**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**

# CERTIFICATE

This is to certify that Ms. ..Fatima Khan..Reg. No.: 210101060

Section: ...A…has satisfactorily completed the lab exercises prescribed for OPERATING SYSTEMS LAB [CSE 2164] of Second Year B. Tech. Degree at MAHE, Dubai, in the academic year 2023-24.

Date: .30/11/2023.

Signature Faculty in Charge

### INDEX

|  |  |  |  |
| --- | --- | --- | --- |
| **LAB NO.** | **TITLE** | **PAGE NO.** | **REMARKS** |
|  | COURSE OBJECTIVES AND OUTCOMES |  |  |
|  | EVALUATION PLAN |  |  |
|  | INSTRUCTIONS TO THE STUDENTS |  |  |
| 1 | LINUX BASIC COMMANDS, SHELL CONCEPTS AND FILE FILTERS | 5 |  |
| 2 | SHELL SCRIPTING – 1 | 15 |  |
| 3 | SHELL SCRIPTING – 2 | 21 |  |
| 4 | LINUX SYSTEM CALLS | 26 |  |
| 5 | THREAD PROGRAMMING | 39 |  |
| 6 | INTERPROCESS COMMUNICATION | 52 |  |
| 7 | PROCESS SYNCHRONIZATION | 66 |  |
| 8 | CPU SCHEDULING ALGORITHMS | 72 |  |
| 9 | DEADLOCK ALGORITHMS | 77 |  |
| 10 | MEMORY MANAGEMENT SCHEMES | 80 |  |
| 11 | PAGE REPLACEMENT ALGORITHMS | 82 |  |
| 12 | DISK SCHEDULING ALGORITHMS |  |  |
|  | REFERENCES | 92 |  |

**Course Objectives**

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

### Course Outcomes

At the end of this course, students will

* Understand Linux commands, shell scripting using appropriate Linux system calls.
* Implement thread programming and inter process communication techniques.
* Implement the CPU scheduling, deadlock, memory management, page replacement and disk scheduling algorithms.

### Evaluation Plan

* Internal Assessment Marks : 60%
  + Continuous evaluation component (for each experiment): 10 marks
  + The assessment will depend on punctuality, program execution, maintaining the observation note and answering the questions in viva voce
  + Total marks of the 12 experiments will sum up to 60
* End semester assessment of 2 hour duration: 40 Marks

### INSTRUCTIONS TO THE STUDENTS

**Pre-Lab Session Instructions**

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

### In-Lab Session Instructions

* Follow the instructions on the allotted exercises given in Lab Manual.
* Show the program and results to the instructors on completion of experiments.
* On receiving approval from the instructor, copy the program and results in the Lab record.
* Prescribed textbooks and class notes can be kept ready for reference if required.

### General Instructions for the exercises in Lab

* The programs should meet the following criteria:
  + Programs should be interactive with appropriate prompt messages, error messages if any, and descriptive messages for outputs.
  + Programs are properly indented and comments should be given whenever it is required.
  + Use meaningful names for variables and procedures.
* Plagiarism (copying from others) is strictly prohibited and would invite severe penalty during evaluation.
* The exercises for each week are divided under three sets:
  + Solved exercise
  + Lab exercises - to be completed during lab hours
  + Additional Exercises - to be completed outside the lab or in the lab to enhance the skill.
* In case a student misses a lab class, he/she must ensure that the experiment is completed at students end or in a repetition class (if available) with the permission of the faculty concerned but credit will be given only to one day’s experiment(s).
* Questions for lab tests and examination are not necessarily limited to the questions in the manual, but may involve some variations and/or combinations of the questions.
* A sample note preparation is given later in the manual as a model for observation.

### THE STUDENTS SHOULD NOT

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

**LAB NO.: 1 Date:**

**LINUX BASIC COMMANDS, SHELL CONCEPTS AND FILE FILTERS**

### Objectives:

**In this lab, student will be able to:**

### Learn Linux basic commands

1. **Understand the working of commands and important shell concepts, file filters.**

### Write and execute basic commands in a Shell.

#### 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 system 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.

#### 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**: Unix 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 above then,

$ **who am i** [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 the content which will be placed in file with filename)

[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

will display os.txt followed with 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 function: 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

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

$ **mkdir** data [Enter]

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 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 a 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

### > 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. First, file file1 is created if not exists otherwise, its content is erased and then output of the command is written.

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.

### < 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.

### >> 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.

### << document operator

There are occasions when the data of your 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

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

1. **Pipes:** Standard input and standard output constitute two separate streams that can be individually manipulated by the shell. If so then one command can take input from the other. This is possible with the help of pipes.

Assume if the **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 to be used in pipe.

1. **Command substitution:** The shell enables the 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 2019

### $ echo “There are total `ls | wc –l ` files and sub-directory in the current directory

There are 15 files in the current directory.

1. **Sequences:** Two separate commands can be written in one line using

“;” in sequences.

### $ chmod 666 xstart; ls –l xstart

1. **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>**

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

To see last 20 lines of a file - $ **tail -20 <file name>**

1. **more:** To see the contents of a file in the form of page views - $ more

<file name>

### $ more f1.txt

1. **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**

1. **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

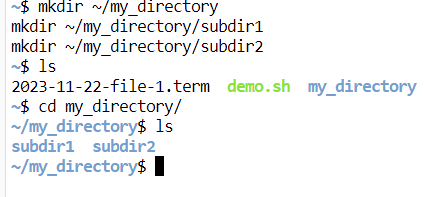
$ ls –l | grep "^-"

To display only directories

$ ls –l | grep "^d"

### Lab Exercises:

1. Write shell commands for the following.
   1. To create a directory in your home directory having 2 subdirectories.



* 1. In the first subdirectory, create 3 different files with different content in each of them.

A computer screen shot of a computer code

Description automatically generated

* 1. Copy the first file from the first subdirectory to the second subdirectory.

A computer code with black text

Description automatically generated

* 1. Create one more file in the second subdirectory, which has the output of the number of users and number of files.



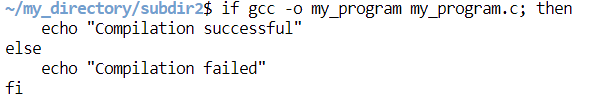
* 1. To list all the files which starts with either a or A.



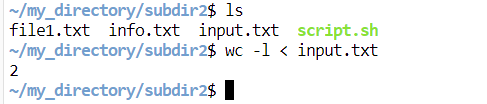
vi)To count the number of files in the current directory.



vii)Display the output if the compilation of a program succeeds.



viii)Count the number of lines in an input file.



1. Execute the following commands in sequence: i) date ii) ls iii) pwd

A screen shot of a computer

Description automatically generated

### Additional Exercises:

1. Write shell commands for the following.
   1. To Display an error message if the compilation of a program fails.
   2. To write a text block into a new file. iii.List all the files.

iv. To count the number of users logged on to the system.

**LAB NO.: 2 Date:**

**SHELL SCRIPTING – 1**

### Objectives:

**In this lab, student will be able to:**

### Understand the importance of scripts.

1. **Write and execute shell scripts.**

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. |

### Lab Exercises:

1. Try the following shell commands

$ echo $HOME, $PATH

$ echo $MAIL

$ echo $USER, $SHELL, $TERM

1. 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

1. 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).

1. **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.

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

$ **sleep** 2

### Lab Exercises:

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

### $ (sleep 10; echo done) &

$ **ps**

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

### $ (sleep 10; echo done) &

$ **kill pid** …..pid is the process id of background process

* 1. 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

Sample Program:

$ 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 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

$

Write Shell Scripts to do the following:

1. List all the files under the given input directory, whose extension has only one character

#!/bin/bash

if [ "$#" -ne 1 ]; then

echo "Usage: $0 <input\_directory>"

exit 1

fi

input\_directory="$1"

find "$input\_directory" -type f -name '?\*'



1. 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.

#!/bin/bash

if [ "$#" -ne 2 ]; then

echo "Usage: $0 <directory> <regex>"

exit 1

fi

directory="$1"

regex="$2"

find "$directory" -type f -name "$regex"

1. Count the number of users logged on to the system. Display the output as Number of users logged into the system.

#!/bin/bash

echo "Number of users logged into the system: $(who | wc -l)"

1. Count only the number of files in the current directory.

#!/bin/bash

echo "Number of files in the current directory: $(ls -l | grep ^- | wc -l)"

1. Write a shell script that takes two sorted numeric files as input and produces a single sorted numeric file without any duplicate contents.

#!/bin/bash

if [ "$#" -ne 2 ]; then

echo "Usage: $0 <file1> <file2>"

exit 1

fi

file1="$1"

file2="$2"

sort -nu "$file1" "$file2" > merged\_sorted\_file.txt

1. 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.

#!/bin/bash

if [ "$#" -ne 2 ]; then

echo "Usage: $0 <format> <destination\_directory>"

exit 1

fi

format="$1"

destination\_directory="$2"

mkdir -p "$destination\_directory"

cp \*.$format "$destination\_directory"

### Additional Exercises:

1. Write Shell Scripts to do the following
   1. To list all the .c files in any given input subdirectory. ii)Write a script to include n different commands.

**LAB NO.: 3 Date:**

**SHELL SCRIPTING – 2**

### Objectives:

**In this lab, student will be able to**

### Grasp the utility of the various variables in the Linux operating system.

1. **Understand the different arithmetic and relational operators.**

### Understand the syntax and working of the various looping and decision statements.

**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

1. **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*

1. 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.

1. **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”)

\*) esac

done

stop =1

;;

echo “illegal choice”

;;

### Example 2:

#!/bin/bash X=0

while [ $X -le 20 ] do

done

echo $X X=$((X+1))

# echo all the command line arguments while test $# != 0

do

done

echo $1

#The shift command shifts arguments to the left shift

1. **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

1. **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.

