LINUX PROGRAMMING LAB REFERENCE MATERIAL

(MIT, Manipal)

BASIC LINUX COMMANDS

shell

a utility program that enables the user to interact with the Linux operating system. Commands entered by the user are passed by the shell to the operating for execution. The results are then passed back by the shell and displayed on the user's display. There are several shells available like Bourne shell, C shell, Korn shell, etc. Each shell differs from the other in Command interpretation. The most popular shell is bash. On most Linux distributions, the program /bin/sh (the default shell) is actually a link to the program /bin/bash.

shell prompt

a character at the start of the command line which indicates that the shell is ready to receive the commands. The character is usually a '%' (percentage sign) or a '$' (dollar sign).

For. e.g.
Last login : Thu April 11 06:45:23
$ _ (This is the shell prompt, the cursor shown by the _ character).

Linux commands are executable binary files located in directories with the name bin (for binary). Many of the commands that are generally used are located in the directory /usr/bin.

echo is a command for displaying any string in the command prompt.

For e.g. $ echo "Welcome to MIT Manipal"

Environment variables: Shell has built in variables which are called environment variables. For e.g. the user who has logged in can be known by typing

$**echo** $USER

The above will display the current user's name.

When the command name is entered, the shell checks for the location of the command in each directory in the PATH environment variable. If the command is found in any of the directories mentioned in PATH, then it will execute. If not found, will give a message "Command not found".

COMMONLY USED LINUX COMMANDS

who: Linux is a system that can be concurrently used by multiple users and to know the users who are using the system can be known by a **who** command. For e.g. Current users are kumar, vipul and raghav. These are the user ids of the current users.

```
$ who [Enter]
kumar   pts/10   May 1  09.32
vipul    pts/4     May 1   09.32
raghav   pts/5   May 1  09.32
```

The first columns indicates the user name of the user, second column indicates the terminal name and the third column indicates the login time. To know the user who has invoked the command can be known by the following command. For e.g. if kumar is the user who has typed the who command:

```
$ whoami [Enter]
kumar   pts/10   May 1  09.32
```

ls: UNIX system has a large number of files that control its functioning and users also create files on their own. These files are stored in separate folders called directories. We can list the names of the files available in this directory with **ls** command. The list is displayed in the order of creation of files.

```
$ ls [Enter]
README
chap01
chap02
chap03
helpdir
progs
```

In the above output, **ls** displays a list of six files. We can also list specific files or directories by specifying the file name or directory names. In this we can use regular expressions.

For e.g. to list all files beginning with chap we can use the following command.

```
$ ls chap* [Enter]
chap01
chap02
chap03
```

To list further detailed information we can use **ls -l** command, where **-l** is an option between the command and filenames. The details include, file type, file or directory access permissions, number of links, owner name, group name, file or directory size, modification time and name of file or directory.

```
$ ls  -l  chap*  [Enter]
-rw-r- - r- -     l  kumar   users    5670 Apr 3   09.30  chap01
-rw-r- - r- -     l  kumar   users    5670 Feb 23  09.30  chap02
-rw-r- - r- -     l  kumar   users    5670 Apr 30  09.34  chap03
```

The argument beginning with hyphen is known as option. The main feature of option is it starts with hyphen. The command **ls** prints the columnar list of files and directories. With the **–l** option it displays all the information as shown above.

General syntax of **ls** command:

ls –[options][file list][directory list]

In Linux, file names beginning with period are hidden files, are not normally displayed in **ls** command. To display all files, including the hidden ones, use option **–a** in **ls** command as shown below:

```
$ ls -a
```

$ ls / will display the name of the files and sub-directories under the root directory.

pwd: This command gives the present working directory where the user is currently located.

```
$ pwd
/home/kumar/pis
```

cd: To move around in the file system use cd (change directory) command. When used with argument, it changes the current directory to the directory specified as argument, for instance:

```
$ pwd
/home/kumar
$ cd progs
$ pwd
$ /home/kumar/progs
```

cd .. : To change the working directory to the parent of the current directory we need to use

```
$ cd ..
```

.. (double dot) indicates parent directory. A single dot indicates current directory.

cat: **cat** is a multipurpose command. Using this we can display a file, create a file as well as concatenate files vertically.

```
$ cat > filename[Enter]
cat > os.txt
```

Welcome to Manipal. (This content will be placed in the file os.txt)

[Ctrl + D] End of input

$_ (comes to the shell prompt)

The above command will create a file named os.txt in the current directory. To see the contents of the file.

```
$ cat os.txt[Enter]
```

Welcome to Manipal.
To display a file we can use **cat** command as shown above.

We can use **cat** for displaying more than one file, one after the other by listing the files after **cat**. For e.g.

```
$ cat os.txt lab.txt
```

the contents of os.txt followed by lab.txt

cp: To copy the contents of one file to another.
Syntax: **cp** sourcefilename targetfilename [Enter]

This command is also used to copy one or more files to a directory. The syntax of this form of **cp** command is

Syntax : **cp** filename(s) directoryname

If the file **os.txt** in current directory i.e. /home/kumar/pis needs to be copied into /home directory then it will be done as follows.

$ **cp** os.txt /home/   **OR**   **$ cp** os.txt ../../

mv: This command renames or moves files. It has two distinct functions: It renames a file or a directory and it moves a group of files to a different directory.
Syntax: **mv** oldfilename newfilename

Syntax of another form of this command is

mv file(s) directory

mv doesn't create a copy of the file, it merely renames it. No additional space is consumed on disk for the file after renaming. To rename the file chap01 to man01,

$ mv chap01 man01.

If the destination file doesn't exist, it will be created. For the above example, **mv** simply replaces the filename in the existing directory with the new name. By default **mv** doesn't prompt for overwriting the destination file if it exist.

The following command moves three files to the progs directory:

$ **mv** chap01 chap02 chap03 progs

mv can also be used to rename a directory for instance pis to pos:
$ **mv** pis pos

rm: This command deletes one or more files.
Syntax: **rm** filename(s)

The following command deletes three files

```
$ rm chap01 chap02 chap03[Enter]
```

A file once deleted can be recovered subject to conditions by using additional software. **rm** won't normally remove a directory but it can remove files from one or more directories. It can remove two chapters from the progrs directory by using:

```
$ rm progrs/chap01 progrs/chap02
```

mkdir: Directories are created by **mkdir** command. The command is followed by the name of the directories to be created.

Syntax: **mkdir** directoryname(s)

```
$ mkdir data
```

This creates a directory named data under the current directory.

```
$ mkdir data dbs doc
```

The above command creates three directories with names data, dbs and doc.

rmdir : Directories are removed by **rmdir** command. The command is followed by the name of the directory to be removed. If a directory is not empty, then the directory will not be removed.

Syntax: **rmdir** directoryname

```
$ rmdir patch [Enter]
```

The command removes the directory by the name patch.

In Linux, every file and directory has access permissions. Access permissions define which users have permission to access a file or directory. Permissions are of three types, read, write and execute. Access permissions are defined for user, group and others.

For e.g. If access permission is only read for user, group and others, then it will be
r--r--r--

Access permissions can also be represented as a number. This number is in octal system. An access permission represented in numerical octal format is called absolute permission. The absolute permission for the above is

444

If the access permission is read, write for user, read, execute for group and only execute for others then it will be,

rw-r-x--x

The absolute permission for the above is

651

chmod: changes the permission specified in the argument and leaves the other permissions unaltered. In this mode the following is the syntax.

Sytax: **chmod** category operation permission filename(s)

chmod takes as its argument an expression comprising some letters and symbols that completely describe the user category and the type of permission being assigned or removed. The expression contains three components:

User category (user, group, others)

The operation to be performed (assign or remove a permission). The type of permission (read, write and execute)

The abbreviations used for these three components are shown in Table 1.1.
E.g. to assign execute permission to the user of the file xstart;

```
$ chmod u+x xstart
$ ls –l xstart
- rwxr- - r- -    l    kumar       metal     1980    May 01 20:30 xstart.
```

The command assigns (+) execute (x) permission to the user (u), but other permissions remain unchanged. Now the owner of the file can execute the file but the other categories i.e. group and others still can’t. To enable all of them to execute this file:

```
$ chmod ugo+x xstart
$ ls –l xstart
- rwxr-x r- x    l    kumar       metal     1980    May 01 20:30 xstart.
```

The string **ugo** combines all the three categories user, group and others. This command accepts multiple filenames in the command line:

```
$ chmod u+x note note1 note3
```

$ chmod a-x, go+r xstart; ls –l xstart (Two commands can be run simultaneously with ;)

```
- rw-r--rwx      l    kumar       metal     1980    May 01 20:30 xstart.
```

Table 1.1: Abbreviations Used by chmod

Category	Operation	Permission
u- User	+ Assigns permission	r- Read permission
g- Group	- Removes permission	w- Write permission
o- Others	= Assigns absolute permission	x- Execute permission
a- All(ugo)		

Absolute Permissions:

Sometimes without needing to know what a file's current permissions the need to set all nine permission bits explicitly using **chmod** is done.

Read permission – 4 (Octal 100)

Write permission – 2 (Ocal 010)

Execute permission – 1 (Octal 001)

For instance, 6 represents read and write permissions, and 7 represents all permissions as can easily be understood from Table 1.2.

Table 1.2: Absolute Permissions

Binary	Octal	Permissions	Significance
000	0	---	No permissions
001	1	--x	Executable only
010	2	-w-	Writable only
011	3	-wx	Writable and executable
100	4	r--	Readable only
101	5	r-x	Readable and executable
110	6	rw-	Readable and writable
111	7	rwx	Readable, writable and executable

$ **chmod 666 xstart; ls –l xstart**

```
- rw-rw- rw -   l    kumar       metal     1980    May 01 20:30 xstart.
```

The 6 indicates read and write permissions (4 + 2).

date: This displays the current date as maintained in the internal clock run perpetually.
$ **date** [Enter]

clear: The screen clears and the prompt and cursor are positioned at the top-left corner.
$ **clear [Enter]**

man: is used to display help file related to a command or system call.

Syntax: **man {command name/system call name}**

e.g. man date

man open

wc: displays the count of lines, words and characters in a file.

e.g. wc os.txt

```
1  3  19 os.txt
```

Syntax: **wc [-c | -m | -C] [-l] [-w] [file….]**

Options: The following options are supported:

-c Count bytes.

-m Count characters.

-C Same as – m,

-l   Count lines

-w Count words delimited by white space characters or new line characters.
If no option is specified the default is –lwc (count lines, words, and bytes).

Redirection Operators

For any program whether it is developed using C, C++ or Java, by default three streams are available known as input stream, output stream and error stream. In programming languages, to refer to them some symbolic names are used (i.e. they are system defined variables).

