UNIVERSITY COLLEGE OF ENGINEERING NAGERCOIL

(ANNA UNIVERSITY CONSTITUENT COLLEGE)

KONAM, NAGERCOIL-629004

RECORD NOTE BOOK

Name :

Register Number :

Year/Semester :

Subject Code and Name :

Department :

UNIVERSITY COLLEGE OF ENGINEERING NAGERCOIL

(ANNA UNIVERSITY CONSTITUENT COLLEGE)

KONAM, NAGERCOIL-629004

BONAFIDE CERTIFICATE

Certified that,this is the bonafide record of work done by Mr/Ms …………………………….. of **2nd year /4th** semester in the Department of **Computer Science And Engineering** of this College in **CS3461-Operating Systems Laboratory** during the academic year 2024-2025 in partial fulfillment of the requirement of the B.E Degree Course of Anna University Chennai.

Faculty-in-charge **Head of the Department**

Submitted for Practical Examination to be held on ……………….

Internal Examiner **External Examiner**

INDEX

AIM:

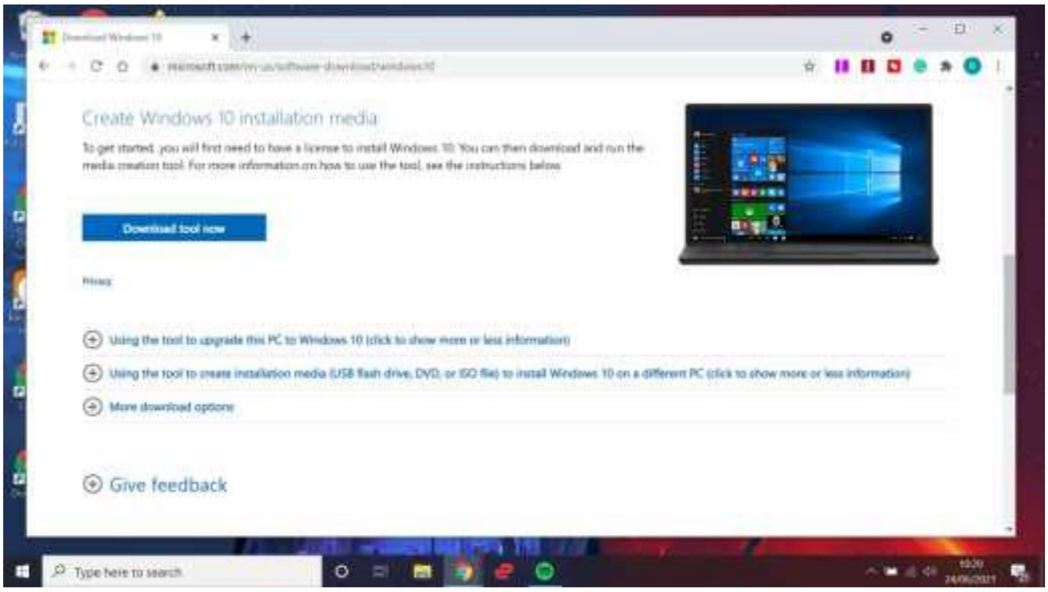
PROCEDURE:

Step1: Check your device meets the Windows 10 system requirements. Below you'll find the minimum specs needed to run Windows 10, so check your device is capable:

CPU: 1GHz or faster processor
RAM: 1GB for Windows 10 32-bit or 2GB for Windows 10 64-bit
Storage: 32GB of space or more
GPU: DirectX 9 compatible or later with WDDM 1.0 driver
Display: 800x600 resolution or higher

If it meets the requirement, we can proceed to the next step.

Step2: Create USB installation media. Visit Microsoft's Windows 10 download page (opens in new tab) and select "Download tool now" under the "create Windows 10 installation media" section. Transfer the downloaded installer tool to a USB drive.



Step3: Run the installer tool. Open the installer tool by clicking on it. Accept Microsoft's terms, and then **select "Create installation media for another PC"** on the "What do you want to do?" page. After selecting which language, you want Windows 10 to run in, and which edition you want as well (32-bit or 62-bit), you'll be asked what type of media you want to use.

Installing from a USB drive is definitely the preferred option but you can also install froma CD or ISO file. Once you choose your device, the installer tool will download the required files and put them onto your drive.

Step4: Use your installation media. Insert your installation media into your device and then **access the computer's BIOS or UEFI**. These are the systems that allow you to control your computer's core hardware.

The process of accessing these systems is unique to each device, but the manufacturer's website should be able to give you a helping hand here. Generally, you'll need to **press the F2, F12 or Delete keys** as your computer boots up.

Step5: Change your computer's boot order. Once you have access to your computer's BIOS/UEFI you'll need to locate the settings for boot order. You need the Windows 10 installation tool to be higher up on the list than the device's current current boot drive: this is the SSD or HDD that your existing OS is stored on. You should **move the drive with the installer files to the very top of the boot order menu**. Now, when you restart your device the Windows 10 installer should load up first.

Step6: Restart your device. Save your settings in the BIOS/UEFI and reboot your device.

Step7: Complete the installation. Your device should now load up the Windows 10 installation tool on restart. This will guide you through the rest of the installation process.

AIM:

COMMANDS:

1. Date Command:

This command is used to display the current data and time.

Syntax:

```
$date
$date +%ch
```

Options:

a = Abbrevated weekday.
A = Full weekday.
b = Abbrevated month.
B = Full month.
c = Current day and time.
C = Display the century as a decimal number.
d = Day of the month.
D = Day in „mm/dd/yy‟ format
h = Abbrevated month day.
H = Display the hour.
L = Day of the year.
m = Month of the year.
M = Minute.
P = Display AM or PM
S = Seconds
T = HH:MM: SS format
u = Week of the year.
y = Display the year in 2 digits.
Y = Display the full year.
Z = Time zone.

OUTPUT:

```
cse@ubuntu2:~$ date +%H:%M:%S
21:59:02
cse@ubuntu2:~$
```

2. Calendar Command:

This command is used to display the calendar of the year or a particular month of the calendar year.

Syntax:

a.$cal<year> b.$cal<montah><year>

Here the first syntax gives the entire calendar for a given year & the second Syntax gives the calendar of the reserved months of that year.

OUTPUT:

```
cse@ubuntu2:~$ cal 2025
                            2025
      January               February               March
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
          1  2  3  4                     1                     1
 5  6  7  8  9 10 11   2  3  4  5  6  7  8   2  3  4  5  6  7  8
12 13 14 15 16 17 18   9 10 11 12 13 14 15   9 10 11 12 13 14 15
19 20 21 22 23 24 25  16 17 18 19 20 21 22  16 17 18 19 20 21 22
26 27 28 29 30 31     23 24 25 26 27 28     23 24 25 26 27 28 29
                                            30 31

       April                  May                   June
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
       1  2  3  4  5               1  2  3   1  2  3  4  5  6  7
 6  7  8  9 10 11 12   4  5  6  7  8  9 10   8  9 10 11 12 13 14
13 14 15 16 17 18 19  11 12 13 14 15 16 17  15 16 17 18 19 20 21
20 21 22 23 24 25 26  18 19 20 21 22 23 24  22 23 24 25 26 27 28
27 28 29 30           25 26 27 28 29 30 31  29 30

        July                 August              September
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
       1  2  3  4  5                  1  2      1  2  3  4  5  6
 6  7  8  9 10 11 12   3  4  5  6  7  8  9   7  8  9 10 11 12 13
13 14 15 16 17 18 19  10 11 12 13 14 15 16  14 15 16 17 18 19 20
20 21 22 23 24 25 26  17 18 19 20 21 22 23  21 22 23 24 25 26 27
27 28 29 30 31        24 25 26 27 28 29 30  28 29 30
                      31

      October               November              December
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
          1  2  3  4                     1      1  2  3  4  5  6
 5  6  7  8  9 10 11   2  3  4  5  6  7  8   7  8  9 10 11 12 13
12 13 14 15 16 17 18   9 10 11 12 13 14 15  14 15 16 17 18 19 20
19 20 21 22 23 24 25  16 17 18 19 20 21 22  21 22 23 24 25 26 27
26 27 28 29 30 31     23 24 25 26 27 28 29  28 29 30 31
                      30
cse@ubuntu2:~$
```

```
cse@ubuntu2:~$ cal 5 2025
      May 2025
Su Mo Tu We Th Fr Sa
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31

cse@ubuntu2:~$
```

3. Echo Command:

This command is used to print the arguments on the screen.

Syntax:

$echo<text>

To have the output in different line, the following command can be used.

Syntax:

```
$echo "text
>line2
>line3"
```

OUTPUT:

```
cse@ubuntu2:~$ echo "How
> are
> you?"
How
are
you?
cse@ubuntu2:~$
```

4. 'who' Command:

It is used to display who are the users connected to our computer currently.

Syntax: $who –option's

```
$who –option's
```

Options:

H–Display the output with headers.

b–Display the last booting date or time or when the system was lately rebooted.

OUTPUT:

```
cse@ubuntu2:~$ who -H
NAME     LINE         TIME             COMMENT
cse      :0           2025-05-08 21:48 (:0)
cse@ubuntu2:~$ who -b
         system boot  2025-05-08 21:47
cse@ubuntu2:~$
```

5. 'who am i' Command:

Display the details of the current working directory.