#!/bin/bash

echo "Enter a number: "

read number

if [ $((number % 2)) -eq 0 ]; then

echo "$number is even."

else

echo "$number is odd."

A black and red text

Description automatically generatedfi

* 1. Print the first ‘n’ odd numbers.

#!/bin/bash

echo "Enter the value of n: "

read n

count=0

number=1

while [ $count -lt $n ]; do

echo $number

number=$((number + 2))

count=$((count + 1))

done

A screenshot of a computer program

Description automatically generated

* 1. Find all the possible quadratic equation roots using case.

#!/bin/bash

echo "Enter the coefficients (a, b, c) of the quadratic equation ax^2 + bx + c = 0: "

read a

read b

read c

delta=$((b \* b - 4 \* a \* c))

if [ $delta -gt 0 ]; then

root1=$(echo "scale=2; (-$b + sqrt($delta)) / (2 \* $a)" | bc)

root2=$(echo "scale=2; (-$b - sqrt($delta)) / (2 \* $a)" | bc)

echo "Two distinct real roots: $root1 and $root2"

elif [ $delta -eq 0 ]; then

root=$(echo "scale=2; -$b / (2 \* $a)" | bc)

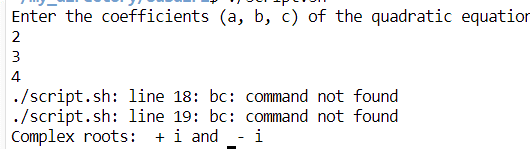
echo "One real root: $root"

else

realPart=$(echo "scale=2; -$b / (2 \* $a)" | bc)

imaginaryPart=$(echo "scale=2; sqrt($((-delta))) / (2 \* $a)" | bc)

echo "Complex roots: $realPart + i$imaginaryPart and $realPart - i$imaginaryPart"

fi

* 1. Find the factorial of a given number.

#!/bin/bash

echo "Enter a number: "

read number

factorial=1

for ((i = 1; i <= number; i++)); do

factorial=$((factorial \* i))

done

echo "Factorial of $number is $factorial."

A black and white text with numbers and dollar signs

Description automatically generated

### Additional Exercises

1. Write Shell scripts to do the following
   1. Find whether the given string is palindrome. ii)Find out the sum of the numbers given by user.

**LAB NO.: 4 Date:**

**LINUX SYSTEM CALLS**

### Objectives:

* + - **To Learn about Appropriate System Calls.**

***Fork ( ):***

The fork () system call creates a new process called a child.

The original process is called the parent, and the child is a near-exact copy of the parent. The child’s runtime is set to zero, whereas for the parent it returns the PID of child. The child has its own process ID and its own copy of the parent’s file descriptors.

**Syntax:**

**#include<unistd.h>**

**pid\_t fork (void);**

***Sleep ( ):-***

A process that voluntarily blocks for a specified time is said

to sleep. The sleep function causes the calling thread to be suspended either until the specified number of seconds has elapsed or the specified number of seconds has elapsed on until the calling thread catches a signal.

* A typical sleep system call takes a time value as a parameter, specifying the minimum amount of time that the process is to sleep before resuming execution.
* Sleep function returns zero, if the requested time has elapsed or the amount of UN slept time if interrupted.
* In sleep system call, the parameter are used are typically specifies seconds, although some operating system provide finer resolution, such as milliseconds or microseconds.

# Wait ( ):

An operating system may provide variations of the wait call that allow a process to wait for any of its children process to exit and suspend the parent process with a wait () function.

* The parameter passed to the wait () function is a pointer to an integer. The call to the wait () function result in the following things:
  + Check is made to see if the parent process has any children if it does not,-1 is returned by wait ().
  + If the parent process has a child that has terminated that child PID is return & it is removed from the process table.
  + If the parent process has child/children that have not terminated, it is suspended till it receives a signal. The signal is received as soon as child dies.
* A process can wait for its child process to finish by executing the wait system call.

Syntax:

**int wait (int \*status);**

* The value returned by wait is the process id of the child process that terminated. If the process that calls wait does not have any child process, wait returns a value of -1 immediately.
* If the process that calls wait has one or more child process that has not yet terminated, than the calling process is suspended by kernel until one of its child process terminated.
* When a child process terminates & waits returns, if the status argument is not NULL, the value passed to exit by the terminating child process is stored in status variable. Some additional information is also returned by wait.
* There are three conditions for which wait returns a process id as its return value.
  + A child process called exit.
  + A child process was terminated by a signal.
  + A child process was being traced & the process s tracing the execution of another process, such as when a debugger is being used to step through a process.

**Exec ( ) :**

* The exec family of function provides a facility for overlaying the process image of the calling with a new image.
* The various exec\*() functions take a list of arguments for the new program loaded into the process. In each case, the first of these arguments is passed to the new program s its own name rather than as an argument a user may have typed on a command

**Execl(path, arg(), argl,**

**…….) Execle(path, arg(), argl,….,env) Execlp(file, arg(), argl,…) Execlpe(file, arg(), argl, ….,env) Execv(path, args)**

**Execve(path,**

line.

* The “l” and “v” variants of the exec\*( ) functions differ in how command-line arguments are passed. The”l” variants are prepays the easiest to work with if the number of parameters is fixed when the code is written; the individual parameter simply become additional parameters to the execl\*() functions. The “v” variants are good when the number of parameters is variable, with the arguments being passed in a list as the args parameter. In either case, the argument to

the child process should start with the name of the command being run, but this is not enforced. The variants which include a “p” near the end (execlp (), execlpe (), execup () and execupe ()) will use the PATH environment variable to locate the program file. When the environment is being replaced (using one of

the exec\*e() variants), the new environment is used as the source of the PATH variable. The other variants, execl (), execv (),execle (), and execve (), will not use the PATH variable to locate the executable; path must contain an appropriate absolute or relative path. For execle (), execlpe (), execve (), and execupe (), the env parameter must be a mapping which is used to define the environment variables for the new process; the execl (), execlp (), execv (), and execup () all cause the new process to inherit the environment of the current process.

* The exec () function will initiate a program from within a program. The function return an integer error code (0 = ok, -1 = fail).
* Execl ( ) and execlp ():
* The function call “execl ()” initiate a new program in the same environment in which it is operating. An executable (with fully qualified path i.e., /bin/ls) and arguments are passed to the function.
* The routine execlp () will perform the same purpose except that it will use environment variable PATH to determine which executable to process. Thus a fully qualified path name would not have to be use. The first argument to the function could instead be “ls”. The function execlp () can also resolves explicitly.
* **Execv () and execup ():**
  + This is the same as execl () except that the argument are passed as NULL terminated array of pointer to char. The first element argv [0] is the command name.
  + The routine execup () will perform same as “routine execlp ()”.
* **Execue () :**
  + It executes a process in an environment which it assigns.
* Return value for Exec function :

If any of the exec () function returns an error will have occurred. The return value is

-1, & the global variable err no will be set to indicate the error.

**#include<unistd.h>**

**int execl (const char \*path, const char \*arg(), …/, char**

**\*const);**

**int execle (const char \*path, const char \*arg(), …/\*, char \*const enup[] \*/); int execlp (const char \*file, const char \*arg(), …char);**

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

* In Exec () function:

|  |  |
| --- | --- |
| **l** | **It works when number of**  **parameters is fixed when the code is written.** |
| **v** | **When number of**  **parameters isvariables.** |
| **e** | **Environment variables.** |
| **h** | **Path name.** |

**PROGRAM TO IMPLEMENT THE FORK FUNCTION**

#include<stdio.h> #include<sys/types.h> main()

{

int pid; pid=fork(); if(pid==0)

{

}

else

printf("\n I am the child");

printf("\n I am the parent :%d",getppid()); printf("\n I am the child :%d",getpid());

{

printf("\n I am the parent ");

printf("\n I am the parents parent :%d",getppid()); printf("\n I am the parent :%d\n",getpid());

}

}

**OUTPUT:**

cc frk.c

./a.out

I am the child

I am the parent: 3944 I am the child: 3945

I am the parent

I am the parents parent: 3211 I am the parent: 3944

PROGRAM TO IMPLEMENT EXECV FUNCTION

#include<stdio.h> #include<unistd.h>

main()

{

char

\*temp[3]; temp[0]="l s"; temp[1]="-l";

temp[2]=(char \*)0; execv("/bin/ls",temp); printf("this will not print\n");

}

**OUTPUT:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| cc execv.c  ./a.out |  |  |  |  |
| total 76  -rwxr-xr-x | 1 be322 | group | 4716 Mar 7 10:13 | a.out |
| -rw-r--r-- | 1 be322 | group | 688 Feb 20 13:52 | comm.c |
| -rw-r--r-- | 1 be322 | group | 925 Feb 20 13:54 | echomsg.c |
| -rw-r--r-- | 1 be322 | group | 722 Feb 20 13:55 | echopipe.c |
| -rw-r--r-- | 1 be322 | group | 178 Feb 20 13:57 | exel.c |
| -rw-r--r-- | 1 be322 | group | 167 Mar 7 10:13 | exev.c |
| -rw-r--r-- | 1 be322 | group | 1109 Feb 20 13:57 | fflag.c |
| -rw-r--r-- | 1 be322 | group | 341 Dec 26 14:47 | frk.c |
| -rw-r--r-- | 1 be322 | group | 140 Feb 20 13:57 | linearg.c |
| -rw-r--r-- | 1 be322 | group | 528 Feb 20 13:57 | lock.c |
| -rw-r--r-- | 1 be322 | group | 254 Feb 20 13:57 | msg.c |
| -rw-r--r-- | 1 be322 | group | 1036 Feb 20 13:57 | msgpass.c |
| -rw-r--r-- | 1 be322 | group | 203 Feb 20 13:58 | sem.c |
| -rw-r--r-- | 1 be322 | group | 1167 Feb 20 13:58 | sharememory.c |
| -rw-r--r-- | 1 be322 | group | 312 Feb 20 13:58 | slp.c |
| -rw-r--r-- | 1 be322 | group | 1182 Feb 20 13:58 | threadf.c |
| -rw-r--r-- | 1 be322 | group | 287 Feb 20 13:59 | wt.c |