The following operators are the redirection operators

1. > standard output operator

> is the standard output operator which sends the output of any command into a file.

Syntax: command > file1

e.g. **ls > file1**

Output of the **ls** command is sent to a file1. If the file1 doesn’t exist, it is created otherwise, it is overwritten.

E.g.: **cat file1 > file2**

Here, file2 get the content of file1.

E.g.: **cat file1 file2 file3 > file4**

This creates the file file4 which gets the content of all the files file1, file2 and file3 in order.

2. < standard input operator

< operator (standard input operator) allows a command to take necessary input from a file.

Syntax: **$ command < file**

E.g.: **cat<file1**

This displays output of file file1 on the screen.

E.g.: **cat <file1 >file2**

This makes cat command to take input from the file file1 and write its output to the file file2. That is, it works like a **cp** command.

3. >> appending operator

Similarly, >> operator can be used to append standard output of a command to a file. E.g.: **command>>file1**

This makes, output of the given command to be appended to the file1. If the file1 doesn’t exist, it will be created and then standard output is written.

4. << document operator

There are occasions when the data which the program reads is fixed and fairly limited. The shell uses the << symbols to read data from the same file containing the script. This is referred to as **here document**, signifying that the data is here rather than in a separate file. Any command using standard input can also take input from a here document.

Example.:

```
#!/bin/bash
cat <<DELIMITER
```

Output:

```
hello
this is a here
document
DELIMITER
```

This gives the output:

```
hello
this is a here
document
```

Shell Concepts

This section will describe some of the features that are common in all of the shells.

1. Wild-card: The metacharacters that are used to construct the generalized pattern for matching filenames belong to a category called wild-cards.

List of shell's wild-cards:

Wild-card	**Matches**
*	Any number of characters including none
?	A single or zero character
[ijk]	A single character - either an i, j or k
[x –z]	A single character between x and z
[!ijk]	A single character that is not an i, j or k.
[!x–z]	A single character not between x and z.
{pat1, pat2, ….}	pat1, pat2, etc.

Example: Consider a directory structure /home/kumar which have the following files:

```
README
chap01
```

```
chap02
chap03
helpdir
progs
```

Then with the below command the following output would be displayed.

```
$ ls chap*
chap    chap01 chap02 chap03
$ ls .*
.bash_profile   .exrc    .netscape        .profile
```

2. Pipes (|): Standard input and standard output constitute two separate streams that can be individually manipulated by the shell. Using pipes, it is possible to give the output of one command as input to the other.

Use **ls** command which produces the list of files, one file per line. Use redirection to save this output to a file:

```
$ ls > user.txt
$ cat user.txt
```

The file shows the list of files.

Now to count the number of files:

```
$ ls | wc – l
```

The above command gives the number of files. This is how | (pipe) is used. There's no restriction on the number of commands that could be used in pipe.

3. Command substitution: The shell enables connecting of two commands in yet another way. While a pipe enables a command to obtain its standard input from the standard output of another command, the shell enables one or more command arguments to be obtained from the standard output of another command. This feature is called command substitution.

```
$ echo The date today is `date`
The date today is Sat May 6 19:01:56 IST 2017
$ echo “There are total `ls | wc –l ` files and sub-directory in the current directory
There are 15 files and sub-directory in the current directory.
```

4. Sequences: Two separate commands can be written in one line using “;” in sequences.
$ **chmod 666 xstart; ls –l xstart**

5. Conditional Sequences: The shell provides two operators that allow conditional execution - the && and ||, which typically have this syntax:

cmd1 && cmd2

cmd1 || cmd2

The && delimits two commands; the command cmd2 is executed only when cmd1 succeeds.

The || operator plays inverse role; the second command cmd2 is executed only when the first command cmd1 fails.

Note: All built-in shell commands returns non-zero if they fail. They return zero on success.
e.g: if there is a program hello.c which displays ‘Hello World’ on compilation and execution. Then the following command in conditional sequences could be used to display the same:

$ **cc hello.c && ./a.out**

This command displays the output ‘Hello World’ if the compilation of the program succeeds. Similarly in case the compilation fails for the program the following output ‘Error’ could be displayed with the following command:

$ **cc hello.c || echo ‘Error’**

File Filters commands in Linux:

1. head: To see the top 10 lines of a file - $ **head <file name>**
To see the top 5 lines of a file - $ **head -5 <file name>**

2. tail: To see last 10 lines of a file - $ **tail < file name>**
To see last 20 lines of a file - $ **tail -20 <file name>**

3. more: To see the contents of a file in the form of page views - $ more <file name>
$ **more f1.txt**

4. grep: To search a pattern of word in a file, **grep** command is used.

Syntax: **$ grep < word name> < file name>**

$ **grep hi file_1**

To search multiple words in a file

$ **grep -E 'word1|word2|word3' <file name>**

$ **grep -E 'hi|beyond|good' file_1**

5. sort: This command is used to sort the file.

$ **sort <file name>**

$ **sort file_1**

To sort the files in reverse order

$ sort -r <file name>

To display only files

$ ls –l | grep "^-"

To display only directories

$ ls –l | grep "^d"

Lab Exercises:

1. Write shell commands for the following.
   i) Create a directory in your home directory having 2 subdirectories.
   ii) In the first subdirectory, create 3 different files with different content in each of them.
   iii) Copy the first file from the first subdirectory to the second subdirectory.
   iv) Create one more file in the second subdirectory, which contains the command listing the number of users and number of files.
   v) To list all the files which starts with either 'a' or 'A'.
   vi) Display the output if the compilation of a sort program succeeds else display an error message.
   vii) Display the first five and last five lines of a given file.
   viii) Redirect the output of commands 'pwd', 'date' and 'ls' in succession to a file.

Additional Exercises:

1. Write shell commands for the following.
   i) A file contains few student records with each record containing Roll No., Name and Branch, separated by comma. Sort the file based on 'Name' of the student.
   ii) Display the number of lines containing a given number, say *50*, in all the files in the current directory.
   iii) Display the first two lines of the given file containing the string "OS"

SHELL SCRIPTING – 1

The Linux shell is a program that handles interaction between the user and the system. Many of the commands that are typically thought of as making up the Linux system are provided by the shell. Commands can be saved as files called scripts, which can be executed like a program.

SHELL PROGRAMS: SCRIPTS
SYNTAX: **scriptname**

NOTE: A file that contains shell commands is called a script. Before a script can be run, it must be given execute permission by using **chmod** utility (chmod +x script). To run the script, only type its name. They are useful for storing commonly used sequences of commands to full-blown programs.

VARIABLES

Table 2.1 : Parameter Variables

$@	an individually quoted list of all the positional parameters
$#	the number of positional parameters
$!	the process ID of the last background command
$0	The name of the shell script.
$$	The process ID of the shell script, often used inside a script for generating unique temporary filenames; for example /tmp/tmpfile_$$.
$1, $2, …	The parameters given to the script.
$*	A list of all the parameters, in a single variable, separated by the first character in the environment variable IFS.

Example:

1. Try the following shell commands

```
$ echo $HOME, $PATH
$ echo $MAIL
$ echo $USER, $SHELL, $TERM
```

2. Try the following snippet, which illustrates the difference between local and environment variable:

```
$ firstname=Rakesh         ……local variables
$ lastname=Sharma
$ echo $firstname $lastname
$ export lastname          …..make "lastname" an envi var
$ sh                       …..start a child shell
$ echo $firstname $lastname
$ ^D                       …..terminate child shell
$ echo $firstname $lastname
```

3. Try the following snippet, which illustrates the meaning of special local variables:

```
$ cat >script.sh
 echo the name of this script is $0
 echo the first argument is $1
 echo a list of all the arguments is $*
 echo this script places the date into a temporary file
 echo called $1.$$
 date > $1.$$          # redirect the output of date
 ls $1.$$              # list the file
 rm $1.$$              # remove the file
 ^D
 $ chmod +x script.sh
 $ ./script.sh Rahul Sachin Kumble
```

NOTE: A shell supports two kinds of variables: local and environment variables. Both hold data in a string format. The main difference between them is that when a shell invokes a subshell, the child shell gets a copy of its parent shell's environment variables, but not its local variables. Environment variables are therefore used for transmitting useful information between parent shells and their children.

Few predefined environment variables:

$HOME	pathname of our home directory
$PATH	list of directories to search for commands
$MAIL	pathname of our mailbox
$USER	our username
$SHELL	pathname of our login shell
$TERM	type of the terminal

Creating a local variable:

```
variableName=value
```

Operations:

- Simple assignment and access
- Testing of a variable for existence
- Reading a variable from standard input
- Making a variable read only
- Exporting a local variable to the environment

Creating / Assigning a variable:

Syntax: {name=value}+

```
Example: $ firstName=Anand  lastname=Sharma  age=35
         $ echo $firstname  $lastname $age
         $ name = Anand Sharma
         $ echo $name
         $ name = “Anand Sharma”
         $ echo $name
```

Accessing variable:

Syntax: $name / ${name}

```
Example: $ verb=sing
         $ echo I like $verbing
```

Reading a variable from standard input:

Syntax: read {variable}+

```
Example:  $ cat > script.sh
            echo “Please enter your name:”
            read name
            echo your name is $name
            ^D
```

Read-only variables:

Syntax: readonly {variable}+

```
Example:  $ password=manipal
          $ echo $password
          $ readonly password
          $ readonly              …..list
          $ password=mangalore
```

Running jobs in Background

A multitasking system lets a user do more than one job at a time. Since there can be only one job in foreground, the rest of the jobs have to run in the background. There are two ways of doing this: with the shell's **& operator** and **nohup** command. The latter permits to log out while the jobs are running, but the former doesn't allow that.

```
$ sort –o emp.lst &
 550
```

The shell immediately returns a number the PID of the invoked command (550). The prompt is returned and the shell is ready to accept another command even though the previous command has not been terminated yet. The shell however remains the parent of the background process. Using an & many jobs can be run in background as the system load permits.

In the above case, if the shell which has started the background job is terminated, the background job will also be terminated. **nohup** is a command for running a job in background in which case the background job will not be terminated if the shell is close. nohup stands for no hang up.

e.g.

```
$ nohup sort-o emp.lst &
586
```

The shell returns the PID too. When the **nohup** command is run it sends the standard output of the command to the file **nohup.out**. Now the user can log out of the system without aborting the command.

JOB CONTROL

1. ps: ps is a command for listing processes. Every process in a system will have unique id called process id or PID. This command when used displays the process attributes.

```
$ ps
```

```
PID     TTY     TIME  CMD
291     console         0:00    bash
```

This command shows the PID, the terminal TTY with which the process is associated, the cumulative processor time that has been consumed since the process has started and the process name (CMD).

