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

B. TECH. FIFTH SEMESTER

COMPUTER SCIENCE AND ENGINEERING (CSE)

OPERATING SYSTEMS CSE_ 3163

LABORATORY MANUAL

CONTENTS

Course Objectives

This laboratory course enables students to

- Illustrate and explore system calls related to Linux operating system.
- Learn process management and thread programming concepts which include scheduling algorithms and inter process communication.
- Understand the working of memory management schemes, disk scheduling algorithms, and page replacement algorithms through simulation.

Course Outcomes

At the end of this course, students will have the

- Understand Linux system calls on Files and directories
- Implement thread programming and inter process communication techniques.
- Implement Locking and synchronization, deadlock, memory management, page replacement and disk scheduling algorithms.

Evaluation Plan

- Internal Assessment Marks : 60 Marks
  - Continuous evaluation: 36 Marks

    The Continuous evaluation assessment will depend on performance in solving lab and additional Exercises, maintaining the lab report and answering the questions in viva voce or and/or execution the instructions told by the instructors.
  - Internal Exam: 24 Marks.

    Internal exam assessment will depend on Execution, writeup and viva
- End semester assessment of 2-hour duration: 40 Marks

INSTRUCTIONS TO THE STUDENTS

Pre-Lab Session Instructions

1. Students should carry the Class notes, Lab Manual and the required stationery to every lab session.
2. Be in time and follow the Instructions from Lab Instructors.
3. Must Sign in the log register provided.
4. Make sure to occupy the allotted seat and answer the attendance.
5. Adhere to the rules and maintain the decorum.

In-Lab Session Instructions

- Follow the instructions on the allotted exercises given in Lab Manual.
- 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.

General Instructions for the exercises in Lab

- 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 are properly indented and comments should be given whenever it is required.
  - Use meaningful names for variables and procedures.
- Plagiarism (copying from others) is strictly prohibited and would invite severe penalty during 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 examinations 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 later in the manual as a model for observation.

THE STUDENTS SHOULD NOT

- Carry mobile phones while working with computer.
- Go out of the lab without permission.

LAB NO.: 1 **Date:**

UNIX SHELL COMMANDS

Objective:

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

1. UNIX shell and special characters

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

Shell Prompt:

The prompt, **$**, which is called ***command prompt***, is issued by the shell. While the prompt is displayed, you can type a command. The command is a binary executable. Once the Enter key is pressed, the shell reads the command line arguments and performs accordingly. It determines the command to be executed by looking for input executable name placed in standard location (ex: /usr/bin). Multiple arguments can be provided to the command (executable) separated by spaces.
Following is a simple example of date command which displays current date and time:
[Command 1]. *$date*
Thu Jun 25 08:30:19 MST 2009

Shell Types:

In UNIX there are two major types of shells:
1. The Bourne shell. If you are using a Bourne-type shell, the default prompt is the $ character.
2. The C shell. If you are using a C-type shell, the default prompt is the % character.
There are again various subcategories for Bourne Shell which are listed as follows:

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

The different C-type shells follow:

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

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

Special Characters:

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

Character	Description
\	Escape character. If you want to refer a special character, you must "escape" it with a backslash first. Example*: touch /tamp/filename**
/	Directory separator, used to separate a string of directory names. Example*: /usr/src/unix*
.	Current directory. Can also "hide" files when it is the first character in a filename.
..	Parent directory
~	User's home directory
*	Represents 0 or more characters in a filename, or by itself, all files in a directory. Example: pic*2002 can represent the files pic2002, picJanuary2002, picFeb292002, etc.
?	Represents a single character in a filename. Example: hello?.txt can represent hello1.txt, helloz.txt, but not hello22.txt

[ ]	Can be used to represent a range of values, e.g. [0-9], [A-Z], etc. Example: hello[0-2].txt represents the names hello0.txt, hello1.txt, and hello2.txt
|	"Pipe". Redirect the output of one command into another command. Example*: ls | more*
>	Redirect the output of a command into a new file. If the file already exists, over-write it. Example: *ls > myfiles.txt*
>>	Redirect the output of a command onto the end of an existing file. Example: *echo "Mary 555-1234" >> phonenumbers.txt*
<	Redirect a file as input to a program. Example: more < phonenumbers.txt
<<	Reads from a stream literal (an inline file, passed to the standard input) Example: *tr a-z A-Z << END_TEXT* *This is OS lab manual* *For IT students* *END_TEXT*
<<<	Reads from a string. Example: *bc <<< 9+5*
;	Command separator. Allows you to execute multiple commands on a 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 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 must give the complete location of the command. [PATH is an environmental variable. To display the value of PATH variable, execute *echo $PATH*]

[Command 2]: *echo $PATH*

Examples: /home/bob/myprogram
./program (Execute a program in the current directory)
~/bin/program (Execute program from a personal bin directory)

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

Command Syntax

When interacting with the UNIX operating system, one of the first things you need to know is that, unlike other computer systems you may be accustomed to, everything in UNIX is case-sensitive. Be careful when you're typing in commands - whether a character is upper or lower case does make a difference. For instance, if you want to list your files with the *ls* command, if you enter LS you will be told "command not found". Commands can be run by themselves, or you can pass in additional arguments to make them do different things. Each argument to the command should be separated by space. Typical command syntax can look something like this:

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

Examples:

ls #List files in current directory

ls -l #Lists files in "long" format

ls -l --color #As above, with colorized output

cat filename #Show contents of a file

cat -n filename #Show contents of a file, with line numbers

Getting Help

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

Many commands have simple "help" screens that can be invoked with special command flags. These flags usually look like -h or --help. Example: grep --help. "Man Pages" are

the best source of information for most commands can be found in the online manual pages. To display a command's manual page, type man <commandName>.