PROGRAM TO IMPLEMENT EXECLP FUNCTION

#include<stdio.h> #include<sys/types.h> main()

{

int pid; pid=fork(); if(pid==0)

{

}

else

{

}

printf("\n fork program started"); execlp("/bin/ls","ls",NULL);

printf("\nend");

}

**OUTPUT:**

cc exel.c

./a.out

end$

fork program started a.out comm.c echoms

g.c echopip e.c exel.c exev.c fflag.c frk.c linearg.c lock.c msg.c msgpas s.c sem.c

sharememo ry.c slp.c thread

f.c wt.c

PROGRAMTO IMPLEMENT WAIT FUNCTION

#include<unistd.h> #include<stdio.h> main()

{

int i=0,pid; pid=fork(); if(pid==0)

{

}

else

{

}

}

printf("child process started\n"); for(i=0;i<10;i++)

printf("\n%d",i);

printf("\n child process ends");

printf("\n parent process starts"); wait(0);

printf("\n parent process ends");

**OUTPUT:**

cc wt.c

./a.out

parent process starts child process started

0

1

2

3

4

5

6

7

8

9

child process ends parent process ends

PROGRAM TO IMPLEMENT SLEEP FUNCTION

#include<unistd.h> #include<stdio.h> main()

{

int i=0,pid;

printf("\n ready for fork\n"); pid=fork();

if(pid==0)

{

}

else

{

}

}

printf("\n child process started \n"); sleep(4);

for(i=0;i<10;i++) printf("\n%d",i);

printf("\n child process ends");

printf("\n I am the parent"); printf("\n parent process ends");

**OUTPUT:**

cc slp.c

./a.out

ready for fork I am the parent

parent process ends child process started

0

1

2

3

4

5

6

7

8

9

child process ends

**LAB NO.: 5 Date:**

**THREAD PROGRAMMING**

### Objectives:

* + **To implement Threading and Synchronization Applications. ALGORITHM:**

Step 1: Start the process

Step 2: Declare process thread, thread-id.

Step 3: Read the process thread and thread state.

Step 4: Check the process thread equals to thread-id by using if condition. Step 5: Check the error state of the thread.

Step 6: Display the completed thread process. Step 7: Stop the process

#### PROGRAM:

#include<stdio.h> #include<string.h> #include<pthread.h> #include<stdlib.h> #include<unistd.h> pthread\_t tid[2];

void\* doSomeThing(void \*arg)

{

unsigned long i = 0; pthread\_t id = pthread\_self();

if(pthread\_equal(id,tid[0]))

{

printf("\n First thread processing\n");

}

else

{

printf("\n Second thread processing\n");

}

for(i=0; i<(0xFFFFFFFF);i++);

return NULL;

}

int main(void)

{

int i = 0;

int err;

while(i < 2)

{

err = pthread\_create(&(tid[i]), NULL, &doSomeThing, NULL); if (err != 0)

printf("\ncan't create thread :[%s]", strerror(err)); else

printf("\n Thread created successfully\n");

i++;

}

sleep(5); return 0;

}

### OUTPUT

Thread created successfully Thread created successfully

First thread processing Second thread processing

**LAB No.: 6 Date:**

**INTERPROCESS COMMUNICATION**

### Objectives:

In this lab, student will be able to:

1. Implement IPC using Shared Memory.

1. Start the program.
2. Declare the variables.
3. Read the choice.
4. Create a piping processing using IPC.
5. Assign the variable lengths
6. “strcpy” the message lengths.
7. To join the operation using IPC .
8. Stop the program

#### PROGRAM: ( PIPE PROCESSING)

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

#define MSG\_LEN 64 int main()

{

int result; int fd[2];

char message[MSG\_LEN]; char recvd\_msg[MSG\_LE]; result = pipe (fd);

//Creating a pipe//fd[0] is for reading and fd[1] is for writing if (result < 0)

{

perror("pipe "); exit(1);

}

strncpy(message,"Linux World!! ",MSG\_LEN); result=write(fd[1],message,strlen(message));

if (result< 0)

{

perror("write"); exit(2);

}

strncpy(message,"Understanding ",MSG\_LEN); result=write(fd[1],message,strlen(message)); if (result < 0)

{

perror("write"); exit(2);

}

strncpy(message,"Concepts of ",MSG\_LEN); result=write(fd[1],message,strlen(message)); if (result < 0)

{

perror("write"); exit(2);

}

strncpy(message,"Piping ", MSG\_LEN); result=write(fd[1],message,strlen(message)); if (result < 0)

{

perror("write"); exit(2);

}

result=read(fd[0],recvd\_msg,MSG\_LEN); if (result < 0)

{

perror("read"); exit(3);

}

printf("%s\n",recvd\_msg); return 0;

}

* 1. ***FIFO***

### Program:

#include

<stdio.h> #include

<stdlib.h> #include

<sys/stat.h> #include

<unistd.h>

#include <linux/stat.h> #define FIFO\_FILE "MYFIFO"

int main(void)

{

FILE \*fp;

char readbuf[80];

/\* Create the FIFO if it does not exist \*/ umask(0);

mknod(FIFO\_FILE, S\_IFIFO|0666, 0);

while(1)

{

fp = fopen(FIFO\_FILE, "r"); fgets(readbuf, 80, fp);

printf("Received string: %s\n", readbuf); fclose(fp);

}

return(0);

}

#include

<stdio.h> #include

<stdlib.h>

#define FIFO\_FILE "MYFIFO"

int main(int argc, char \*argv[])

{

FILE \*fp;

if ( argc != 2 ) { printf("USAGE: fifoclient [string]\n"); exit(1);

}

if((fp = fopen(FIFO\_FILE, "w")) == NULL)

{ perror("fopen"); exit(1);

}

fputs(argv[1], fp);

fclose(fp); return(0);

}

C Program for Message Queue (Writer Process)

#include

<stdio.h> #include

<sys/ipc.h>

#include

<sys/msg.h>

// structure for message queue struct mesg\_buffer

{

long msg\_type; char msg\_text[100];

} message;

int main()

{

key\_t key; int msgid;

// ftok to generate unique key key = ftok("progfile", 65);

// msgget creates a message queue

// and returns identifier msgid = msgget(key, 0666 |

IPC\_CREAT); message.mesg\_type

= 1;

printf("Write Data :

");

gets(message.mesg\_ text);

// msgsnd to send message

msgsnd(msgid, &message, sizeof(message), 0);

// display the message

printf("Data send is : %s \n", message.mesg\_text);

return 0;

}

C Program for Message Queue (Reader Process) #include <stdio.h>

#include

<sys/ipc.h> #include

<sys/msg.h>

// structure for message queue struct mesg\_buffer

{

long mesg\_type;

char mesg\_text[100];

} message;

int main()

{

key\_t key; int msgid;

// ftok to generate unique key key = ftok("progfile", 65);

// msgget creates a message queue and returns identifier msgid = msgget(key, 0666

| IPC\_CREAT);

// msgrcv to receive message

msgrcv(msgid, &message, sizeof(message), 1, 0);

// display the message printf("Data Received is :

%s \n",

message.mesg\_text);

// to destroy the message queue msgctl(msgid, IPC\_RMID, NULL);

return 0;

}

C Program for Message Queue (Reader Process)

#include

<stdio.h> #include

<sys/ipc.h> #include

<sys/msg.h>

// structure for message

queue struct mesg\_buffer

{

long mesg\_type; char mesg\_text[100];

} message;

int main()

{

key\_t key; int msgid;

// ftok to generate unique key key = ftok("progfile", 65);

// msgget creates a message queue

// and returns identifier

msgid = msgget(key, 0666 | IPC\_CREAT);

// msgrcv to receive message

msgrcv(msgid, &message, sizeof(message), 1, 0);

// display the message printf("Data Received is :

%s \n",

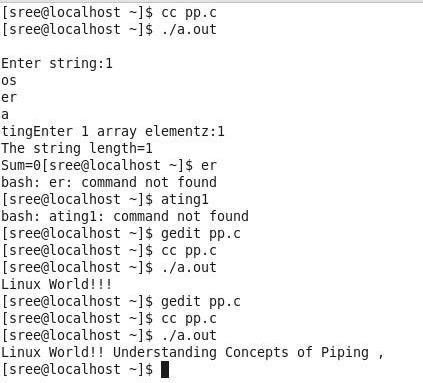
message.mesg\_text);

// to destroy the message queue msgctl(msgid, IPC\_RMID, NULL);

return 0;

}

### OUTPUT:



**LAB No.: 7 Date:**

### PROCESS SYNCHRONIZATION

**Objectives:**

In this lab, student will be able to:

Implement the Producer – Consumer problem using semaphores

### Algorithm:

* + 1. The Semaphore mutex, full & empty are initialized.
    2. In the case of producer process
    3. Produce an item in to temporary variable.

If there is empty space in the buffer check the mutex value for enter into the critical section. If the mutex value is 0, allow the producer to add value in the temporary variable to the buffer.