2. kill: This command sends a signal usually with the intention of killing one or more process. This command is an internal command in most shells. The command uses one or more PIDs as its arguments and by default sends the SIGTERM(15) signal. Thus: $ **kill 105** terminates the job having PID 105. The command can take many PIDs at a time to be terminated.

3. sleep: This command makes the calling process sleep until the specified number of seconds or   a signal arrives which is not ignored.

$ **sleep** 2

Example:

1. Try the following, which illustrates the usage of **ps**:

```
$ (sleep 10; echo done) &
$ ps
```

2. Try the following, which illustrates the usage of **kill**:

```
$ (sleep 10; echo done) &
$ kill pid        …..pid is the process id of background process
```

3. Try the following, which illustrates the usage of **wait**:

```
$ (sleep 10; echo done  1 ) &
$ (sleep 10; echo done  2 ) &
$ echo done 3; wait ;  echo done 4          ….wait for children
```

NOTE: The following two utilities and one built-in command allow the listing controlling the current processes.

ps: generates a list of processes and their attributes, including their names, process ID numbers, controlling terminals, and owners

kill: allows to terminate a process on the basis of its ID number

wait: allows a shell to wait for one or all of its child processes to terminate

Example:

```
$ cat>script.sh
echo there are $# command line arguments: $@
^D
$ script.sh arg1 arg2
```

Example:

```
#!/bin/sh
salutation="Hello"
echo $salutation
echo "The program $0 is now running"
echo "The second parameter was $2"
echo "The first parameter was $1"
echo "The parameter list was $*"
echo "The user's home directory is $HOME"
echo "Please enter a new greeting"
read salutation
echo $salutation
echo "The script is now complete"
exit 0
```

If we save and execute the above shell script as try.sh, we get the following output:

```
$ ./try.sh foo bar baz
Hello
The program ./try.sh is now running
The second parameter was bar
The first parameter was foo
The parameter list was foo bar baz
The user's home directory is /home/rick
Please enter a new greeting
Sire
Sire
The script is now complete
$
```

Lab Exercises:

Write Shell Scripts to do the following:

1. List all the files under the given input directory, whose extension has only one character
2. Write a shell script that accepts two command line parameters. First parameter indicates the directory and the second parameter indicates a regular expression. The

script should display all the files and directories in the directory specified in the first argument matching the format specified in the second argument.

3. Count the number of users logged on to the system. Display the output as Number of users logged into the system.
4. Count only the number of files in the current directory.
5. Write a shell script that takes two sorted numeric files as input and produces a single sorted numeric file without any duplicate contents.
6. Write a shell script that accepts two command line arguments. First argument indicates format of file and the second argument indicates the destination directory. The script should copy all the files as specified in the first argument to the location indicated by the second argument. Also, try the script where the destination directory name has space in it.

Additional Exercises:

1. Write Shell Scripts to do the following
   - i) To list all the .c files in any given input subdirectory.
   - ii) Count all the sub directories (recursively) in a given path.

SHELL SCRIPTING – 2

The shell is not just a collection of commands but a really good programming language. A lot of tasks could be automated with it, along with this the shell is very good for system administration tasks. Many of the ideas could be easily tried with it thus making it as a very useful tool for simple prototyping and it is very useful for small utilities that perform some relatively simple tasks where efficiency is less important as compared to the ease of configuration, maintenance and portability.

COMMENTS

Comments in shell programming start with # and go until the end of the line.

List variables

Syntax: declare [-ax] [listname]

Example: $ declare –a teamnames

```
           $ teamnames[0] = “India”        …..assignment
           $ teamnames[1] = “England”
           $ teamnames[2] = “Nepal”
           $ echo “There are  ${#teamnames[*]} teams          ….accessing
           $ echo “They are: ${teamnames [*]}”
           $ unset teamnames[1]                              …delete
```

Aliases

Allows to define your own commands

Syntax: alias [word[=string]]

```
         Unalias [-a] {word}+
```

Example: $ alias dir=“ls –aF”

```
           $ dir
```

ARITHMETIC

expr utility is s used for arithmetic operations. All of the components of expression must be separated by blanks, and all of the shell metacharacters must be escaped by a \.

Syntax: **expr** expression

Example:
```
$ x=1
$ x=`expr $x +1`
$ echo $x
$ x=`expr  2 + 3  \* 5`
$echo $x
$echo `expr \( 4 \> 5 \)`
$echo `expr length “cat”`
$echo `expr substr “donkey” 4 3`
```

TEST EXPRESSION

Syntax: test expression

Table 3.1 :Forms of Test Expressions

Test	Meaning
!=	not equal
=	equal
-eq	equal
-gt	greater than
-ge	greater than or equal
-lt	less than
-le	less than or equal
!	logic negation
-a	logical and
-o	logical or
-r file	true if the file exists and is readable
-w file	true if the file exists and is writable
-x file	true if the file exists and is executable
-s file	true if the file exists and its size > 0
-d file	true if the file is a directory
-f file	true if the file is an ordinary file
-t filed	true if the file descriptor is associated with a terminal
-n str	true if the length of str is > 0
-z str	true if the length of str is zero

CONTROL STRUCTURES

(i) The **if** conditional

Syntax: **if** *command1*
then *command2*
fi

Example:

```
echo “enter a number:”
read number
if  [ $number  -lt  0 ]
then
echo “negative”
elif  [ $number -eq 0 ]
then
echo “zero”
else
echo “positive”
fi
```

(ii) The **case** conditional

Syntax: **case** string **in**
pattern1) commands1 ;;
pattern2) commands2 ;;

……..
esac

case selectively executes statements if string matches a pattern. You can have any number of patterns and statements. Patterns can be literal text or wildcards. You can have multiple patterns separated by the "|" character.

Example:

```
case $1 in
*.c)
cc $1
;;
*.h | *.sh)
# do nothing
;;
*)
```

The above example performs a compile if the filename ends in .c, does nothing for files ending in .h or .sh. else it writes to stdout that the file is an unknown type. Note that the: character is a NULL command to the shell (similar to a comment field).

```
case $1 in
[AaBbCc])
option=0
;;
*)
option=1
;;
esac
echo $option
```

In the above example, if the parameter $1 matches A, B or C (uppercase or lowercase), the shell variable *option* is assigned the value 0, else is assigned the value 1.

(iii) **while**: looping

Syntax: **while** *condition is true*
do
commands
done

Example 1:

```
# menu program
echo “menu test program”
stop=0
while test $stop  -eq 0
do
cat << ENDOFMENU
1: print the date
2,3     : print the current working directory
4: exit
ENDOFMENU
echo
echo “your choice ?”
```

```
read reply
echo
case $reply in
    "1")
            date
              ;;
    "2" | "3")
            pwd
               ;;
     "4")
           stop =1
               ;;
       *)
           echo "illegal choice"
               ;;
  esac
done
```

Example 2:

```
#!/bin/bash
X=0
while [ $X -le 20 ]
do
            echo $X
            X=$((X+1))
done
# echo all the command line arguments
while test $# != 0
do
          echo $1
          #The shift command shifts arguments to the left
          shift
done
```

(iv) **until:** Looping

Syntax: **until** *command-list1*
do
command-list2
done

Example:

```
x=1
until [ $x –gt 3  ]
do
echo x = $x
x=`expr $x + 1`
done
```

(v) **for:** Looping

Syntax: **for** *variable* in *list*
do
command-list
done

Sample Program

```
homedir=`pwd`
for files in /*
do
echo $files
done
cd $homedir
```

The above example lists the names of all files under / (the root directory)

Lab Exercises:

Write shell scripts to perform the following

1. Find whether the given number is even or odd.
2. Print the first ‘n’ odd numbers.
3. Find all the possible quadratic equation roots using case.
4. Find the factorial of a given number.
5. Implement a simple calculator

Additional Exercises

1. Write Shell scripts to do the following
   i) Find whether the given string is palindrome.
   ii) Find out the sum of the numbers given by user.
   iii) Delete all sub directories of size 0 for a given directory
   iv) Implement ADDRESS BOOK with search, add, delete and edit options

LINUX SYSTEM CALLS

Basic system calls:

The directory functions are declared in the header file **dirent.h**. This uses a structure **DIR** as a basis for directory manipulation. A pointer to this structure called the directory stream (a **DIR ***), acts in much the same way as a file stream (FILE *) does for the file manipulation. Directory entries themselves are returned in dirent structure, also declared in dirent.h. One should never alter the fields in DIR structure directly. Some functions reviewed below are:

opendir: This function opens a directory and establishes a directory stream. If successful, it returns a pointer to a DIR structure to be used for reading directory entries.

```
#include <sys/types.h>
#include <dirent.h>
```
DIR *opendir(const char *name);
opendir returns a null pointer on failure. Note that a directory stream uses a low-level file descriptor to access the directory itself, so opendir could fail with too many open files.

readdir: The **readdir** function returns a pointer to a structure detailing the next directory entry in the directory stream **dirp**. Successive calls to **readdir** return further directory entries. On error, and at the end of the directory, **readdir** returns NULL. POSIX-compliant systems leave errno unchanged when returning **NULL** at end of directory and set it when an error occurs.

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

struct dirent *readdir(DIR *dirp);

Note that **readdir** scanning isn't guaranteed to list all the files (and subdirectories) in a directory if there are other processes creating and deleting files in the directory at the same time. The **dirent** structure containing directory entry details includes the following entries:

ino_t d_ino: The inode of the file
char d_name[]: The name of the file

On Linux, the dirent structure is defined as follows:

```
struct dirent {
ino_t    d_ino;  /* inode number */
off_t    d_off;   /* offset to the next dirent */
unsigned short d_reclen;         /* length of this record */
unsigned char d_type;  /* type of file */
char     d_name[256];   /* filename */
};
```

To create and remove directories the **mkdir** and **rmdir** system calls are used.

```
#include <sys/stat.h>
int mkdir(const char *path, mode_t mode);
```

The **mkdir** system call is used for creating directories and is the equivalent of the **mkdir** program. The **mkdir** makes a new directory with path as its name. The directory permissions are passed in the parameter mode and are given as in the O_CREAT option of the open system call and, again, subject to umask.

```
#include <unistd.h>
int rmdir(const char *path);
```

The **rmdir** system call removes directories, but only if they are empty. The **rmdir** program uses this system call to do its job.

closedir: The **closedir** function closes a directory stream and frees up the resources associated with it. It returns 0 on success and 1 if there is an error.

