MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

School of Computer Engineering
Manipal – 576 104

Operating Systems Lab [CSS-2211]

Fourth Semester

B. Tech. (CSE Stream)

MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

SCHOOL OF COMPUTER ENGINEERING

CERTIFICATE

This is to certify that Ms./Mr. ……...…… Reg. No.: …..…… Section: ……..…… Roll No.: ……... has satisfactorily completed the lab exercises prescribed for OPERATING SYSTEMS LAB [**CSS 2211**] of Second Year B. Tech. Degree at MIT, Manipal, in the academic year 2025-2026.

Date: …….....

Signature
Faculty in Charge

CONTENTS

Course Objectives

- Execute shell scripts in UNIX based operating system.
- Implement inter process communication using system calls.
- Implement algorithms for CPU scheduling as well as process synchronization

Course Outcomes

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

1. Demonstrate the working of UNIX based operating systems
2. Demonstrate process management in operating systems
3. Implement CPU scheduling as well as synchronization algorithms
4. Implement algorithms used to understand the functionality of modern operating systems

Evaluation plan

- Internal Assessment Marks : 60%
  - ✓ Continuous evaluation component (for each experiment): 10 marks
  - ✓ Record check, Program check and midterm evaluation are included.
  - ✓ 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 Class notes, Lab Manual Book and the required stationery to everylab 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 /C++ OS Concept implementations.

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
- In case a student misses a lab class, he/she must ensure that the experiment is completed at students end or in a repetition class (if available) with the permission of the faculty concerned but credit will be given only to one day's experiment(s).
- 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

- Carry 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. Understand and execute 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*: ls ~/filename* #T*tries to list a file literally named 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 file-name.
..	Parent directory
~	User's home directory
*	Represents 0 or more characters in a filename, or by itself, all files in a direc-tory. 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:ls file[0-3].txt represents the names file0.txt, file1.txt, and file2.txt
;	Command separator. Allows you to execute multiple commands on a singleline. Example: *cd /var/log ; less* messages
&&	Command separator as above, but only runs the second command if the firstone 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". Command 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*

Redirection operator	**Description**
|	"Pipe". Redirect the output of one command into another command. Example*: ls | sort* #Lists the files in the directory and sends them to `sort`, which prints them in alphabetical order.
>	Redirect the output of a command into a new file. If the file already exists, over-write it. Example: *ls > myfiles.txt # runs ls and saves the output into myfiles.txt, replacing the file if it already*
>>	Redirect the output of a command onto the end of an existing file. Example: *echo "Mary 555-1234" >> phonenumbers.txt #Appends the text to the end of phonenumbers.txt without deleting what was already in the file.*
<	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*

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. Create a folder with your <name >_registration number in the /home folder. Use this folder to save rest of lab exercises. For each lab create a separate folder ex: Lab1, Lab2… Lab11.
2. Execute and write output of all the commands explained so far in this manual. Save the output in Lab1_1.txt . Commands and the output I copied from the command prompt.
3. 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)

Save the output in Lab1_2.txt, Commands and the output I copied from the command prompt.

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

1. *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.

2. ls -l | tr -s ' ' | cut -d' ' -f5 # Lists files and prints only the size column. tr squeezes multiple spaces to one space.

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. File permission commands

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:

1. $bc (hit *enter*
key)5 + 2
7 #7 is response of bc i.e. addition of 5 + 2 you can even try
5 / 2
5 # to perform floating point operations use *bc -l*
5 > 2 #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. echo "2^5" | bc #32
3. echo "scale=2; 5 / 2" | bc #output 2.50
4. echo "5 / 2" |bc #2
5. echo "5 / 2" |bc -l # 2.50000000000000000000

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).

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=5; 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 $b" \| bc -l`* *#or use result=$(bc -l <<< "$aopb")* *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: $*bash test.sh* [Here the test.sh isthe file to be executed]. OR give executable permission to the script and run ./script- Name. Example: *$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.

Write a shell script to modify all occurrences of "ex:" with "Example:" in all the filespresent

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 re- placed” and “second ex: should not be replaced as it occurs in the middle of the sen-tence.”

6. 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:**

PROCESS AND THREAD MANAGEMENT

Objectives:

1. Understand the working of the different system calls.
2. Understand the creation of new processes with fork and altering the code space of process using exec.
3. Understand the concepts of the multithreading.
4. Grasp the execution of the different processes with respect to multithreading.

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()

A new process is created by calling fork. This system call duplicates the current process, creating a new entry in the process table with many of the same attributes as the current process. The new process is almost identical to the original, executing the same code but with its own data space, environment, and file descriptors. Combined with the **exec** functions, **fork** is all we need to create new processes.

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

The return value of fork() is pid_t (defined in the header file sys/types.h). As seen in the Fig. 4.1, the call to fork in the parent process returns the PID of the new child process. The new process continues to execute just like the parent process, with the exception that in the child process, the PID returned is 0. The parent and child process can be determined by using the PID returned from fork() function. To the parent the fork() returns the PID of the child, whereas to the child the PID returned is zero. This is shown in the following Fig. 4.1.

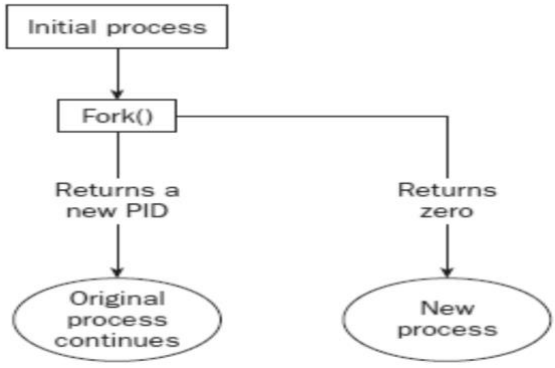


Figure 4.1: Fork system call

In Linux in case of any error observed in calling the system functions, then a special variable called errno will contain the error number. To use errno a header file named errno.h has to be

included in the program. If fork fails, it returns -1. This is commonly due to a limit on the number of child processes that a parent may have (CHILD_MAX), in which case errno will be set to EAGAIN. If there is not enough space for an entry in the process table, or not enough virtual memory, the errno variable will be set to ENOMEM.

A typical code snippet using fork is

```
pid_t new_pid;
new_pid = fork();
switch(new_pid) {
case -1 : /* Error */
break;
case 0 : /* We are child */
break;
default : /* We are parent */
break;
}
```

Sample Program on fork1.c

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
        pid_t pid;
        char *message;
        int n;
        printf("fork program starting\n");
        pid = fork();
        switch(pid)
        {
                case -1:
                        perror("fork failed");
                        exit(1);
                case 0:
                        message = "This is the child";
                        n = 5;
                        break;
                default:
```

```
                message = "This is the parent";
                n = 3;
                break;
        }
        for(; n > 0; n--) {
                puts(message);
                sleep(1);
        }
        exit(0);
}
```

This program runs as two processes. A child process is created and prints a message five times. The original process (the parent) prints a message only three times. The parent process finishes before the child has printed all of its messages, so the next shell prompt appears mixed in with the output.

```
$ ./fork1
fork program starting
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
This is the child
This is the child
```

When fork is called, this program divides into two separate processes. The parent process is identified by a nonzero return from fork and is used to set a number of messages to print, each separated by one second.

The wait() System Call

A parent process usually needs to synchronize its actions by waiting until the child process 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():

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

void main()
{
        int status; pid_t
        pid;
        pid = fork();
        if(pid = -1)
          printf("\nERROR child not created");
         else if (pid == 0) /* child process */
        {
                printf("\n I'm the child!");
                 exit(0);
        }
        else /* parent process */
        {
                wait(&status);
                printf("\n I'm the parent!")
                printf("\n Child returned: %d\n", status)
        }
}
```

A few notes on this program:

wait(&status) causes the parent to sleep until the child process has finished execution. The exit status of the child is returned to the parent.

The exit() System Call

This system call is used to terminate the current running process. A value of zero is passed to indicate that the execution of process was successful. A non-zero value is passed if the execution of process was unsuccessful. All shell commands are written in C including grep. grep will return 0 through exit if the command is successfully runs (grep could find pattern in file). If grep fails to find pattern in file, then it will call exit() with a non-zero value. This is applicable to all commands.

The exec() System Call

The exec function will execute a specified program passed as argument to it, in the same process (Fig. 4.2). The exec() will not create a new process. As new process is not created, the process ID

(PID) does not change across an execute, but the data and code of the calling process are replaced by those of the new process.

fork() is the name of the system call that the parent process uses to "divide" itself ("fork") into two identical processes. After calling fork(), the created child process is actually an exact copy of the parent - which would probably be of limited use - so it replaces itself with another process using the system call exec().

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.

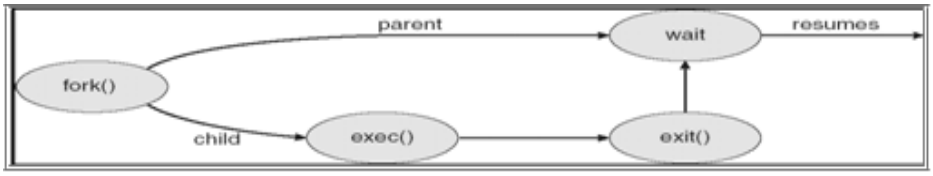


Figure 4.2: exec() system call

The parent process can either continue execution or wait for the child process to complete. If the parent chooses to wait for the child to die, then the parent will receive the exit code of the program that the child executed. If a parent does not wait for the child, and the child terminates before the parent, then the child is called **zombie** process. If a parent terminates before the child process then the child is attached to a process called init (whose PID is 1). In this case, whenever

the child does not have a parent then child is called **orphan** process.

Sample Program:

C program forking a separate process.

```
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
int main()
{
        pid_t  pid;
        /* fork another process */
        pid = fork();
        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                exit(-1);
        }
        else if (pid == 0) { /* child process */
                execlp("/bin/ls", "ls", NULL);
        }
        else { /* parent process */
        /* parent will wait for the child to complete */
                wait (NULL);
                printf ("Child Complete");
                exit(0);
        }
}
```

execl: is used with a list comprising the command name and its arguments:

int execl(const char *path, const char *arg0, …../*, (char *) 0 */);

This is used when the number of arguments are known in advance. The first argument is the pathname which could be absolute or a relative pathname, The arguments to the command to run are represented as separate arguments beginning with the name of the command (*arg0). The ellipsis representation in the syntax (…/*) points to the varying number of arguments.

Example: How to use execl to run the wc –l command with the filename foo as argument:

execl (“/bin/wc”, “wc”, “-l”, “foo”, (char *) 0);
execl doesn’t use PATH to locate wc so pathname is specified as the first argument.

execv: needs an array to work with.

int execv(const char *path, char *const argv[ ]);

Here path represents the pathname of the command to run. The second argument represents an array of pointers to char. The array is populated by addresses that point to strings representing the command name and its arguments, in the form they are passes to the main function of the program to be executed. In this case also the last element of the argv[ ] must be a null pointer.