* + 1. In the case of consumer process
       1. It should wait if the buffer is empty
       2. If there is any item in the buffer check for mutex value, if the mutex==0, remove item from buffer
       3. Signal the mutex value and reduce the empty value by 1.
       4. Consume the item.
    2. Print the result

### Program:

#include<stdio. h> #include<stdlib. h>

int mutex = 1, full = 0, empty = 3, x = 0; int main ()

{

int n;

void producer (); void consumer (); int wait (int); int signal (int); printf

("\n1.Producer\n2.Consumer\n3.Exit "); while (1)

{

printf ("\nEnter your choice:"); scanf ("%d", &n);

switch (n)

{

case 1:

if ((mutex == 1) && (empty

!= 0)) producer (); else

printf ("Buffer is full!!"); break; case 2:

if ((mutex == 1) && (full

!= 0)) consumer (); else

printf ("Buffer is empty!!"); break; case 3:

exit (0);

break;

}

}

return 0;

}

int wait (int s)

{

return (--s);

}

int signal (int s)

{

return (++s);

}

void producer ()

{

mutex = wait (mutex); full = signal (full); empty

= wait (empty); x++;

printf ("\nProducer produces the item

%d", x); mutex = signal (mutex);

}

void consumer ()

{

mutex = wait (mutex); full =

wait (full);

empty = signal (empty);

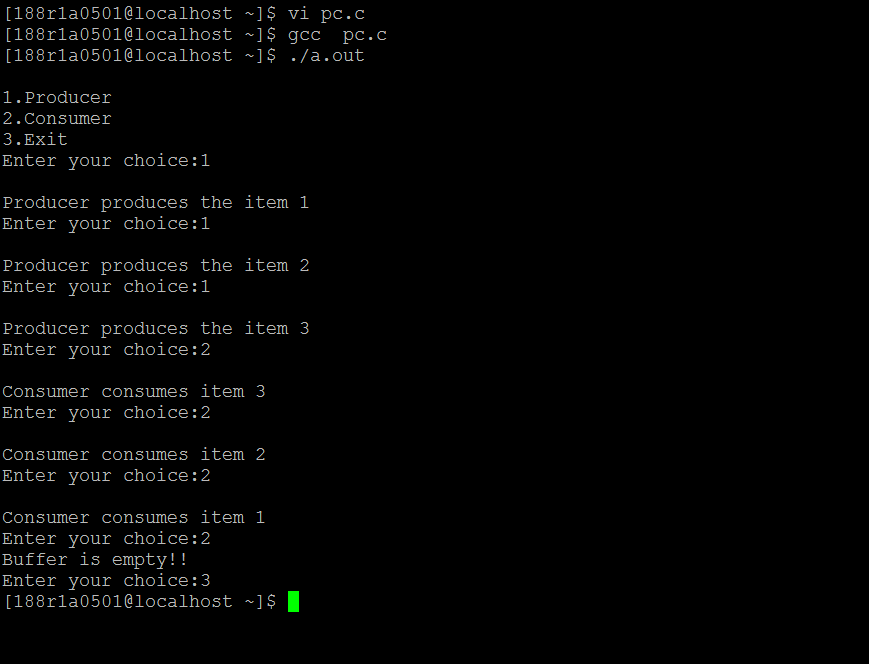
printf ("\nConsumer consumes item

%d", x); x--;

mutex = signal (mutex);

}

***Output:***



### LAB No.: 8 Date:

**CPU SCHEDULING ALGORITHMS**

### Objectives:

In this lab, student will be able to:

Implement the various process scheduling mechanisms such as FCFS scheduling, SJF, Round Robin, Priority.

#### FCFS (First Come First Serve)

***Algorithm:***

1: Start the process

2: Accept the number of processes in the ready Queue

3: For each process in the ready Q, assign the process id and accept the CPU burst time 4: Set the waiting of the first process as ‘0’ and its burst time as its turn around time

5: for each process in the Ready Q calculate

* 1. Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)
  2. Turnaround time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

6: Calculate

1. Average waiting time = Total waiting Time / Number of process
2. Average Turnaround time = Total Turnaround Time / Number of process

7: Stop the process

#### Program:

#include<stdio.h> int main()

{

int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,p os,temp; float avg\_wt,avg\_tat; printf("Enter number of

process:"); scanf("%d",&n); printf("\nEnter Burst Time:\n"); for(i=0;i<n;i++)

{

printf("p % d:",i+1);

scanf("%d",&bt[i]);

p[i]=i+1; //contains process number

}

wt[0]=0; //waiting time for first process will be zero

//calculate waiting time for(i=1;i<n;i++)

{

wt[i]=0; for(j=0;j<i;j

++)

wt[i]+=bt[j];

total+=wt[i];

}

avg\_wt=(float)total/n; //average waiting time total=0;

printf("\nProcess\t Burst Time \tWaiting Time\tTurnaround Time"); for(i=0;i<n;i++)

{

tat[i]=bt[i]+wt[i]; //calculate turnaround time total+=tat[i];

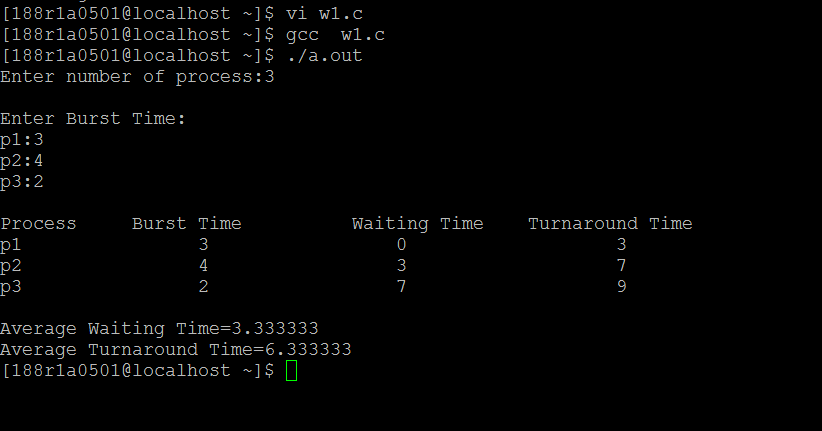
printf("\np%d\t\t %d\t\t %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);

}

avg\_tat=(float)total/n; //average turnaround time printf("\n\nAverage Waiting Time=%f",avg\_wt); printf("\nAverage Turnaround Time=%f\n",avg\_tat);

}

***Output:***



### SJF (Shortest Job First)

#### Algorithm:

1: Start the process

2: Accept the number of processes in the ready Queue

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

4: Start the Ready Q according the shortest Burst time by sorting according to lowest to highest burst time.

5: Set the waiting time of the first process as ‘0’ and its turnaround

time as its burst time.

6: For each process in the ready queue, calculate

1. Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)
2. Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

7: Calculate

1. Average waiting time = Total waiting Time / Number of process

* Average Turnaround time = Total Turnaround Time / Number of process
* 8: Stop the process

#### Program:

#include<stdio. h> int main()

{

int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,p os,temp; float avg\_wt,avg\_tat; printf("Enter number of

process:"); scanf("%d",&n);

printf("\nEnter Burst Time:\n"); for(i=0;i<n;i++)

{

printf("p%d:",i+1);

scanf("%d",&bt[i]);

p[i]=i+1; //contains process number

}

//sorting burst time in ascending order using selection sort for(i=0;i<n;i++)

{

pos=i; for(j=i+1;j<n;j

++)

{

if(bt[j]<bt[po

s]) pos=j;

}

temp=bt[i]; bt[i]=bt[pos

];

bt[pos]=tem p;

temp=p[i]; p[i]=p[pos]

;

p[pos]=tem p;

}

wt[0]=0; //waiting time for first process will be zero

//calculate waiting time for(i=1;i<n;i++)

{

wt[i]=0;

for(j=0;j<i;j

++)

wt[i]+=bt[j

];

total+=wt[i];

}

avg\_wt=(float)total/n; //average waiting time total=0;

printf("\nProcess\t Burst Time \tWaiting Time\tTurnaround Time"); for(i=0;i<n;i++)

{

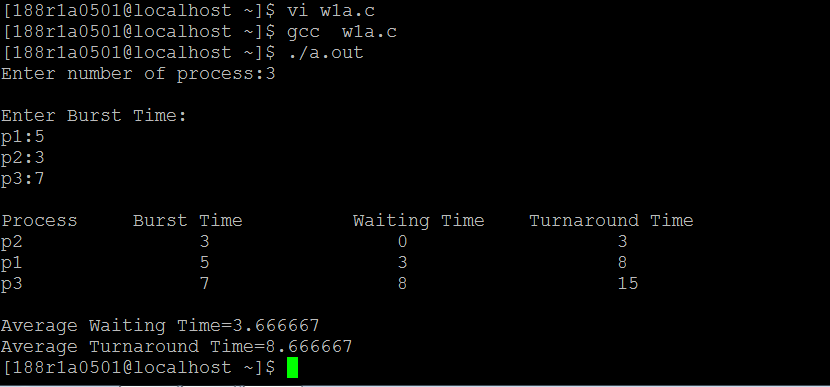
tat[i]=bt[i]+wt[i]; //calculate turnaround time total+=tat[i];

printf("\np%d\t\t %d\t\t %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);

}

avg\_tat=(float)total/n; //average turnaround time printf("\n\nAverage Waiting Time=%f",avg\_wt); printf("\nAverage Turnaround Time=%f\n",avg\_tat);

}

***Output:***

### Round Robin Algorithm

1: Start the process

2: Accept the number of processes in the ready Queue and time quantum (or) time slice 3: For each process in the ready Q, assign the process id and accept the CPU burst time 4: Calculate the no. of time slices for each process where

No. of time slice for process(n) = burst time process(n)/time slice 5: If the burst time is less than the time slice then the no.

of time slices =1. 6: Consider the ready queue is a circular Q, calculate

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

7: Calculate

1. Average waiting time = Total waiting Time / Number of process
2. Average Turnaround time = Total Turnaround Time / Number of process Step 8: Stop the process