Examples (**Command set 3**): *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.

[Command Set 4]:

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

[Command Set 5]:

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 be- tween cd and .. .

iv. **cd -**: Return to previous directory. An easy way to get back to your previous location!

[Command Set 6]

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.

vi. **ls -l --color** : As ls -l, with colorized output.

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

i) *$ls program.?*

program.o program.c

ii) *$ls fr**

frank fred

iii) *$ls *ed*

ed fred

iv) *$ls *r**

darlene frank fred

v) *$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 the letters and nothing more.

The process of matching expressions containing wildcards to filenames is called wildcard expansion. 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:

[Command Set 7]:

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 display 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 direc- tory or you can remove a lot of stuff quickly. Example: *rmdir /tmp/myfiles/*.

h. **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.

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:

[Command Set 8]

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:

[Command Set 9]

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 the MP3 files in a directory, we can do something like this, using the > redirection character.

[Command Set 10]

Example:

a. *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:

b. *ls -l /home/vic/extraMP3s/*.mp3 >> mp3files.txt*

7. Other useful commands:

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 f

ruit-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.

[Command Set 11]

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. ***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

[Command Set 12]

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

d. **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: echothis 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. Ifquotes (either single or double) are used, they are not repeated on the screen.

e. **bc (Basic Calculator):**

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

[Command Set 13]

$bc (hit enter key)

5 + 2

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

2 # to perform floating point operations use bc -l

5 > 2

#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.

f. **Starting vi** :

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

[Command Set 14]

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 must 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

8. Shortcuts

a. ctrl+c Halts the current command

b. ctrl+z Stops the current command,

c. ctrl+d Logout the current session, similar to exit

d. ctrl+w Erases one word in the current line

e. ctrl+u Erases the whole line

f. !! Repeats the last command

g. exit Logout the current session

Lab Exercises

1. Execute and write output of all the commands from command set 1 to command set 10.
2. List all the file names satisfying following criteria

a. has the extension .txt.
b. containing at least one digit.
c. having minimum length of 4.
d. does not contain any of the vowels as the start letter.

3. Write grep commands to do the following activities:
a. To select the lines from a file that have exactly two characters.
b. To select the lines from a file that start with the upper case letter.
c. To select the lines from a file that end with a period.
d. To select the lines in a file that has one or more blank spaces.

Additional Exercise:

1. Explore the following commands along with their various options. (Some of the op- tions 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)

2. Execute and write output of all the commands from command set 11 to command set 14.

LAB NO.: 2 **Date:**

SYSTEM CALLS FOR PROCESS CONTROL

Objectives:

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

1. Executing a C program on UNIX platform

Compile/Link a Simple C Program - hello.c

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

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

To compile the hello.c:

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

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

To run the program:

$ ./a.out

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

To specify the output filename, use -o option:

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

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

2. Use of System Calls in C programming

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

- fork()

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

fork():

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

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

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

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

The following is a simple example of fork()

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

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

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

```
#include <stdio.h>
#include
<sys/types.h>
#include
<unistd.h>
#include <errno.h>
#include
<string.h> #define
BUFLEN 10 int
main(void)
{
  int i;
  char buffer[BUFLEN+1];
  pid_t pid1;
  pid1  =
  fork( );if
  (pid1 ==
```

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

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

A few notes on this program:

- The function call sleep will result in a process "sleeping" a specified number of seconds. It can be used to prevent the process from running to completion

withinone time slice.

- One process will always end before the other. If there is enough intervening timebefore the second process ends, the system call will redisplay the prompt, produc-ing the last line of output where the output from the child process is appended tothe end of the prompt (i.e.. %child)

A few additional notes about fork():

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

wait():

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

A parent process usually needs to synchronize its actions by waiting until the child process which has either stopped or terminated its actions. The wait() system call al-lows 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 andreturns 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 pro- cesses are still active, the calling process will suspend its activity until a child processterminates.

Example of wait():

```
int status;
pid_t
fork_return;
fork_return =
fork();
if (fork_return == 0) /* child process
```

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

A few notes on this program:

- wait(&status) causes the parent to suspend until the child process finishes execu-tion
- details of how the child stopped are returned via the status variable to the parent.Several macros are available to interpret the information. Two useful ones are:
  - WIFEXITED evaluates as true, or 0, if the process ended normally with an exitor return call.
  - WEXITSTATUS if a process ended normally you can get the value that wasreturned with this macro.

exec*():

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

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

Commonly a process generates a child process because it would like to transform

thechild 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 passtheir own environment variable list
- 'p' indicates the current PATH string should be used when the system searches forexecutable 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 programto be executed is known in advance

execvp

- this system call is used when the numbers of arguments for the program to beexecuted 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 piddoes not change.
- the exec() is issued by the calling process and what is exec'ed is referred to as thenew 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 un-less 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",NUL
L);

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 pos-itive 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 processforked 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 groupID value. When a process generates a child process, the operating system will auto- matically create a process group.
The initial parent process is known as the process leader. getpgrp() will obtain the process group id.