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dirp);
```

<u>Sample program to simulate ls command</u>

```
#include<stdio.h>
#include<dirent.h>  //for directory commands
#include<stdlib.h>    //for exit
int main()
{
      char dirname[10];
      DIR*p;                        // pointer to directory
```

```
        struct dirent *d; // pointer to directory entry structure
        printf("Enter directory name\n");
        scanf("%s", dirname);
        p=opendir(dirname);
        if(p==NULL)
          {
            printf("Cannot find directory");
            exit(-1);
          }
        while(d = readdir(p))
         printf("%s\n", d->d_name);
         return 0;
}
```

<u>**System Calls Related to Processes**</u>

getpid()
This function returns the process identifiers of the calling process.

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);   // this function returns the process identifier (PID)
pid_t getppid(void); // this function returns the parent process identifier (PPID)
```

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

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

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

returns the PID of the child, whereas to the child the PID returned is zero. This is shown in the following Fig. 4.1.

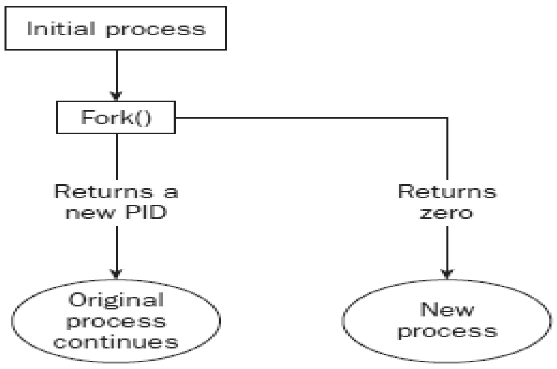


Figure 4.1: Fork system call

In Linux in case of any error observed in calling the system functions, then a special variable called errno will contain the error number. To use errno a header file named errno.h has to be included in the program. If fork fails, it returns -1. This is commonly due to a limit on the number of child processes that a parent may have (CHILD_MAX), in which case errno will be set to EAGAIN. If there is not enough space for an entry in the process table, or not enough virtual memory, the errno variable will be set to ENOMEM.

A typical code snippet using fork is

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

Sample Program on fork1.c

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

```
#include <unistd.h>
#include <stdio.h>
int main()
{
        pid_t pid;
        char *message;
        int n;
        printf(“fork program starting\n”);
        pid = fork();
        switch(pid)
        {
                case -1:
                        perror(“fork failed”);
                        exit(1);
                case 0:
                        message = “This is the child”;
                        n = 5;
                        break;
                default:
                        message = “This is the parent”;
                        n = 3;
                        break;
        }
        for(; n > 0; n--) {
                puts(message);
                sleep(1);
        }
        exit(0);
}
```

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

```
$ ./fork1
fork program starting
```

```
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
This is the child
This is the child
```

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

The wait() System Call

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

Example of wait():

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

 void main()
 {
          int status;
          pid_t  pid;
          pid = fork();
          if(pid = -1)
            printf("\nERROR child not created");
          else if (pid == 0) /* child process */
          {
                  printf("\n I'm the child!");
```

```
                    exit(0);
        }
        else /* parent process */
        {
                wait(&status);
                printf("\n I'm the parent!")
                printf("\n Child returned: %d\n", status)
        }
}
```

A few notes on this program:

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

The exit() System Call

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

The exec() System Call

The exec function will execute a specified program passed as argument to it, in the same process (Fig. 4.2). The exec() will not create a new process. As new process is not created, the process ID (PID) does not change across an execute, but the data and code of the calling process are replaced by those of the new process.

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

The versions of exec are:

- execl

- execv
- execle
- execve
- execlp
- execvp

The naming convention: exec*

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

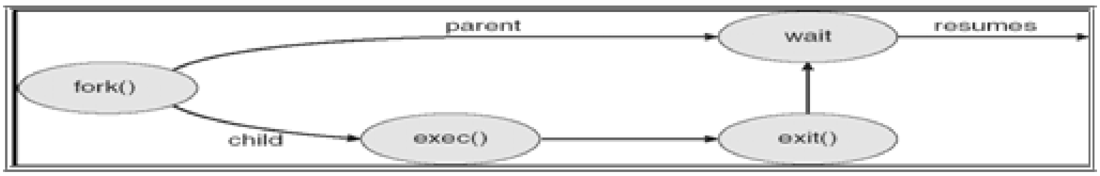


Figure 4.2: exec() system call

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

Sample Program:

C program forking a separate process.

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

```
#include<unistd.h>
int main()

{
        pid_t  pid;
        /* fork another process */
        pid = fork();
        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                exit(-1);
        }
        else if (pid == 0) { /* child process */
                execlp("/bin/ls", "ls", NULL);
        }
        else { /* parent process */
        /* parent will wait for the child to complete */
                wait (NULL);
                printf ("Child Complete");
                exit(0);
        }

}
```

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

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

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

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

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

execv: needs an array to work with.

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

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

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

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

Drawbacks:

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

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

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

```
execlp ("wc", "wc", "-l", "foo", (char *) 0);
```

execle and execve: All of the previous four exec calls silently pass the environment of the current process to the executed process by making available the environ[ ] variable to the overlaid process. Sometime there may be a need to provide a different environment to the new program - a restricted shell for instance. In that case these functions are used.

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

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

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

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

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

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

I/O SYSTEM CALLS

I/O through system calls is simpler and operates at a lower level than making calls to the C file-I/O library.

There are seven fundamental file-I/O system calls:

creat() Create a file for reading or writing.
open() Open a file for reading or writing.
close() Close a file after reading or writing.
unlink() Delete a file.
write() Write bytes to file.
read() Read bytes from file.

The creat() System Call

The "creat()" system call creates a file. It has the syntax: int fp; /* fp is the file descriptor variable */
fp = creat(<filename>, <protection bits>);
Ex: fp=creat(“students.dat”,RD_WR);

This system call returns an integer, called a "file descriptor", which is a number that identifies the file generated by "creat()". This number is used by other system calls in the program to access the file. Should the "creat()" call encounter an error, it will return a file descriptor value of -1.

The "filename" parameter gives the desired filename for the new file.

The "permission bits" give the "access rights" to the file. A file has three "permissions" associated with it:

Write permission - Allows data to be written to the file.

Read permission - Allows data to be read from the file.

Execute permission - Designates that the file is a program that can be run.

These permissions can be set for three different levels:

User level: Permissions apply to individual user.

Group level: Permissions apply to members of user's defined "group".

System level: Permissions apply to everyone on the system

The open() System Call

The "open()" system call opens an existing file for reading or writing. It has the syntax:

<file descriptor variable> = open(<filename>, <access mode>);

The "open()" call is similar to the "creat()" call in that it returns a file descriptor for the given file, and returns a file descriptor of -1 if it encounters an error. However, the second parameter is an "access mode", not a permission code. There are three modes (defined in the "fcntl.h" header file):

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing

For example, to open "data" for writing, assuming that the file had been created by another program, the following statements would be used:

int fd;

fd = open("students.dat", O_WRONLY);

A few additional comments before proceeding:

A "creat()" call implies an "open()". There is no need to "creat()" a file and then "open()" it.

The close() System Call

The "close()" system call is very simple. All it does is "close()" an open file when there is no further need to access it. The "close()" system call has the syntax:
close(<file descriptor>);

The "close()" call returns a value of 0 if it succeeds, and returns -1 if it encounters an error.

The write() System Call

The "write()" system call writes data to an open file. It has the syntax:
write(<file descriptor>, <buffer>, <buffer length>);

The file descriptor is returned by a "creat()" or "open()" system call. The "buffer" is a pointer to a variable or an array that contains the data; and the "buffer length" gives the number of bytes to be written into the file.

While different data types may have different byte lengths on different systems, the "sizeof()" statement can be used to provide the proper buffer length in bytes. A "write()" call could be specified as follows:

```
float array[10];
write( fd, array, sizeof( array ) );
```

The "write()" function returns the number of bytes it actually writes. It will return -1 on an error.

The read() Sytem Call

The "read()" system call reads data from a open file. Its syntax is exactly the same as that of the "write()" call:

read(<file descriptor>, <buffer>, <buffer length>);

The "read()" function returns the number of bytes it actually returns. At the end of file it returns 0, or returns -1 on error.

lseek: The lseek system call sets the read/write pointer of a file descriptor, fildes; that is, we can use it to set where in the file the next read or write will occur. We can set the pointer to an absolute location in the file or to a position relative to the current position or the end of file.