**Program:** #include<stdio. h> main()

{

int st[10],bt[10],wt[10],tat[10],n, tq; int i,count=0,swt=0,stat=0,temp, sq=0; float awt,atat; printf("enter the number of processes"); scanf("%d",&n);

printf("enter the burst time of each process /n"); for(i=0;i<n;i++)

{

printf(("p%d",i

+1);

scanf("%d",&bt

[i]);

st[i]=bt[i];

}

printf("enter the time quantum"); scanf("%d",&tq); while(1)

{

for(i=0,count=0;i<n;i++)

{

temp=tq; if(st[i]== 0)

{

count++

;

continue

;

}

if(st[i]>tq)

st[i]=st[i]- tq;

else if(st[i]>=

0)

{

temp=st[i

]; st[i]=0;

}

sq=sq+tem p; tat[i]=sq;

}

if(n==coun t) break;

}

for(i=0;i<n;i++)

{

wt[i]=tat[i]- bt[i]; swt=swt+wt[i]

;

stat=stat+tat[i

];

}

awt=(float)swt

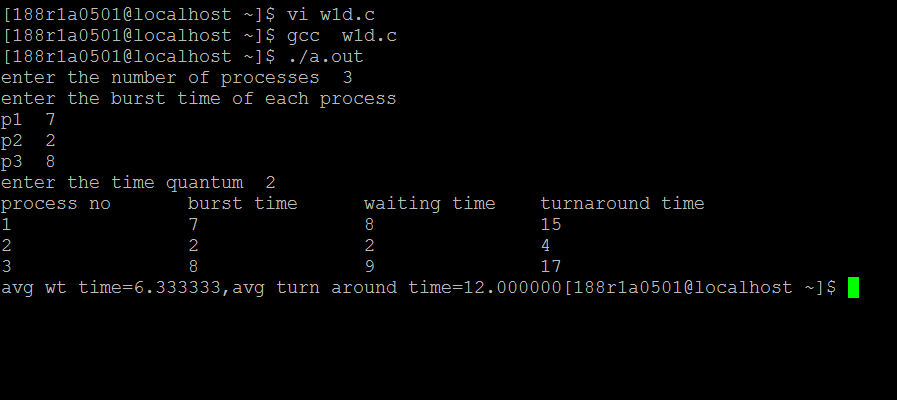
/n; atat=(float)sta t/n;

printf("process no\t burst time\t waiting time\t turnaround time\n"); for(i=0;i<n;i++) printf("%d\t\t %d\t\t %d\t\t

%d\n",i+1,bt[i],wt[i],tat[i]); printf("avg wt time=%f,avg turn around time=%f",awt,atat);

}

**Output:**



1. **Priority**

### Algorithm:

1: Start the process

2: Accept the number of processes in the ready Queue

3: For each process in the ready Q, assign the process id and accept the CPU burst time 4: Sort the ready queue according to the priority number.

5: Set the waiting of the first process as ‘0’ and its burst time as its

turn around time 6: For each process in the Ready Q calculate

1. Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)
2. Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n) 7: Calculate
3. Average waiting time = Total waiting Time / Number of process
4. Average Turnaround time = Total Turnaround Time / Number of process Step

8: Stop the process

### Program:

#include<stdio. h> int main()

{

int bt[20],p[20],wt[20],tat[20],pri[20],i,j,k,n,total=0

,pos,temp; float avg\_wt,avg\_tat; printf("Enter number of process:"); scanf("%d",&n);

printf("\nEnter Burst Time:\n"); for(i=0;i<n;i++)

{

printf("p%d:",i+1);

scanf("%d",&bt[i]);

p[i]=i+1; //contains process number

}

printf(" enter priority of the process "); for(i=0;i<n;i++)

{

p[i] = i;

//printf("Priority of Process"); printf("p%d ",i+1);

scanf("%d",&pri[i]);

}

for(i=0;i<n;i++) for(k=i+1;k<n;k

++) if(pri[i] >

pri[k])

{

temp=p[i]

;

p[i]=p[k]; p[k]=tem p;

temp=bt[i]; bt[i]=bt[k]; bt[k]=temp; temp=pri[i]; pri[i]=pri[k]; pri[k]=temp;

}

wt[0]=0; //waiting time for first process will be zero

//calculate waiting time for(i=1;i<n;i++)

{

wt[i]=0; for(j=0;j<i;j

++)

wt[i]+=bt[j];

total+=wt[i];

}

avg\_wt=(float)total/n; //average waiting time total=0;

printf("\nProcess\t Burst Time \tPriority \tWaiting Time\tTurnaround Time"); for(i=0;i<n;i++)

{

tat[i]=bt[i]+wt[i]; //calculate turnaround time total+=tat[i];

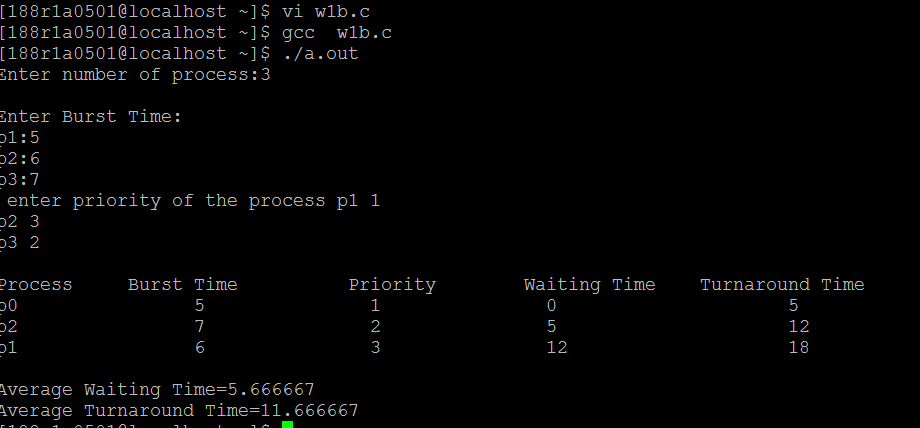
printf("\np%d\t\t %d\t\t %d\t\t %d\t\t\t%d",p[i],bt[i],pri[i],wt[i],tat[i]);

}

avg\_tat=(float)total/n; //average turnaround time printf("\n\nAverage Waiting Time=%f",avg\_wt); printf("\nAverage Turnaround Time=%f\n",avg\_tat);

}

### Output:



**LAB NO.: 9 Date:**

**Deadlock Algorithms**

**Objectives:**

### In this lab, student will be able to:

* **Simulate the Bankers Algorithm for Deadlock Avoidance.**

### Simulate Bankers Algorithm for Deadlock Prevention

**Data structures**

* 1. n- Number of process, m-number of resource types.
  2. Available: Available[j]=k, k – instance of resource type Rj is available.
  3. Max: If max [i, j]=k, Pi may request at most k instances resource Rj.
  4. Allocation: If Allocation [i, j]=k, Pi allocated to k instances of resource Rj
  5. Need: If Need[I, j]=k, Pi may need k more instances of resource type Rj,
  6. Need [I, j] =Max [I, j]-Allocation [I, j];

### Safety Algorithm

1. Work and Finish be the vector of length m and n respectively, Work=Available and Finish[i] =False.
2. Find an i such that both
3. Finish[i] =False
4. Need<=Work
5. If no such I exist go to step 4.
6. work=work+Allocation, Finish[i] =True;
7. If Finish [1] =True for all I, then the system is in safe state.

### Resource request algorithm

1. Let Request i be request vector for the process Pi, If request i=[j]=k, then process Pi wants k instances of resource type Rj.
2. If Request<=Need I go to step 2. Otherwise raise an error condition.
3. If Request<=Available go to step 3. Otherwise Pi must since the resources are available.
4. Have the system pretend to have allocated the requested resources to process Pi bymodifying the state as follows;
5. Available=Available-Request I;
6. Allocation I =Allocation+Request I;
7. Need i=Need i-Request I;

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

### Algorithm:

1. Start the program.
2. Get the values of resources and processes.
3. Get the avail value.
4. After allocation find the need value.
5. Check whether it is possible to allocate.
6. If it is possible then the system is in safe state.
7. Else system is not in safety state.
8. If the new request comes then check that the system is in safety.
9. Or not if we allow the request.
10. Stop the program.

### Program:

#include<stdio.h>

int main ()

{

int allocated[15][15], max[15][15], need[15][15],

avail[15], tres[15], work[15], flag[15]; int pno, rno, i, j, prc, count,

t, total; count = 0;

//clrscr ();

printf ("\n Enter number of process:"); scanf ("%d", &pno); printf ("\n Enter number of resources:"); scanf ("%d", &rno);

for (i = 1; i <= pno; i++)

{

flag[i] = 0;

}

printf ("\n Enter total numbers of each resources:"); for (i = 1; i <= rno; i++) scanf ("%d", &tres[i]);

printf ("\n Enter Max resources for each process:"); for (i = 1; i <= pno; i++)

{

printf ("\n for process

%d:", i); for (j = 1; j <= rno; j++)

scanf ("%d", &max[i][j]);

}

printf ("\n Enter allocated resources for each process:"); for (i = 1; i <= pno; i++)

{

printf ("\n for process

%d:", i); for (j = 1; j <= rno; j++)

scanf ("%d", &allocated[i][j]);

}

printf ("\n available resources:\n"); for (j = 1; j

<= rno; j++)

{

avail[j] = 0;

total = 0;

for (i = 1; i <= pno; i++)

{

total += allocated[i][j];

}

avail[j] = tres[j] - total; work[j] = avail[j];

printf (" %d \t", work[j]);

}

do

{

for (i = 1; i <= pno; i++)

{

for (j = 1; j <= rno; j++)

{

need[i][j] = max[i][j] - allocated[i][j];