Here the following program uses execv program to run grep command with two options to look up the author’s name in /etc/passwd. The array *cmdargs[ ] are populated with the strings comprising the command line to be executed by execv. The first argument is the pathname of the command:

```
#include<stdio.h>
int main(int argc, char **argv){
char *cmdargs[ ] = {“grep”, “-I”, “-n”, “SUMIT”, “/etc/passed”, NULL};
execv(“/bin/grep”, cmdargs);
printf (“execv error\n”);
}
```

Drawbacks:

Need to know the location of the command file since neither execl nor execv will use PATH to locate it. The command name is specified twice - as the first two arguments. These calls can’t be used to run a shell script but only binary executable. The program has to be invoked every time there is a need to run a command.

execlp and execvp: requires pathname of the command to be located. They behave exactly like their other counterparts but overcomes two of the four limitations discussed above. First the first argument need not be a pathname it can be a command name. Second these functions can also run a shell script.

int execlp(const char *file, const char *arg0, …./*, (char *) 0 */);
int execvp(const char *file, char *const argv[ ]);

execlp (“wc”, “wc”, “-l”, “foo”, (char *) 0);

execle and execve: All of the previous four exec calls silently pass the environment of the current

process to the executed process by making available the environ[ ] variable to the overlaid process. Sometime there may be a need to provide a different environment to the new program - a restricted shell for instance. In that case these functions are used.

int execle(const char *path, const char *arg0, … /*, (char *) 0, char * const envp[ ] */);
int execve(const char *path, char * const argv[ ], char *const envp[ ]);

These functions unlike the others use an additional argument to pass a pointer to an array of environment strings of the form variable = value to the program. It's only this environment that is available in the executed process, not the one stored in envp[ ].

The following program (assume fork2.c) is the same as fork1.c, except that the number of messages printed by the child and parent processes is reversed. Here are the relevant lines of code:

```
switch(pid)
{
        case -1:
                perror("fork failed");
                exit(1);

        case 0:
                message = "This is the child";
                n = 3;
                break;
        default:
                message = "This is the parent";
                n = 5;
                break;
}
```

When the preceding program is run with ./fork2 & and then call the ps program after the child has finished but before the parent has finished, a line such as this. (Some systems may say <zombie> rather than <defunct>) is seen.

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.

THREAD MANAGEMENT

A process will start with a single thread which is called main thread or master thread. Calling pthread_create() creates a new thread. It takes the following parameters.

- A pointer to a pthread_t structure. The call will return the handle to the thread in this structure.
- A pointer to a pthread attributes structure, which can be a null pointer if the default attributes are to be used. The details of this structure will be discussed later.

- The address of the routine to be executed.
- A value or pointer to be passed into the new thread as a parameter.

```
#include <pthread.h>
#include <stdio.h>

void* thread_code( void * param )
{
printf( "In thread code\n" );
}
int main()
{
pthread_t thread;
pthread_create(&thread, 0, &thread_code, 0 );
printf("In main thread\n" );
}
```

In this example, the main thread will create a second thread to execute the routine thread_code(), which will print one message while the main thread prints another. The call to create the thread has a value of zero for the attributes, which gives the thread default attributes. The call also passes the address of a pthread_t variable for the function to store a handle to the thread. The return value from the thread_create() call is zero if the call is successful; otherwise, it returns an error condition.

Thread termination:
Child threads terminate when they complete the routine they were assigned to run. In the above example child thread thread will terminate when it completes the routine thread_code().

The value returned by the routine executed by the child thread can be made available to the main thread when the main thread calls the routine pthread_join().

The pthread_join() call takes two parameters. The first parameter is the handle of the thread that is to be waited for. The second parameter is either zero or the address of a pointer to a void, which will hold the value returned by the child thread.

The resources consumed by the thread will be recycled when the main thread calls pthread_join(). If the thread has not yet terminated, this call will wait until the thread terminates and then free the assigned resources.

```
#include <pthread.h>
#include <stdio.h>

void* thread_code( void * param )
```

```
{
printf( "In thread code\n" );
}
int main()
{
pthread_t thread;
pthread_create( &thread, 0, &thread_code, 0 );
printf( "In main thread\n" );
pthread_join( thread, 0 );
}
```

Another way a thread can terminate is to call the routine pthread_exit(), which takes a single parameter—either zero or a pointer—to void. This routine does not return and instead terminates the thread. The parameter passed in to the pthread_exit() call is returned to the main thread through the pthread_join(). The child threads do not need to explicitly call pthread_exit() because it is implicitly called when the thread exits.

Passing Data to and from Child Threads

In many cases, it is important to pass data into the child thread and have the child thread return status information when it completes. To pass data into a child thread, it should be cast as a pointer to void and then passed as a parameter to pthread_create().

```
for ( int i=0; i<10; i++ )
pthread_create( &thread, 0, &thread_code, (void *)i );
```

Following is a program where the main thread passes a value to the Pthread and the thread returns a value to the main thread.

```
#include <pthread.h>
#include <stdio.h>
void* child_thread( void * param )
{
int id = (int)param;
printf( "Start thread %i\n", id );
return (void *)id;
}

int main()
{
pthread_t thread[10];
int return_value[10];
for ( int i=0; i<10; i++ )
{
```

```
pthread_create( &thread[i], 0, &child_thread, (void*)i );
}
for ( int i=0; i<10; i++ )
{
pthread_join( thread[i], (void**)&return_value[i] );
printf( "End thread %i\n", return_value[i] );
}
}
```

Setting the Attributes for Pthreads

The attributes for a thread are set when the thread is created. To set the initial thread attributes, first create a thread attributes structure, and then set the appropriate attributes in that structure, before passing the structure into the pthread_create() call.

```
#include <pthread.h>
...
int main()
{
pthread_t thread;
pthread_attr_t attributes;
pthread_attr_init( &attributes );
pthread_create( &thread, &attributes, child_routine, 0 );
}
```

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. Block parent process until child completes using wait system call.
2. Write a C program to load the binary executable of the previous program in a child process using exec system call.
3. Create a zombie (defunct) child process (a child with exit() call, but no corresponding wait() in the sleeping parent) and allow the init process to adopt it (after parent terminates). Run the process as background process and run "ps" command.
4. Write a multithreaded program that performs different sorting algorithms. The program should work as follows: the user enters on the command line the number of elements to sort and the elements themselves. The program then creates separate threads, each using a different sorting algorithm. Each thread sorts the array using its corresponding algorithm and displays the time taken to produce the

result. The main thread waits for all threads to finish and then displays the final sorted array.

5. Write multithreaded program that generates the Fibonacci series. The program should work as follows: The user will enter on the command line the number of Fibonacci numbers that the program is to generate. The program then will create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that is shared by the threads (an array is probably the most convenient data structure). When the thread finishes execution the parent will output the sequence generated by the child thread. Because the parent thread cannot begin outputting the Fibonacci sequence until the child thread finishes, this will require having the parent thread wait for the child thread to finish.

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]
2. Create a orphan process (parent dies before child – adopted by "init" process) and display the PID of parent of child before and after it becomes orphan. Use sleep(n) in the child to delay the termination.
3. Modify the program in the previous question to include wait (&status) in the parent and to display the exit return code (left most byte of status) of the child.
4. Create a child process which returns a 0 exit status when the minute of time is odd and returning a non-zero (can be 1) status when the minute of time is even.
5. Write a multithreaded program for matrix multiplication.

LAB NO:5 **Date:**

CPU SCHEDULING ALGORITHMS

Objectives:

1. Understand the different CPU scheduling algorithms.
2. Compute the turnaround time, response time and waiting time for each process.

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.

Scheduling Criteria & Optimization:

- CPU utilization – keep the CPU as busy as possible
    - Maximize CPU utilization
- Throughput – # of processes that complete their execution per time unit
    - Maximize throughput
- Turnaround time – amount of time to execute a particular process

  - Minimize turnaround time
- Waiting time – amount of time a process has been waiting in the ready queue
  - Minimize waiting time
- Response time – time from the submission of a request until the first response is produced (response time, is the time it takes to start responding, not the time it takes to output the response)
  - Minimize response time

CPU Scheduling algorithms:

(i) First-Come First Served (FCFS) Scheduling:

The process that requests the CPU first is allocated the CPU first.

(ii) Shortest-Job-First (SJF) Scheduling:

This algorithm associates with each process the length of its 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.

Two schemes:

- Non-preemptive – once CPU given to the process it cannot be preempted until it completes its CPU burst.
- Preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF).

(iii) Priority Scheduling:

A priority number (integer) is associated with each process. The CPU is allocated to the process with the highest priority. A smaller value means a higher priority.

Two schemes:

- Preemptive
- Non-preemptive

Note: SJF is a priority scheduling where priority is the predicted next CPU burst time.

(iv) Round-Robin (RR) Scheduling:

The RR scheduling is the Preemptive version of FCFS. In RR scheduling, each process gets a small unit of CPU time (time quantum). Usually 10-100 ms. After quantum expires, the process is preempted and added to the end of the ready queue.

(v) Multilevel Queue (MQ) Scheduling:

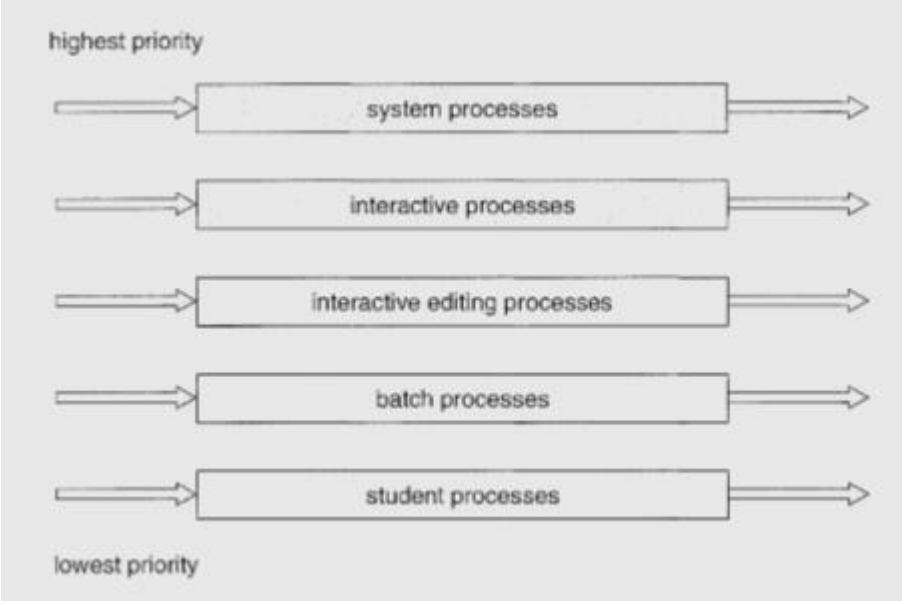


Figure 5.1: Multilevel queue scheduling

MQ scheduling is used when processes can be classified into groups. For example, **foreground** (interactive) processes and **background** (batch) processes. A MQ scheduling algorithm partitions the ready queue into several separate queues:

- foreground (interactive)
- background (batch)

Each process assigned to one queue based on its memory size, process priority, or process type. Each queue has its own scheduling algorithm

- foreground – RR
- background – FCFS

Scheduling must be done between the queues

- Fixed priority scheduling; (i.e., serve all from foreground then from background as shown Fig. 5.1).
- Time slice – each queue gets a certain portion of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR and 20% to background in FCFS.

(vi) Multilevel Feedback Queue (MFQ) Scheduling:

In MQ scheduling algorithm processes do not move from one queue to the other. In contrast, the MFQ scheduling algorithm, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue**.**

Example of Multilevel Feedback Queue:

Consider multilevel feedback queue scheduler with three queues as shown in Fig. 5.2.

- Q_0 – RR with time quantum 8 milliseconds
- Q_1 – RR time quantum 16 milliseconds
- Q_2 – FCFS

MFQ Scheduling

- A process queue in Q_0 *is given a time quantum of* 8 milliseconds. If it does not finish in 8 milliseconds, the job is moved to the tail of queue Q_1.
- When Q_0 is empty, the process at the head of Q_1 is given a quantum of 16 milliseconds. If it does not complete, it is pre-empted and moved to queue Q_2 .