Lab Exercises

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

Additional Exercises

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

LAB NO.: 3 **Date:**

PROGRAMS ON THREADS

Objectives:

In this lab, the student will be able to:

1. Understand the concepts of multithreading.
2. Grasp the execution of the different processes concerning multithreading.

Creating Thread

A process will start with a single thread which is called the 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 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 into 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 );
}
```

Compile Pthread program:
gcc -pthread fileSourceName.c -o outputName

Run the program:
./outputName

Function name	Description
pthread_create	Create a new thread
pthread_join	blocks the calling thread until the specified *threadid* thread terminates
pthread_exit	Terminates the thread and returns "status" to any joining thread

Lab Exercises:

1. Write a 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.
2. Write a multithreaded program for generating prime numbers from a given starting number to the given ending number.
3. Write a multithreaded program that performs the sum of even numbers and odd numbers in an input array. Create two separate threads to perform the sum of even numbers and odd numbers. The parent thread has to wait until both the threads are done.

Additional Exercises:

1. Write a multithreaded program for matrix multiplication.
2. Write a multithreaded program for finding row sum and column sum

LAB NO.: 4 **Date:**

PROCESS SCHEDULING

Objectives:

1. To implement process scheduling algorithms.

1. Basic Concepts：

CPU scheduling is the basis of multi programmed operating systems. By switching the CPU among processes, the operating system can make the computer more productive. Ina 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 processand gives the CPU to another process. This pattern continues. Every time one process hasto 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 sched-uler. The scheduler selects a process from the processes in memory that are ready to ex- ecute and allocates the CPU to that process.

2. Problem Description and Algorithm：

2.1 Round Robin Scheduling (RR):

The **round-robin (RR) scheduling algorithm** is designed especially for timesharing systems. It is similar to (First Come First Serve) FCFS scheduling, but preemption is addedto 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 burstof the currently running process is longer than 1 time quantum, the timer will go off andwill 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 selectthe next process in the ready queue.

Example: Time quantum=3

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

P1	P3	P1	P2	P3	P4	P1	P2	P3	P4

0 3 6 9 12 15 18 20 21 24 37

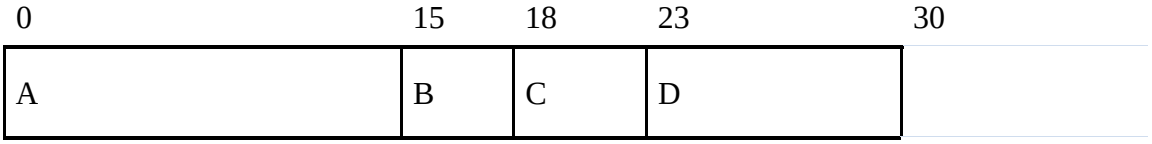
Average waiting time: (12+12+12+14)/4 = 12.5.

2.2 Shortest Job First (SJF):

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

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

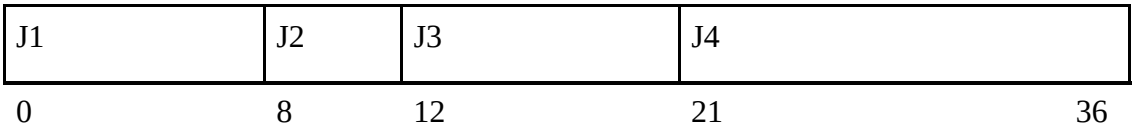
SJF Waiting time A= 0, B=10, C=10, D=13. TT A=15, B=13, C=15, D=20.



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

Additional Exercises

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

LAB NO.: 5 **Date:**

DEADLOCK, LOCKING, SYNCHRONIZATION

In this lab, students will be able to:

1. Synchronize various processes with the use of semaphore

1. Semaphores

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

The definition of wait () is as follows:

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

The definition of signal() is as follows:

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

2. 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;
```

```
}
```

3. Implementation of Semaphores:

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, an 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 );
}
```

4. Classical Problems：

4.1 The Bounded – Buffer Problem

Here the pool consists of *n* buffers, each capable of holding one item. The mutex sema- phore 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. Thesemaphore empty is initialized to the value n; the semaphore full is initialized to the value
0. The classical example is the production line.

4.2 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 producerand 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 ina 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.

4.3 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 pro- cesses by referring to the former as readers and to the latter as writers. Obviously, if tworeaders access the shared data simultaneously, no adverse effects will result. However, ifa writer and some other process (either a reader or a writer) access the database simulta-neously, 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 problemis referred to as the *readers-writers problem.*

5. Algorithm

5.1 Solution to bounded buffer Producers_Consumers Problem using semaphores

Program producers_consumers

{parent process}

```
const int capacity
=5;int  item ;
buffer: array[1..capacity] of item;
empty, full : semaphore; {general} //Initialize empty to capacity and full to 0
```

```
        pmutex, cmutex :semaphore; {binary} //Initialize to 1
        int in =1, out=1;
 // initiate  processes producers, consumers

}end { producers_consumers}

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

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

5.2 Solution to Readers Writers problem using semaphores

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

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

```
    end {while}
  end; {writerZ}

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

  initiate readers, writers
end {readers_writers}
```

Lab Exercise

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

Additional Exercise

Write a C program to solve the Dining-Philosophers problem

LAB NO.: 6 **Date:**

BANKERS ALGORITHM

Objectives:

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

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

1. Deadlocks

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

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

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

Release: The process releases the resource.