Syntax:

```
$who am i
```

OUTPUT:

```
cse@ubuntu2:~$ whoami
cse
cse@ubuntu2:~$
```

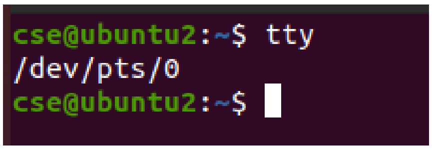
6. 'tty' Command:

It will display the terminal name.

Syntax:

```
$tty
```

OUTPUT:



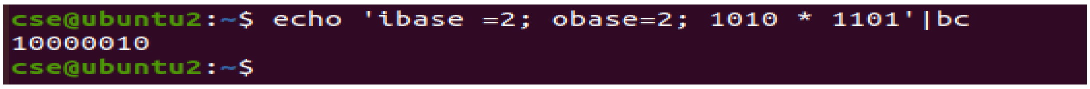
7. 'Binary' Calculator Command:

It will change the „$ mode and in the new mode, arithmetic operations such as +, -, *, /, %, n,
sqrt (), length (), =, etc can be performed. This command is used to go to the binary calculus mode.

Syntax:

$echo 'ibase=2; obase=2; <binary_expression>' | bc

OUTPUT:



8. 'CLEAR' Command:

It is used to clear the screen.

Syntax:

$clear

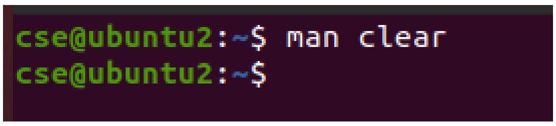
9. 'MAN' Command:

It helps us to know about the particular command and its options & working. It is like "help"
command in Windows.

Syntax: $man <command name>

$man<command name>

OUTPUT:



```
clear(1)                    General Commands Manual                    clear(1)

NAME
       clear - clear the terminal screen

SYNOPSIS
       clear [-Ttype] [-V] [-x]

DESCRIPTION
       clear  clears your screen if this is possible, including its scrollback buffer (if the
       extended “E3” capability is defined).  clear looks in the environment for the terminal
       type  given by the environment variable TERM, and then in the terminfo database to de-
       termine how to clear the screen.

       clear writes to the standard output.  You can redirect the standard output to  a  file
       (which  prevents  clear  from actually clearing the screen), and later cat the file to
       the screen, clearing it at that point.

OPTIONS
       -T type
            indicates the type of terminal.  Normally this option is unnecessary, because the
            default  is  taken  from the environment variable TERM.  If -T is specified, then
            the shell variables LINES and COLUMNS will also be ignored.

       -V   reports the version of ncurses which was used in this program,  and  exits.   The
            options are as follows:

       -x   do  not attempt to clear the terminal's scrollback buffer using the extended “E3”
            capability.

HISTORY
       A clear command appeared in 2.79BSD dated February 24, 1979.  Later that was  provided
       in Unix 8th edition (1985).

       AT&T  adapted  a  different  BSD program (tset) to make a new command (tput), and used
       this to replace the clear command with a shell script which calls tput clear, e.g.,

           /usr/bin/tput ${1:+-T$1} clear 2> /dev/null
           exit

       In 1989, when Keith Bostic revised the BSD tput command to make it similar to the AT&T
       tput, he added a shell script for the clear command:

           exec tput clear

       The remainder of the script in each case is a copyright notice.

 Manual page clear(1) line 1 (press h for help or q to quit)
```

10. MANIPULATION Command:

It is used to manipulate the screen.

Syntax: $tput <argument>

$tput <argument>

Arguments:

1. Clear – to clear the screen.
2. Longname – Display the complete name of the terminal.
3. smso – background becomes white and foreground becomes black colour.
4. rmso – background becomes black and foreground becomes white colour.

OUTPUT:

```
cse@ubuntu2:~$ tput smso
cse@ubuntu2:~$
```

11. LIST Command:

It is used to list all the contents in the current working directory.

Syntax:

$ ls –options <arguments>

If the command does not contain any argument means it is working in the Current directory.

Options:

a– used to list all the files including the hidden files.

c– list all the files columnwise.

d- list all the directories.

m- list the files separated by commas.

p- list files include "/" to all the directories.

r- list the files in reverse alphabetical order.

f- list the files based on the list modification date.

x-list in column wise sorted order.

OUTPUT:

```
cse@ubuntu2:~$ ls -l -a Desktop/cseb
total 196
drwxrwxr-x 2 cse cse  4096 May  8 19:07 .
drwxr-xr-x 3 cse cse  4096 Apr 28 19:11 ..
-rwxrwxr-x 1 cse cse 16872 May  8 19:07 a.out
-rw-rw-r-- 1 cse cse  1798 Apr 23 21:26 bankers.c
-rw-rw-r-- 1 cse cse  1082 May  4 17:48 bestfit.c
-rw-rw-r-- 1 cse cse   547 Apr 23 20:47 client.c
-rw-rw-r-- 1 cse cse   949 May  8 19:07 clook.c
-rw-rw-r-- 1 cse cse   964 May  8 19:02 cscan.c
-rw-rw-r-- 1 cse cse  1817 Apr 24 19:47 dedlock_detection.c
-rw-rw-r-- 1 cse cse  4176 May  8 18:33 diskscheduling.c
-rw-rw-r-- 1 cse cse   567 May  8 18:53 fcfs.c
-rw-rw-r-- 1 cse cse   771 May  7 18:38 fifo.c
-rw-rw-r-- 1 cse cse  1256 May  4 17:14 firstfit.c
-rw-rw-r-- 1 cse cse   630 Apr 29 20:26 firstinfirstout.c
-rw-rw-r-- 1 cse cse   855 May  8 18:02 indexfileall.c
-rw-rw-r-- 1 cse cse   690 May  8 17:54 linkedfileall.c
-rw-rw-r-- 1 cse cse  1022 May  8 19:05 look.c
-rw-rw-r-- 1 cse cse  1284 May  8 16:52 lru.c
-rw-rw-r-- 1 cse cse  1121 May  8 16:57 optimal.c
-rwxrwxr-x 1 cse cse 16888 May  3 11:30 paging
-rw-rw-r-- 1 cse cse   680 May  3 11:39 paging.c
-rwxrwxr-x 1 cse cse 16888 May  1 10:47 paing
-rw-rw-r-- 1 cse cse  1025 May  8 19:00 scan.c
-rw-rw-r-- 1 cse cse   908 Apr 24 19:16 semaphore.c
-rw-rw-r-- 1 cse cse   702 May  8 17:48 seqfileall.c
-rw-rw-r-- 1 cse cse   745 Apr 23 20:45 server.c
-rw-rw-r-- 1 cse cse   659 May  8 17:22 singleleveldir.c
-rw-rw-r-- 1 cse cse   833 May  8 18:57 sstf.c
-rwxrwxr-x 1 cse cse 16864 May  1 10:38 thread
-rw-rw-r-- 1 cse cse   560 May  1 10:34 thread.c
-rw-rw-r-- 1 cse cse   977 May  8 17:32 twoleveldir.c
-rw-rw-r-- 1 cse cse  1112 May  4 17:21 worstfit.c
cse@ubuntu2:~$
```

DIRECTORY RELATED COMMANDS:

1. Present Working Directory Command:

To print the complete path of the current working directory.

Syntax: $pwd

$pwd

OUTPUT:

2. MKDIR Command:

To create or make a new directory in a current directory.

Syntax:

$mkdir <directory name>

3. CD Command:

To change or move the directory to the mentioned directory.

Syntax:

$cd <directory name>

4. RMDIR Command:

To remove a directory in the current directory & not the current directory itself.

Syntax:

$rmdir <directory name>

FILE RELATED COMMANDS:

1. CREATE A FILE:

To create a new file in the current directory we use CAT command.

Syntax:

$cat > filename

The > symbol is the directory we use the cat command.

2. DISPLAY A FILE:

To display the content of the file mentioned we use the CAT command without the “>” operator.

Syntax:

$cat filename

AIM:

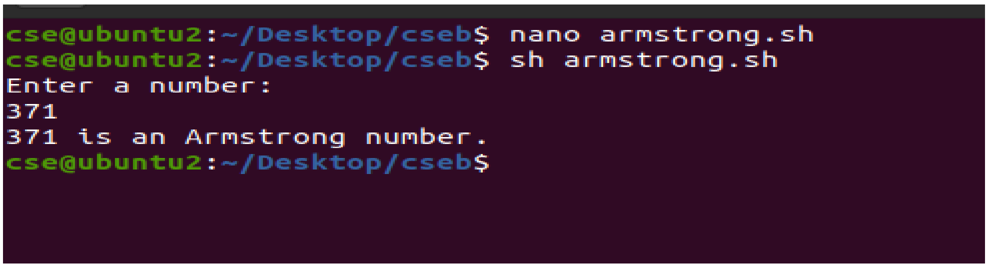
ALGORITHM:

Step 1: Prompt the user to enter a number.
Step 2: Read and store the number in the variable number.
Step 3: Store the original value of number in the variable originalNumber.
Step 4: Calculate the number of digits in number using ${#number} and store the result in numOfDigits.
Step 5: Initialize the variable sum to 0.
Step 6: Enter a loop that continues until number becomes 0:
   a. Extract the last digit of number using $number % 10 and store it in digit.
   b. Calculate digit raised to the power of numOfDigits using $digit^$numOfDigits.
   c. Add the result to sum.
   d. Remove the last digit from number by dividing it by 10 using $number / 10.
Step 7: Check if sum is equal to originalNumber.
Step 8: If the condition is true, print that originalNumber is an Armstrong number.
Step 9: If the condition is false, print that originalNumber is not an Armstrong number.