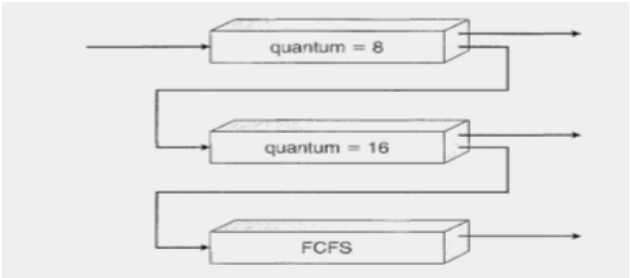


Figure 5.2: Multilevel feedback queues

2. Problem Description and Algorithm :

2.1 Round Robin Scheduling (RR):

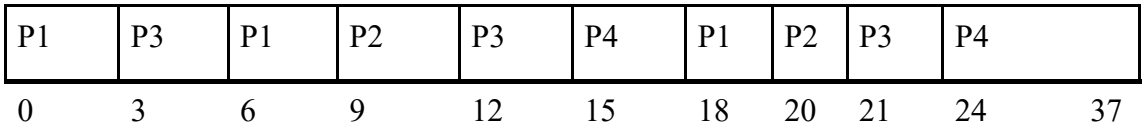
The **round-robin (RR) scheduling algorithm** is designed especially for timesharing systems. It is similar to (First Come First Serve) FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a **time quantum** or time slice, is defined. 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 firstprocess from the ready queue, sets a timer to interrupt after 1 time quantum, and dis- patches 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



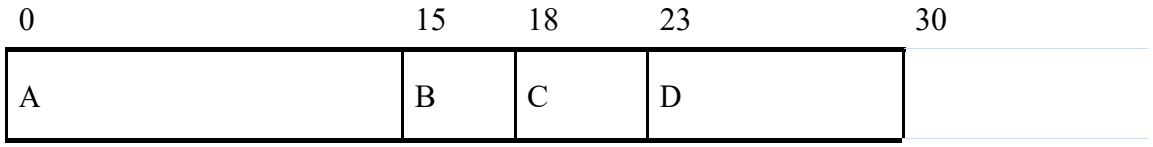
Average waiting time: (12+12+12+14)/4 = 12.5.

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.

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

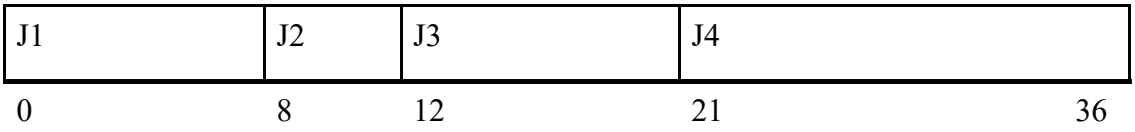
SJF Waiting time A= 0, B=10, C=10, D=13. TT A=15, B=13, C=15, D=20.



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.

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. Consider the following set of processes, with length of the CPU burst given in milliseconds:

Process	**Arrival time**	**Burst time**	***Priority***
P1	0	60	3
P2	3	30	2
P3	4	40	1
P4	9	10	4

Write a menu driven C program to simulate the following CPU scheduling algorithms. Display Gantt chart showing the order of execution of each process. Compute waiting time and turnaround time for each process. Hence, compute average waiting time and average turnaround

time.
(i) FCFS (ii) SRTF (iii) Round-Robin (quantum = 10) iv) non-preemptive priority (higher the number higher the priority)
2. Write a C program to simulate multi-level feedback queue scheduling algorithm.

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 SJF, where the arrival time is different for the processes.

LAB NO.: 6 **Date:**

INTERPROCESS COMMUNICATION

Objectives:

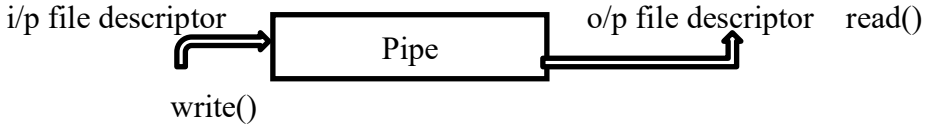
In this lab, student will be able to:

1. Gain knowledge as to how IPC (Interprocess Communication) happens between two processes.
2. Execute programs with respect to IPC using the different methods of message queues, pipes and shared memory.

Inter-Process communication (IPC), is the mechanism whereby one process can communicate with another process, i.e exchange of data. IPC in Linux can be implemented by using a pipe, shared memory and message queue.

Pipe

- Pipes are unidirectional byte streams which connect the standard output from one process into the standard input of another process. A pipe is created using the system call pipe that returns a pair of file descriptors.



- Call to the pipe () function which returns an array of file descriptors fd[0] and fd [1]. fd [1] connects to the write end of the pipe, and fd[0] connects to the read end of the pipe. Anything can be written to the pipe, and read from the other end in the order it came in.
- A pipe is one directional providing one-way flow of data and it is created by the pipe() system call.

```
int pipe ( int *filedes ) ;
```

- Array of two file descriptors are returned- fd[0] which is open for reading , and fd[1] which is open for writing. It can be used only between parent and child processes.

  PROTOTYPE: int pipe(int fd[2]);

  RETURNS: 0 on success

  -1 on error: errno = EMFILE (no free descriptors)

EMFILE (system file table is full)
EFAULT (fd array is not valid)

fd[0] is set up for reading, fd[1] is set up for writing. i.e., the first integer in the array (element 0) is set up and opened for reading, while the second integer (element 1) is set up and opened for writing.

```
#include <stdlib.h>
#include <stdio.h>      /* for printf */
#include <string.h>     /* for strlen */

int main(int argc, char **argv)
{
        int n;
        int fd[2];
        char buf[1025];
        char *data = "hello... this is sample data";
        pipe(fd);
        write(fd[1], data, strlen(data));
        if ((n = read(fd[0], buf, 1024)) >= 0) {
                buf[n] = 0;       /* terminate the string */
                printf("read %d bytes from the pipe: \"%s\"\n", n, buf);
        }
        else
                perror("read");
        exit(0);
}
```

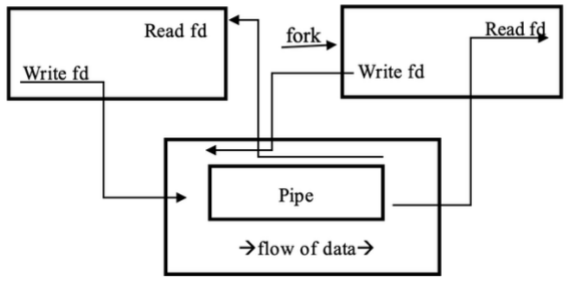


Fig. 6.1: Working of pipe in single process which is immediately after fork()

- First, a process creates a pipe and then forks to create a copy of itself.
- The parent process closes the read end of the pipe.
- The child process closes the write end of the pipe.
- The fork system call creates a copy of the process that was executing.
- The process which executes the fork is called the parent process and the new process is which is created is called the child process.

```
#include <sys/wait.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[])
{
   int pfd[2];
   pid_t cpid;
   char buf;
   assert(argc == 2);
   if (pipe(pfd) == -1) { perror("pipe");
      exit(EXIT_FAILURE); }
   cpid = fork();
   if (cpid == -1) { perror("fork");
   exit(EXIT_FAILURE); }

   if (cpid == 0) {    /* Child reads from pipe */
      close(pfd[1]);          /* Close unused write end */
      while (read(pfd[0], &buf, 1) > 0)
       write(STDOUT_FILENO, &buf, 1);
       write(STDOUT_FILENO, "\n", 1);
      close(pfd[0]);
      exit(EXIT_SUCCESS);

   } else {            /* Parent writes argv[1] to pipe */
      close(pfd[0]);          /* Close unused read end */
```

```
        write(pfd[1], argv[1], strlen(argv[1]));
        close(pfd[1]);          /* Reader will see EOF */
        wait(NULL);             /* Wait for child */
        exit(EXIT_SUCCESS);
    }
}
```

Message Queues

- It is an IPC facility. Message queues are similar to named pipes without the opening and closing of pipe. It provides an easy and efficient way of passing information or data between two unrelated processes.
- The advantages of message queues over named pipes is, it removes few difficulties that exists during the synchronization, the opening and closing of named pipes.
- A message queue is a linked list of messages stored within the kernel. A message queue is identified by a unique identifier. Every message has a positive long integer type field, a non-negative length, and the actual data bytes. The messages need not be fetched on FCFS basis. It could be based on type field.

Creating a Message Queue

- In order to use a message queue, it has to be created first. The msgget() system call is used for that. This system call accepts two parameters - a queue key and flags.
- IPC_PRIVATE - use to create a private message queue. A positive integer - used to create or access a publicly accessible message queue.

The message queue function definitions are

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
int msgget(key_t key, int msgflg);
int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);
int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);
```

msgget

We create and access a message queue using the msgget function:

int msgget(key_t key, int msgflg);

The program must provide a key value that, as with other IPC facilities, names a particular message queue. The special value IPC_PRIVATE creates a private queue, which in theory is accessible only by the current process. The second parameter, msgflg, consists of nine permission

flags. A special bit defined by IPC_CREAT must be bitwise ORed with the permissions to create a new message queue. It's not an error to set the IPC_CREAT flag and give the key of an existing message queue. The IPC_CREAT flag is silently ignored if the message queue already exists.

The msgget function returns a positive number, the queue identifier, on success or –1 on failure.

msgsnd

The msgsnd function allows us to add a message to a message queue:

int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);

The structure of the message is constrained in two ways. First, it must be smaller than the system limit, and second, it must start with a long int, which will be used as a message type in the receive function. When you're using messages, it's best to define your message structure something like this:

```
struct my_message {
long int message_type;
/* The data you wish to transfer */
}
```

The first parameter, msqid, is the message queue identifier returned from a msgget function. The second parameter, msg_ptr, is a pointer to the message to be sent, which must start with a long int type as described previously. The third parameter, msg_sz, is the size of the message pointed to by msg_ptr. This size must not include the long int message type. The fourth parameter, msgflg, controls what happens if either the current message queue is full or the system wide limit on queued messages has been reached. If msgflg has the IPC_NOWAIT flag set, the function will return immediately without sending the message and the return value will be –1. If the msgflg has the IPC_NOWAIT flag clear, the sending process will be suspended, waiting for space to become available in the queue. On success, the function returns 0, on failure –1. If the call is successful, a copy of the message data has been taken and placed on the message queue.

msgrcv

The msgrcv function retrieves messages from a message queue:

int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);

The first parameter, msqid, is the message queue identifier returned from a msgget function. The second parameter, msg_ptr, is a pointer to the message to be received, which must start with a long int type as described above in the msgsnd function. The third parameter, msg_sz, is the size of the message pointed to by msg_ptr, not including the long int message type. The fourth parameter, msgtype, is a long int, which allows a simple form of reception priority to be implemented. If msgtype has the value 0, the first available message in the queue is retrieved. If it's greater than zero, the first message with the same message type is retrieved. If it's less than

zero, the first message that has a type the same as or less than the absolute value of msgtype is retrieved. This sounds more complicated than it actually is in practice. If you simply want to retrieve messages in the order in which they were sent, set msgtype to 0. If you want to retrieve only messages with a specific message type, set msgtype equal to that value. If you want to receive messages with a type of n or smaller, set msgtype to -n. The fifth parameter, msgflg, controls what happens when no message of the appropriate type is waiting to be received. If the IPC_NOWAIT flag in msgflg is set, the call will return immediately with a return value of –1. If the IPC_NOWAIT flag of msgflg is clear, the process will be suspended, waiting for an appropriate type of message to arrive. On success, msgrcv returns the number of bytes placed in the receive buffer, the message is copied into the user-allocated buffer pointed to by msg_ptr, and the data is deleted from the message queue. It returns –1 on error.

msgctl

The final message queue function is msgctl.

int msgctl(int msqid, int command, struct msqid_ds *buf);

The msqid_ds structure has at least the following members:

```
struct msqid_ds {
uid_t msg_perm.uid;
uid_t msg_perm.gid
mode_t msg_perm.mode;
}
```

The first parameter, msqid, is the identifier returned from msgget. The second parameter, command, is the action to take. It can take three values:

Command Description

Command	Description
IPC_STAT	Sets the data in the msqid_ds structure to reflect the values associated with the message queue.
IPC_SET	If the process has permission to do so, this sets the values associated with the message queue to those provided in the msqid_ds data structure.
IPC_RMID	Deletes the message queue.

0 is returned on success, –1 on failure. If a message queue is deleted while a process is waiting in a msgsnd or msgrcv function, the send or receive function will fail.

Receiver program:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
struct my_msg_st {
        long int my_msg_type;
        char some_text[BUFSIZ];
};

int main()
{
        int running = 1;
        int msgid;
        struct my_msg_st some_data;
        long int msg_to_receive = 0;
        msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
        if (msgid == -1) {
                fprintf(stderr, "msgget failed with error: %d\n", errno);
                exit(EXIT_FAILURE);
        }
        while(running) {
                if (msgrcv(msgid, (void *)&some_data, BUFSIZ,
                        msg_to_receive, 0) == -1) {
                        fprintf(stderr, "msgrcv failed with error: %d\n", errno);
                        exit(EXIT_FAILURE);
                }
                printf("You wrote: %s", some_data.some_text);
                if (strncmp(some_data.some_text, "end", 3) == 0) {
                        running = 0;
                }
        }
        if (msgctl(msgid, IPC_RMID, 0) == -1) {
                fprintf(stderr, "msgctl(IPC_RMID) failed\n");
                exit(EXIT_FAILURE);
        }
        exit(EXIT_SUCCESS);
```

```
}
```

Sender Program:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_TEXT 512

struct my_msg_st {
        long int my_msg_type;
        char some_text[MAX_TEXT];
};

int main()
{
        int running = 1;
        struct my_msg_st some_data;
        int msgid;
        char buffer[BUFSIZ];
        msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
        if (msgid == -1) {
                fprintf(stderr, "msgget failed with error: %d\n", errno);
                exit(EXIT_FAILURE);
        }
        while(running) {
                printf("Enter some text:");
                fgets(buffer, BUFSIZ, stdin);
                some_data.my_msg_type = 1;
                strcpy(some_data.some_text, buffer);
                if (msgsnd(msgid, (void *)&some_data, MAX_TEXT, 0) == -1) {
                        fprintf(stderr, "msgsnd failed\n");
```

```
                exit(EXIT_FAILURE);
            }
            if (strncmp(buffer, "end", 3) == 0) {
                running = 0;
            }
        }
        exit(EXIT_SUCCESS);
}
```

Shared memory

Shared memory allows two or more processes to access the same logical memory. Shared memory is an efficient of transferring data between two running processes. Shared memory is a special range of addresses that is created by one process and the Shared memory appears in the address space of that process. Other processes then attach the same shared memory segment into their own address space. All processes can then access the memory location as if the memory had been allocated just like malloc. If one process writes to the shared memory, the changes immediately become visible to any other process that has access to the same shared memory.

The functions for shared memory are,

#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
void *shmat(int shm_id, const void *shm_addr, int shmflg);
int shmctl(int shm_id, int cmd, struct shmid_ds *buf);
int shmdt(const void *shm_addr);

The include files sys/types.h and sys/ipc.h are normally also required before shm.h is included.

shmget

We create shared memory using the shmget function:

int shmget(key_t key, size_t size, int shmflg);

The argument key names the shared memory segment, and the shmget function returns a shared memory identifier that is used in subsequent shared memory functions. There's a special key value, IPC_PRIVATE, that creates shared memory private to the process. The second parameter, size, specifies the amount of memory required in bytes. The third parameter, shmflg, consists of nine permission flags that are used in the same way as the mode flags for creating files. A special bit defined by IPC_CREAT must be bitwise ORed with the permissions to create a new shared memory segment. It's not an error to have the IPC_CREAT flag set and pass the key of an existing shared memory segment. The IPC_CREAT flag is silently ignored if it is not required.

The permission flags are very useful with shared memory because they allow a process to create shared memory that can be written by processes owned by the creator of the shared memory but only read by processes that other users have created. We can use this to provide efficient read-only access to data by placing it in shared memory without the risk of its being changed by other users.

If the shared memory is successfully created, shmget returns a nonnegative integer, the shared memory identifier. On failure, it returns –1.

shmat

When we first create a shared memory segment, it's not accessible by any process. To enable access to the shared memory, we must attach it to the address space of a process. We do this with the shmat function:

void *shmat(int shm_id, const void *shm_addr, int shmflg);

The first parameter, shm_id, is the shared memory identifier returned from shmget. The second parameter, shm_addr, is the address at which the shared memory is to be attached to the current process. This should almost always be a null pointer, which allows the system to choose the address at which the memory appears. The third parameter, shmflg, is a set of bitwise flags. The two possible values are SHM_RND, which, in conjunction with shm_addr, controls the address at which the shared memory is attached, and SHM_RDONLY, which makes the attached memory read-only. It's very rare to need to control the address at which shared memory is attached; you should normally allow the system to choose an address for you, as doing otherwise will make the application highly hardware-dependent. If the shmat call is successful, it returns a pointer to the first byte of shared memory. On failure –1 is returned.

The shared memory will have read or write access depending on the owner (the creator of the shared memory), the permissions, and the owner of the current process. Permissions on shared memory are similar to the permissions on files. An exception to this rule arises if shmflg & SHM_RDONLY is true. Then the shared memory won't be writable, even if permissions would have allowed write access.

shmdt

The shmdt function detaches the shared memory from the current process. It takes a pointer to the address returned by shmat. On success, it returns 0, on error –1. Note that detaching the shared memory doesn't delete it; it just makes that memory unavailable to the current process.

shmctl

int shmctl(int shm_id, int command, struct shmid_ds *buf);

The first parameter, shm_id, is the identifier returned from shmget. The second parameter,

command, is the action to take. It can take three values:

Command	Description
IPC_STAT	Sets the data in the shmid_ds structure to reflect the values associated with the shared memory.
IPC_SET	Sets the values associated with the shared memory to those provided in the shmid_ds data structure, if the process has permission to do so.
IPC_RMID	Deletes the shared memory segment.

The shmid_ds structure has the following members:

```
struct shmid_ds {
        uid_t shm_perm.uid;
        uid_t shm_perm.gid;
        mode_t shm_perm.mode;
}
```

The third parameter, buf, is a pointer to structure containing the modes and permissions for the shared memory. On success, it returns 0, on failure returns –1.

We will write a pair of programs shm1.c and shm2.c. The first will create a shared memory segment and display any data that is written into it. The second will attach into an existing shared memory segment and enters data into shared memory segment.

First, we create a common header file to describe the shared memory we wish to pass around. We call this shm_com.h.

```
#define TEXT_SZ 2048

struct shared_use_st {
        int written_by_you;
        char some_text[TEXT_SZ];
};

//shm1.c – Consumer process
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
#include “shm_com.h”

int main()
{
        int running = 1;
        void *shared_memory = (void *)0;
        struct shared_use_st *shared_stuff;
        int shmid;
        srand((unsigned int)getpid());
        shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);
        if (shmid == -1) {
                fprintf(stderr, “shmget failed\n”);
                exit(EXIT_FAILURE);
        }
        shared_memory = shmat(shmid, (void *)0, 0);
        if (shared_memory == (void *)-1) {
                fprintf(stderr, “shmat failed\n”);
                exit(EXIT_FAILURE);
        }
        printf(“Memory attached at %X\n”, (int)shared_memory);
        shared_stuff = (struct shared_use_st *)shared_memory;
        shared_stuff->written_by_you = 0;
        while(running) {
                if (shared_stuff->written_by_you) {
                printf(“You wrote: %s”, shared_stuff->some_text);
                sleep( rand() % 4 ); /* make the other process wait for us ! */
                shared_stuff->written_by_you = 0;
                if (strncmp(shared_stuff->some_text, “end”, 3) == 0) {
                        running = 0;
                }
        }
        if (shmdt(shared_memory) == -1) {
                fprintf(stderr, “shmdt failed\n”);
                exit(EXIT_FAILURE);
        }
        if (shmctl(shmid, IPC_RMID, 0) == -1) {
                fprintf(stderr, “shmctl(IPC_RMID) failed\n”);
                exit(EXIT_FAILURE);
        }
        exit(EXIT_SUCCESS);
}
```

//shm2.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "shm_com.h"
int main()
{
        int running = 1;
        void *shared_memory = (void *)0;
        struct shared_use_st *shared_stuff;
        char buffer[BUFSIZ];
        int shmid;
        shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);
        if (shmid == -1) {
                fprintf(stderr, "shmget failed\n");
                exit(EXIT_FAILURE);
        }
        shared_memory = shmat(shmid, (void *)0, 0);
        if (shared_memory == (void *)-1) {
                fprintf(stderr, "shmat failed\n");
                exit(EXIT_FAILURE);
        }
        printf("Memory attached at %X\n", (int)shared_memory);
        shared_stuff = (struct shared_use_st *)shared_memory;
        while(running) {
                while(shared_stuff->written_by_you == 1) {
                        sleep(1);
                        printf("waiting for client...\n");
                }
                printf("Enter some text:");
                fgets(buffer, BUFSIZ, stdin);
                strncpy(shared_stuff->some_text, buffer, TEXT_SZ);
                shared_stuff->written_by_you = 1;
                if (strncmp(buffer, "end", 3) == 0) {
                        running = 0;
                }
        }
        if (shmdt(shared_memory) == -1) {
                fprintf(stderr, "shmdt failed\n");
                exit(EXIT_FAILURE);
```

```
        }
        exit(EXIT_SUCCESS);
}
```

Named Pipes: FIFOs

Pipes can share data between related processes, i.e. processes that have been started from a common ancestor process. We can use named pipe or FIFOs to overcome this. A named pipe is a special type of file that exists as a name in the file system but behaves like the unnamed pipes we have discussed already.We can create named pipes from the command line using

$ mkfifo filename

From inside a program, we can use

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

int mkfifo(const char *filename, mode_t mode);

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
int main()
    {
        int res = mkfifo("/tmp/my_fifo", 0777);
        if (res == 0) printf("FIFO created\n");
        exit(EXIT_SUCCESS);
    }
```

We can look for the pipe with

$ **ls -lF /tmp/my_fifo**

prwxr-xr-x 1 rick users 0 July 10 14:55 /tmp/my_fifo|

Notice that the first character of output is a p, indicating a pipe. The | symbol at the end is added by the ls command's -F option and also indicates a pipe. We can remove the FIFO just like a conventional file by using the rm command, or from within a program by using the unlink system

call.

Producer-Consumer Problem (PCP):

- Producer process produces information that is consumed by a consumer process. 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 consumer. Two types of buffers can be used.
  - *unbounded-buffer* places no practical limit on the size of the buffer.
  - *bounded-buffer* assumes that there is a fixed buffer size.
- For bounded-buffer PCP basic synchronization requirement is:
  - Producer should not write into a full buffer (i.e. producer must wait if the buffer is full)
  - Consumer should not read from an empty buffer (i.e. consumer must wait if the buffer is empty)
  - All data written by the producer must be read exactly once by the consumer

Following is a program for Producer-Consumer problem using named pipes.

//producer.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF
#define TEN_MEG (1024 * 1024 * 10)

int main()
{
        int pipe_fd;
        int res;
        int open_mode = O_WRONLY;
        int bytes_sent = 0;
        char buffer[BUFFER_SIZE + 1];
        if (access(FIFO_NAME, F_OK) == -1) {
```

```
            res = mkfifo(FIFO_NAME, 0777);
            if (res != 0) {
                fprintf(stderr, "Could not create fifo %s\n", FIFO_NAME);
                exit(EXIT_FAILURE);
            }
        }
        printf("Process %d opening FIFO O_WRONLY\n", getpid());
        pipe_fd = open(FIFO_NAME, open_mode);
        printf("Process %d result %d\n", getpid(), pipe_fd);
        if (pipe_fd != -1) {
            while(bytes_sent < TEN_MEG) {
                res = write(pipe_fd, buffer, BUFFER_SIZE);
                if (res == -1) {
                    fprintf(stderr, "Write error on pipe\n");
                    exit(EXIT_FAILURE);
                }
                bytes_sent += res;
            }
            (void)close(pipe_fd);
        }
        else {
            exit(EXIT_FAILURE);
        }
        printf("Process %d finished\n", getpid());
        exit(EXIT_SUCCESS);
}
```

