holes
2. and processes respectively.
3. Get number of holes, say *nh*.
4. Get the size of each hole
5. Get number of processes, say *np*.
6. Get the memory requirements for each process.
7. Allocate processes to holes, by examining each hole as follows:
    a. Sort the holes according to their sizes in ascending order
    b. If hole size > process size then
        i. Mark process as allocated to that hole.
            i. ii. Decrement hole size by process size.
    c. Otherwise check the next from the set of sorted hole
8. Print the list of process and their allocated holes or unallocated status.
9. Print the list of holes, their actual and current availability.
10. Stop

Lab exercise

1. Write a C program to implement dynamic storage allocation strategy for first fit and best fit using dynamic allocations for all the required data structures.

Additional exercises

1. Write a C program to implement dynamic storage allocation strategy for worst fit using dynamic allocations for all the required data structures.
1. Write a C program to implement basic page replacement algorithm.

LAB NO: 10 **Date:**

PAGE REPLACEMENT ALGORITHMS

Objectives:

1. To learn FIFO and optimal page replacement algorithms.
2. To write a C program to simulate FIFO and optimal page replacement algorithms.

1.Introduction :

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme.

FIFO algorithm:

The simpler page replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replace, the oldest page is chosen. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue when a page is brought into memory; we insert it at the tail of the queue.

Optimal Page Replacement :

Optimal page replacement algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. The basic idea is to replace the page that will not be used for the longest period of time. Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames. Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

2.Algorithms :

FIFO :

1. Start
2. Read the number of frames
3. Read the number of pages
4. Read the page numbers
5. Initialize the values in frames to -1
6. Allocate the pages in to frames in first in first out order.
7. Display the number of page faults.
8. stop

Optimal Page Replacement :

1. Start
2. Read the number of frames
3. Read the number of pages
4. Read the page numbers
5. Initialize the values in frames to -1
6. Replace the page that will not be used for the longest period of time.
7. Display the number of page faults.
8. stop.

Lab exercise :

1. Write a C program to simulate page replacement algorithms: FIFO and optimal. Frame allocation has to be done as per user input and use dynamic allocation for all data structures.

2. Write a C program to simulate LRU Page Replacement. Frame allocation has to be done as per user input and dynamic allocation for all data structures.

LAB NO: 11 **Date:**

DISK SCHEDULING ALGORITHM

Objectives:

1. To understand the concept of various disk scheduling algorithms.

2. To write a menu driven C program to simulate the following disk scheduling algorithm : SSTF, SCAN, C-SCAN, C-LOOK.

1. 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, we will 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

By using these algorithms we can keep the Head Movements (# tracks) to the least amount as possible. The less the head has to move the faster the seek time will be.

Problem :

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.

2. Illustration of Disk Scheduling Algorithm :

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.

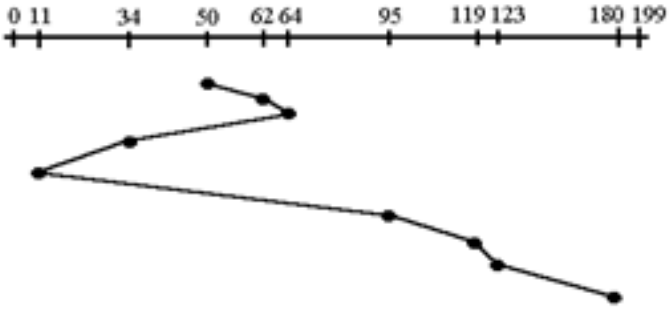


Figure 11.1 SSTF

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.

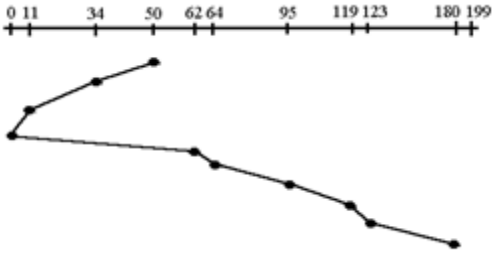


Figure 11.2 SCAN

Circular Scan (C-SCAN) :

Circular scanning works just like the elevator to some extent. 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 most sufficient.

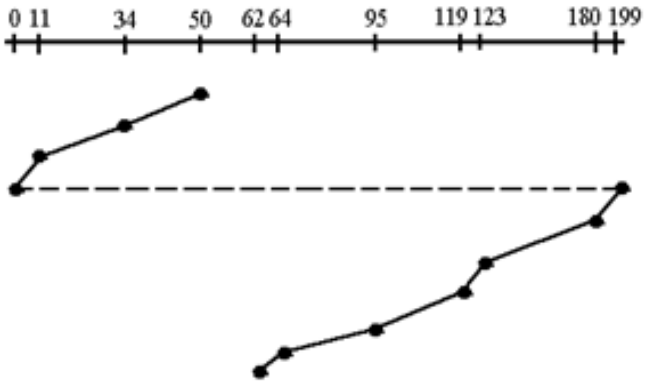


Figure 11.3 C-SCAN

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.

From this you were able to see a scan change from 644 total head movements to just 157. You should now have an understanding as to why your operating system truly relies on the type of algorithm it needs when it is dealing with multiple processes.