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

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

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

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

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

is waiting for a resource held by P_n and Pn is waiting for a resource held by *Po.*

2. Deadlock Avoidance

In deadlock avoidance, simplest and most useful model requires that each process declarethe *maximum number* of resources of each type that it may need. The deadlock-avoidancealgorithm 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 ofavailable and allocated resources, and the maximum demands of the processes

Safe Sate

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

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

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

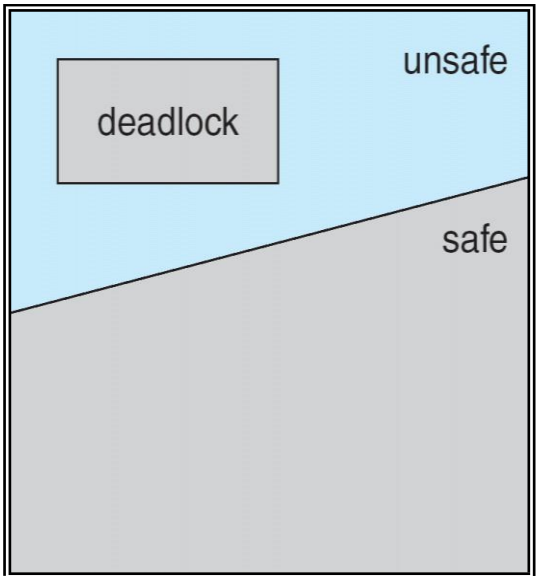


Fig 7.1 Safe, Unsafe and Deadlock State

3. Bankers Algorithm

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

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

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

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

3.1 Safety Algorithm:

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

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

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

3.2 Resource-Request Algorithm :

This algorithm is used for determining whether requests can be safely granted.

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

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

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

Lab Exercise

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

Additional exercises

Write a C program to implement the deadlock detection algorithm

LAB NO.: 7 **Date:**

MESSAGE QUEUE, SHARED MEMORY

Objectives:

In this lab, the student will be able to:

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

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 are, it removes a few difficulties that exist 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 the type field.

Creating a Message Queue

- 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 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 an 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 efficient in 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 subsequently 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 non-negative 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 the 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 the 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);
}
```

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 inter process communication using a message queue.

2. Implement a parent process, which sends an English alphabet to a child process using shared memory. The child process responds with the next English alphabet to the parent. The parent displays the reply from the Child.

Additional Exercises:

1. 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.

2. 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 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.: 8 **Date:**

DYNAMIC STORAGE ALLOCATION STRATEGY FOR FIRST FIT ANDBEST FIT

Objectives:

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

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

1. Description

One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. In this multiple- partition method, when a partition is free, a process is selected from the input queue andis loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needsmemory, 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 begin-ning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

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

2. Algorithm

First Fit Allocation

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

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

Best Fit Allocation

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

Lab exercise

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

Additional exercises

1. Write a C program to implement dynamic storage allocation strategy for worst fitusing dynamic allocations for all the required data structure.

LAB NO.: 9 **Date:**

PAGE REPLACEMENT ALGORITHMS

Objectives:

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

1. Introduction :

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. There are many different page-replacement 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 must be replace, the oldest page is chosen. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue when a page is brought into memory; we insert it at the tail of the queue.

Optimal Page Replacement :

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

Lab exercise :

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

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

LAB NO.: 10 **Date:**

DISK SCHEDULING ALGORITHM

Objectives:

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

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

1. Introduction

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

Types of Disk Scheduling Algorithms

Although there are other algorithms that reduce the seek time of all requests, we willconcentrate on the following disk scheduling algorithms:

a. First Come-First Serve (FCFS)
b. Shortest Seek Time First (SSTF)
c. Elevator (SCAN)
d. Circular SCAN (C-SCAN)
e. LOOK
f. C-LOOK

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

Problem：

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

Illustration of Disk Scheduling Algorithm：

Shortest Seek Time First (SSTF):

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

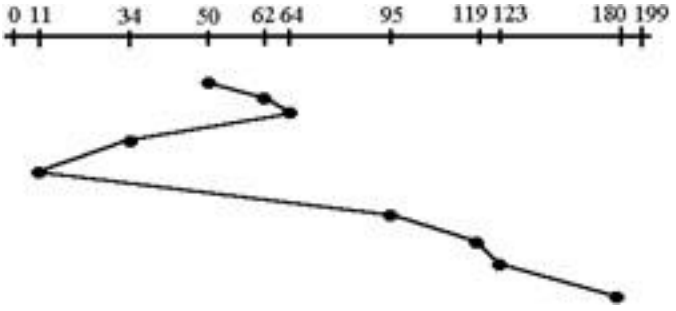


Figure 11.1 SSTF

SCAN:

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

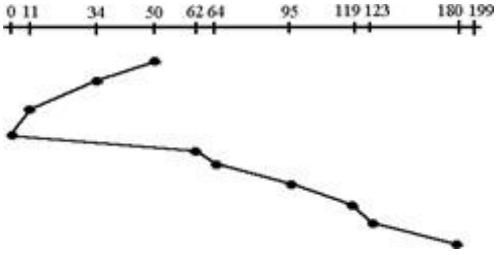


Figure 11.2 SCAN

Circular Scan (C-SCAN) :

Circular scanning works just like the elevator to some extent. It begins its scan toward the nearest end and works it way all the way to the end of the system. Once it hits the

bottom or top it jumps to the other end and moves in the same direction. Keep in mind that the huge jump doesn't count as a head movement. The total head movement for thisalgorithm is only 187 track, but still this isn't the most sufficient.

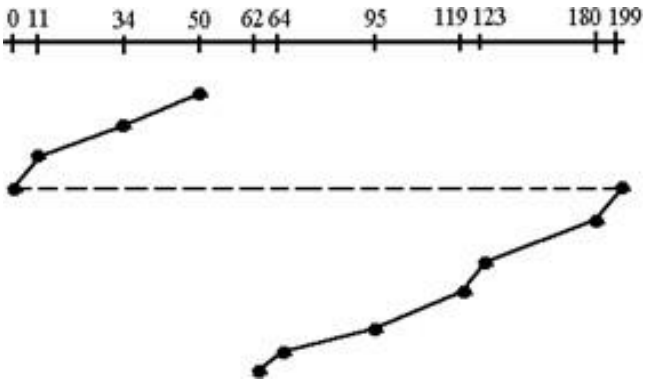


Figure 11.3 C-SCAN

C-LOOK :

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

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

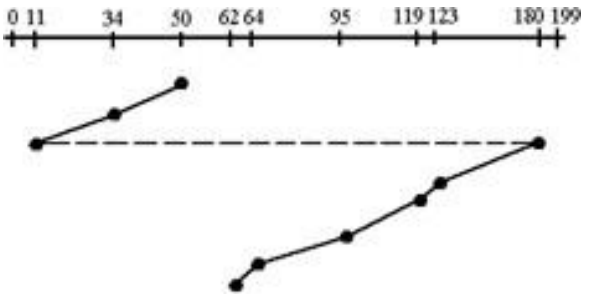


Figure 11.4 C-LOOK

Lab Exercise :

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

Additional Exercise：

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

LAB NO.: 11 **Date:**

FILE SYSTEM

Objectives:

In this lab, the student will be able to:

1. Understand the file system and their metadata associated with them.
2. Operations that could be performed on the files in the file system.

Each file is referenced by an inode, which is addressed by a filesystem-unique numerical value known as an inode number. An inode is both a physical object located on the disk of a Unix-style filesystem and a conceptual entity represented by a data structure in the Linux kernel. The inode stores the metadata associated with a file, such as a file's access permissions, last access timestamp, owner, group, and size, as well as the location of the file's data.

The file name is not available in an inode is the file's name and it is stored in the directory entry.

Obtain the inode number for a file using the -i flag to the ls command:

```
$ ls -i
```

To obtain information of files in the current directory in detail, use -il:

```
$ ls -il
```

To display the filesystem inode space information, the df command could be used:

```
$ df -i
```

Stat functions

Linux provides a family of functions for obtaining the metadata of a file:

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

int stat (const char *path, struct stat *buf);
int fstat (int fd, struct stat *buf);
```

```
int lstat (const char *path, struct stat *buf);
```

Each of these functions returns information about a file.
stat() returns information about the file denoted by the path, while fstat() returns information about the file represented by the file descriptor fd.
lstat() is identical to stat(), except that in the case of a symbolic link, lstat() returns information about the link itself and not the target file.

Each of these functions stores information in a stat structure, which is provided by the user. The stat structure is defined in <bits/stat.h>, which is included from <sys/stat.h>:

```
struct stat {
dev_t st_dev; /* ID of device containing file */
ino_t st_ino; /* inode number */
mode_t st_mode; /* permissions */
nlink_t st_nlink; /* number of hard links */
uid_t st_uid; /* user ID of owner */
gid_t st_gid; /* group ID of owner */
dev_t st_rdev; /* device ID (if special file) */
off_t st_size; /* total size in bytes */
blksize_t st_blksize; /* blocksize for filesystem I/O */
blkcnt_t st_blocks; /* number of blocks allocated */
time_t st_atime; /* last access time */
time_t st_mtime; /* last modification time */
time_t st_ctime; /* last status change time */
};
```

A file is mapped to its inode by its parent directory.
The inode number of / (topmost) directory is fixed, and is always 2.

```
$ stat /
 File: '/'
 Size: 4096 Blocks: 8 IO Block: 4096 directory
Device: 806h/2054d Inode: 2 Links: 27
Access: (0755/drwxr-xr-x) Uid: ( 0/ root) Gid: ( 0/ root)
Access: 2019-12-07 01:40:01.565097799 +0100
Modify: 2019-12-07 01:27:33.651924301 +0100
```

```
Change: 2019-12-07 01:27:33.651924301 +0100
Birth:
```

The fields are as follows:

• The **st_dev** field describes the device node on which the file resides. If the file is not backed by a device—for example, if it resides on an NFS volume—this value is 0.
• The **st_ino** field provides the file's inode number.
• The **st_mode** field provides the file's mode bytes, which describe the file type (such as a regular file or a directory) and the access permissions.
• The **st_nlink** field provides the number of hard links pointing at the file. Every file on a filesystem has at least one hard link.
• The **st_uid** field provides the user ID of the user who owns the file.
• The **st_gid** field provides the group ID of the group who owns the file.
• If the file is a device node, the st_rdev field describes the device that this file represents.
• The **st_size** field provides the size of the file, in bytes.
• The **st_blksize** field describes the preferred block size for efficient file I/O. This value is the optimal block size for user-buffered I/O.
• The **st_blocks** field provides the number of filesystem blocks allocated to the file. This value multiplied by the block size will be smaller than the value provided by st_size if the file has holes.
• The **st_atime** field contains the last file access time. This is the most recent time at which the file was accessed.
• The **st_mtime** field contains the last file modification time—that is, the last time the file was written to.
• The **st_ctime** field contains the last file change time. The field contains the last time that the file's metadata (for example, its owner or permissions) was changed.