//consumer.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
```

```
#include <sys/stat.h>
#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF
int main()
{
        int pipe_fd;
        int res;
        int open_mode = O_RDONLY;
        char buffer[BUFFER_SIZE + 1];
        int bytes_read = 0;
        memset(buffer, '\0', sizeof(buffer));
        printf("Process %d opening FIFO O_RDONLY\n", getpid());
        pipe_fd = open(FIFO_NAME, open_mode);
        printf("Process %d result %d\n", getpid(), pipe_fd);
        if (pipe_fd != -1) {
                do {
                        res = read(pipe_fd, buffer, BUFFER_SIZE);
                        bytes_read += res;
                } while (res > 0);
                (void)close(pipe_fd);
        }
        else {
                exit(EXIT_FAILURE);
        }
        printf("Process %d finished, %d bytes read\n", getpid(), bytes_read);
        exit(EXIT_SUCCESS);
}
```

The Readers-Writers Problem:

- Concurrent processes share a file, record, or other resources
- Some may read only (readers), some may both read and write (writers)
- Two concurrent reads have no adverse effects
- Problems if
  - concurrent reads and writes
  - multiple writes

Two Variations

- First Readers-Writers problem: No reader be kept waiting unless a writer has already obtained exclusive write permissions (Readers have high priority)
- Second Readers-Writers problem: If a writer is waiting/ready , no new readers may start reading (Writers have high priority)

Lab Exercises:

1. Process A wants to send a number to Process B. Once received, Process B has to check whether the number is palindrome or not. Write a C program to implement this interprocess communication using message queue. .
2. Write a producer and consumer program in C using FIFO queue. The producer should write a set of 4 integers into the FIFO queue and the consumer should display the 4 integers.
3. Implement a parent process, which sends an English alphabet to child process using shared memory. Child process responds back with next English alphabet to the parent. Parent displays the reply from the Child.
4. Write a producer-consumer program in C in which producer writes a set of words into shared memory and then consumer reads the set of words from the shared memory. The shared memory need to be detached and deleted after use.

Additional Exercises:

1. Demonstrate creation, writing to and reading from a pipe.
2. Demonstrate creation of a process which writes through a pipe while the parent process reads from it.
3. Write a program which creates a message queue and writes message into queue which contains number of users working on the machine along with observed time in hours and minutes. This is repeated for every 10 minutes. Write another program which reads this information form the queue and calculates on average in each hour how many users are working.

LAB NO: 7 **Date:**

CLASSICAL PROBLEMS OF SYNCHRONIZATION

Objectives:

2. To synchronize various processes with the use of semaphore.
2. To understand how communication between two processes can take place with the help of named pipe.

For a multithreaded application spanning a single process or multiple processes to do useful work, it is necessary for some kind of common state to be shared between the threads. The degree of sharing that is necessary depends on the task. At one extreme, the only sharing necessary may be a single number that indicates the task to be performed. For example, a thread in a web server might be told only the port number to respond to. At the other extreme, a pool of threads might be passing information constantly among themselves to indicate what tasks are complete and what work is still to be completed.

Data Races

Data race occurs when multiple threads spanning single process or multiple processes use the same data item and one or more of those threads are updating it.

Suppose there is a function update, which takes an integer pointer and updates the value of the content pointer by 4. If multiple threads call the function, then there is a possibility of data race. If the current value of *a is 10, then when two threads simultaneously call update function, then the final value of *a might be 14, instead of 18. To visualize this, we need to write the corresponding assembly language code for this function.

```
void update(int * a)
{
*a = *a + 4;
}
```

Another situation might be when one thread is running, but the other thread has been context switched off of the processor. Imagine that the first thread has loaded the value of the variable **a** and then gets context switched off the processor. When it eventually runs again, the value of the variable a will have changed, and the final store of the restored thread will cause the value of the variable a to regress to an old value. The following code has data race.

```
//race.c
#include <pthread.h>
int counter = 0;
void * func(void * params)
{
        counter++;
}
void main()
{
        pthread_t thread1, thread2;
        pthread_create(&thread1, 0, func, 0);
        pthread_create(&thread2, 0, func, 0);
        pthread_join(thread1, 0 );
        pthread_join(thread2, 0 );
}
```

Using tools to detect data races

We can compile the above code using gcc, and then use Helgrind tool which is part of Valgrind suite to identify the data race.

```
$ gcc –g race.c -lpthread

$ valgrind –tool=helgrind ./a.out
```

Avoiding Data Races

Although it is hard to identify data races, avoiding them can be very simple. The easiest way to do this is to place a synchronization lock around all accesses to that variable and ensure that before referencing the variable, the thread must acquire the lock.

Synchronization Primitives:

Mutex Locks:

A mutex lock is a mechanism that can be acquired by only one thread at a time. Other threads that attempt to acquire the same mutex must wait until it is released by the thread that currently has it.

Mutex locks need to be initialized to the appropriate state by a call to pthread_mutex_init() or for statically defined mutexes by assignment with the PTHREAD_MUTEX_INITIALIZER. The call to pthread_mutex_init() takes an optional parameter that points to attributes describing the type of mutex required. Initialization through static assignment uses default parameters, as does passing in a null pointer in the call to pthread_mutex_init().

Once a mutex is no longer needed, the resources it consumes can be freed with a call to pthread_mutex_destroy().

```
#include <pthread.h>

...

pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2;
pthread_mutex_init( &m2, 0 );
...
pthread_mutex_destroy( &m1 );
pthread_mutex_destroy( &m2 );
```

A thread can lock a mutex by calling pthread_mutex_lock(). Once it has finished with the mutex, the thread calls pthread_mutex_unlock(). If a thread calls pthread_mutex_lock() while another thread holds the mutex, the calling thread will wait, or *block*, until the other thread releases the mutex, allowing the calling thread to attempt to acquire the released mutex.

```
#include <pthread.h>
#include <stdio.h>
pthread_mutex_t mutex;
volatile int counter = 0;
void * count( void * param)
{
        for ( int i=0; i<100; i++)
        {
                pthread_mutex_lock(&mutex);
```

```
                counter++;
                printf("Count = %i\n", counter);
                pthread_mutex_unlock(&mutex);
        }
}
int main()
{
        pthread_t thread1, thread2;
        pthread_mutex_init( &mutex, 0 );
        pthread_create( &thread1, 0, count, 0 );
        pthread_create( &thread2, 0, count, 0 );
        pthread_join( thread1, 0 );
        pthread_join( thread2, 0 );
        pthread_mutex_destroy( &mutex );
        return 0;
}
```

Semaphores:

A semaphore is a counting and signaling mechanism. One use for it is to allow threads access to a specified number of items. If there is a single item, then a semaphore is essentially the same as a mutex, but it is more commonly useful in a situation where there are multiple items to be managed.

A semaphore is initialized with a call to sem_init(). This function takes three parameters. The first parameter is a pointer to the semaphore. The next is an integer to indicate whether the semaphore is shared between multiple processes or private to a single process. The final parameter is the value with which to initialize the semaphore. A semaphore created by a call to sem_init() is destroyed with a call to sem_destroy().

The code below initializes a semaphore with a count of 10. The middle parameter of the call to sem_init() is zero, and this makes the semaphore private to the process; passing the value one rather than zero would enable the semaphore to be shared between multiple processes.

```
#include <semaphore.h>
int main()
{
        sem_t semaphore;
        sem_init( &semaphore, 0, 10 );
        ...
        sem_destroy( &semaphore );
}
```

The semaphore is used through a combination of two methods. The function sem_wait() will attempt to decrement the semaphore. If the semaphore is already zero, the calling thread will wait until the semaphore becomes nonzero and then return, having decremented the semaphore. The call to sem_post() will increment the semaphore. One more call, sem_getvalue(), will write the current value of the semaphore into an integer variable.

In the following program a order is maintained in displaying Thread 1 and Thread 2. Try removing the semaphore and observe the output.

```
#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>

sem_t semaphore;

void *func1( void * param )
{
        printf( "Thread 1\n" );
        sem_post( &semaphore );
}
void *func2( void * param )
{
        sem_wait( &semaphore );
        printf( "Thread 2\n" );
}
int main()
{
        pthread_t threads[2];
        sem_init( &semaphore, 0, 1 );
        pthread_create( &threads[0], 0, func1, 0 );
        pthread_create( &threads[1], 0, func2, 0 );
        pthread_join( threads[0], 0 );
        pthread_join( threads[1], 0 );
        sem_destroy( &semaphore );
}
```

1. Classical Problems :

1.1 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.

1.2 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.

1.3 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.*

Solution to Producer-Consumer problem

```
#include<stdio.h>
#include<pthread.h>
#include<semaphore.h>
int buf[5],f,r;
sem_t mutex,full,empty;
void *produce(void *arg)
{
   int i;
   for(i=0;i<10;i++)
   {
      sem_wait(&empty);
      sem_wait(&mutex);
      printf("produced item is %d\n",i);
      buf[(++r)%5]=i;
      sleep(1);
      sem_post(&mutex);
      sem_post(&full);
      printf("full %u\n",full);
   }
}
void *consume(void *arg)
{
      int item,i;
      for(i=0;i<10;i++)
      {
            sem_wait(&full);
            printf("full %u\n",full);
            sem_wait(&mutex);
            item=buf[(++f)%5];
            printf("consumed item is %d\n",item);
            sleep(1);
            sem_post(&mutex);
            sem_post(&empty);
            int val;
            sem_getvalue(&wt,&val)
if( val <10)
sem_post(wrt);
      }
}
main()
```

```
{
   pthread_t tid1,tid2;
   sem_init(&mutex,0,1);
   sem_init(&full,0,0);
   sem_init(&empty,0,5);
   pthread_create(&tid1,NULL,produce,NULL);
   pthread_create(&tid2,NULL,consume,NULL);
   pthread_join(tid1,NULL);
   pthread_join(tid2,NULL);
}
```

Solution to First Readers-Writers Problem using semaphores:

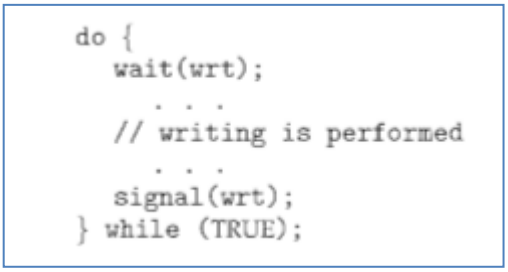
- The reader processes share the following data structures:

  semaphore mutex , wrt;

  int readcount;

- The binary semaphores mutex and wrt are initialized to 1; readcount is initialized to 0;
- Semaphore wrt is common to both reader and writer process
  - wrt functions as a mutual exclusion for the writers
  - It is also used by the first or last reader that enters or exits the critical section
  - It is not used by readers who enter or exit while other readers are in their critical section
- The readcount variable keeps track of how many processes are currently reading the object
- The mutex semaphore is used to ensure mutual exclusion when readcount is updated

The structure of a writer process



```
do {
   wait(wrt);
      . . .
   // writing is performed
      . . .
   signal(wrt);
} while (TRUE);
```

The structure of a reader process