PROGRAM:

```
#!/bin/bash
echo "Enter the first number: "
read num1
echo "Enter the second number: "
read num2
echo "Enter the third number: "
read num3
if [ $num1 -gt $num2 ] && [ $num1 -gt $num3 ]
then
   echo "$num1 is the greatest."
elif [ $num2 -gt $num1 ] && [ $num2 -gt $num3 ]
then
   echo "$num2 is the greatest."
else

   echo "$num3 is the greatest."
Fi
```

OUTPUT:



```
cse@ubuntu2:~/Desktop/cseb$ nano armstrong.sh
cse@ubuntu2:~/Desktop/cseb$ sh armstrong.sh
Enter a number:
371
371 is an Armstrong number.
cse@ubuntu2:~/Desktop/cseb$
```

Shell Script To Find the Greatest of Three Numbers

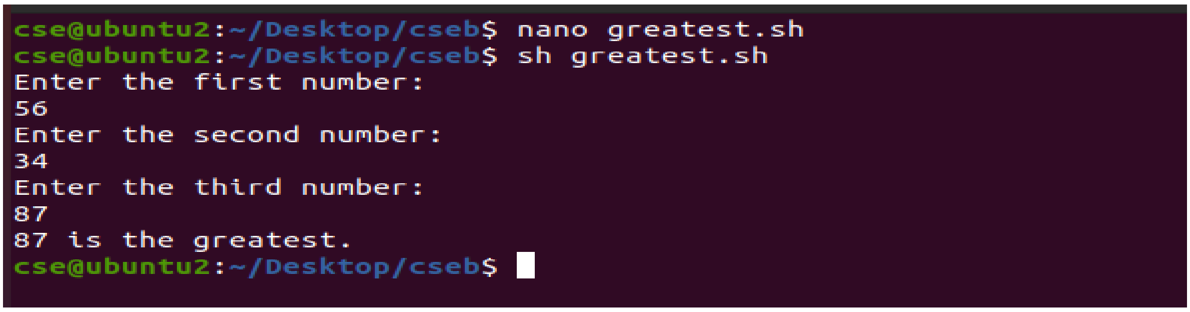
ALGORITHM:

Step 1: Prompt the user to enter the first number.
Step 2: Read and store the first number in the variable num1.
Step 3: Prompt the user to enter the second number.
Step 4: Read and store the second number in the variable num2.
Step 5: Prompt the user to enter the third number.
Step 6: Read and store the third number in the variable num3.
Step 7: Compare the values of the numbers using the following conditions:
  a. If num1 is greater than both num2 and num3, then num1 is the greatest number.
  b. If num2 is greater than both num1 and num3, then num2 is the greatest number.
  c. Otherwise, num3 is the greatest number.
Step 8: Display the greatest number.

PROGRAM:

```
#!/bin/bash
echo "Enter the first number: "
read num1
echo "Enter the second number: "
read num2
echo "Enter the third number: "
read num3
if [ $num1 -gt $num2 ] && [ $num1 -gt $num3 ]
then
    echo "$num1 is the greatest."
elif [ $num2 -gt $num1 ] && [ $num2 -gt $num3 ]
then
   echo "$num2 is the greatest."
else
    echo "$num3 is the greatest."
fi
```

OUTPUT:



```
cse@ubuntu2:~/Desktop/cseb$ nano greatest.sh
cse@ubuntu2:~/Desktop/cseb$ sh greatest.sh
Enter the first number:
56
Enter the second number:
34
Enter the third number:
87
87 is the greatest.
cse@ubuntu2:~/Desktop/cseb$
```

Shell Script to Find the sum of two numbers

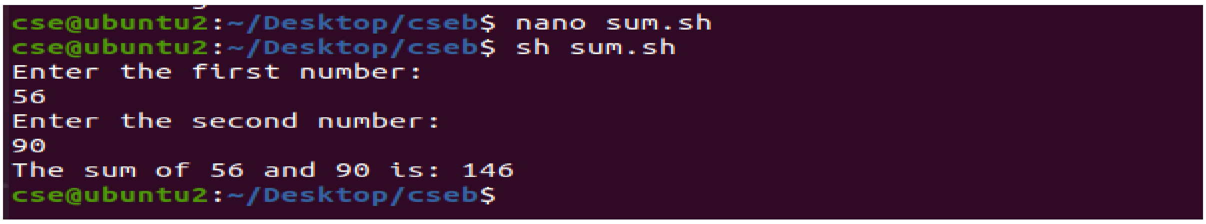
ALGORITHM:

Step 1: Prompt the user to enter the first number.
Step 2: Read and store the first number in the variable num1.
Step 3: Prompt the user to enter the second number.
Step 4: Read and store the second number in the variable num2.
Step 5: Calculate the sum of num1 and num2 by adding them together.
Step 6: Store the sum in the variable sum.
Step 7: Display the sum of the two numbers along with the original numbers.

PROGRAM:

```
#!/bin/bash
echo "Enter the first number: "
read num1
echo "Enter the second number: "
read num2
sum=$((num1 + num2))
echo "The sum of $num1 and $num2 is: $sum"
```

OUTPUT:



Shell Script to calculate the average of three numbers

ALGORITHM:

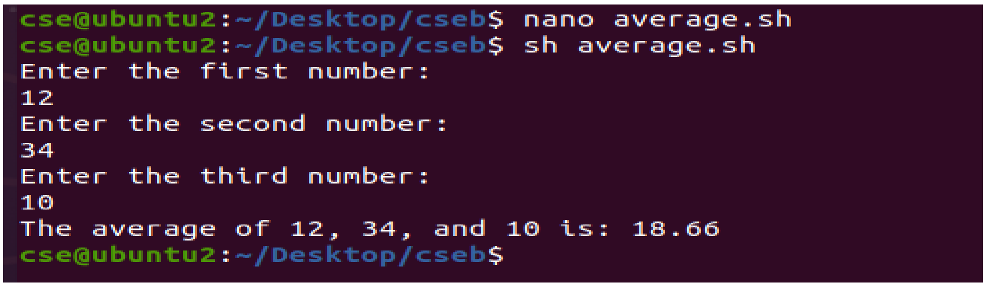
Step 1: Prompt the user to enter the first number.
Step 2: Read and store the first number in the variable num1.
Step 3: Prompt the user to enter the second number.
Step 4: Read and store the second number in the variable num2.
Step 5: Prompt the user to enter the third number.
Step 6: Read and store the third number in the variable num3.
Step 7: Calculate the sum of the three numbers by adding them together.
Step 8: Divide the sum by 3 to calculate the average.
Step 9: Store the average in the variable average.
Step 10: Display the average of the three numbers along with the original numbers.

PROGRAM:

```
#!/bin/bash
echo "Enter the first number: "
read num1
echo "Enter the second number: "
read num2
echo "Enter the third number: "
read num3
sum=$((num1 + num2 + num3))
```

average=$(echo "scale=2; $sum / 3" | bc)
echo "The average of $num1, $num2, and $num3 is: $average"

<u>OUTPUT:</u>



AIM:

ALGORITHM:

Step 1: Declare the variable pid.
Step 2: Get the pid value using system call fork ().
Step 3: If pid value is less than zero then print as "Fork failed".
Step 4: Else if pid value is equal to zero include the new process in the system"s Fil. using execlp system call.
Step 5: Else if pid is greater than zero then it is the parent process and it waits till the child completes using the system call wait ()
Step 6: Then print "Child complete"

PROGRAM:

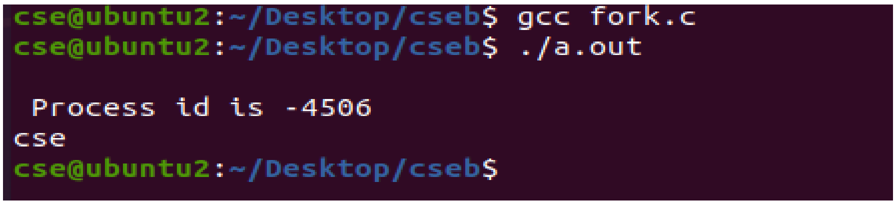
```
#include <stdio.h>
#include <stdlib.h>
#include<sys/wait.h>
#include <unistd.h>


void main(int argc, char *arg[]) {
int pid;
pid = fork(); // create a new process
if (pid < 0) {
printf("fork failed");
exit(1);
} else if (pid == 0) { // child process
execlp("whoami", "ls", NULL); // replace child process code with whoami command
exit(0); // exit child process
} else { // parent process
printf("\n Process id is -%d\n", getpid()); // print parent process ID
wait(NULL); // wait for child process to finish
exit(0);
}
}
```

System call used:

1.fork ():

The fork () system call is a fundamental operating system call in Unix-like operating systems, including Linux, macOS, and FreeBSD. It is used to create a new process, which is called a child process. The fork () system call creates a new process by duplicating the calling process, which becomes the parent process of the new child process. The new child process is an exact copy of the parent process, except that it has its own unique process ID (PID) and parent process ID (PPID).

OUTPUT:



b) PROGRAM USING SYSTEM CALL: EXIT ()

AIM:

ALGORITHM:

Step 1: Include the necessary header files: stdio.h, stdlib.h, and unistd.h.

Step 2: Declare a variable of type pid_t to store the process ID returned by fork ().

Step 3: Call fork () to create a child process. If fork () returns -1, an error has occurred, so print an error message using perror() and exit the program with a failure status using exit().