Return Values
On success, all three calls return 0 and store the file's metadata in the provided stat structure. On error, they return −1 and set errno to one of the following:

EACCES
The invoking process lacks search permission for one of the directory components of path (stat() and lstat() only).

EBADF
fd is invalid (fstat() only).

EFAULT
path or buf is an invalid pointer.

ELOOP
path contains too many symbolic links (stat() and lstat() only).

ENAMETOOLONG
path is too long (stat() and lstat() only).

ENOENT
A component in path does not exist (stat() and lstat() only).

ENOMEM
There is insufficient memory available to complete the request.

ENOTDIR
A component in path is not a directory (stat() and lstat() only).

Program that uses stat() to retrieve the size of a file provided on the command line:

```
/* Filename: stat.c */
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
struct stat sb;
int ret;
if (argc < 2) {
fprintf (stderr, "usage: %s <file>\n", argv[0]);
return 1;
}
ret = stat (argv[1], &sb);
if (ret) {
```

```
perror ("stat");

return 1;
}
printf ("%s is %ld bytes\n", argv[1], sb.st_size);
return 0;
}
```

Here is the result of running the program on its own source file:

```
$ ./stat stat.c
stat.c is 392 bytes
```

Program to find the file type (such as symbolic link or block device node) of the file given by the first argument to the program:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
struct stat sb;
int ret;
if (argc < 2) {
fprintf (stderr, "usage: %s <file>\n", argv[0]);
return 1;
}
ret = stat (argv[1], &sb);
if (ret) {
perror ("stat");
return 1;
}
printf ("File type: ");
switch (sb.st_mode & S_IFMT) {
case S_IFBLK:
printf("block device node\n");
break;
```

```
case S_IFCHR:
printf("character device node\n");

break;
case S_IFDIR:
printf("directory\n");
break;
case S_IFIFO:
printf("FIFO\n");
break;
case S_IFLNK:
printf("symbolic link\n");
break;
case S_IFREG:
printf("regular file\n");
break;
case S_IFSOCK:
printf("socket\n");
break;
default:
printf("unknown\n");
break;
}
return 0;
}
```

Links

Each name-to-inode mapping in a directory is called a link. A link is essentially just a name in a list (a directory) that points at an inode. A single inode (and thus a single file) could be referenced from, say, both /etc/customs and /var/run/ledger. However, because links map to inodes and inode numbers are specific to a particular filesystem, /etc/customs and /var/run/ledger must both reside on the same filesystem. Within a single filesystem, there can be a large number of links to any given file. The only limit is in the size of the integer data type used to hold the number of links. Among various links, no one link is the “original” or the “primary” link. All of the links get the same status, pointing at the same file and called as hard links. Files can have 0, 1, or many links. Most

files have a link count of 1 i.e. they are pointed at by a single directory entry, but some files have 2 or even more links. Files with a link count of 0 have no corresponding directory entries on the filesystem. When a file's link count reaches 0, the file is marked as free, and its disk blocks are made available for reuse.5 Such a file, however, remains on the filesystem if a process has the file open. Once no process has the file open, the file is removed.

The link() system call standardized by POSIX, creates a new link for an existing file:

```
#include <unistd.h>
int link (const char *oldpath, const char *newpath);
```

A successful call to link() creates a new link under the path new path for the existing file oldpath and then returns 0. Upon completion, both the oldpath and newpath refer to the same file. There is no way to tell which was the "original" link.

On failure, the call returns −1 and sets errno to one of the following:

EACCES
The invoking process lacks search permission for a component in oldpath, or the invoking process does not have write permission for the directory containing newpath.

EEXIST
newpath already exists—link() will not overwrite an existing directory entry.

EFAULT
oldpath or newpath is an invalid pointer.

EIO
An internal I/O error occurred (this is bad!).

ELOOP
Too many symbolic links were encountered in resolving oldpath or newpath.

EMLINK
The inode pointed at by oldpath already has the maximum number of links pointing at it.

ENAMETOOLONG

oldpath or newpath is too long.

ENOENT
A component in oldpath or newpath does not exist.

ENOMEM
There is insufficient memory available to complete the request.

ENOSPC
The device containing newpath has no room for the new directory entry.

ENOTDIR
A component in oldpath or newpath is not a directory.

EPERM
The filesystem containing newpath does not allow the creation of new hard links, or oldpath is a directory.

EROFS
newpath resides on a read-only filesystem.

EXDEV
newpath and oldpath are not on the same mounted filesystem. (Linux allows a single filesystem to be mounted in multiple places, but even in this case, hard links cannot be created across the mount points.)

This example creates a new directory entry, pirate, that maps to the same inode (and thus the same file) as the existing file privateer, both of which are in /home/nayan:

```
int ret;
/* create a new directory entry,
'/home/nayan/somecontents', that points at the same inode
as '/home/nayan/contents' */

ret = link ("/home/ nayan/somecontents",
/home/nayan/contents");
if (ret)
perror ("link");
```

Unlinking

The converse to linking is unlinking, the removal of pathnames from the filesystem. A single system call, unlink(), handles this task:

```
#include <unistd.h>
int unlink (const char *pathname);
```

A successful call to unlink() deletes pathname from the filesystem and returns 0. If that name was the last reference to the file, the file is deleted from the filesystem. If, however, a process has the file open, the kernel will not delete the file from the filesystem until that process closes the file. Once no process has the file open, it is deleted.

If pathname refers to a symbolic link, the link, not the target, is destroyed. If pathname refers to another type of special file, such as a device, FIFO, or socket, the special file is removed from the filesystem, but processes that have the file open may continue to utilize it.

On error, unlink() returns −1 and sets errno to one of the following error codes:

EACCES
EFAULT
EIO
EISDIR
ELOOP
ENAMETOOLONG
ENOENT
ENOMEM
ENOTDIR
EPERM
EROFS

unlink() does not remove directories. For that, applications should use rmdir.

To ease the want on destruction of any type of file, the remove() function is provided:

```
#include <stdio.h>
int remove (const char *path);
```

A successful call to remove() deletes path from the filesystem and returns 0. If path is a file, remove() invokes unlink(); if path is a directory, remove() calls rmdir().

On error, remove() returns −1 and sets errno to any of the valid error codes set by unlink() and rmdir(), as applicable.

Lab Exercises:

1. Write a program to find the inode number of an existing file in a directory. Take the input as a filename and print the inode number of the file.
2. Write a program to create a new link to an existing file and unlink the same. Accept the old path as input and print the newpath.

Additional Exercises:

1. Write a program to find the inode number of all files in a directory. Take the input as a directory name and print the inode numbers of all the files in it.
2. Write a program to print the full stat structure of a directory.

LAB NO.: 12 **Date:**

DISK MANAGEMENT

Objectives:

In this lab, students will be able to

- Understand find the details of underlying operating systems disk space and file system information

System calls related to disk

df command

The 'df' (Disk Free) command in an inbuilt utility to find the available and the disk usage space on Linux servers/storage.

The following table provides an overview of the options of df command in Linux.

Options	Description
-a	-all: includes real files and virtual files like pseudo,pro, sysfs. lxc
-h	It prints the sizes in the human-readable format in power of 1024 (eg: 1K 1M 1G)
-H	--si : likewise '-h' but here in power of 1000
-i	-- inodes: correspondence the inode details
-k	--block-size=1K display the disk space
-l	--local display local file systems only
-m	--megabytes display the disk space
-t	--type=TYPE To filter a particular file system type
-T'	--print-type List the file system types
-x	--exclude-type=TYPE To exclude a particular file system type

1. **How to check the details of disk space used in each file system?**

```
# df
```

```
Output:

Filesystem     1K-blocks      Used Available Use% Mounted on
/dev/sda2      164962420 142892148  15301196  91% /
udev               10240         0     10240   0% /dev
tmpfs            3291620    329084   2962536  10% /run
tmpfs            8229048         0   8229048   0% /dev/shm
tmpfs               5120         0      5120   0% /run/lock
tmpfs            8229048         0   8229048   0% /sys/fs/cgroup
/dev/sda1          97167     76552     15598  84% /boot
tmpfs            1645812         0   1645812   0% /run/user/301703
tmpfs            1645812         0   1645812   0% /run/user/301677
tmpfs            1645812         0   1645812   0% /run/user/301483
```

Note: Using 'df' command without any option/parameter will display all the partitions and the usage information of the disk space. The result of the above command contains 6 columns which are explained here below:

```
Filesystem      -->    Mount Point Name
1K-blocks       -->    Available total space counted in 1kB (1000 bytes)
Used            -->    Used block size
Available       -->    Free blocks size
Use%            -->    Usage on percentage-wise
Mounted on      -->    Show the path of the mounted point

# df -k
```

Note: Even using the '-k' option also provides the same output as the default 'df' command. Both outputs provide the same data usage of file systems in block size which is measured in 1024 bytes.

2. How to check the disk space in Human-Readable format?

```
# df -h
```

```
Output:

Filesystem      Size  Used Avail Use% Mounted on
/dev/sda2       158G  137G   15G  91% /
udev             10M     0   10M   0% /dev
tmpfs           3.2G  322M  2.9G  10% /run
tmpfs           7.9G     0  7.9G   0% /dev/shm
tmpfs           5.0M     0  5.0M   0% /run/lock
tmpfs           7.9G     0  7.9G   0% /sys/fs/cgroup
/dev/sda1        95M   75M   16M  84% /boot
tmpfs           1.6G     0  1.6G   0% /run/user/301703
tmpfs           1.6G     0  1.6G   0% /run/user/301483
tmpfs           1.6G     0  1.6G   0% /run/user/301613
tmpfs           1.6G     0  1.6G   0% /run/user/301677
```

Note: Using the'-h' option will list all the output in "Human Readable Format" used the power of 1024

3. How to sum up the total of the disk space usage?