```
#include <unistd.h>
#include <sys/types.h>
off_t lseek(int fildes, off_t offset, int whence);
```

The offset parameter is used to specify the position, and the whence parameter specifies how the offset is used. whence can be one of the following:

SEEK_SET: offset is an absolute position

SEEK_CUR: offset is relative to the current position

SEEK_END: offset is relative to the end of the file

lseek returns the offset measured in bytes from the beginning of the file that the file pointer is set to, or -1 on failure. The type off_t, used for the offset in seek operations, is an implementation-dependent type defined in sys/types.h.

Errors*:*

EACCES Permission denied.

EMFILE Too many file descriptors in use by process.

ENFILE Too many files are currently open in the system.

ENOENT Directory does not exist, or name is an empty string.

ENOMEM Insufficient memory to complete the operation.

Lab Exercises:

1. Write a C program to block a parent process until child completes using wait system call.
2. Write a C program to load the binary executable of the previous program in a child process using exec system call.
3. Write a program to create a child process. Display the process IDs of the process, parent and child (if any) in both the parent and child processes.
4. Create a zombie (defunct) child process (a child with exit() call, but no corresponding wait() in the sleeping parent) and allow the init process to adopt it (after parent terminates). Run the process as background process and run “ps” command.
5. Write a C program to create a file and write contents to it.
6. Write a C program to copy the content of one file to other.
7. Write a C program to simulate ls command.
8. Write a C program to simulate **wc** command to count number of characters, words and lines in a file.

Additional Exercises:

1. Create a orphan process (parent dies before child – adopted by “init” process) and display the PID of parent of child before and after it becomes orphan. Use sleep(n) in the child to delay the termination.
2. Modify the program in the previous question to include wait (&status) in the parent and to display the exit return code (left most byte of status) of the child.
3. Use lseek() to copy different parts (initial, middle and last) of the file to other. (For lseek() refer to man pages)
4. Create a child process which returns a 0 exit status when the minute of time is odd and returning a non-zero (can be 1) status when the minute of time is even.

THREAD PROGRAMMING

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

- A pointer to a pthread_t structure. The call will return the handle to the thread in this structure.
- A pointer to a pthread attributes structure, which can be a null pointer if the default attributes are to be used. The details of this structure will be discussed later.
- The address of the routine to be executed.
- A value or pointer to be passed into the new thread as a parameter.

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

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

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

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

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

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

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

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

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

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

Passing Data to and from Child Threads

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

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

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

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

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

Setting the Attributes for Pthreads

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

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

Lab Exercises:

1. Write 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 that calculates summation of non-negative integers in a separate thread and passes the result to main thread.
3. Write a multithreaded program for generating prime numbers from a given starting number to the given ending number.
4. Write a multithreaded program that perform the sum of even numbers and odd numbers in an input array. Create separate thread 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.

CPU SCHEDULING ALGORITHMS

Scheduling Criteria & Optimization:

- CPU utilization – keep the CPU as busy as possible
  - Maximize CPU utilization
- Throughput – # of processes that complete their execution per time unit
  - Maximize throughput
- Turnaround time – amount of time to execute a particular process
  - Minimize turnaround time
- Waiting time – amount of time a process has been waiting in the ready queue
  - Minimize waiting time
- Response time – time from the submission of a request until the first response is produced (response time, is the time it takes to start responding, not the time it takes to output the response)
  - Minimize response time

CPU Scheduling algorithms:

(i) First-Come First Served (FCFS) Scheduling:
The process that requests the CPU first is allocated the CPU first.

(ii) Shortest-Job-First (SJF) Scheduling:
This algorithm associates with each process the length of its next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

Two schemes:

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

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

Two schemes:

- Preemptive
- Non-preemptive

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

(iv) Round-Robin (RR) Scheduling:

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

(v) Multilevel Queue (MQ) Scheduling:

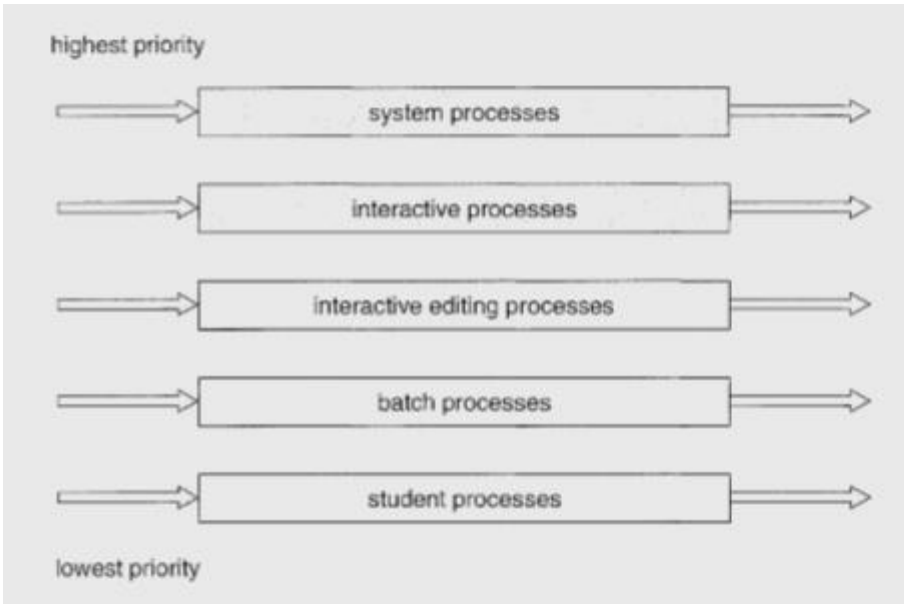


Figure 6.1: Multilevel queue scheduling

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

- foreground (interactive)
- background (batch)

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

- foreground – RR
- background – FCFS

Scheduling must be done between the queues

- Fixed priority scheduling; (i.e., serve all from foreground then from background as shown Fig. 6.1).
- Time slice – each queue gets a certain portion of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR and 20% to background in FCFS.

(vi) Multilevel Feedback Queue (MFQ) Scheduling:

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

Example of Multilevel Feedback Queue:

Consider multilevel feedback queue scheduler with three queues as shown in Fig. 6.2.

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

MFQ Scheduling

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

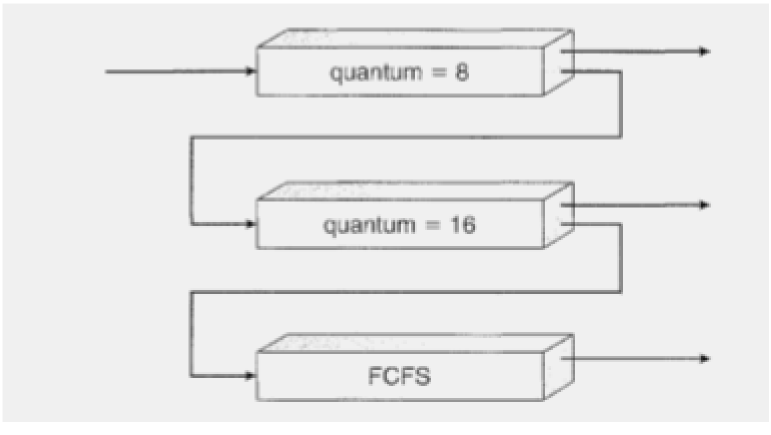


Figure 6.2: Multilevel feedback queues

Lab Exercises:

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

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

Write a C program to simulate the following CPU scheduling algorithms. Display Gantt chart showing the order of execution of each process. Compute waiting time and turnaround time for each process. Hence, compute average waiting time and average turnaround time.
(i) FCFS (ii) SRTF (iii) Round-Robin (quantum = 10)

Additional Exercises:

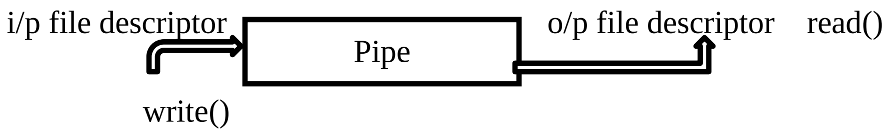
1. Write a C program to simulate the following CPU scheduling algorithms. Display Gantt chart showing the order of execution of each process. Compute waiting time and turnaround time for each process. Hence, compute average waiting time and average turnaround time.
   (i) SJF (ii) preemptive priority (iii) non-preemptive priority
2. Write a C program to simulate multi-level queue scheduling algorithm.
3. Write a C program to simulate multi-level feedback queue scheduling algorithm.

INTERPROCESS COMMUNICATION

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

Pipe

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



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

  int pipe (int *filedes) ;

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

```
PROTOTYPE: int pipe( int fd[2] );
RETURNS: 0 on success
                -1 on error: errno = EMFILE (no free descriptors)
                                EMFILE (system file table is full)
                                EFAULT (fd array is not valid)
```

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

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

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

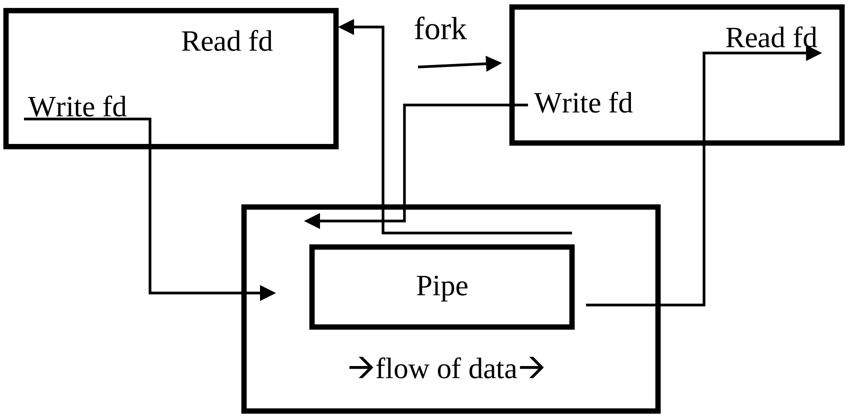


Fig. 7.1: Working of pipe in single process which is immediately after fork()

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

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

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

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

```
        write(STDOUT_FILENO, &buf, 1);
        write(STDOUT_FILENO, "\n", 1);
       close(pfd[0]);
       exit(EXIT_SUCCESS);

    } else {            /* Parent writes argv[1] to pipe */
       close(pfd[0]);         /* Close unused read end */
       write(pfd[1], argv[1], strlen(argv[1]));
       close(pfd[1]);         /* Reader will see EOF */
       wait(NULL);             /* Wait for child */
       exit(EXIT_SUCCESS);
    }
}
```

Message Queues

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

Creating a Message Queue

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

The message queue function definitions are

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

msgget

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

int msgget(key_t key, int msgflg);

The program must provide a key value that, as with other IPC facilities, names a particular message queue. The special value IPC_PRIVATE creates a private queue, which in theory is accessible only by the current process. The second parameter, msgflg, consists of nine permission flags. A special bit defined by IPC_CREAT must be bitwise ORed with the permissions to create a new message queue. It's not an error to set the IPC_CREAT flag and give the key of an existing message queue. The IPC_CREAT flag is silently ignored if the message queue already exists.

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

msgsnd

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

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

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

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

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

msgrcv
The msgrcv function retrieves messages from a message queue:

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

The first parameter, msqid, is the message queue identifier returned from a msgget function. The second parameter, msg_ptr, is a pointer to the message to be received, which must start with a long int type as described above in the msgsnd function. The third parameter, msg_sz, is the size of the message pointed to by msg_ptr, not including the long int message type. The fourth parameter, msgtype, is a long int, which allows a simple form of reception priority to be implemented. If msgtype has the value 0, the first available message in the queue is retrieved. If it's greater than zero, the first message with the same message type is retrieved. If it's less than zero, the first message that has a type the same as or less than the absolute value of msgtype is retrieved. This sounds more complicated than it actually is in practice. If you simply want to retrieve messages in the order in which they were sent, set msgtype to 0. If you want to retrieve only messages with a specific message type, set msgtype equal to that value. If you want to receive messages with a type of n or smaller, set msgtype to -n. The fifth parameter, msgflg, controls what happens when no message of the appropriate type is waiting to be received. If the IPC_NOWAIT flag in msgflg is set, the call will return immediately with a return value of –1. If the IPC_NOWAIT flag of msgflg is clear, the process will be suspended, waiting for an appropriate type of message to arrive. On success, msgrcv returns the number of bytes placed in the receive buffer, the message is copied into the user-allocated buffer pointed to by msg_ptr, and the data is deleted from the message queue. It returns –1 on error.

msgctl
The final message queue function is msgctl.

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

The msqid_ds structure has at least the following members:

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

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

Command Description

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

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

Receiver program:

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

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

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

Sender Program:

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

#define MAX_TEXT 512

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

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

```
                fprintf(stderr, “msgsnd failed\n”);
                exit(EXIT_FAILURE);
            }
            if (strncmp(buffer, “end”, 3) == 0) {
                running = 0;
            }
        }
        exit(EXIT_SUCCESS);
}
```

Shared memory

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

The functions for shared memory are,

#include <sys/shm.h>

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

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

shmget

We create shared memory using the shmget function:

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

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

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

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

shmat

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

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

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

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

shmdt

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

shmctl

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

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

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

The shmid_ds structure has the following members:

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

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

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

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