Step 4: If the process is the child process, print its process ID using getpid(), a message indicating that it is running and another message indicating that it is exiting, then exit the process with a success status using exit().

Step 5: Print a message with parent process ID, wait for child process to finish using wait() with NULL argument, and print message for child process finishing.

Step 6: Exit the program with a success status using return 0.

PROGRAM:

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

int main() {
pid_t pid;
pid = fork();
if (pid == -1) {
perror("fork error");
exit(EXIT_FAILURE);
} else if (pid == 0) {
printf("Child process is running with process ID: %d\n", getpid());
```

```
printf("Child process is exiting...\n");
exit(EXIT_SUCCESS);
} else {
printf("Parent process is running with process ID: %d\n", getpid());
printf("Waiting for child process to finish...\n");
wait(NULL);
printf("Child process has finished.\n");
}
return 0;
}
```

System call used:

1.exit ():

The System call exit () is used to terminate a process.

<u>OUTPUT:</u>

```
cse@ubuntu2:~/Desktop/cseb$ gcc exit.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out
Child process is running with process ID: 4554
Child process is exiting...
Parent process is running with process ID: 4553
Waiting for child process to finish...
Child process has finished.
cse@ubuntu2:~/Desktop/cseb$
```

<u>c) PROGRAM USING SYSTEM CALLS: GETPID ()</u>

<u>AIM:</u>

<u>ALGORITHM:</u>

Step 1: Declare a variable of type pid_t to store the process ID returned by fork ().

Step 2: Call fork () to create a child process. If fork () returns -1, an error has occurred, so print an error message using perror() and exit the program with a failure status using exit().

Step 3: If the process is the child process, print a message indicating that the child process is running with its process ID using getpid().

Step 4: If the process is the parent process, print a message indicating that the parent process is running with its process ID using getpid().

Step 5: Exit the program with a success status using return 0.

<u>PROGRAM:</u>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
pid_t pid;
pid = fork();
if (pid == -1) {
perror("fork error");
exit(EXIT_FAILURE);
```

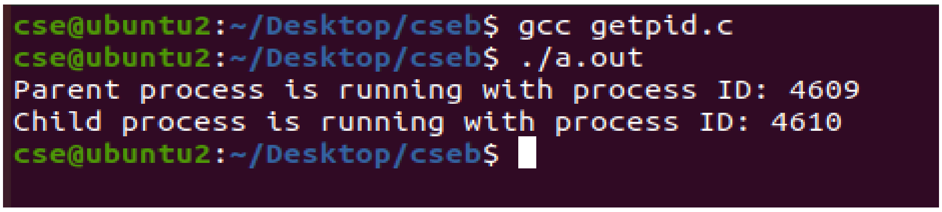
```
} else if (pid == 0) {
printf("Child process is running with process ID: %d\n", getpid());
} else {
printf("Parent process is running with process ID: %d\n", getpid());
}
return 0;
}
```

System call used:

1.getpid ():

'getpid()' is a system call in Unix and Unix-like operating systems that returns the process ID (PID) of the calling process.

OUTPUT:



d) PROGRAM USING SYSTEM CALLS: WAIT ()

AIM:

ALGORITHM:

Step 1: Declare variables pid, cpid, and status of types pid_t and int.

Step 2: Call fork () to create a child process, and handle the error case where fork () returns -1.

Step 3: In the child process, print a message indicating that it is a child process, sleep for 5 seconds, and then exit with a success status.

Step 4: In the parent process, print a message indicating that it is a parent process, and wait for the child process to terminate using wait () and store the child process ID and exit status in cpid and status variables respectively.

Step 5: Check if the child process terminated normally using WIFEXITED (status), and if so, print a message indicating the child process ID and exit status. Otherwise, print a message indicating that the child process terminated abnormally.

Step 6: Exit the program with a success status.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
pid_t pid, cpid;
int status;
pid = fork();
if (pid == -1) {
```

```
perror("fork error");
exit(EXIT_FAILURE);
} else if (pid == 0) {
printf("Child process\n");
sleep(5);
exit(EXIT_SUCCESS);
} else {
printf("Parent process\n");
cpid = wait(&status);
if (WIFEXITED(status)) {
printf("Child process %d terminated with status %d\n", cpid,
WEXITSTATUS(status));
} else {
printf("Child process %d terminated abnormally\n", cpid);
}
}
return 0;
}
```

System call used:

1.wait ():

In computing, wait () is a system call that suspends the calling process until one of its child processes terminates. It also provides information about the child process that terminated, such as its exit status.

OUTPUT:

```
cse@ubuntu2:~/Desktop/cseb$ gcc wait.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out
Parent process
Child process
Child process 4687 terminated with status 0
cse@ubuntu2:~/Desktop/cseb$
```

e) PROGRAM USING SYSTEM CALL: CLOSE ()

AIM:

ALGORITHM:

Step 1: Declare an integer variable `fd` for file descriptor

Step 2: Open a file named `test.txt` with the `open ()` system call using the flags `O_WRONLY`, `O_CREAT`, and `O_TRUNC`, and permissions `S_IRUSR` and `S_IWUSR`. Store the file descriptor returned by the `open ()` call in `fd`

Step 3: Check if `fd` is equal to `-1`, if it is, print an error message using `perror()`, and exit the program with a failure status using `exit()`

Step 4: Print a message indicating that the file has been opened with the file descriptor `fd`
Step 5: Write the string `"Hello, world! \n"` to the file using the `write () ` system call, passing `fd` as the file descriptor and `14` as the number of bytes to write. Check if the return value is `-1`, if it is, print an error message using `perror()`, and exit the program with a failure status using `exit()`
Step 6: Close the file using the `close () ` system call, passing `fd` as the file descriptor. Check if the return value is `-1`, if it is, print an error message using `perror()`, and exit the program with a failure status using `exit()`
Step 7: Print a message indicating that the file has been closed
Step 8: Exit the program with a success status using `exit () `

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
int main () {
int fd;
// Open a file for writing
fd = open("test.txt", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
if (fd == -1) {
perror("open error");
exit(EXIT_FAILURE);
}
printf("File opened with file descriptor %d\n", fd);
// Write some data to the file
if (write(fd, "Hello, world!\n", 14) == -1) {
perror("write error");
exit(EXIT_FAILURE);
}
// Close the file
if (close(fd) == -1) {
perror("close error");
exit(EXIT_FAILURE);
}
printf("File closed\n");
return 0;
}
```

System call used:

1.close ():

In operating systems, close () is a system call that is used to release a file descriptor and free the associated kernel resources. It returns 0 on success and -1 on failure, setting errno to indicate the cause of the error.

OUTPUT:

```
cse@ubuntu2:~/Desktop/cseb$ gcc close.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out
File opened with file descriptor 3
File closed
cse@ubuntu2:~/Desktop/cseb$ 
```

AIM:

ALGORITHM:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process name and the burst time

Step 4: Set the waiting of the first process as 0‘and its burst time as its turnaroundtime

Step 5: for each process in the Ready Q calculate

Step 5.1: Waiting time (n) = waiting time (n-1) + Burst time (n-1)

Step 5.2: Turnaround time (n)= waiting time(n)+Burst time(n)

Step 6: Calculate

Step 6.1: Average waiting time = Total waiting Time / Number of process

Step 6.2: Average Turnaround time = Total Turnaround Time / Number of process

Step 7: Stop the process

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
int main()
{
int bt[20], wt[20], tat[20], i, n;
float wtavg, tatavg;
clrscr();
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\nEnter Burst Time for Process %d -- ", i);
scanf("%d", &bt[i]);
}
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0];
for(i=1;i<n;i++)
{
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f", tatavg/n);
getch();
return 0;
}
```

OUTPUT:

```
Enter the number of processes -- 3

Enter Burst Time for Process 0 -- 24

Enter Burst Time for Process 1 -- 3

Enter Burst Time for Process 2 -- 3
         PROCESS        BURST TIME       WAITING TIME    TURNAROUND TIME

         P0              24              0               24
         P1             3                24              27
         P2             3                27              30
Average Waiting Time -- 17.000000
Average Turnaround Time -- 27.000000_
```

AIM:

ALGORITHM:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

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

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

Step 5: Set the waiting time of the first process as 0‘and its turnaround time as its burst time.

Step 6: Sort the processes names based on their Burt time

Step 7: For each process in the ready queue, calculate

Step 7.1: Waiting time(n) = waiting time (n-1) + Burst time (n-1)

Step 7.2: Turnaround time (n) = waiting time(n)+Burst time(n)

Step7.3: Average waiting time = Total waiting Time / Number of process

Step 7.4: Average Turnaround time = Total Turnaround Time / Number of process

Step 8: Stop the process

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
main()
{
int p[20], bt[20], wt[20], tat[20], i, k, n, temp; float wtavg,
tatavg;
clrscr();
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
p[i]=i;
printf("Enter Burst Time for Process %d -- ", i);
scanf("%d", &bt[i]);
}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(bt[i]>bt[k])
{
temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;
temp=p[i];
p[i]=p[k];
p[k]=temp;
}
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0]; for(i=1;i<n;i++)
{
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);
```

```
printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f", tatavg/n);
getch();
 return 0;
}
```

OUTPUT:

```
Enter the number of processes -- 4
Enter Burst Time for Process 0 -- 6
Enter Burst Time for Process 1 -- 8
Enter Burst Time for Process 2 -- 7
Enter Burst Time for Process 3 -- 3

         PROCESS        BURST TIME        WAITING TIME    TURNAROUND TIME

         P3              3               0               3
         P0              6               3               9
         P2              7               9               16
         P1              8               16              24
Average Waiting Time -- 7.000000
Average Turnaround Time -- 13.000000
```