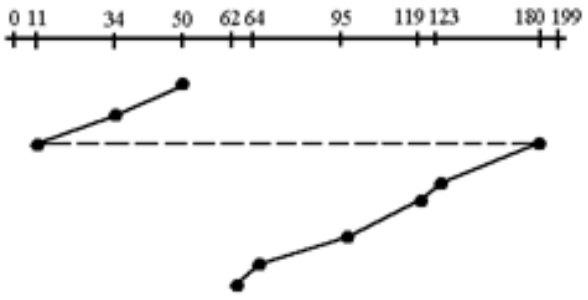


Figure 11.4 C-LOOK

Lab Exercise :

1. Develop a menu driven program to simulate the following disk scheduling algorithms: SSTF, SCAN, C-SCAN, C-LOOK.

Additional Exercise :

1. Develop a menu driven program to simulate the following disk scheduling algorithms: FCFS, LOOK.

LAB NO: 12 **Date:**

SCHEDULING IN REAL TIME SYSTEMS

Objectives:

1. To know the definition of Real Time Systems (RTSs).
2. To understand Rate-Monotonic Scheduling.
3. To understand Earliest-Deadline-First Scheduling.

1. Introduction to Real Time Systems (RTSs)

A real-time system is a computer system that requires not only that the computing results be "correct" but also that the results be produced within a specified deadline period. Results produced after the deadline has passed even if correct-may be of no real value. To illustrate, consider an autonomous robot that delivers mail in an office complex. If its vision-control system identifies a wall after the robot has walked into it, despite correctly identifying the wall, the system has not met its requirement. Contrast this timing requirement with the much less strict demands of other systems. In an interactive desktop computer system, it is desirable to provide a quick response time to the interactive user, but it is not mandatory to do so. Some systems -such as a batch-processing system-may have no timing requirements whatsoever.

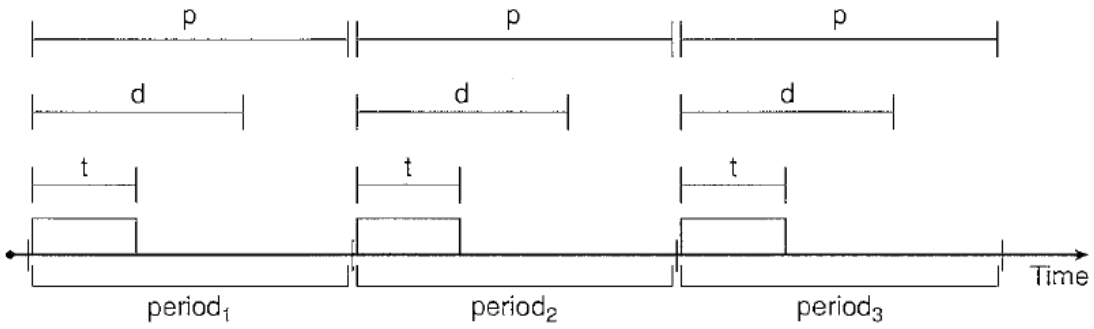
Types of RTSs

Real-time computing is of two types: hard and soft. A hard real-time system has the most stringent requirements, guaranteeing that critical realtime tasks be completed within their deadlines. Safety-critical systems are typically hard real-time systems. A soft real-time system is less restrictive, simply providing that a critical real-time task will receive priority over other tasks and that it will retain that priority until it completes. Many commercial operating systems-as well as Linux-provide soft real-time support.

Process Characteristics

Certain characteristics of the processes are as follows: First, the processes are considered periodic. That is, they require the CPU at constant intervals (periods). Each periodic process has a fixed processing timet once it acquires the CPU, a deadline *d* by which time it must be serviced by the CPU, and a period *p*. The relationship of the processing time, the deadline, and the period can be expressed as $0 <= t <= d <= p$. The rate of a periodic task

is 1 / p. The Figure below illustrates the execution of a periodic process over time. Schedulers can take advantage of this relationship and assign priorities according to the deadline or rate requirements of a periodic process.



2. Rate-Monotonic Scheduling

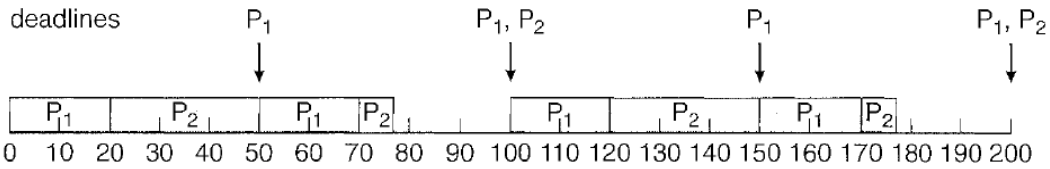
The rate-monotonic scheduling algorithm schedules periodic tasks using a static priority policy with preemption. If a lower-priority process is running and a higher-priority process becomes available to run, it will preempt the lower-priority process. Upon entering the system, each periodic task is assigned a priority inversely based on its period. The shorter the period, the higher the priority; the longer the period, the lower the priority. The rationale behind this policy is to assign a higher priority to tasks that require the CPU more often.