```
do {
   wait(mutex);
   readcount++;
   if (readcount == 1)
     wait(wrt);
   signal(mutex);
      . . .
   // reading is performed
      . . .
   wait(mutex);
   readcount--;
   if (readcount == 0)
     signal(wrt);
   signal(mutex);
} while (TRUE);
```

The Dining Philosophers Problem:

- Five philosophers sit at a round table - thinking and eating
- Each philosopher has one chopstick
  - five chopsticks total
- A philosopher needs two chopsticks to eat
  - philosophers must share chopsticks to eat
- No interaction occurs while thinking

The situation of the dining philosophers is shown in Figure 7.1

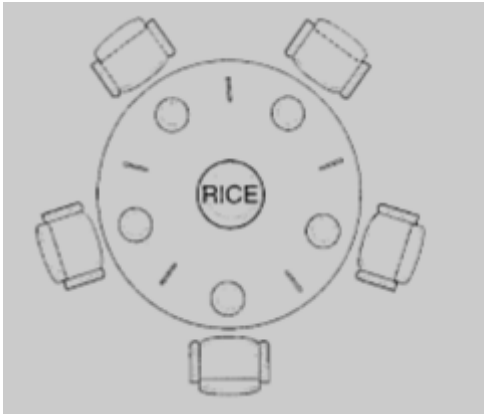


Figure 7.1

Lab Exercise

1. Modify the above Producer-Consumer program so that, a producer can produce at the

most 10 items more than what the consumer has consumed.

2. Write a C program for first readers-writers problem using semaphores.

Additional Exercise

1. Write a C program for Dining-Philosophers problem using semaphores.

LAB NO: 8 **Date:**

DEADLOCK MANAGEMENT ALGORITHMS

Objectives:

1. To understand how deadlocks occurs in a computer system.
2. To implement different algorithms for preventing or avoiding deadlocks in a computer system.

The deadlock problem:

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

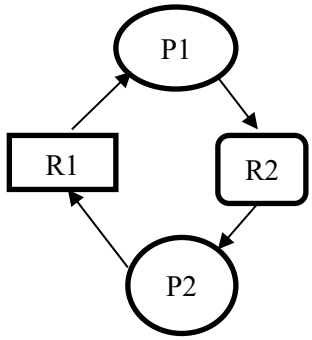


Figure 8.1: Deadlock Situation

Above Fig. 8.1 shows, a situation of deadlock where Process P1 Waiting for resource R2, which is held with process P2 and in the meantime, Process P2 is waiting for resource R1, which is held with process P1. Neither P1 nor P2 can proceed their execution until their needed resources are fulfilled forming a cyclic wait. It is the deadlock situation among processes as both are not progressed. In a single instance of resource type, a cyclic wait is always a deadlock.

Consider Figure 8.2 below, the situation with 4 processes P1, P2, P3 and P4 and 2 resources R1 and R2 both are of two instances. Here, there is no deadlock even though the cycle exists between processes P1 and P3. Once P2 finishes its job, 1 instance of resource will be available which can be accessed by process P1, which turns request edge to assignment edge, thereby removing cyclic-wait. So, in multiple instances of resource type, the cyclic-wait need not be deadlock.

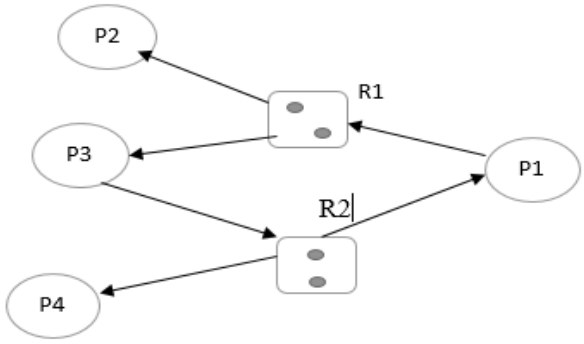


Figure 8.2: Cyclic-wait but no deadlock

Methods for Handling Deadlocks:

(i) Deadlock Avoidance:

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 State:

System is in safe state if there exists a safe sequence of all processes. **Sequence** of processes $<P_1, P_2, \ldots, P_n>$ is **safe** if for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$.

- 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 $\Rightarrow$ no deadlocks.

If a system is in unsafe state $\Rightarrow$ possibility of deadlock.

Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

Banker's Algorithm:

Used when there exists **multiple** instances of a resource type. Each process must **declare in advance the maximum** number of instances of each resource type that it may need. When a process requests a resource, it may have to wait. When a process gets all its resources, it must return them in a finite amount of time

Data Structures for the Banker's Algorithm:
Let n = number of processes, and m = number of resources types.

Available: Vector of length m. If available [j] = k, there are k instances of resource type R_j available.
Max: n x m matrix. If Max [i, j] = k, then process P_i may request at most k instances of resource type R_j.
Allocation: n x m matrix. If Allocation[i, j] = k then P_i is currently allocated k instances of R_j.
Need: n x m matrix. If $Need$[i, j] = k, then P_i may need k more instances of R_j to complete its task.
$Need$ [i, j] = Max[i, j] – $Allocation$ [i, j].

Safety Algorithm:
Allocation$_i$ means resources allocated to process P_i
Need$_i$ means resources needed for the process P_i

1. Let $Work$ and $Finish$ be vectors of length m and n, respectively.
Initialize:
$Work = Available$
$Finish$ [i] = $false$ for i = 0,1, …, n-1.
2. Find an i such that both:
(a) $Finish$ [i] = $false$
(b) $Need_i \leq Work$
If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$
$Finish$[i] = $true$
go to step 2.
4. If $Finish$ [i] == true for all i, then the system is in a safe state.

The above algorithm may require an O(m x n^2) operations to determine whether a state is safe.

Resource-Request Algorithm:
$Request_i$ = request vector for process P_i.
$Request_i$ [j] == k means that process P_i wants k instances of resource type R_j.

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to 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 safe* ➡ *the resources are allocated to* P_i.
- *If unsafe* ➡ P_i *must wait for* $Request_i$, *and the old resource-allocation state is restored*

(ii) Deadlock Detection:

For deadlock detection, the system must provide

- An algorithm that examines the state of the system to detect whether a deadlock has occurred
- And an algorithm to recover from the deadlock

Deadlock detection algorithm:

If a resource type can have multiple instances, then an algorithm very similar to the banker's algorithm can be used.

Required data structures:

Available: A vector of length *m* indicates the number of available resources of each type.
Allocation: An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.
Request: An *n* x *m* matrix indicates the current request of each process. If *Request* [*i*][j] == *k*, then process P_i is requesting *k* more instances of resource type R_j.

Detection Algorithm:

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize: *Work* = *Available*. For *i* = 0,2, …, *n-1*, if $Allocation_i \neq 0$, then *Finish*[i] = false; otherwise, *Finish*[i] = *true*.
2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$
   If no such *i* exists, go to step 4.
3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2.
4. If *Finish*[*i*] == false, for some *i*, $0 \leq i < n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then process P_i is deadlocked

The above algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state.

Lab Exercises

1. Consider the following snapshot of the system. Write C program to implement Banker's algorithm for deadlock avoidance. The program has to accept all inputs from the user. Assume the total number of instances of A,B and C are 10,5 and 7 respectively.

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

(a) What is the content of the matrix *Need*?
(b) Is the system in a safe state?
(c) If a request from process P1 arrives for (1, 0, 2), can the request be granted immediately? Display the updated Allocation, Need and Available matrices.
(d) If a request from process P4 arrives for (3, 3, 0), can the request be granted immediately?
(e) If a request from process P0 arrives for (0, 2, 0), can the request be granted immediately?

2. Consider the following snapshot of the system. Write C program to implement deadlock detection algorithm.
   (a) Is the system in a safe state?
   (b) Suppose that process P2 make one additional request for instance of type C, can the system still be in a safe state?

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Additional Exercises:

1. Write a multithreaded program that implements the banker's algorithm. Create n threads that request and release resources from the bank. The banker will grant the request only if it leaves the system in a safe state. You may write this program using either Pthreads. It is important that shared data be safe from concurrent access. To ensure safe access to shared data, you can use mutex locks, which are available in the Pthreads libraries.

LAB NO: 9 **Date:**

MEMORY MANAGEMENT I

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.
3. To learn Understand various memory management schemes.

Memory allocation methods:

1. Fixed-size partition: Divide memory into a fixed no. of partitions, and allocate a process to each. Partitions can be different *fixed* sizes. Each partition may contain exactly one process.