}

}

printf ("\n Allocated matrix Max

need"); for (i

= 1; i <= pno; i++)

{

printf ("\n");

for (j = 1; j <= rno; j++)

{

printf ("%4d", allocated[i][j]);

}

printf ("|");

for (j = 1; j <= rno; j++)

{

printf ("%4d", max[i][j]);

}

printf ("|");

for (j = 1; j <= rno; j++)

{

printf ("%4d", need[i][j]);

}

}

prc = 0;

for (i = 1; i <= pno; i++)

{

if (flag[i] == 0)

{

prc = i;

for (j = 1; j <= rno; j++)

{

if (work[j] < need[i][j])

{

prc = 0;

break;

}

}

}

if (prc != 0)

break;

}

if (prc != 0)

{

printf ("\n Process %d completed", i); count++; printf ("\n Available matrix:"); for (j = 1; j

<= rno; j++)

{

work[j] += allocated[prc][j]; allocated[prc][j] = 0;

max[prc][j] = 0;

flag[prc] = 1;

printf (" %d", work[j]);

}

}

}

while (count != pno && prc != 0);

if (count == pno)

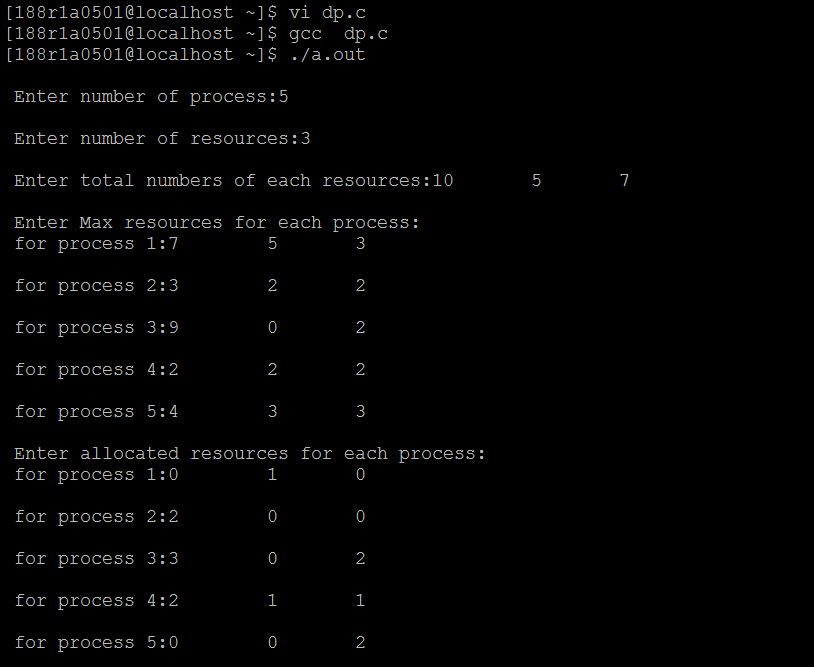
printf ("\nThe system is in a safe

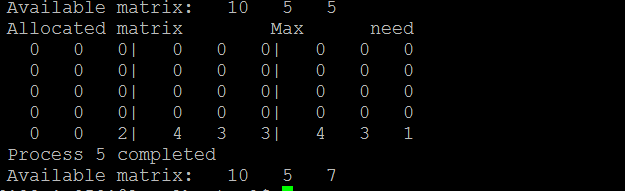
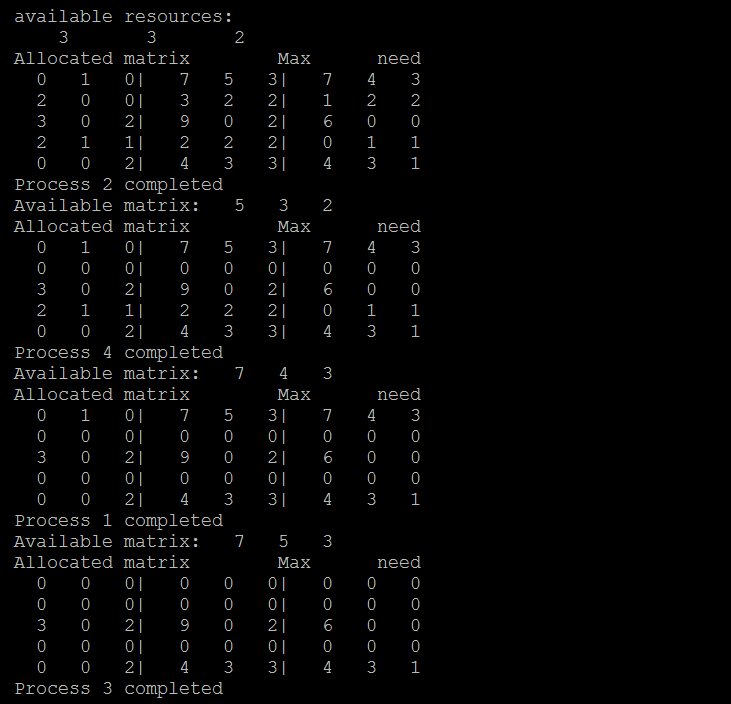
state!!"); else

printf ("\nThe system is in an unsafe state!!"); return 0;

}

### Output:





**Algorithm:**

1. Start
2. Attacking Mutex condition : never grant exclusive access. but this may not be possible for several resources.
3. Attacking preemption: not something you want to do.
4. Attacking hold and wait condition : make a process hold at the most 1 resource at a time.make all the requests at the beginning. All or nothing policy. If you feel,retry. eg. 2- phase locking 34
5. Attacking circular wait: Order all the resources. Make sure that the requests are issued in the correct order so that there are no cycles

present in the resource graph. Resources numbered 1 ... n. Resources can be requested only in increasing order. ie. you cannot request a resource whose no is less than any you may be holding.

1. Stop

### Program:

#include<stdio.h>

int max[10][10],alloc[10][10],need[10][10],avail[10],i,j,p,r,finish[10]=

{0},flag=0; main( )

{

printf("\n SIMULATION OF DEADLOCK PREVENTION \n ");

printf("Enter no. of processes, resources\n "); scanf("%d%d",&p,&r); printf("Enter allocation matrix"); for(i=0;i<p;i++) for(j=0;j<r;j++) scanf("%d",&alloc[i][j]); printf("\n enter max matrix");

for(i=0;i<p;i++) /\*reading the maximum matrix and availale matrix\*/ for(j=0;j<r;j++) scanf("%d",&max[i][j]);

printf(" \n enter available matrix"); for(i=0;i<r;i++) scanf("%d",&avail[i]);

for(i=0;i<p;i++) for(j=0;j<r;j++) need[i][j]=max[i][j]- alloc[i][j]; fun(); /\*calling function\*/ if(flag==0)

{if(finish[i]!=1)

{

printf("\n Failing :Mutual exclusion"); for(j=0;j<r;j++)

{ /\*checking for mutual exclusion\*/ if(avail[j]<need[i][j])

avail[j]=need[i][j];

}fun();

printf("\n By allocating required resources to process %d dead lock is prevented ",i); printf("\n lack of preemption");

for(j=0;j<r;j++)

{

if(avail[j]<need[i][j])

avail[j]=need[i][j];

alloc[i][j]=0;

}

fun( );

printf("\n dead lock is prevented by allocating needed resources");

printf(" \n failing:Hold and Wait condition "); for(j=0;j<r;j++)

{ /\*checking hold and wait condition\*/ if(avail[j]<need[i][j])

avail[j]=need[i][j];

}

fun( );

printf("\n AVOIDING ANY ONE OF THE CONDITION, U CAN PREVENT DEADLOCK");

}

}

}

fun()

{

while(1)

{

for(flag=0,i=0;i<p;i++)

{

if(finish[i]==0)

{

for(j=0;j<r;j++)

{

if(need[i][j]<=avai l[j]) continue;

else break

;

}

if(j==r)

{

for(j=0;j<r;j++) avail[j]+=alloc[i][j]

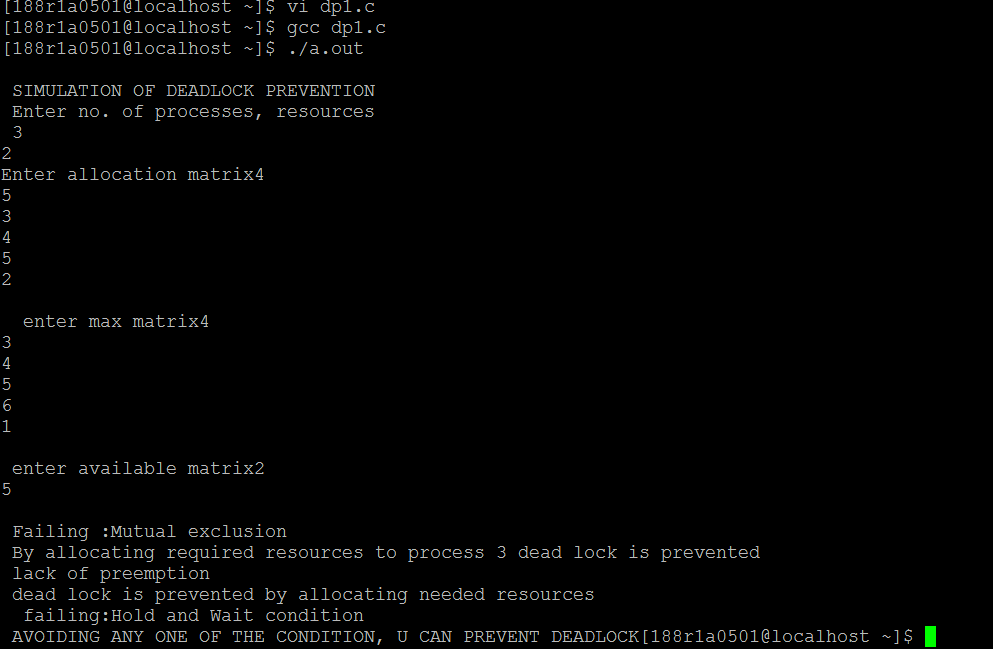
; flag=1; finish[i]=1;

}

}

}

### Output:



**LAB NO.: 10 Date:**

**MEMORY MANAGEMENT SCHEMES**

**Objectives:**

### In this lab, student will be able to:

* **Implement memory allocation methods for fixed partition using first fit, worst Fit, Best Fit.**

#### ALGORITHM:

Step 1:Define the max as 25.