AIM:

ALGORITHM:

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue and time quantum
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Calculate the no. of time slices for each process where No. of time slice for process (n) = burst time process (n)/time slice
Step 5: If the burst time is less than the time slice then the no. of time slices =1.
Step 6: Consider the ready queue is a circular Q, calculate
Step 6.1: 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)
Step 6.2: Turnaround time for process(n) = waiting time of process(n) + burst time of process(n)+ the time difference in getting CPU from process(n).
Step 6.3: Average waiting time = Total waiting Time / Number of process
Step 6.4: Average Turnaround time = Total Turnaround Time / Number of process
Step 7: Stop the process

PROGRAM:

```
#include<stdio.h>
main()
{
int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;
float awt=0,att=0,temp=0;
clrscr();
printf("Enter the no of processes -- ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
```

```
printf("\nEnter Burst Time for process %d -- ", i+1);
scanf("%d",&bu[i]);
ct[i]=bu[i];
}
printf("\nEnter the size of time slice -- ");
scanf("%d",&t);
max=bu[0];
for(i=1;i<n;i++)
if(max<bu[i])
max=bu[i];
for(j=0;j<(max/t)+1;j++)
for(i=0;i<n;i++)
if(bu[i]!=0)
if(bu[i]<=t)
{
tat[i]=temp+bu[i];
temp=temp+bu[i];
bu[i]=0;
}
else
{
bu[i]=bu[i]-t;
temp=temp+t;
}
for(i=0;i<n;i++)
{
wa[i]=tat[i]- ct[i];
att+=tat[i];
awt+=wa[i];
}
printf("\nThe Average Turnaround time is -- %f",att/n);
printf("\nThe Average Waiting time is -- %f ",awt/n);
printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]);
getch();
return 0;
}
```

OUTPUT:

```
Enter the no of processes -- 3

Enter Burst Time for process 1 -- 5

Enter Burst Time for process 2 -- 6

Enter Burst Time for process 3 -- 24

Enter the size of time slice -- 3

The Average Turnaround time is -- 20.000000
The Average Waiting time is -- 8.333333
        PROCESS  BURST TIME       WAITING TIME   TURNAROUND TIME
        1        5                6               11
        2        6                8               14
        3        24               11              35

_
```

AIM:

ALGORIMTHM:

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Sort the ready queue according to the priority number.
Step 5: Set the waiting of the first process as _0‘and its burst time as its turnaround time
Step 6: Arrange the processes based on process priority
Step 7: For each process in the Ready Q calculate
Step 8: for each process in the Ready Q calculate
Step 8.1: Waiting time(n)= waiting time (n-1) + Burst time (n-1)
Step 8.2: Turnaround time (n)= waiting time(n)+Burst time(n)
Step 8.3: Average waiting time = Total waiting Time / Number of process
Step 8.4: Average Turnaround time = Total Turnaround Time / Number of process
Step 9: Print the results in an order.
Step10: Stop the process

PROGRAM:

```
#include<stdio.h>
main()
{
int p[20],bt[20],pri[20], wt[20],tat[20],i, k, n, temp; float wtavg, tatavg;
clrscr();
printf("Enter the number of processes --- ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
```

```
p[i] = i;
printf("Enter the Burst Time & Priority of Process %d --- ",i);
scanf("%d%d",&bt[i], &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]=temp;
temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;
temp=pri[i];
pri[i]=pri[k];
pri[k]=temp;
}
wtavg = wt[0] = 0;
tatavg = tat[0] = bt[0];
for(i=1;i<n;i++)
{
wt[i] = wt[i-1] + bt[i-1];
tat[i] = tat[i-1] + bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\nPROCESS\t\tPRIORITY\tBURST TIME\tWAITING   TIME\tTURNAROUNDTIME");
for(i=0;i<n;i++)
printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],pri[i],bt[i],wt[i],tat[i]);
printf("\nAverage Waiting Time is --- %f",wtavg/n);
 printf("\nAverage Turnaround Time is --- %f",tatavg/n);
getch();
return 0;
}
```

OUTPUT:

```
Enter the number of processes --- 5
Enter the Burst Time & Priority of Process 0 --- 1 5
Enter the Burst Time & Priority of Process 1 --- 2 4
Enter the Burst Time & Priority of Process 2 --- 1 1
Enter the Burst Time & Priority of Process 3 --- 5 2
Enter the Burst Time & Priority of Process 4 --- 1 2

PROCESS         PRIORITY        BURST TIME      WAITING TIME    TURNAROUNDTIME
2                1               1               0               1
3                2               5               1               6
4                2               1               6               7
1                4               2               7               9
0                5               1               9               10
Average Waiting Time is --- 4.600000
Average Turnaround Time is --- 6.600000_
```

AIM:

ALGORITHM:

Step 1: Start
Step 2: Declare and initialize necessary variables and arrays.
Step 3: Input the number of processes (n) from the user.
Step 4: Iterate over each process from 0 to n-1 and input the arrival time and burst time.
Step 5: Set the remaining time for each process to the corresponding burst time.
Step 6: Print the table header for the turnaround time and waiting time.
Step 7: Set a flag process smallest to 9 indicating no valid process available initially.
Step 8: Run a loop until all processes are completed (remain! = n).
Step 9: Find the process with the smallest remaining time among the arrived processes.
Step 10: Decrement the remaining time for the selected process.
Step 11: If the remaining time of the selected process becomes 0, mark it as completed, calculate its end time, print the process details.
Step 12: Increment the time variable.
Step 13: After completing the loop, calculate and print the average waiting time and average turnaround time.
Step 14: Stop

PROGRAM:

```
#include<stdio.h>
int main()
{
 int at[10],bt[10],rt[10],endTime,i,smallest;
int remain=0,n,time,sum_wait=0,sum_turnaround=0;
clrscr();
printf("\n\tEnter no of Processes : ");
```

```
 scanf("%d",&n);
for(i=0;i<n;i++)
{
 printf("\n\tEnter arrival time for Process P%d : ",i+1);
 scanf("%d",&at[i]);
printf("\n\tEnter burst time for Process P%d : ",i+1);
scanf("%d",&bt[i]);
rt[i]=bt[i];
}
printf("\n\tProcess\t|Turnaround Time| Waiting Time\n\n");
rt[9]=9999;
for(time=0;remain!=n;time++)
{
smallest=9;
for(i=0;i<n;i++)
{
if(at[i]<=time && rt[i]<rt[smallest] && rt[i]>0)
{
smallest=i;
} }
rt[smallest]--;
 if(rt[smallest]==0)
{
remain++;
endTime=time+1;
printf("\n\tP[%d]\t|\t%d\t|\t%d",smallest+1,endTime-at[smallest],endTime-bt[smallest]-
at[smallest]);
sum_wait+=endTime-bt[smallest]-at[smallest];
sum_turnaround+=endTime-at[smallest];
}
}
printf("\n\n\tAverage waiting time = %f\n",sum_wait*1.0/5);
printf("\n\tAverage Turnaround time = %f",sum_turnaround*1.0/5);
getch();
return 0;
}
```

OUTPUT:

```
Enter no of Processes : 3

Enter arrival time for Process P1 : 0

Enter burst time for Process P1 : 5

Enter arrival time for Process P2 : 1

Enter burst time for Process P2 : 4

Enter arrival time for Process P3 : 2

Enter burst time for Process P3 : 6

Process |Turnaround Time| Waiting Time


P[1]    |       5       |       0
P[2]    |       8       |       4
P[3]    |       13      |       7

Average waiting time = 3.666667

Average Turnaround time = 8.666667
```

AIM:

ALGORITHM:

Step 1: Start
Step 2: Initialize arrays for process ID (p), burst time (bt), system/user process (su), waiting time (wt), and turnaround time (tat).
Step 3: Read the number of processes (n) from the user.
Step 4: Using a loop, read the burst time and system/user process for each process from the user and store it in the corresponding array.
Step 5: Using nested loops, sort the processes in ascending order based on them system/user process (su), so that system processes come before user processes.
Step 6: Initialize waiting time (wt) and turnaround time (tat) for the first process to 0.
Step 7: Using a loop, calculate the waiting time (wt) and turnaround time (tat) for each process based on the burst time and waiting time of the previous process.
Step 8: Calculate the average waiting time and average turnaround time using the formula: sum of all waiting/turnaround times divided by the number of processes.
Step 9: Print the process ID, system/user process, burst time, waiting time, and turnaround time for each process using a loop.
Step 10: Print the average waiting time and average turnaround time.
Step 11: Stop

PROGRAM:

```
#include<stdio.h>
int main()
{
int p[20],bt[20], su[20], wt[20],tat[20],i, k, n, temp;
float wtavg, tatavg;
clrscr();
printf("Enter the number of processes:");
scanf("%d",&n);
```

```
for(i=0;i<n;i++)
{
p[i] = i;
printf("Enter the Burst Time of Process %d:", i);
scanf("%d",&bt[i]);
printf("System/User Process (0/1) ? ");
scanf("%d", &su[i]);
}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(su[i] > su[k])
{
temp=p[i];
p[i]=p[k];
p[k]=temp;
temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;
temp=su[i];
su[i]=su[k];
su[k]=temp;
}
wtavg = wt[0] = 0;
tatavg = tat[0] = bt[0];
for(i=1;i<n;i++)
{
wt[i] = wt[i-1] + bt[i-1];
tat[i] = tat[i-1] + bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\nPROCESS\t SYSTEM/USER PROCESS \tBURST TIME\tWAITING TIME \t
TURNAROUND TIME");
for(i=0;i<n;i++)
printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],su[i],bt[i],wt[i],tat[i]);
printf("\nAverage Waiting Time is --- %f",wtavg/n);
printf("\nAverage Turnaround Time is --- %f",tatavg/n);
getch();
return 0;
}
```