2. Variable size partition : The Operating system keeps a table indicating which part of memory are available and which are occupied. Initially, all memory is available to user processes and is considered as one large block of available memory called a hole. Hence, memory contains a set of holes of various sizes.

Dynamic storage allocation problem:

Which concerns how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. First fit, best fit and worst fit strategies are the ones most commonly used to select a free hole from a set of available holes.

- **First-fit:** Allocate the *first* hole that is big enough
- **Best-fit:** Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit:** Allocate the *largest* hole; must also search entire list
  - Produces the largest leftover hole

Fragmentation: Both first-fit and best-fit strategies suffer from external fragmentation. External fragmentation exists, when there is enough total memory exists to satisfy a memory request but the available spaces are not contiguous; storage is fragmented into a large number of small holes.

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 avail-

able 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.

1. 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

One solution to the problem of external fragmentation (Lab 9 Technique) is compaction. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Memory fragmentation can be internal also. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is internal fragmentation – unused memory that is internal to a partition.

Paging:

- Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous
    - Avoids external fragmentation
    - Avoids the need for compaction
    - May still have internal fragmentation
- Divide physical memory into fixed-sized blocks called frames
    - the frames may be located anywhere in memory
- Divide logical memory into blocks of same size called pages
    - Backing store is also split into pages of the same size

To run a program of size *N pages, need to find N free frames and load program.*

- Each logical address has two parts:
    - *Page number (p)* - index to page table
        - *page table* contains the mapping from a page number to the base address of its corresponding frame
    - *Page offset (d)* - offset into page/frame
- The size of a page is typically a power of 2:
    - 512 (2^9) -- 8192 (2^{13}) bytes
- This makes it easy to split a machine address into page number and offset parts.
- For example, assume:

- the memory size is 2^m bytes
- a page size is 2^n bytes (n < m)

Paging Hardware is shown in the following Figure 10.1

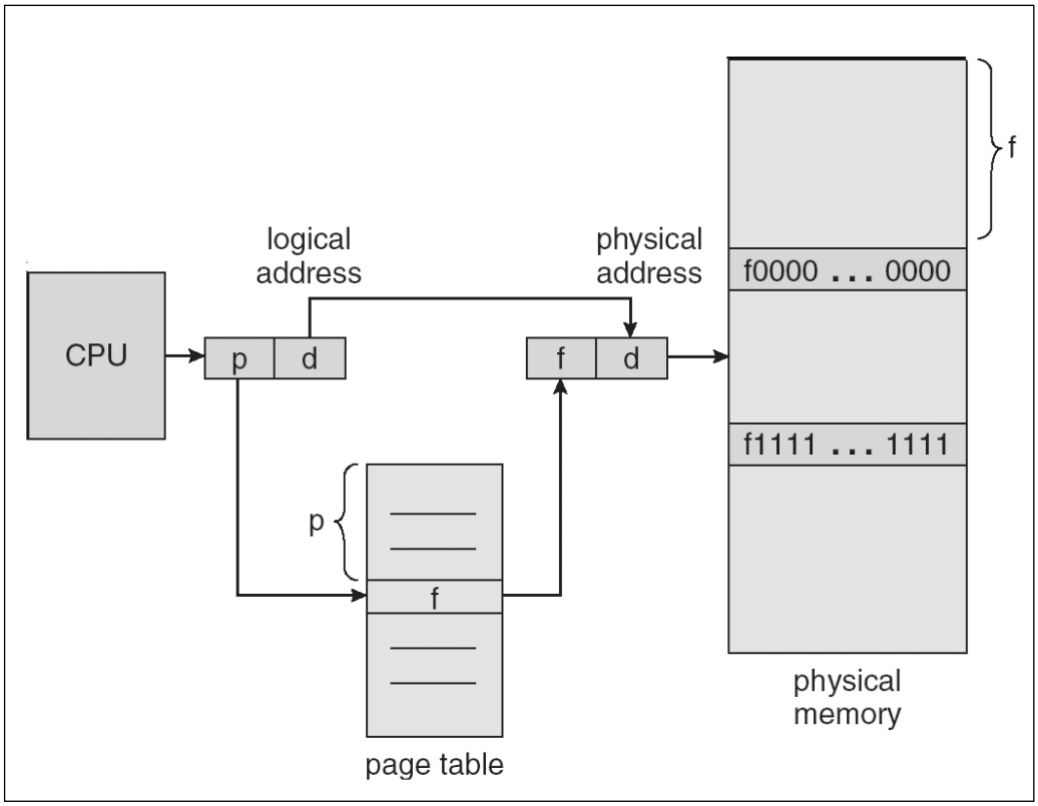


Figure 9.1: Paging Hardware

Segmentation:

- Memory-management scheme that supports user view of memory
- A program is a collection of segments.
- A segment is a logical unit such as:
  - main program,
  - procedure,
  - function,
  - method,
  - object,
  - local variables, global variables,
  - common block,
  - stack,

- symbol table, arrays

Segmentation architecture:

- Logical address consists of a two tuple:
  - <segment-number (s), offset (d)>,
- Segment table maps two-dimensional user defined address into one-dimensional physical address
- Each segment table entry has:
  - Segment base – contains the starting physical address where the segments reside in memory
  - Segment limit – specifies the length of the segment

Segmentation Hardware is shown in Fig. 9.2

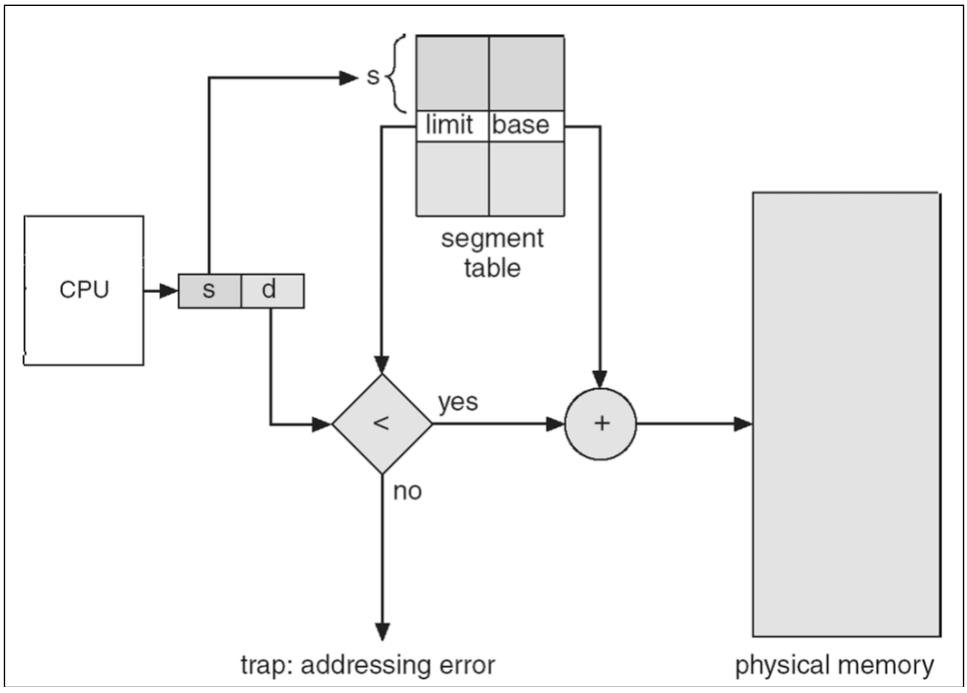


Figure 9.2: Segmentation hardware

Lab Exercises:

1. Write a C program to simulate First-fit, Best-fit and Worst-fit strategies. Given memory partitions of 100K, 500K, 200K, 300K, and 600K(in order), how would each of the First-fit, Best-fit, and Worst-fit algorithms place processes of 212K, 417K, 112K, and 426K (in order)? Which algorithm makes efficient use of memory?

2. Assuming a page size of 32 bytes and there are total of 8 such pages totaling 256 bytes. Write a C program to simulate this memory mapping. The program should read the logical memory address and display the page number and page offset in decimal. How many bytes do you need to represent the address in this scenario? Display the page number and offset to reference the following logical addresses.
   (i) 204 byte (ii) 56 byte

Additional Exercises:

1. We have five segments numbered 0 through 4. The segments are stored in physical memory as shown in the following Fig 10.3. Write a C program to create segment table. Write methods for converting logical address to physical address. Compute the physical address for the following.
   (i) 53 byte of segment 2 (ii) 852 byte of segment 3 (iii) 1222 byte of segment 0

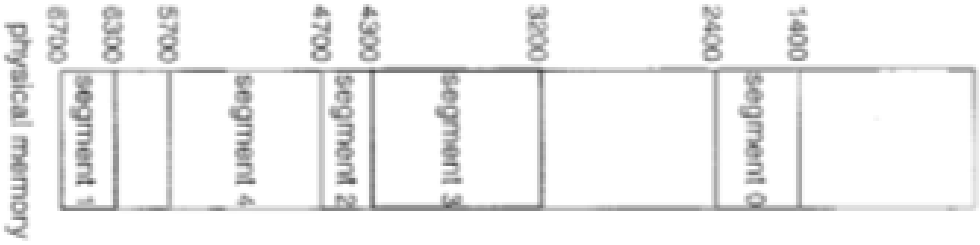


Figure 9.3: Physical memory

LAB NO: 10 **Date:**

MEMORY MANAGEMENT II

Objectives:

1. To understand the various page replacement algorithms.
2. To design programmatic solution to the algorithms.

1. Introduction :

Page replacement is basic to demand paging. It completes the separation between logical memoryand 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 algo- rithms. 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 mustbe 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 atthe tail of the queue.

Optimal Page Replacement :

Optimal page replacement algorithm has the lowest page-fault rate of all algorithms and will neversuffer 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 algo- rithm 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

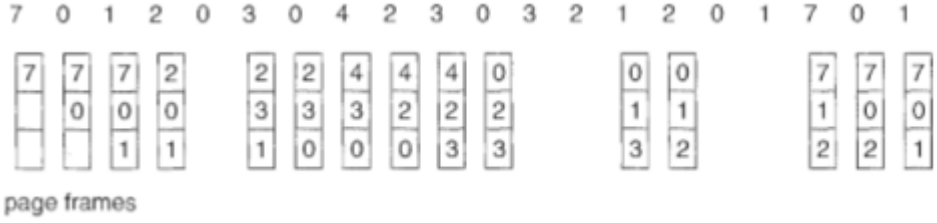
Optimal Page Replacement :

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.

Example: Consider the following Page requests

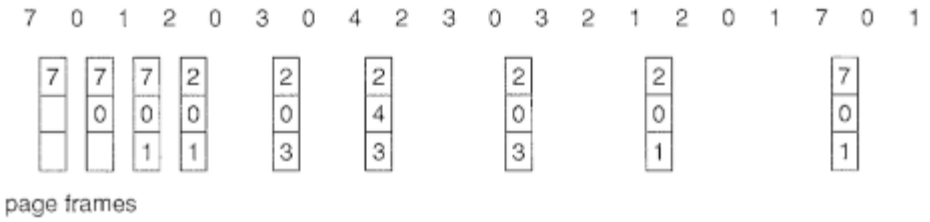
7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

(i) FIFO page replacement algorithm



Total Page faults =15.

(ii) Optimal page replacement algorithm



Total page faults = 9.

(iii) Least recently used page replacement algorithm

Total page faults = 12.

LRU approximation page replacement algorithms:

1. Additional reference byte (e.g., 8 bits) algorithm:

- Keep a ***reference byte*** for each page initialized to 00000000
- Every time a page is referenced
  - Shift the reference bits to the right by 1 bit and discarding low–order bit
  - Place the reference bit (1 if being referenced 0 otherwise) into the high order bit of the reference bits
  - The page with the lowest reference bits value is the one that is Least Recently Used, thus to be replaced

Example 1: the page with ref bits 11000100 is more recently used than the page with ref bits 01110111 (11000100 > 01110111)

Example 2: 00000000 means this page not been used for 8 periods of time. So, it is LRU.

Example 3: 11111111 means this page have been used (referenced) 8 times.

Note: If numbers are not unique, then replace (swap out) all pages with the smallest value (have same value), or use a FIFO selection among them.

2. Second chance algorithm (or clock algorithm):

- The basic algorithm of second-chance is a FIFO (Fig. 11.1).

- A reference bit for each frame is set to 1 whenever:
  - a page is first loaded into the frame.
  - the corresponding page is referenced.
- When a page has been selected for replacement, inspect its reference bit:
- If a page's reference bit is 1
  - set its reference bit to zero and skip it (give it a second chance)

- If a page's reference bit is set to 0
  - select this page for replacement

Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given second chance)

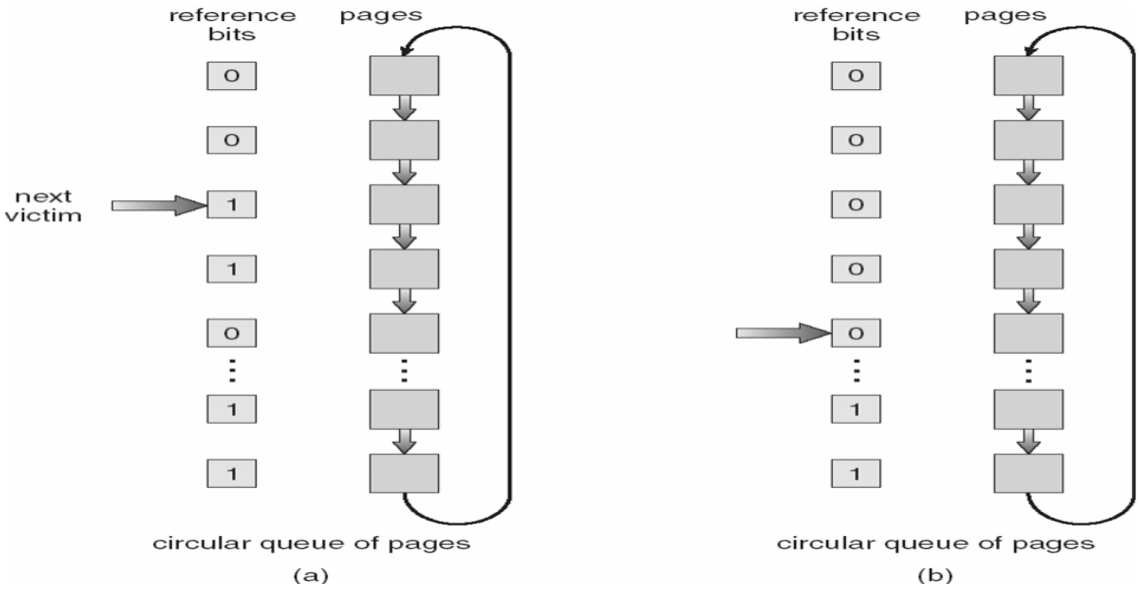


Figure 10.1: Second-chance algorithm

3. **Enhanced Second-Chance Algorithm (2-bits):**
   - Very similar to Clock Algorithm
   - Consider also the reference bits and the modified bits of pages
     - Reference (R) bit: page is referenced in the last interval
     - Modified (M) bit: page is modified after being loaded into memory
   - Four possible cases (R,M):
     - (0,0) – neither recently used nor modified (best page to replace).