Furthermore, rate-monotonic scheduling assumes that the processing time of a periodic process is the same for each CPU burst. That is, every time a process acquires the CPU, the duration of its CPU burst is the same.

Let's consider an example. We have two processes P1 and P2. The periods for P1 and P2 are 50 and 100, respectively-that is, Pl =50 and P2 = 100. The processil1.g times are t1 = 20 for P1 and t2 = 35 for P2 . The deadline for each process requires that it complete its CPU burst by the start of its next period. We must first ask ourselves whether it is possible to schedule these tasks so that each meets its deadlines. If we measure the CPU utilization of a process Pi as the ratio of its burst to its period -ti I Pi -the CPU utilization of P1 is 20/50 = 0.40 and that of P2 is 35/100 = 0.35, for a total CPU utilization of 75 percent. Therefore, it seems we can schedule these tasks in such a way that both meet their deadlines and still leave the CPU with available cycles.

Example:

Suppose we use rate-monotonic scheduling, in which we assign P1 a higher priority than P2, since the period of P1 is shorter than that of P2.

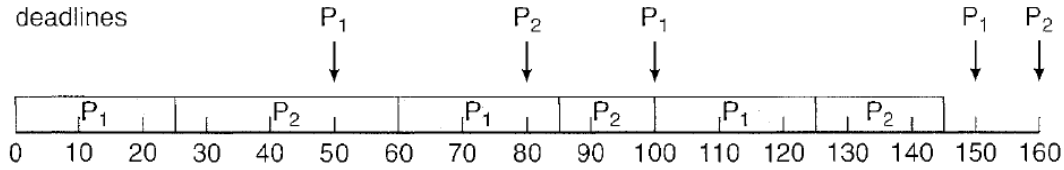


P1 starts first and completes its CPU burst at time 20, thereby meeting its first deadline. P2 starts running at this point and runs until time 50. At this time, it is preempted by P1, although it still has 5 milliseconds remaining in its CPU burst. P1 completes its CPU burst at time 70, at which point the scheduler resumes P2. P2 completes its CPU burst at time 75, also meeting its first deadline. The system is idle until time 100, when P1 is scheduled again. Rate-monotonic scheduling is considered optimal in that if a set of processes cannot be scheduled by this algorithm, it cannot be scheduled by any other algorithm that assigns static priorities.

3. Earliest-Deadline-First Scheduling

Earliest-deadline-first (EDF) scheduling dynamically assigns priorities according to deadline. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority. Under the EDF policy, when a process becomes runnable, it must announce its deadline requirements to the system. Priorities may have to be adjusted to reflect the deadline of the newly rmmable process. Note how this differs from rate-monotonic scheduling, where priorities are fixed.

Example:
P1 has values of p1 = 50 and t1 = 25 and that P2 has values of p2 = 80 and t2 = 35. The EDF scheduling of these processes is shown in Figure below.



Process P1 has the earliest deadline, so its initial priority is higher than that of process P2 • Process P2 begins rmming at the end of the CPU burst for P1. However, whereas rate-monotonic scheduling allows P1 to preempt P2 at the beginning of its next period at time 50, EDF scheduling allows process P2 to continue running. P2 now has a higher priority than P1 because its next deadline (at time 80) is earlier than that of P1 (at time 100). Thus, both P1 and P2 meet their first deadlines. Process P1 again begins running at time 60 and completes its second CPU burst at time 85, also meeting its second deadline at time 100.

P2 begins rum1ing at this point only to be preempted by P1 at the start of its next period at time 100. P2 is preempted because P1 has an earlier deadline (time 150) than P2 (time 160). At time 125, P1 completes its CPU burst and P2 resumes execution, finishing at time 145 and meeting its deadline as well. The system is idle until time 150, when P1 is scheduled to run once again.

Unlike the rate-monotonic algorithm, EDF scheduling does not require that processes be periodic, nor must a process require a constant amount of CPU time per burst. The only requirement is that a process a1mom1ce its deadline to the scheduler when it becomes runnable. The appeal of EDF scheduling is that it is theoretically optimal-theoretically, it can schedule processes so that each process can meet its deadline requirements and CPU utilization will be 100 percent. In practice, however, it is impossible to achieve this level of CPU utilization due to the cost of context switching between processes and interrupt handling.

Lab exercises

1. Write a C program to simulate Rate-Monotonic and Earliest-Deadline-First scheduling for real time systems.

REFERENCES

1. A Silberschartz and Galvin, "Operating Systems Concepts ", 8th edition, Addison Wesley, 2012.

2. H.M. Deitel "An Introduction to Operating Systems" Addison Wesley, 2000.

3. Milan Milankovic"Operating systems Concepts and Design"McGrawHill, 2000.

4. T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 2nd Edition, Prentice-Hall India, 2001.

5. Sartaj Sahni, Data Structures, Algorithms and Applications in C++, 2nd Edition, McGraw-Hill, 2000.

6. Mark Allen Weiss, "Data Structures and Algorithm Analysis in C", Pearson Education, 2nd Edition, 2007.