Step 2: Declare the variable frag[max],b[max],f[max],i,j,nb,nf,temp, highest=0, bf[max],ff[max]. Step 3: Get the number of blocks,files,size of the blocks using for loop.

Step 4: In for loop check bf[j]!=1, if so temp=b[j]-f[i] Step 5: Check highest<temp,if so assign ff[i]=j,highest=temp Step 6: Assign frag[i]=highest, bf[ff[i]]=1,highest=0

Step 7: Repeat step 4 to step 6.

Step 8: Print file no,size,block no,size and fragment. Step 9: Stop the program.

#### PROGRAM:

#include<stdio.h

>

#include<conio.h

> #define max 25 void main()

{

int frag[max],b[max],f[max],i,j,nb,nf,temp,high est=0; static int bf[max],ff[max];

clrscr();

printf("\n\tMemory Management Scheme - Worst Fit"); printf("\nEnter the number of blocks:"); scanf("%d",&nb);

printf("Enter the number of files:"); scanf("%d",&nf);

printf("\nEnter the size of the blocks:-\n"); for(i=1;i<=nb;i++)

{

printf("Block %d:",i);

scanf("%d",&b[i]);

}

printf("Enter the size of the files :-

\n"); for(i=1;i<=nf;i++)

{

printf("File %d:",i);

scanf("%d",&f[i]);

}

for(i=1;i<=nf;i++)

{

for(j=1;j<=nb;j++)

{

if(bf[j]!=1) //if bf[j] is not allocated

{

temp=b[j]-f[i]; if(temp>=0) if(highest<temp)

{

ff[i]=j; highest=temp;

}

}

}

frag[i]=highest; bf[ff[i]]=1; highest=0;

}

printf("\nFile\_no:\tFile\_size

:\tBlock\_no:\tBlock\_size:\tFragement"); for(i=1;i<=nf;i++) printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]], frag[i]); getch();

}

#### ALGORITHM:

Step 1:Define the max as 25.

Step 2: Declare the variable frag[max],b[max],f[max],i,j,nb,nf,temp, highest=0, bf[max],ff[max]. Step 3: Get the number of blocks,files,size of the blocks using for loop.

Step 4: In for loop check bf[j]!=1, if so temp=b[j]-f[i] Step 5: Check temp>=0,if so assign ff[i]=j break the for loop. Step 6: Assign frag[i]=temp,bf[ff[i]]=1; Step 7: Repeat step 4 to step 6.

Step 8: Print file no,size,block no,size and fragment.

Step 9: Stop the program.

#### PROGRAM:

#include<stdio.h

>

#include<conio.h

> #define max 25 void main()

{

int frag[max],b[max],f[max],i,j,nb,nf,te mp; static int bf[max],ff[max]; clrscr();

printf("\n\tMemory Management Scheme - First Fit"); printf("\nEnter the number of

blocks:"); scanf("%d",&nb); printf("Enter the number of files:"); scanf("%d",&nf);

printf("\nEnter the size of the blocks:-\n"); for(i=1;i<=nb;i++)

{

printf("Block %d:",i);

scanf("%d",&b[i]);

}

printf("Enter the size of the files :-

\n"); for(i=1;i<=nf;i++)

{

printf("File %d:",i);

scanf("%d",&f[i]);

}

for(i=1;i<=nf;i++)

{

for(j=1;j<=nb;j++)

{

if(bf[j]!=1)

{

temp=b[j]-f[i]; if(temp>=0)

{

ff[i]=j; break;

}

}

}

frag[i]=temp; bf[ff[i]]=1;

}

printf("\nFile\_no:\tFile\_size

:\tBlock\_no:\tBlock\_size:\tFragement"); for(i=1;i<=nf;i++) printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]], frag[i]); getch();

}

#### ALGORITHM:

Step 1:Define the max as 25.

Step 2: Declare the variable frag[max],b[max],f[max],i,j,nb,nf,temp, highest=0, bf[max],ff[max]. Step 3: Get the number of blocks,files,size of the blocks using for loop.

Step 4: In for loop check bf[j]!=1, if so temp=b[j]-f[i] Step 5: Check lowest>temp,if so assign ff[i]=j,highest=temp Step 6: Assign

frag[i]=lowest, bf[ff[i]]=1,lowest=10000 Step 7: Repeat step 4 to step 6.

Step 8: Print file no,size,block no,size and fragment.

Step 9: Stop the program.

#### PROGRAM:

#include<stdio.h

>

#include<conio.h

> #define max 25 void main()

{

int frag[max],b[max],f[max],i,j,nb,nf,temp,lowest= 10000; static int bf[max],ff[max];

clrscr();

printf("\nEnter the number of blocks:"); scanf("%d",&nb); printf("Enter the number of files:"); scanf("%d",&nf);

printf("\nEnter the size of the blocks:-\n"); for(i=1;i<=nb;i++)

{

printf("Block %d:",i);

scanf("%d",&b[i]);

}

printf("Enter the size of the files :-

\n"); for(i=1;i<=nf;i++)

{

printf("File %d:",i);

scanf("%d",&f[i]);

}

for(i=1;i<=nf;i++)

{

for(j=1;j<=nb;j++)

{

if(bf[j]!=1)

{

temp=b[j]-f[i]; if(temp>=0) if(lowest>temp)

{

ff[i]=j;

lowest=temp;

}

}

}

frag[i]=lowest; bf[ff[i]]=1; lowest=10000;

}

printf("\nFile No\tFile Size \tBlock No\tBlock Size\tFragment"); for(i=1;i<=nf && ff[i]!=0;i++) printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i

]],frag[i]); getch();

}

**LAB NO.: 11 Date:**

**PAGE REPLACEMENT ALGORITHMS**

Implementation of FIFO page replacement algorithm.

#### ALGORITHM:

Step 1: Start the program.

Step 2: Declare the necessary variables. Step 3: Enter the number of frames.

Step 4: Enter the reference string end with zero.

Step 5: FIFO page replacement selects the page that has been in memory the longest time and when the page must be replaced the oldest page is chosen.

Step 6: When a page is brought into memory, it is inserted at the tail of the queue.

Step 7: Initially all the three frames are empty.

Step 8: The page fault range increases as the no of allocated frames also increases.

Step 9: Print the total number of page faults. Step 10: Stop the program.

#### PROGRAM:

#include<stdio.h

> int main()

{

int i=0,j=0,k=0,i1=0,m,n,rs[30],flag=1,p [30]; system("clear");

printf("FIFO page replacement algorithm. \\n"); printf("enter the no. of

frames:"); scanf("%d",&n); printf("enter the reference string:"); while(1)

{

scanf("%d",&rs[i]); if(rs[i]==0)

break; i++;

}

m=i; for(j=0;j<n;j+

+) p[j]=0;

for(i=0;i<m;i++)

{

flag=1; for(j=0;j<n;j+

+)

if(p[j]==rs[i])

{

printf("data already in page \n"); flag=0;

break;

}

if(flag==1)

{

p[i1]=rs[i]; i1++; k++;

if(i1==n) i1=0;

for(j=0;j<n;j++)

{

printf("\n page

%d:%d",j+1,p[j]);

if(p[j]==rs[i])

printf("\*");

}

printf("\n\n");

}

}

printf("total no page faults=%d",k);

}

To implement LRU page replacement algorithm.

**ALGORITHM:**

Step 1: Start the process Step 2: Declare the size

Step 3: Get the number of pages to be inserted Step 4: Get the value

Step 5: Declare counter and stack

Step 6: Select the least recently used page by counter value Step 7: Stack them according the selection.

Step 8: Display the values Step 9: Stop the process

#### PROGRAM:

#include<stdio.h> main()

{

int q[20],p[50],c=0,c1,d,f,i,j,k=0,n,r,t,b[20],c2[20]

; printf("Enter no of pages:"); scanf("%d",&n);

printf("Enter the reference string:"); for(i=0;i<n;i++)

scanf("%d",&p[i]); printf("Enter no of frames:"); scanf("%d",&f);

q[k]=p[k]; printf("\n\t%d\n",q[k]

); c++; k++;

for(i=1;i<n;i++)

{ c1=0;

for(j=0;j<f;j++)

{

if(p[i]!=q[j])

c1++;

}

if(c1==f)

{c++;

if(k<f)

{q[k]=p[i]; k++;

for(j=0;j<k;j++) printf("\t%d",q[j]); printf("\n");

}

else

{for(r=0;r<f;r++)

{c2[r]=0;

for(j=i-1;j<n;j--)

{if(q[r]!=p[j]) c2[r]++;

else break;

}}

for(r=0;r<f;r++) b[r]=c2[r]; for(r=0;r<f;r++)

{

for(j=r;j<f;j++)

{

if(b[r]<b[j])

{

t=b[r]; b[r]=b[j]; b[j]=t;

}}}

for(r=0;r<f;r++)

{

if(c2[r]==b[0])

q[r]=p[i]; printf("\t%d",q[r]);

}

printf("\n");

}}}

printf("\nThe no of page faults is %d",c);

}

implement LFU page replacement algorithm.