```
#define TEXT_SZ 2048

struct shared_use_st {
        int written_by_you;
```

```
	char some_text[TEXT_SZ];
};

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

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

//shm2.c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include “shm_com.h”
int main()
{
	int running = 1;
	void *shared_memory = (void *)0;
	struct shared_use_st *shared_stuff;
	char buffer[BUFSIZ];
	int shmid;
	shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);
	if (shmid == -1) {
		fprintf(stderr, “shmget failed\n”);
		exit(EXIT_FAILURE);
	}
	shared_memory = shmat(shmid, (void *)0, 0);
	if (shared_memory == (void *)-1) {
		fprintf(stderr, “shmat failed\n”);
		exit(EXIT_FAILURE);
	}
	printf(“Memory attached at %X\n”, (int)shared_memory);
	shared_stuff = (struct shared_use_st *)shared_memory;
	while(running) {
        while(shared_stuff->written_by_you == 1) {
            sleep(1);
            printf("waiting for client...\n");
        }
        printf("Enter some text:");
        fgets(buffer, BUFSIZ, stdin);
        strncpy(shared_stuff->some_text, buffer, TEXT_SZ);
        shared_stuff->written_by_you = 1;
        if (strncmp(buffer, "end", 3) == 0) {
            running = 0;
        }
    }
    if (shmdt(shared_memory) == -1) {
        fprintf(stderr, "shmdt failed\n");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

Named Pipes: FIFOs

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

```
$ mkfifo filename
```

From inside a program, we can use

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

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

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

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

We can look for the pipe with

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

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

Producer-Consumer Problem (PCP):

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

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

```
//producer.c

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
```

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

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

```
        }
        printf(“Process %d finished\n”, getpid());
        exit(EXIT_SUCCESS);
}

//consumer.c

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

The Readers-Writers Problem:

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

Two Variations

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

Lab Exercises:

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

Additional Exercises:

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

PROCESS SYNCHRONIZATION

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

Data Races

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

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

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

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

```
//race.c
#include <pthread.h>
int counter = 0;
```

```
void * func(void * params)
{
        counter++;
}
void main()
{
        pthread_t thread1, thread2;
        pthread_create(&thread1, 0, func, 0);
        pthread_create(&thread2, 0, func, 0);
        pthread_join(thread1, 0 );
        pthread_join(thread2, 0 );
}
```

Using tools to detect data races

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

```
$ gcc –g race.c -lpthread

$ valgrind –tool=helgrind ./a.out
```

Avoiding Data Races

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

Synchronization Primitives:

Mutex Locks:

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

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

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

```
#include <pthread.h>

...

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

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

```
#include <pthread.h>
#include <stdio.h>
pthread_mutex_t mutex;
volatile int counter = 0;
void * count( void * param)
```

```
{
        for ( int i=0; i<100; i++)
        {
                pthread_mutex_lock(&mutex);
                counter++;
                printf("Count = %i\n", counter);
                pthread_mutex_unlock(&mutex);
        }
}
int main()
{
        pthread_t thread1, thread2;
        pthread_mutex_init( &mutex, 0 );
        pthread_create( &thread1, 0, count, 0 );
        pthread_create( &thread2, 0, count, 0 );
        pthread_join( thread1, 0 );
        pthread_join( thread2, 0 );
        pthread_mutex_destroy( &mutex );
        return 0;
}
```

Semaphores:

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

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

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

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

```
int main()
{
        sem_t semaphore;
        sem_init( &semaphore, 0, 10 );
        ...
        sem_destroy( &semaphore );
}
```

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

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

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

sem_t semaphore;

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

Solution to Producer-Consumer problem

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

```
                printf("full %u\n",full);
                sem_wait(&mutex);
                item=buf[(++f)%5];
                printf("consumed item is %d\n",item);
                sleep(1);
                sem_post(&mutex);
                sem_post(&empty);

                int val;

               sem_getvalue(&wt,&val)

if( val <10)

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

Solution to First Readers-Writers Problem using semaphores:

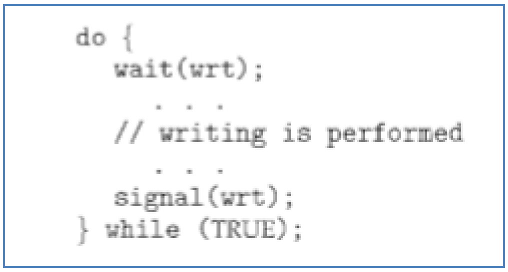
- The reader processes share the following data structures:

  ```
  semaphore mutex , wrt;
  int readcount;
  ```

- The binary semaphores mutex and wrt are initialized to 1; readcount is initialized to 0;
- Semaphore wrt is common to both reader and writer process
  - wrt functions as a mutual exclusion for the writers

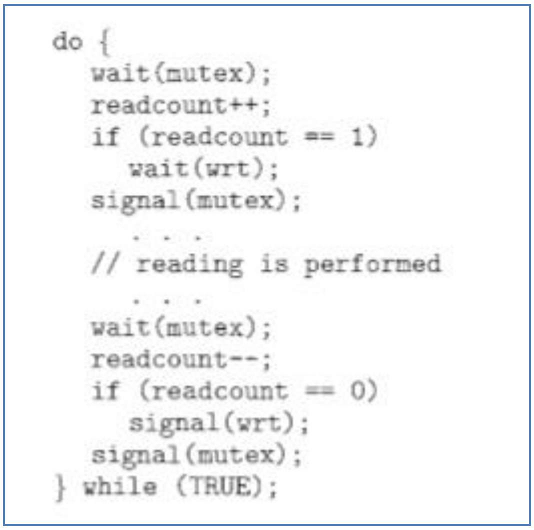
  - It is also used by the first or last reader that enters or exits the critical section
  - It is not used by readers who enter or exit while other readers are in their critical section
- The readcount variable keeps track of how many processes are currently reading the object
- The mutex semaphore is used to ensure mutual exclusion when readcount is updated

The structure of a writer process



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

The structure of a reader process



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

The Dining Philosophers Problem:

- Five philosophers sit at a round table - thinking and eating
- Each philosopher has one chopstick
  - five chopsticks total

- A philosopher needs two chopsticks to eat
    - philosophers must share chopsticks to eat
- No interaction occurs while thinking

The situation of the dining philosophers is shown in Fig. 8.1

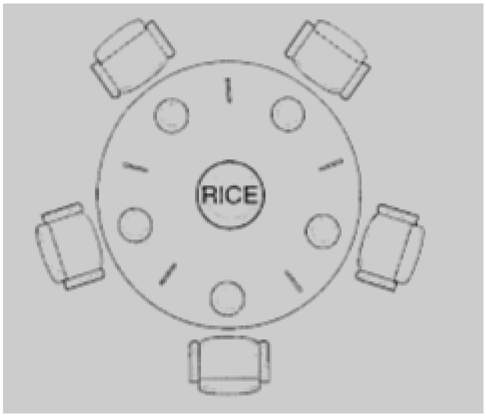


Figure 8.1

Lab Exercises:

1. Modify the above Producer-Consumer program so that, a producer can produce at the most 10 items more than what the consumer has consumed.
2. Write a C program for first readers-writers problem using semaphores.

Additional Exercises:

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

DEADLOCK ALGORITHMS

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

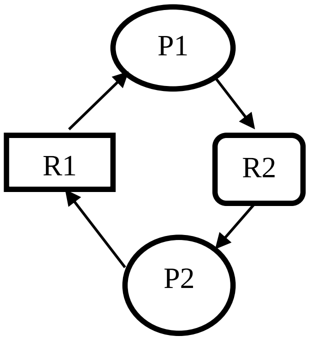


Figure 9.1: Deadlock Situation

Above Fig. 9.1 shows, a situation of deadlock where Process P1 Waiting for resource R2, which is held with process P2 and in the meantime, Process P2 is waiting for resource R1, which is held with process P1. Neither P1 nor P2 can proceed their execution until their needed resources are fulfilled forming a cyclic wait. It is the deadlock situation among processes as both are not progressed. In a single instance of resource type, a cyclic wait is always a deadlock.

Consider Figure 9.2 below, the situation with 4 processes P1, P2, P3 and P4 and 2 resources R1 and R2 both are of two instances. Here, there is no deadlock even though the cycle exists between processes P1 and P3. Once P2 finishes its job, 1 instance of resource will be available which can be accessed by process P1, which turns request edge to assignment edge, thereby removing cyclic-wait. So, in multiple instances of resource type, the cyclic-wait need not be deadlock.

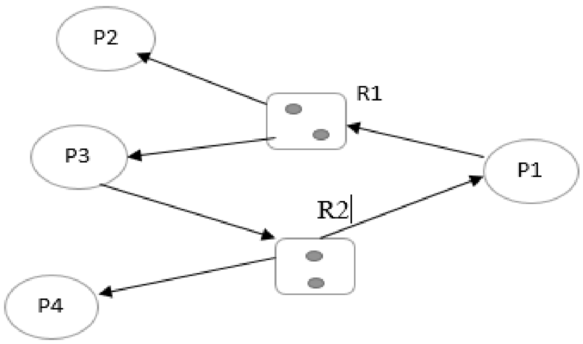


Figure 9.2: Cyclic-wait but no deadlock

Methods for Handling Deadlocks:

(i) Deadlock Avoidance:

The deadlock avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition. Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe State:

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

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

If a system is in safe state $\Rightarrow$ no deadlocks.

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

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

Banker's Algorithm:

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

Data Structures for the Banker's Algorithm:

Let n = number of processes, and m = number of resources types.

Available: Vector of length m. If available [j] = k, there are k instances of resource type R_j available.

Max: n x m matrix. If *Max* [i, j] = k, then process P_i may request at most k instances of resource type R_j.

Allocation: n x m matrix. If Allocation[i, j] = k then P_i is currently allocated k instances of R_j.

Need: n x m matrix. If *Need*[i, j] = k, then P_i may need k more instances of R_j to complete its task.

Need [i, j] = *Max*[i, j] – *Allocation* [i, j].

Safety Algorithm:

Allocation$_i$ means resources allocated to process P_i

Need$_i$ means resources needed for the process P_i

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

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

Resource-Request Algorithm:

$Request_i$ = request vector for process *P_i*.

$Request_i$ [*j*] == *k* means that process *P_i* wants *k* instances of resource type *R_j*.

1. If *$Request_i \le Need_i$* go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If *$Request_i \le Available$*, go to step 3. Otherwise *P_i* must wait, since resources are not available.
3. Pretend to allocate requested resources to *P_i* by modifying the state as follows:

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

- *If safe* $\rightarrow$ *the resources are allocated to P_i.*
- *If unsafe* $\rightarrow$ *P_i must wait for $Request_i$, and the old resource-allocation state is restored*

(ii) Deadlock Detection:

For deadlock detection, the system must provide

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

Deadlock detection algorithm:

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

Required data structures:

Available: A vector of length *m* indicates the number of available resources of each type.

Allocation: An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.

Request: An *n* x *m* matrix indicates the current request of each process. If *Request* [*i*][j] == *k*, then process *P_i* is requesting *k* more instances of resource type *R_j*.

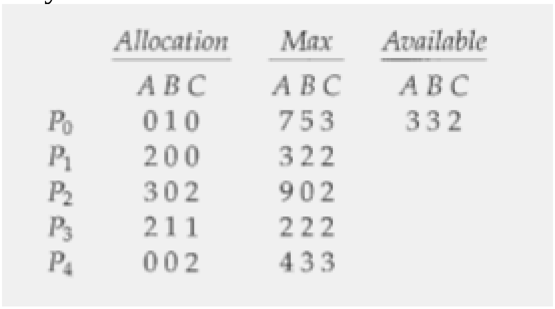
Detection Algorithm:

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

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

Lab Exercises

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



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

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

2. Consider the following snapshot of the system. Write C program to implement

deadlock detection algorithm.

(a) Is the system in a safe state?

(b) Suppose that process P2 make one additional request for instance of type C, can the system still be in a safe state?

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

Additional Exercises:

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

MEMORY MANAGEMENT

Memory allocation methods:

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

Dynamic storage allocation problem:

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

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

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

One solution to the problem of external fragmentation is compaction. The goal is to shuffle the memory contents so as to place all free memory together in one large block.

Memory fragmentation can be internal also. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is internal fragmentation – unused memory that is internal to a partition.

Paging:

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

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

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

Paging Hardware is shown in the following Figure 10.1

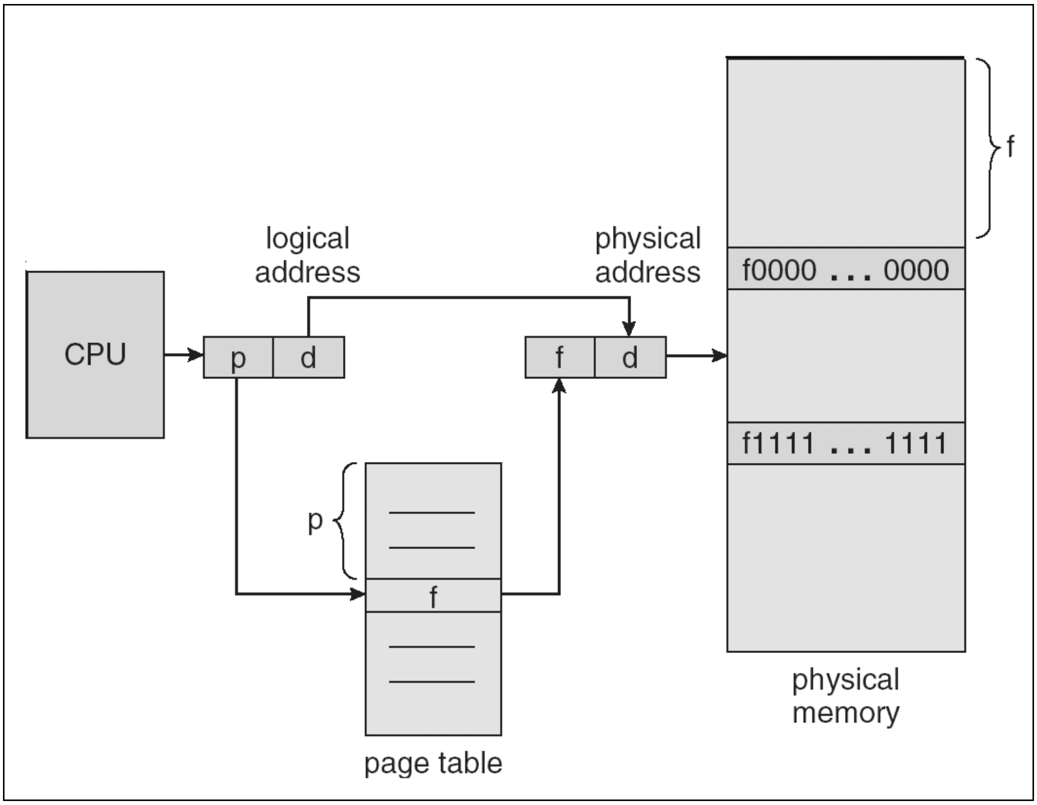


Figure 10.1: Paging Hardware

Segmentation:

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

Segmentation architecture:

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

Segmentation Hardware is shown in Fig. 10.2

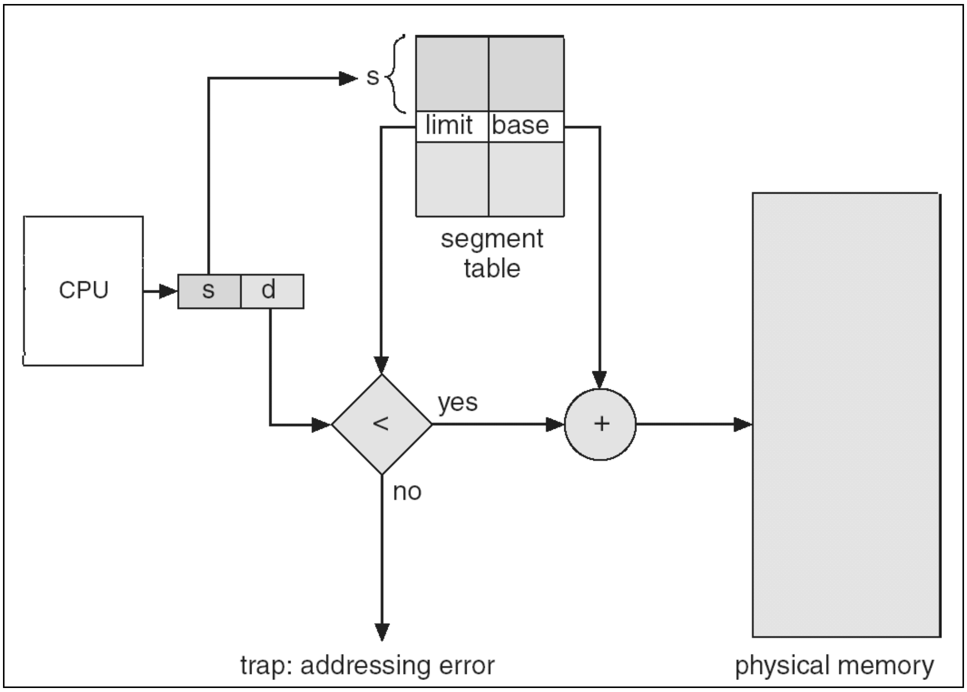


Figure 10.2: Segmentation hardware