```
# df -h --total
```

```
Output:

Filesystem      Size  Used Avail Use% Mounted on
/dev/sda2       158G  137G   15G  91% /
udev             10M     0   10M   0% /dev
tmpfs           3.2G  322M  2.9G  11% /run
tmpfs           7.9G     0  7.9G   0% /dev/shm
tmpfs           5.0M     0  5.0M   0% /run/lock
tmpfs           7.9G     0  7.9G   0% /sys/fs/cgroup
/dev/sda1        95M   75M   16M  84% /boot
tmpfs           1.6G     0  1.6G   0% /run/user/301703
tmpfs           1.6G     0  1.6G   0% /run/user/301483
tmpfs           1.6G     0  1.6G   0% /run/user/301613
tmpfs           1.6G     0  1.6G   0% /run/user/301677
total           183G  137G   40G  78% -
```

Note: using '--total' along with '-h' will sum up the total disk usage of all the file

systems.

4. How to list the Inodes information of all file systems?

```
# df -i
```

```
 Output:

Filesystem       Inodes   IUsed   IFree IUse% Mounted on
/dev/sda2      10240000 2491518 7748482   25% /
udev            2054985     305 2054680    1% /dev
tmpfs           2057262     767 2056495    1% /run
tmpfs           2057262       1 2057261    1% /dev/shm
tmpfs           2057262      11 2057251    1% /run/lock
tmpfs           2057262      13 2057249    1% /sys/fs/cgroup
/dev/sda1         25168     336   24832    2% /boot
tmpfs           2057262       4 2057258    1% /run/user/301703
tmpfs           2057262       4 2057258    1% /run/user/301483
tmpfs           2057262       4 2057258    1% /run/user/301613
tmpfs           2057262       4 2057258    1% /run/user/301677
```

Note: Using '-i' will list the information about the Inodes of all the filesystem.

5. How to list the file system usage in MB (MegaByte)?

```
# df -m
```

```
 Output:

Filesystem     1M-blocks   Used Available Use% Mounted on
/dev/sda2         161097 139711     14776  91% /
udev                 10      0        10   0% /dev
tmpfs              3215    322      2894  10% /run
tmpfs              8037      0      8037   0% /dev/shm
tmpfs                 5      0         5   0% /run/lock
tmpfs              8037      0      8037   0% /sys/fs/cgroup
/dev/sda1            95     75        16  84% /boot
tmpfs              1608      0      1608   0% /run/user/301703
tmpfs              1608      0      1608   0% /run/user/301483
tmpfs              1608      0      1608   0% /run/user/301613
```

Note: Using the '-m' option we can get the output of all the file systems disk space usage in MB (megabytes).

6. How to check the file system type?

```
# df -T
```

```
 Output:

Filesystem     Type     1K-blocks      Used Available Use% Mounted on
/dev/sda2      ext4     164962420 143084916  15108428  91% /
udev           devtmpfs     10240         0     10240   0% /dev
tmpfs          tmpfs      3291620    329144   2962476  10% /run
tmpfs          tmpfs      8229048         0   8229048   0% /dev/shm
tmpfs          tmpfs         5120         0      5120   0% /run/lock
tmpfs          tmpfs      8229048         0   8229048   0% /sys/fs/cgroup
/dev/sda1      ext2         97167     76552     15598  84% /boot
tmpfs          tmpfs      1645812         0   1645812   0% /run/user/301703
tmpfs          tmpfs      1645812         0   1645812   0% /run/user/301483
tmpfs          tmpfs      1645812         0   1645812   0% /run/user/301613
tmpfs          tmpfs      1645812         0   1645812   0% /run/user/301677
```

Note: Using the'-T' option we can get the list of file system types. As you can see types of file systems in the above example's 2nd column "TYPE" as "ext4, tmpfs, ext2".

7. How to check the disk space details of a specific file system type?

```
# df -t ext4
```

```
 Output:

Filesystem     1K-blocks      Used Available Use% Mounted on
/dev/sda2      164962420 143225180  14968164  91% /
```

Note: Using the'-t' option we can filter the output of a specific file system. In this example, I have used the "ext4" file system. You can check it accordingly.

8. How to customize the output with certain columns?

```
# df -h / --output=size,used
 Output:
 Size Used
158G 137G
```

Note: Using '--output[=FIELD_LIST]'with df command you can customize your output with certain fields/columns. Check the example I used here.

```
# df --help or man help
```

Sample Program:

Read sectors from a disk

```
#include<stdio.h>
int main()
{
        int numSecteur=10;
        char secteur[1024];
        FILE* disqueF=fopen("/dev/sda6","r");
        fseek(disqueF, numSecteur*512,SEEK_SET);
        fread(secteur, 1024, 1, disqueF);
        //printf("%s",se
        for (int i = 0; i < sizeof(secteur); i++) {
           printf ("%x ", secteur[i]);

           if ((i + 1) % 16 == 0)
              printf ("\n");
```

```
        }
        fclose(disqueF);
        return 0;
}
```

Lab Exercises:

1. Identify the number of hard disks connected to the system.
2. Identify the available memory in the system

Additional Exercises:

1. Write a program to read a sector from the pen drive? Identify the number of partitions and other details with the pen drive.

2. Display the list of devices connected to your system including the physical names and its instance number.