**ALGORITHM:**

Step 1: Start the process Step 2: Declare the size

Step 3: Get the number of pages to be inserted Step 4: Get the value

Step 5: Declare counter and stack

Step 6: Select the least frequently used page by counter value Step 7: Stack them according the selection.

Step 8: Display the values Step 9: Stop the process

#### PROGRAM:

#include<stdio.h

> int main()

{

int f,p; int

pages[50],frame[10],hit=0,count[50],tim e[50]; int i,j,page,flag,least,minTime,temp; printf("Enter no of frames : "); scanf("%d",&f);

printf("Enter no of pages : "); scanf("%d",&p); for(i=0;i<f;i++)

{

frame[i]=-1;

}

for(i=0;i<50;i++)

{

count[i]=0;

}

printf("Enter page no : \n"); for(i=0;i<p;i++)

{

scanf("%d",&pages[i]);

}

printf("\n"); for(i=0;i<p;i++)

{

count[pages[i]]++; time[pages[i]]=i; flag=1;

least=frame[0]; for(j=0;j<f;j++)

{

if(frame[j]==-1 || frame[j]==pages[i])

{

if(frame[j]!=-1)

{

hit++;

}

flag=0; frame[j]=pages[i]; break;

}

if(count[least]>count[frame[j]])

{

least=frame[j];

}

}

if(flag)

{

minTime=50; for(j=0;j<f;j++)

{

if(count[frame[j]]==count[least] && time[frame[j]]<minTime)

{

temp=j; minTime=time[frame[j]];

}

}

count[frame[temp]]=0; frame[temp]=pages[i];

}

for(j=0;j<f;j++)

{

printf("%d ",frame[j]);

}

printf("\n");

}

printf("Page hit =

%d",hit); return 0;

}

**LAB NO.: 12 Date:**

**DISK SCHEDULING ALGORITHMS**

### Objectives:

**In this lab, student will be able to:**

### IMPLEMENT DISK SCHEDULING ALGORITHMS.

* 1. **FCFS B) SCAN**

FCFS DISK SCHEDULING ALGORITHM

Given an array of disk track numbers and initial head position, our task is to find the total number of seek operations done to access all the requested tracks if **First Come First Serve (FCFS)** disk scheduling algorithm is used. **First Come First Serve (FCFS)**

FCFS is the simplest [disk scheduling algorithm](https://www.geeksforgeeks.org/disk-scheduling-algorithms/). As the name suggests, this algorithm entertains requests in the order they arrive in the disk queue.

The algorithm looks very fair and there is no starvation (all requests are serviced sequentially) but generally, it does not provide the fastest service. **Algorithm:**

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ‘head’ is the position of disk head.
2. Let us one by one take the tracks in default order and calculate the absolute distance of the track from the head.
3. Increment the total seek count with this distance.
4. Currently serviced track position now becomes the new head position.
5. Go to step 2 until all tracks in request array have not been serviced.

**Example:**

**Input:**

Request sequence = {176, 79, 34, 60, 92, 11, 41, 114}Initial head position = 50

### Output:

Total number of seek operations = 510 Seek Sequence is

176

79

34

60

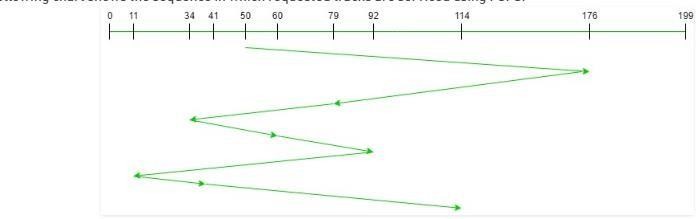
92

11

41

114

The following chart shows the sequence in which requested tracks areserviced using FCFS.



Therefore, the total seek count is calculated as:

= (176-50)+(176-79)+(79-34)+(60-34)+(92-60)+(92-11)+(41-11)+(114-41)

= 510

**PROGRAM FOR FCFS DISK SCHEDULING ALGORITHM**

#include<stdio.h

> int main()

{

int queue[20],n,head,i,j,k,seek=0,max,d iff; float avg;

printf("Enter the max range of disk\n"); scanf("%d",&max); printf("Enter the size of queue request\n"); scanf("%d",&n);

printf("Enter the queue of disk positions to be read\n"); for(i=1;i<=n;i++) scanf("%d",&queue[i]);

printf("Enter the initial head position\n"); scanf("%d",&head); queue[0]=head;

for(j=0;j<=n- 1;j++)

{

diff=abs(queue[j+1]-queue[j]); seek+=diff;

printf("Disk head moves from %d to %d with seek

%d\n",queue[j],queue[j+1],diff);

}

printf("Total seek time is

%d\n",seek); avg=seek/(float)n;

printf("Average seek time is

%f\n",avg); return 0;

}

SCAN DISK SCHEDULING ALGORITHM

Given an array of disk track numbers and initial head position, our task is to find the total number of seek operations done to access all the requested tracks if SCAN disk scheduling algorithm is used.

**SCAN (Elevator) algorithm**

In SCAN disk scheduling algorithm, head starts from one end of the disk and moves towards the other end, servicing requests in between one by one and reach the other end. Then the direction of the head is reversed and the process continues as head continuously scan back and forth to access the disk. So, this algorithm works as an elevator and hence also known as the **elevator algorithm**. As a result, the requests at the midrange areserviced more and those arriving behind the disk arm will have to wait.

**Advantages of SCAN (Elevator) algorithm**

1. This algorithm is simple and easy to understand.
2. SCAN algorithm have no starvation.
3. This algorithm is better than FCFS Scheduling algorithm .

**Disadvantages of SCAN (Elevator) algorithm**

1. More complex algorithm to implement.
2. This algorithm is not fair because it cause long waiting time for the cylinders just visited by the head.
3. It causes the head to move till the end of the disk in this way the requests arriving ahead of the arm position would get immediate service but some other requests that arrive behind the arm position will have to wait for the request to complete.

ALGORITHM

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ‘head’ is the position of disk head.
2. Let direction represents whether the head is moving towards left or right.
3. In the direction in which head is moving service all tracks one by one.
4. Calculate the absolute distance of the track from the head.
5. Increment the total seek count with this distance.
6. Currently serviced track position now becomes the new head position.
7. Go to step 3 until we reach at one of the ends of the disk.
8. If we reach at the end of the disk reverse the direction and go to step 2 until all tracks in request array have not been serviced.

Example:

### Input:

Request sequence = {176, 79, 34, 60, 92, 11, 41, 114}Initial head position = 50

Direction = left (We are moving from right to left)

### Output:

Total number of seek operations = 226 Seek Sequence is

41

34

11

0

60

79

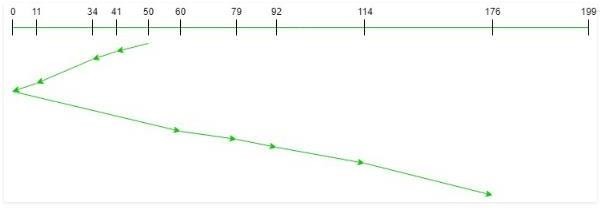
92

114

176

## The following chart shows the sequence in which requested tracks are servicedusing

SCAN.



## Therefore, the total seek count is calculated as:

= (50-41)+(41-34)+(34-11)

+(11-0)+(60-0)+(79-60)

+(92-79)+(114-92)+(176-114)

= 226

**PROGRAM FOR SCAN DISK SCHEDULING ALGORITHM**

#include<conio.h> #include<stdio.h> int main()

{

int i,j,sum=0,n; int d[20];

int disk; //loc of head int temp,max;

int dloc; //loc of disk in array clrscr();

printf("enter number of location\t"); scanf("%d",&n);

printf("enter position of head\t"); scanf("%d",&disk); printf("enter elements of disk queue\n");

for(i=0;i<n;i++)

scanf("%d",&d[i]);

}

d[n]=disk; n=n+1;

for(i=0;i<n;i++) // sorting disk locations

{

for(j=i;j<n;j++)

{

if(d[i]>d[j])

{

temp=d[i]; d[i]=d[j]; d[j]=temp;

}

}

}

max=d[n];

for(i=0;i<n;i++) // to find loc of disc in array

{

if(disk==d[i]) { dloc=i; break; }

}

for(i=dloc;i>=0;i--)

{

printf("%d -->",d[i]);

}

printf("0 -->"); for(i=dloc+1;i<n;i+

+)

{

printf("%d-->",d[i]);

}

sum=disk+max; printf("\nmovement of total cylinders

%d",sum); getch(); return 0;

}

Output:

Enter no of location 8 Enter position of head 53

Enter elements of disk queue 98

183

37

122

14

124

65

67

53->37->14->0->65->67->98->122->124->183->

Movement of total cylinders 236.

**REFERENCES**

1. Maurice Bach, Design of Unix Operating System, Prentice Hall India Learning Private Limited, 2015
2. Graham Glass, Unix for Programmers and Users- A complete guide, (3e), Prentice Hall India Learning Private Limited , 2003
3. Sumitabha Das, Unix concepts and applications, (4e), McGraw Hill Education ,2017
4. Neil Matthew, Richard Stones, Beginning Linux Programming, (4e),Wiley Publication, 2007
5. A. Silberschatz, P. B. Galvin and G. Gagne, Operating System Concepts, (9e), Wiley and Sons (Asia) Pte Ltd, 2013.
6. Rachel Morgan- Unix System, McGraw Hill Education , 2007
7. N.B.Venkateshwarlu-Advanced UNIX Programming, (2e), BS Publications, 2010
8. Gisela May A.Albano, Angelito G Pastrana, Fundamentals of Operating System,A & C Printers, 2009.
9. Darryl Gove, Multicore Application Programming for Windows, Linuz and Oracle Solaris, Addison-Wesley, 2010