OUTPUT:

```
Enter the number of processes:5
Enter the Burst Time of Process 0:8
System/User Process (0/1) ? 0
Enter the Burst Time of Process 1:15
System/User Process (0/1) ? 0
Enter the Burst Time of Process 2:6
System/User Process (0/1) ? 1
Enter the Burst Time of Process 3:4
System/User Process (0/1) ? 0
Enter the Burst Time of Process 4:12
System/User Process (0/1) ? 1

PROCESS  SYSTEM/USER PROCESS    BURST TIME      WAITING TIME    TURNAROUND TIME
0                0               8               0               8
1                0               15              8               23
3                0               4               23              27
2                1               6               27              33
4                1               12              33              45
Average Waiting Time is --- 18.200000
Average Turnaround Time is --- 27.200000
```

AIM:

ALGORITHM:

SERVER:

Step1: Initialize size of shared memory shmsize to 27.
Step2: Initialize key to 2013 (some random value).
Step 3: Create a shared memory segment using shmget with key & IPC_CREAT as parameter.
a. If shared memory identifier shmid is -1, then stop.
Step 4: Display shmid.
Step 5: Attach server process to the shared memory using shmmat with shmid as parameter.
a. If pointer to the shared memory is not obtained, then stop.
Step 6: Clear contents of the shared region using memset function.
Step 7: Write a–z onto the shared memory.
Step 8: Wait till client reads the shared memory contents
Step 9: Detach process from the shared memory using shmdt system call.
Step10: Remove shared memory from the system using shmctl with IPC_RMID argument

CLIENT:

Step1: Initialize size of shared memory shmsize to 27.
Step2: Initialize key to 2013 (same value as in server).
Step3: Obtain access to the same shared memory segment using same key.
a. If obtained then display the shmid else print "Server not started"
Step4: Attach client process to the shared memory using shmmat with shmid as parameter.
a. If pointer to the shared memory is not obtained, then stop.
Step5: Read contents of shared memory and print it.
Step6: After reading, modify the first character of shared memory to '*'

PROGRAM:

Server: /* Shared memory server – server.c */

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/un.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define shmsize 27
void main()
{
char c;
int shmid;
key_t key = 2013;
char *shm, *s;
if ((shmid = shmget(key, shmsize, IPC_CREAT|0666)) < 0)
{
perror("shmget");
exit(1);
}
printf("Shared memory id : %d\n", shmid);
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
{
perror("shmat");
exit(1);
}memset(shm, 0, shmsize);
s = shm;
printf("Writing (a-z) onto shared memory\n");
for (c = 'a'; c <= 'z'; c++)
*s++ = c;
*s = '\0';
while (*shm != '*');
printf("Client finished reading\n");
if(shmdt(shm) != 0)
fprintf(stderr, "Could not close memory segment.\n");
shmctl(shmid, IPC_RMID, 0);
}
```

PROGRAM:

Client: /* Shared memory client - client.c */

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define shmsize 27
void main()
{

int shmid;
key_t key = 2013;
char *shm, *s;
if ((shmid = shmget(key, shmsize, 0666)) < 0)
{
printf("Server not started\n");
exit(1);
}
else
printf("Accessing shared memory id : %d\n",shmid);
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
{
perror("shmat");
exit(1);
}
 printf("Shared memory contents:\n");
for (s = shm; *s != '\0'; s++)
putchar(*s);
putchar('\n');
*shm = '*';
}
```

OUTPUT:

SERVER OUTPUT:

```
cse@ubuntu2:~/Desktop/cseb$ gcc server.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out
Shared memory id : 131079
Writing (a-z) onto shared memory
Client finished reading
cse@ubuntu2:~/Desktop/cseb$
```

CLIENT OUTPUT:

```
cse@ubuntu2:~/Desktop/cseb$ gcc client.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out
Accessing shared memory id : 131079
Shared memory contents:
abcdefghijklmnopqrstuvwxyz
cse@ubuntu2:~/Desktop/cseb$
```

AIM:

ALGORITHM:

Step 1: Start the program.
Step 2: Declare the required variables.
Step 3: Initialize the buffer size and get maximum item you want to produce.
Step 4: Get the option, which you want to do either producer, consumer or exit from the operation.
Step 5: If you select the producer, check the buffer size if it is full the producer should not produce the item or otherwise produce the item and increase the value buffer size.
Step 6: If you select the consumer, check the buffer size if it is empty the consumer should not consume the item or otherwise consume the item and decrease the value of buffer size.
Step 7: If you select exit come out of the program. Step 8: Stop the program.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
int mutex =1,full =0,empty =3,x=0;
void main()
{
int n;
void producer();
void consumer();
int wait(int);
int signal(int);
clrscr();
printf("\n1.Producer\n2.consumer\n3.exit\n");
while(1)
{
```

```
printf("\nEnter your choice: ");
scanf("%d",&n);
switch(n)
{
case 1:
if((mutex == 1)&&(empty!=0))
producer();
else
printf("\nBuffer is full\n");
break;
case 2:
if((mutex ==1)&&(full!=0))
consumer();
else
printf("\nBuffer is empty\n");
break;
case 3:
exit(0);
break;
}}}
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 items %d",x);
mutex =signal(mutex);
}
void consumer()
{
mutex =wait(mutex);
full =wait(full);
```

empty =signal(empty);

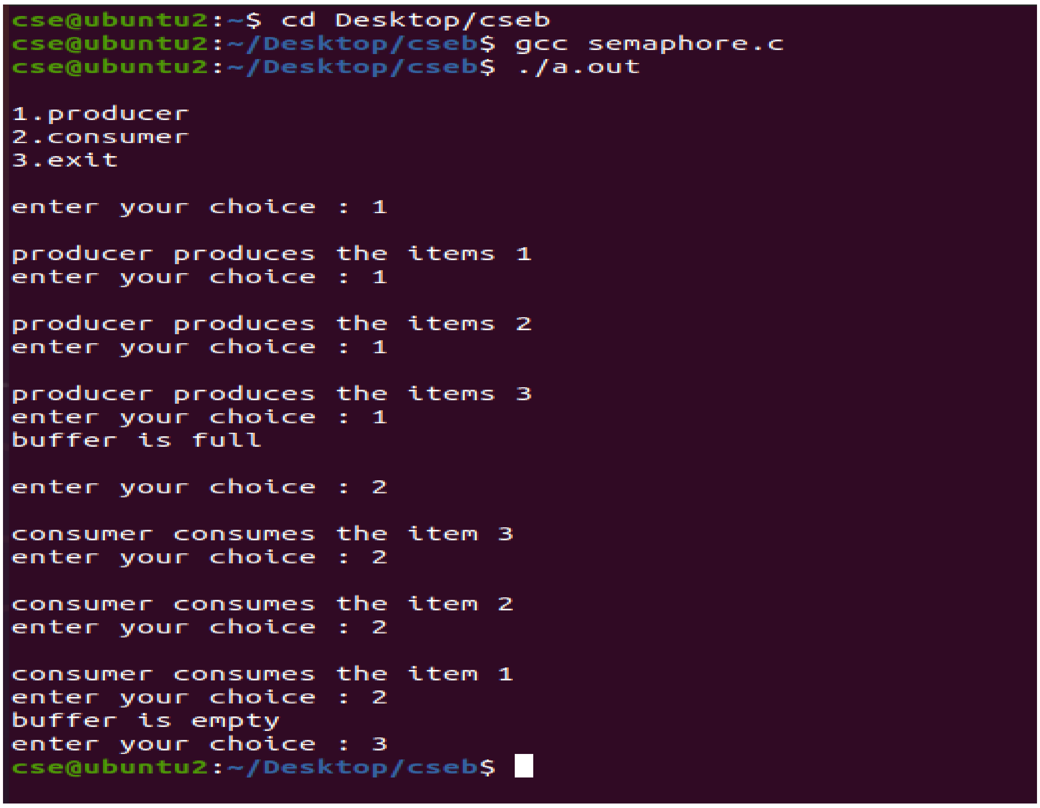
printf("\nconsumer consumes The item %d ",x);

x--;

mutex =signal(mutex);

}

<u>**OUTPUT:**</u>



```
cse@ubuntu2:~$ cd Desktop/cseb
cse@ubuntu2:~/Desktop/cseb$ gcc semaphore.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out

1.producer
2.consumer
3.exit

enter your choice : 1

producer produces the items 1
enter your choice : 1

producer produces the items 2
enter your choice : 1

producer produces the items 3
enter your choice : 1
buffer is full

enter your choice : 2

consumer consumes the item 3
enter your choice : 2

consumer consumes the item 2
enter your choice : 2

consumer consumes the item 1
enter your choice : 2
buffer is empty
enter your choice : 3
cse@ubuntu2:~/Desktop/cseb$
```

AIM:

ALGORITHM:

Step 1: Start the program.
Step 2: Declare the memory for the process.
Step 3: Read the number of processes, resources, allocation matrix and available matrix.
Step 4: Compare each and every process using the banker's algorithm.
Step 5: If the process is in safe state, then it is a not a deadlock process otherwise it is a deadlock process
Step 6: produce the result of state of process
Step 7: Stop the program

PROGRAM:

```
#include<stdio.h>
int max[100][100];
int alloc[100][100];
int need[100][100];
int avail[100];
int n,r;
void input();
void show();
void cal();
int main()
{
int i,j;
printf("********** Banker's Algorithm ************\n");
input();
show();
cal();
```

```
 return 0;
}
void input()
{
int i,j;
printf("Enter the no of Processes\t");
scanf("%d",&n);
printf("Enter the no of resources instances\t");
scanf("%d",&r);
printf("Enter the Max Matrix\n");
for(i=0;i<n;i++) {
for(j=0;j<r;j++) {
scanf("%d",&max[i][j]);
}}
printf("Enter the Allocation Matrix\n");
for(i=0;i<n;i++) {
for(j=0;j<r;j++) {
scanf("%d",&alloc[i][j]);
}}
printf("Enter the available Resources\n");
for(j=0;j<r;j++) {
scanf("%d",&avail[j]);
}}
void show() {
int i,j;
printf("Process\t Allocation\t Max\t Available\t");
for(i=0;i<n;i++) {
printf("\nP%d\t ",i+1);
for(j=0;j<r;j++) {
printf("%d ",alloc[i][j]); }
printf("\t\t");
for(j=0;j<r;j++) {
printf("%d ",max[i][j]); }
printf("\t");
if(i==0) {
for(j=0;j<r;j++)
printf("%d ",avail[j]);
}}}
void cal()
{
int finish[100],temp,need[100][100],flag=1,k,c1=0;
```

```
int safe[100];
int i,j;
for(i=0;i<n;i++) {
finish[i]=0; }
//find need matrix
for(i=0;i<n;i++) {
for(j=0;j<r;j++) {
need[i][j]=max[i][j]-alloc[i][j];
}}
printf("\n");
while(flag)
 {
flag=0;
for(i=0;i<n;i++) {
int c=0;
for(j=0;j<r;j++) {
if((finish[i]==0)&&(need[i][j]<=avail[j])) {
c++;
if(c==r) {
for(k=0;k<r;k++) {
avail[k]+=alloc[i][j];
finish[i]=1;
flag=1; }
printf("P%d->",i);
if(finish[i]==1) {
i=n;
}}}}}}
for(i=0;i<n;i++) {
if(finish[i]==1) {
c1++;
}
else
{
printf("P%d->",i);
}}
if(c1==n)
{printf("\n The system is in safe state\n");
}
else
{
printf("\n Process are in dead lock\n");
```

```
printf("\n System is in unsafe state\n");
} }
```

<u>OUTPUT:</u>

```
cse@ubuntu2:~/Desktop/cseb$ gcc bankers.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out
********** Banker's Algorithm ************
Enter the no of Processes       5
Enter the no of resources instances      3
Enter the Max Matrix
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter the Allocation Matrix
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter the available Resources
3 3 2
Process  Allocation      Max     Available
P1       0 1 0          7 5 3    3 3 2
P2       2 0 0          3 2 2
P3       3 0 2          9 0 2
P4       2 1 1          2 2 2
P5       0 0 2          4 3 3
P1->P3->P4->P2->P0->
 The system is in safe state
cse@ubuntu2:~/Desktop/cseb$ 
```


AIM:

ALGORITHM:

Step 1: Declare and initialize the necessary variables and arrays for the program.
Step 2: Input the user's number of processes and resource instances.
Step 3: Input the Max matrix, Allocation matrix, and Available Resources matrix from the user.
Step 4: Show the input matrices and available resources matrix on the console.
Step 5: Calculate the Need matrix by subtracting the Allocation matrix from the Max matrix.
Step 6: Initialize the Finish array to all 0s.
Step 7: While there is still a process that is not finished:
a) Look for a process that has not finished and whose resource needs are less than or equal to the available resources.
b) If such a process is found, mark it as finished, add its allocated resources back to the available resources, and continue the loop.
c) If no such process is found, the system is in deadlock. Print out the processes that are deadlocked and terminate the program.
Step 8: If all processes are finished, print out that no deadlock occurred and terminate the program.

PROGRAM:

```
#include <stdio.h>
int max[100][100];
int alloc[100][100];
int need[100][100];
int avail[100];
int n, r;
void input();
void show();
void cal();
```

```
int main()
{
int j,i;
printf("*****Deadlock Detection Algorithm*****\n");
input();
show();
cal();
return 0;
}
void input()
{
int i,j;
printf("Enter the no of Processes\t");
scanf("%d",&n);
printf("Enter the no of resource instances\t");
scanf("%d",&r);
printf("Enter the Max Matrix\n");
for(i=0;i<n;i++)
for(j=0;j<r;j++)
scanf("%d",&max[i][j]);
printf("Enter the Allocation Matrix\n");
for(i=0;i<n;i++)
for(j=0;j<r;j++)
scanf("%d",&alloc[i][j]);
printf("Enter the Available Resources\n");
for(j=0;j<r;j++)
scanf("%d",&avail[j]);
}
void show()
{

int i,j;
printf("Process\t Allocation\t Max\t   Available\t\n");
for(i=0;i<n;i++)
{
printf("P%d\t ",i+1);
for(j=0;j<r;j++)
{
printf("%d  ",alloc[i][j]);
}
printf("\t");
```

```
for(j=0;j<r;j++)
{
printf("%d ",max[i][j]);
}
printf("\t");
if(i==0)
{
for(j=0;j<r;j++)
printf("%d ",avail[j]);
}
printf("\n");
}
}
void cal()
{
int finish[100],temp,flag=1,k,cl=0;
int dead[100];
int safe[100];
int i,j;
for(i=0;i<n;i++)
{
finish[i]=0;
}
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{
need[i][j]=max[i][j]-alloc[i][j];
}
}
while(flag){
flag=0;
for(i=0;i<n;i++)
{
int c=0;
for(j=0;j<r;j++)
{
if((finish[i]==0) && (need[i][j] <= avail[j]))
{
c++;
if(c==r)
```

```
{
for(k=0;k<r;k++)
{
avail[k]+=alloc[i][j];
finish[i]=1;
flag=1;
}
if(finish[i]==1)
{
i=n;
}}
}}
}}
j=0;
flag=0;
for(i=0;i<n;i++)
{
if(finish[i]==0)
{
dead[j]=i;
j++;
flag=1;
}
}
if(flag==1)
{
printf("\n\nSystem is in Deadlock and the Deadlock processes are\n");


for(i=0;i<j;i++)
{
printf("P%d\t",dead[i]+1);
}
printf("\n");
}
else
{
printf("\nNo Deadlock Occurs");
}
}
```

OUTPUT:

```
cse@ubuntu2:~/Desktop/cseb$ gcc dedlock_detection.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out
*****Deadlock Detection Algorithm*****
Enter the no of Processes       3
Enter the no of resource instances      3
Enter the Max Matrix
3 6 0
4 3 3
3 4 4
Enter the Allocation Matrix
3 3 3
2 0 3
1 2 4
Enter the Available Resources
1 2 0
Process  Allocation      Max       Available
P1       3  3  3        3 6 0   1 2 0
P2       2  0  3        4 3 3
P3       1  2  4        3 4 4


System is in Deadlock and the Deadlock processes are
P1      P2      P3
cse@ubuntu2:~/Desktop/cseb$
```

AIM:

ALGORITHM:

Step 1: Define a constant "NUM_THREADS" to specify the number of threads to be created.
Step 2: Define a function "print_hello" that takes a thread ID as an argument and prints a greeting message.
Step 3: In the main function, declare an array of "pthread_t" structures to represent the threads and an integer variable "rc" to store the return value of the "pthread_create" function.
Step 4: Using a loop, create "NUM_THREADS" threads, passing the "print_hello" function and the thread ID as arguments to each thread.
Step 5: Check if the "pthread_create" function returns any error, if so, print an error message and exit the program.
Step 6: Call the "pthread_exit" function to terminate the main thread and wait for all other threads to finish their execution.
Step 7: Return 0 to indicate successful execution of the program.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 5
void *print_hello(void *thread_id)
{
long tid = (long)thread_id;
printf("Hello from thread %ld\n", tid);
pthread_exit(NULL);
}
int main()
{
```

```
pthread_t threads[NUM_THREADS];
int rc;
long t;
for (t = 0; t < NUM_THREADS; t++) {
printf("Creating thread %ld\n", t);
rc = pthread_create(&threads[t], NULL, print_hello, (void*)t);
if (rc) {
printf("ERROR; return code from pthread_create() is %d\n", rc);
exit(EXIT_FAILURE);
}
}
pthread_exit(NULL);
return 0;
}
```

OUTPUT:

```
cse@ubuntu2:~/Desktop/cseb$ gcc thread.c -o thread -lpthread
cse@ubuntu2:~/Desktop/cseb$ ./thread
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Hello from thread 2
Creating thread 4
Hello from thread 1
Hello from thread 0
Hello from thread 3
Hello from thread 4
cse@ubuntu2:~/Desktop/cseb$
```


AIM:

ALGORITHM:

Step 1: Declare necessary variables, including size, n, pgno, pagetable, ra, ofs, and frameno.
Step 2: Read in the process size from the user.
Step 3: Calculate the number of pages required for the process by dividing the size by the page size (4KB).
Step 4: Round up the number of pages to the nearest integer using the ceil function from the math.h library.
Step 5: Read in the relative address in hexadecimal format from the user.
Step 6: Calculate the page number and offset from the relative address.
Step 7: Display the page number and the current contents of the page table.
Step 8: Get the frame number corresponding to the current page number from the page table.
Step 9: Calculate the physical address by concatenating the frame number and the offset.
Step 10: Display the physical address.