Lab Exercises:

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

417K, 112K, and 426K (in order)? Which algorithm makes efficient use of memory?

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

Additional Exercises:

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

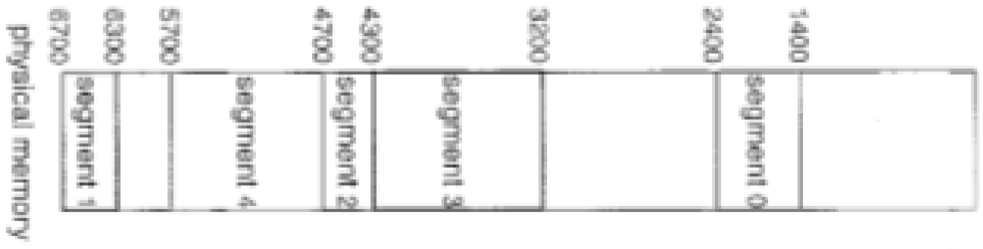


Figure 10.3: Physical memory

PAGE REPLACEMENT ALGORITHMS

Page Replacement Algorithms:

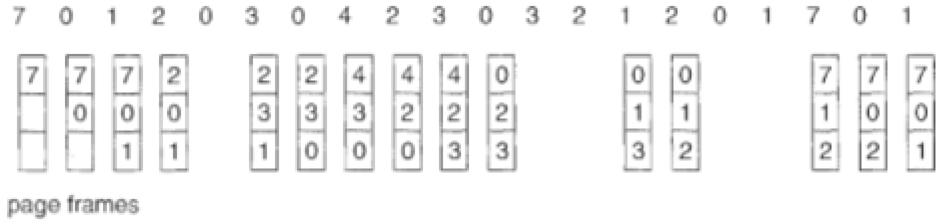
Below are some popular page replacement algorithms. Modern operating systems like Linux use some variants of these simple algorithms.

1. **FIFO (First In First Out).** The page that was brought in first will be replaced first.
2. **Optimal Page Replacement Algorithm.** Replaces the page that will not be used for the longest time in future.
3. **The LRU (least recently used)** replaces the page that is least recently used.

Example: Consider the following Page requests

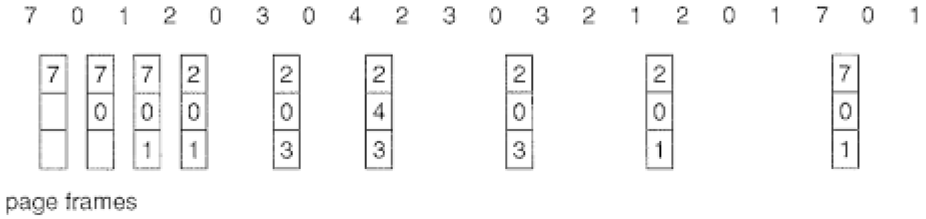
7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

(i) FIFO page replacement algorithm



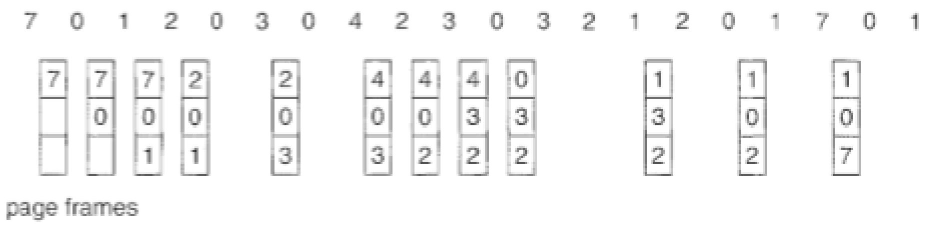
Total Page faults =15.

(ii) Optimal page replacement algorithm



Total page faults = 9.

(iii) Least recently used page replacement algorithm



Total page faults = 12.

LRU approximation page replacement algorithms:

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

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

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

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

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

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

2. Second chance algorithm (or clock algorithm):

- The basic algorithm of second-chance is a FIFO (Fig. 11.1).
- A reference bit for each frame is set to 1 whenever:
  - o a page is first loaded into the frame.
  - o the corresponding page is referenced.
- When a page has been selected for replacement, inspect its reference bit:
- If a page’s reference bit is 1
  - o set its reference bit to zero and skip it (give it a second chance)

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

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

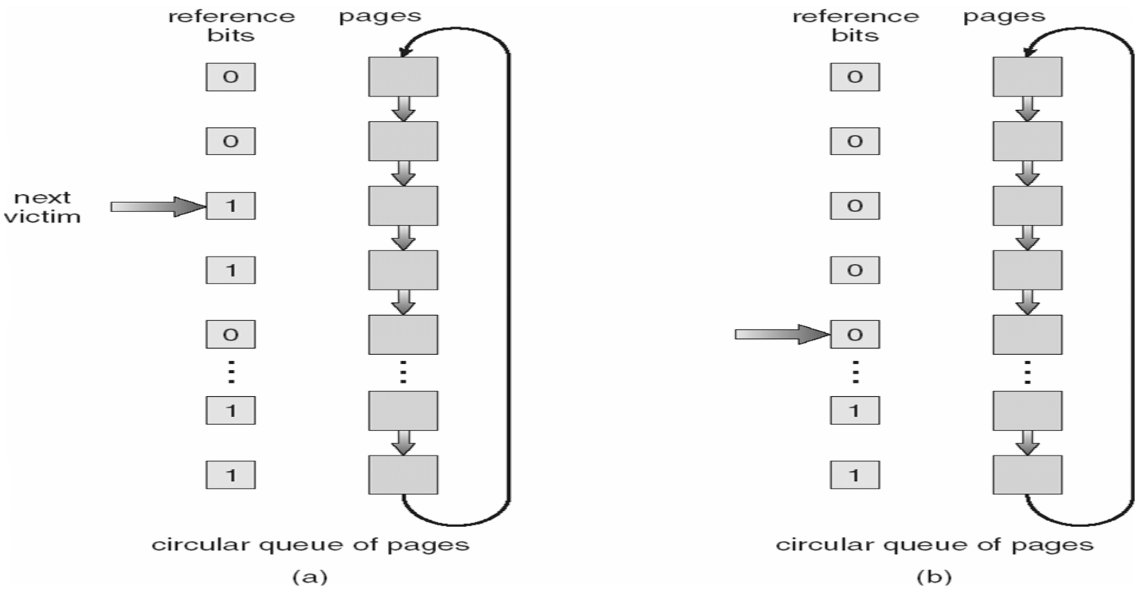


Figure 11.1: Second-chance algorithm

1. **Enhanced Second-Chance Algorithm (2-bits):**
   - Very similar to Clock Algorithm
   - Consider also the reference bits and the modified bits of pages
     - Reference (R) bit: page is referenced in the last interval
     - Modified (M) bit: page is modified after being loaded into memory
   - Four possible cases (R,M):
     - (0,0) – neither recently used nor modified (best page to replace).
     - (0,1) – not recently used but modified (not quite as good, must write out before replacement)
     - (1,0) – recently used but clean (probably will be used again).
     - (1,1) – recently used and modified (probably will be used again and need to write out before replacement)
   - Major difference between this algorithm and clock algorithm is that here we give preference to those pages that have been modified.

Counting-Based Page Replacement:

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

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

Lab Exercises:

1. Write a C program to simulate the following page replacement algorithms. Calculate and display the total number of page faults and page hits for each algorithm.

   (i) FIFO (ii) Optimal (iii) LRU
2. Write a C program to simulate LRU approximation page replacement using second chance algorithm. Find the total number of page faults and hit ratio for the algorithm.

Additional Exercises:

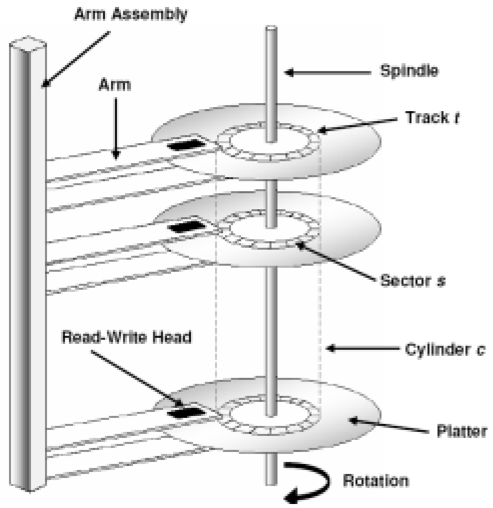
1. Write a C program to simulate the following page replacement algorithms.

   (i) LFU (ii) MFU
2. Write a C program to simulate the following LRU approximation page replacement algorithms. (a) Additional Reference byte algorithm (b) Enhanced second chance algorithm.

DISK SCHEDULING ALGORITHMS

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

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



A common disk drive has a capacity in the size of gigabytes. While the set of tracks that are at one arm position forms a cylinder, in a disk drive there may be thousands of concentric cylinders.
In a movable-head disk, where there is only one access arm to service all the disk tracks, the time spent by the Read and Write (R/W) head to move from one track to another is called seek time.

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

- FCFS-Selects request in the order of arrival

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

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

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

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

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

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

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

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

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

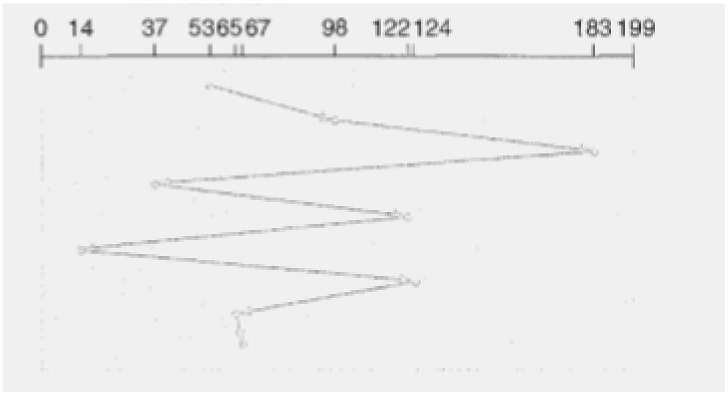


Figure 12.1: FCFS disk scheduling

Shortest Seek Time First (SSTF)

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

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

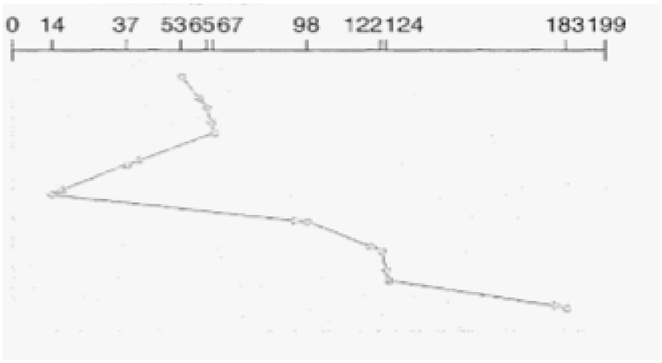


Figure 12.2: SSTF disk scheduling

SCAN Scheduling Algorithm (Elevator Algorithm)

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

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

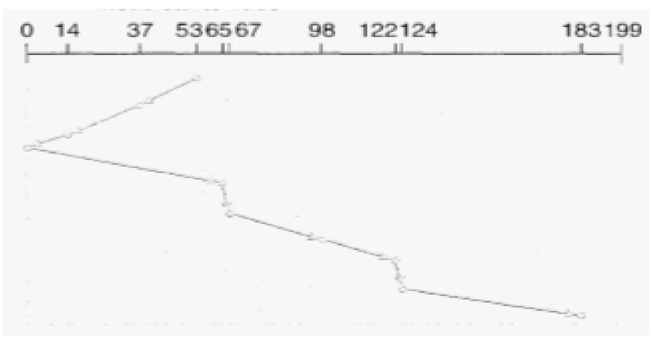


Figure 12.3: SCAN disk scheduling

Circular SCAN (C-SCAN) Algorithm

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

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

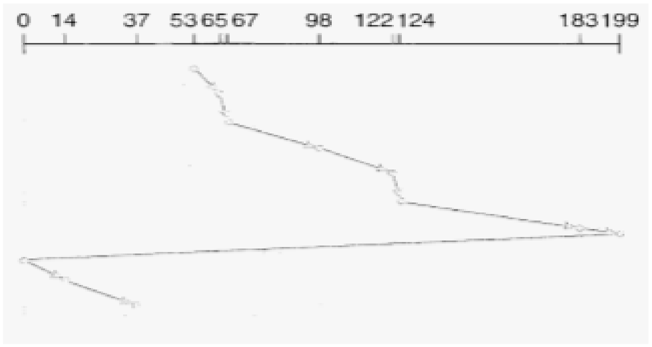


Figure 12.4: C-SCAN disk scheduling

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

LOOK Scheduling Algorithm

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

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

C-LOOK

This scheduling algorithm is Circular LOOK is like a C-SCAN which uses a return sweep before processing a set of disk requests. It does not reach the end of the tracks unless there is a request, either read or write on such disk location similar with the LOOK algorithm. Using the same sets of example in FCFS the solutions are as follows:

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

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

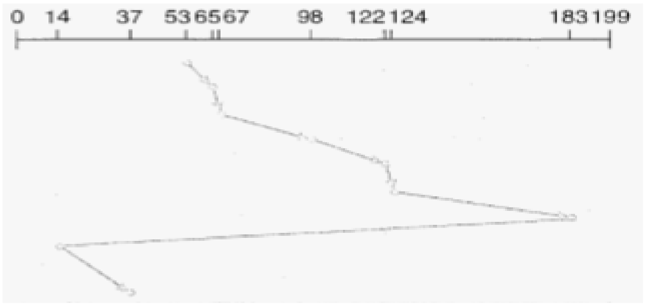


Figure 12.5: C-LOOK disk scheduling

Lab Exercises:

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

Additional Exercises:

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

REFERENCES

1. Maurice Bach, The Design of the Unix Operating System, PHI, 1986.
2. Graham Glass and King Abels, Unix for Programmers and Users – *A complete guide*, PHI, 1993.
3. Sumitabha Das, Unix Concepts and Applications, McGraw Hill, 4th Edition, 2015.
4. Neil Matthew and Richard Stones, Beginning Linux Programming, 3rd Edition, Wiley, 1999.
5. A. Silberschatz, P. B. Galvin and G. Gagne, Operating System Concepts, Wiley, 8th Edition, 2014.
6. Darryl Gove, Multicore Application Programming for Windows, Linux and Oracle Solaris, Addison Wesley, 2011.
7. W. R. Stevens, UNIX Network Programming-Volume II (IPC), PHI, 1998.