     - (0,1) – not recently used but modified (not quite as good, must write out before

replacement)
  - (1,0) – recently used but clean (probably will be used again).
  - (1,1) – recently used and modified (probably will be used again and need to write out before replacement)
- Major difference between this algorithm and clock algorithm is that here we give preference to those pages that have been modified.

Counting-Based Page Replacement:

Keep a counter of the number of references that have been made to each page, and develop the following two schemes:

- **The least frequently used (LFU) Algorithm:** replaces page with smallest count
  - Suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again.
- **The most frequently used (MFU) Algorithm:** replaces the page with the largest count
  - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Lab exercises :

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 must be done as per user input and dynamic allocation for all data structures. Find the total number of page faults and hit ratio for the algorithm.

LAB NO: 11 **Date:**

DISK SCHEDULING ALGORITHM

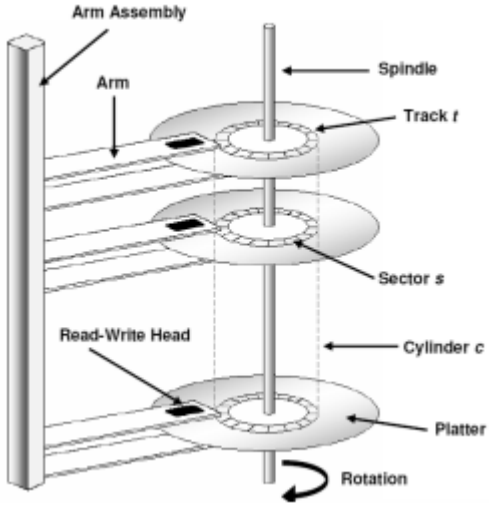
Objectives:

1. To simulate different Disk Scheduling algorithms.
2. To understand the working and calculation of the number of tracks a disk traverses for each of the algorithms.

1. Introduction

A disk is basically a platter, which is made of metal or plastic with a magnetisable coating on its surface, and it is in circular shape. It is possible to store information by recording it magnetically on the platters. A conducting coil, called head, which is a relatively small device, facilitates the data recording on and retrieval from the disk. In a disk system, head rotates just above both surfaces of each platter. All heads, being attached to a disk arm, move collectively as a unit. To enable a read and write operation, the platter rotates beneath the stationary head.

Data are organized on the platter in tracks, which are in the form of concentric set of rings. In medias using constant linear velocity, the track densities are uniform (bits per linear inch of track). The outermost zone has about 40 percent more sectors than innermost zone. The rotation speed increases as the head moves from the outer to the inner tracks to keep the same data transfer rate. This method is also used in CD-ROM and DVDROM drives.



A common disk drive has a capacity in the size of gigabytes. While the set of tracks that are at one arm position forms a cylinder, in a disk drive there may be thousands of concentric cylinders.

In a movable-head disk, where there is only one access arm to service all the disk tracks, the time spent by the Read and Write (R/W) head to move from one track to another is called seek time.

Disk scheduling Algorithms: These select one of the requests from the queue of pending requests for that drive. The various algorithms for disk scheduling are

- FCFS-Selects request in the order of arrival
- SSTF-Selects request with minimum seek time
- SCAN-starts from one end and moves to the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end the direction is reversed and servicing continues.
- C-SCAN-Similar to scan, but on reversing it returns to the beginning, without servicing requests on its return trip.
- LOOK & C- LOOK-Similar to SCAN and C-LOOK, but the arm goes only as far as the final request in each direction.

Suppose that a disk drive has 200 cylinders, numbered 0 to 199. The drive is currently servicing a request at cylinder 53, and the previous request was at cylinder 125(Applicable to only specific algorithm). The queue of pending requests in FIFO order is

98, 183, 37, 122, 14, 124, 65, 67

Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests, for each of the above algorithms?

FCFS:

It is the simplest form of disk scheduling algorithms. The I/O requests are served or processes according to their arrival. The request arrives first will be accessed and served first. Since it follows the order of arrival, it causes the wild swings from the innermost to the outermost tracks of the disk and vice versa. The farther the location of the request being serviced by the read/write head from its current location, the higher the seek time will be.

Given the following track requests in the disk queue, compute for the Total Head Movement (THM) of the read/write head

98, 183, 37, 122, 14, 124, 65, 67

Consider that the read/write head is positioned at location 53.

THM = |98 – 53| + |183-98| + |37-183| + |122-37| + |14-122| + |124-14| + |65-124| + |67-65|= 45+85+146+85+108+110+59+2 = 640 tracks.

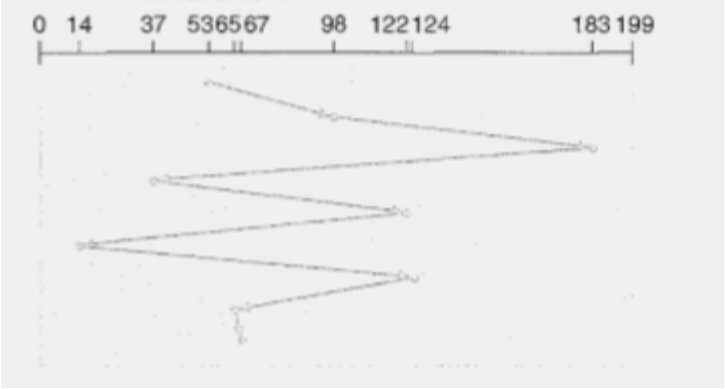


Figure 11.1: FCFS disk scheduling

Shortest Seek Time First (SSTF)

This algorithm is based on the idea that that the R/W head should proceed to the track that is closest to its current position. The process would continue until all the track requests are taken care of. Using the same sets of example in FCFS the solution are as follows:

THM = |53-65|+|67-65|+|67-37|+|14-37|+|98-14|+|122-98|+|124-122|+|183-124|=12+2+30+23+84+24+2+59=236 tracks

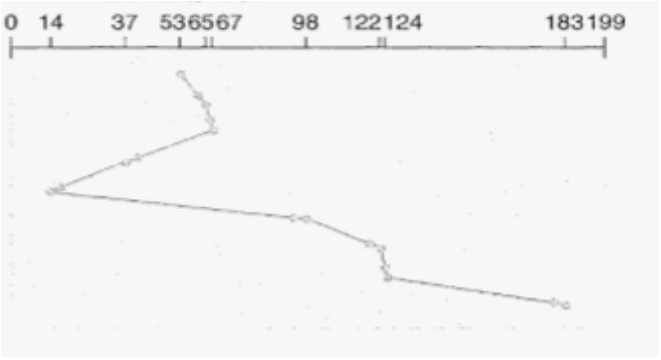


Figure 11.2: SSTF disk scheduling

SCAN Scheduling Algorithm (Elevator Algorithm)

Disk arm starts at one end of the disk and moves towards the other end servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues. This is also known as the **Elevator** algorithm

THM= |53-37|+|37-14|+|14-0|+|0-65|+|67-65|+|98-67|+|122-98|+|124-122|+|183-124| = 16+23+14+65+2+31+24+2+59=236 tracks

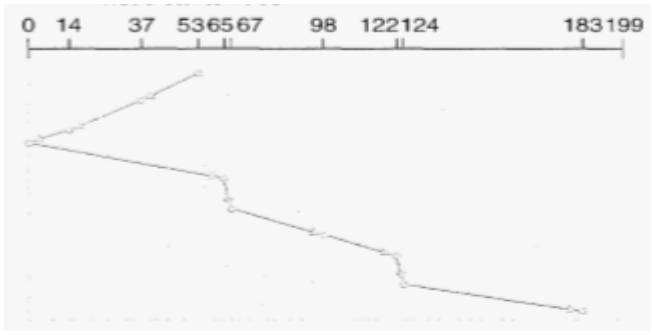


Figure 11.3: SCAN disk scheduling

Circular SCAN (C-SCAN) Algorithm

Disk head moves from one end to the other servicing requests as it goes. When it reaches the other end, it immediately returns to the beginning of the disk, without servicing any requests in the return trip.

THM = |65-53|+|67-65|+|98-67|+|122-98|+|124-122|+|183-124|+|199-183|+|14-0|+|37-14| =12+2+31+24+2+59+16+14+23=183 tracks

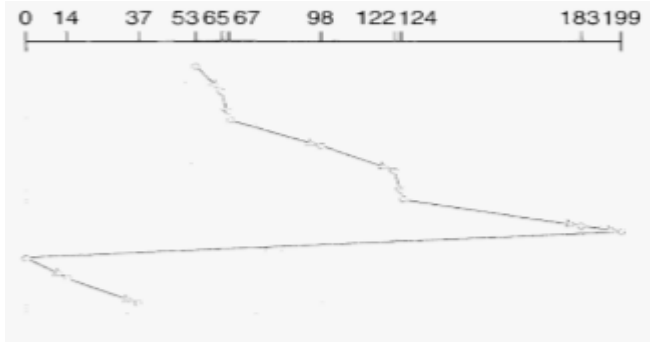


Figure 11.4: C-SCAN disk scheduling

Note: Huge jump from one end to the other end, will not be considered in total head movement.

LOOK Scheduling Algorithm

This algorithm is similar to SCAN algorithm except for the end-to-end reach of each sweep. The R/W head is only tasked to go the farthest location in need of servicing. This is also a directional algorithm, as soon as it is done with the last request in one direction it then sweeps in the other direction. Using the same sets of example in FCFS the solution are as follows:

THM = |37-53|+|14-37|+|65-14|+|67-65|+|98-67|+|122-98|+|124-122|+|183-124| = 16+23+51+2+31+24+2+59=208 tracks

C-LOOK

This scheduling algorithm is Circular LOOK is like a C-SCAN which uses a return sweep before processing a set of disk requests. It does not reach the end of the tracks unless there is a request,

either read or write on such disk location similar with the LOOK algorithm. Using the same sets of example in FCFS the solutions are as follows:

THM = |65-53|+|67-65|+|98-67|+|122-98|+|124-122|+|183-124|+|37-14| =

12+2+31+24+2+59+23 = 153 tracks

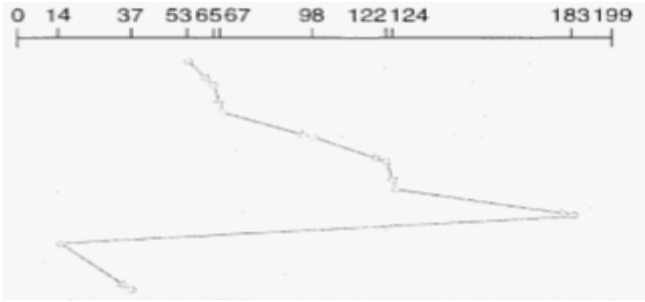


Figure 11.5: C-LOOK disk scheduling

Lab Exercises:

1. Write a C Program to simulate the following algorithms find the total no. of cylinder movements for various input requests
   (i) FCFS (ii) SSTF (iii) SCAN (iv) C-SCAN

Additional Exercises:

1. Write a C Program to simulate the following algorithms find the total no. of cylinder movements for various input requests
   (i) LOOK (ii) C-LOOK

REFERENCES

1. A Silberschartz and Galvin, “Operating Systems Concepts “, 10th edition, AddisonWesley, 2018.
2. Apple Inc. (n.d.). *Shell Scripting Primer* (Apple Developer Documentation). Retrieved from https://developer.apple.com/library/archive/documentation/OpenSource/Conceptual/ShellScripting/Introduction/Introduction.html
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.
7. W. R. Stevens, UNIX Network Programming-Volume II (IPC), PHI, 1998.