PROGRAM:

```
#include<stdio.h>
#include<math.h>
void main() {
int size, n, pgno, pagetable[3] = {5, 6, 7}, i, j, physical_address,frameno;
double ml;
int ra = 0, ofs;
printf("Enter process size (in KB of max 12KB): ");
scanf("%d", &size);
ml = size / 4.0;
n = ceil(ml);
printf("Total No. of pages: %d\n", n);
```

```
printf("\nEnter relative address (in hexa): ");
scanf("%x", &ra);
pgno = ra / 4096;
ofs = ra % 4096;
printf("Page no = %d\n", pgno);
printf("Page table:\n");
for(i = 0; i < n; i++) {
printf("Pageno %d -->Frame %d\n", i, pagetable[i]);
}
frameno = pagetable[pgno];
physical_address =frameno*4096+ofs;
printf("\nPhysical address: %d\n", physical_address);
}
```

OUTPUT:

```
cse@ubuntu2:~/Desktop/cseb$ gcc paging.c -o paging -lm
cse@ubuntu2:~/Desktop/cseb$ ./paging
Enter process size (in KB of max 12KB): 10
Total No. of pages: 3

Enter relative address (in hexa): 2A10
Page no = 2
Page table:
Pageno 0 -->Frame 5
Pageno 1 -->Frame 6
Pageno 2 -->Frame 7

Physical address: 31248
cse@ubuntu2:~/Desktop/cseb$
```


AIM:

ALGORITHM:

Step 1: Start.
Step 2: Define the max as 25.
Step 3: Declare the variable frag[max],b[max],f[max],i,j,nb,nf,temp, highest=0, bf[max],ff[max], .
Step 4: Get the number of blocks,files,size of the blocks using for loop.
Step 5: In for loop check bf[j]!=1, if so temp=b[j]-f[i]
Step 6: Check highest
Step 7: Stop.

PROGRAM:

```
#include <stdio.h>
#define MAX 25

void main() {
int frag[MAX], b[MAX], f[MAX], i, j, nb, nf;
static int bf[MAX], ff[MAX];
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 = 0; i < nb; i++) {
printf("Block %d: ", i + 1);
scanf("%d", &b[i]);
}
printf("Enter the size of the files:\n");
```

```c
for (i = 0; i < nf; i++) {
printf("File %d: ", i + 1);
scanf("%d", &f[i]);
}
for (i = 0; i < nf; i++) {
for (j = 0; j < nb; j++) {
if (bf[j] == 0 && b[j] >= f[i]) {
ff[i] = j;
bf[j] = 1;
frag[i] = b[j] - f[i];
break;
}
}
}
printf("\nFile No\tFile Size\tBlock No\tBlock Size\tFragment");
for (i = 0; i < nf; i++) {
printf("\n%d\t%d\t\t", i + 1, f[i]);
if (ff[i] != 0 || b[0] >= f[i])
printf("%d\t\t%d\t\t%d\n", ff[i] + 1, b[ff[i]], frag[i]);
 else
printf("Not Allocated");
}
}
```

OUTPUT:

```
cse@ubuntu2:~/Desktop/cseb$ gcc firstfit.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out

        Memory Management Scheme - First Fit
Enter the number of blocks: 5
Enter the number of files: 4

Enter the size of the blocks:
Block 1: 2
Block 2: 3
Block 3: 6
Block 4: 7
Block 5: 5
Enter the size of the files:
F LibreOffice Writer
File 2: 2
File 3: 4
File 4: 6

File No File Size       Block No        Block Size      Fragment
1       4               3               6               2

2       2               1               2               0

3       4               4               7               3

4       6               Not Allocated
cse@ubuntu2:~/Desktop/cseb$
```


AIM:

ALGORITHM:

Step 1: Start
Step 2: Define the maximum size MAX = 25
Step 3: Declare the variables:
frag[MAX], b[MAX], f[MAX], i, j, nb, nf, temp, worstIndex
bf[MAX] = {0}, ff[MAX] = {-1}
Step 4: Input the number of memory blocks nb and the number of files nf
Step 5: Input the size of each memory block into array b[]
Step 6: Input the size of each file into array f[]
Step 7: For each file i = 0 to nf - 1, do the following:
- **a.** Initialize worstIndex = -1
- **b.** For each block j = 0 to nb - 1:
  - i. If bf[j] == 0 (block is free) and b[j] >= f[i]:
    - If worstIndex == -1 or b[j] > b[worstIndex], update worstIndex = j
- **c.** If worstIndex != -1:
  - Assign ff[i] = worstIndex
  - Assign frag[i] = b[worstIndex] - f[i]
  - Mark block as used: bf[worstIndex] = 1
- **d.** Else:
  - File is not allocated; ff[i] = -1, frag[i] = -1

Step 8: Repeat Step 7 for all files
Step 9: Display for each file:
File Number, File Size, Block Number, Block Size, Fragment
Step 10: Stop the program

PROGRAM:

```
#include <stdio.h>
#define MAX 25
```

```
void main() {
int frag[MAX], b[MAX], f[MAX], i, j, nb, nf, temp, worstIndex;
static int bf[MAX], ff[MAX];
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 = 0; i < nb; i++) {
printf("Block %d: ", i + 1);
scanf("%d", &b[i]);
}
printf("Enter the size of the files:\n");
for (i = 0; i < nf; i++) {
printf("File %d: ", i + 1);
scanf("%d", &f[i]);
}
for (i = 0; i < nf; i++) {
worstIndex = -1;
for (j = 0; j < nb; j++) {
if (bf[j] == 0 && b[j] >= f[i]) {
if (worstIndex == -1 || b[j] > b[worstIndex]) {
worstIndex = j;
}
}
}
if (worstIndex != -1) {
ff[i] = worstIndex;
frag[i] = b[worstIndex] - f[i];
bf[worstIndex] = 1;
} else {
ff[i] = -1;
frag[i] = -1;
}
}
printf("\nFile No\tFile Size\tBlock No\tBlock Size\tFragment");
for (i = 0; i < nf; i++) {
printf("\n%d\t%d\t\t", i + 1, f[i]);
if (ff[i] != -1)
printf("%d\t\t%d\t\t%d\n", ff[i] + 1, b[ff[i]], frag[i]);
```

```
else
printf("Not Allocated\t-\t\t-\n");
}
}
```

<u>OUTPUT:</u>

```
cse@ubuntu2:~/Desktop/cseb$ gcc worstfit.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out

        Memory Management Scheme - Worst Fit
Enter the number of blocks: 5
Enter the number of files: 4

Enter the size of the blocks:
Block 1: 2
Block 2: 3
Block 3: 4
Block 4: 6
Block 5: 7
Enter the size of the files:
File 1: 5
File 2: 3
File 3: 2
File 4: 4

File No File Size       Block No        Block Size      Fragment
1       5               5               7               2

2       3               4               6               3

3       2               3               4               2

4       4               Not Allocated   -               -
cse@ubuntu2:~/Desktop/cseb$
```

AIM:

ALGORITHM:

Step 1: Start
Step 2: Define MAX = 25
Step 3: Declare variables: frag[MAX], b[MAX], f[MAX], bf[MAX], ff[MAX], i, j, nb, nf, temp, lowest
Step 4: Input number of blocks nb and number of files nf
Step 5: Input the size of each block into array b[]
Step 6: Input the size of each file into array f[]
Step 7: For each file i = 0 to nf - 1
a.Set lowest = very large number (e.g., 10000)
b.For each block j = 0 to nb - 1
If bf[j] == 0 (block is free) and b[j] >= f[i]
If b[j] - f[i] < lowest, then
lowest = b[j] - f[i]
ff[i] = j
File is not allocated (ff[i] = -1, frag[i] = -1)
Step 8: Repeat for all files
Step 9: Display: File No, File Size, Block No, Block Size, Fragment
Step 10: Stop

PROGRAM:

```
#include <stdio.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];
printf("\n\tMemory Management Scheme - Best 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 = 0; i < nb; i++) {
printf("Block %d: ", i + 1);
scanf("%d", &b[i]);
}
printf("Enter the size of the files :-\n");
for (i = 0; i < nf; i++) {
printf("File %d: ", i + 1);
scanf("%d", &f[i]);
}
for (i = 0; i < nf; i++) {
lowest = 10000;
for (j = 0; j < nb; j++) {
if (bf[j] != 1) {
temp = b[j] - f[i];
if (temp >= 0 && temp < lowest) {
ff[i] = j;
lowest = temp;
}
}
}
if (lowest != 10000) {
frag[i] = lowest;
bf[ff[i]] = 1;
} else {
ff[i] = -1;
frag[i] = -1;
}
}
printf("\nFile No\tFile Size\tBlock No\tBlock Size\tFragment");
for (i = 0; i < nf; i++) {
printf("\n%d\t\t%d\t", i + 1, f[i]);
if (ff[i] != -1)
printf("\t%d\t\t%d\t\t%d\n", ff[i] + 1, b[ff[i]], frag[i]);
else
printf("\tNot Allocated\t--\t\t--\n");
}
}
```

OUTPUT:

```
cse@ubuntu2:~/Desktop/cseb$ gcc bestfit.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out

        Memory Management Scheme - Best Fit
Enter the number of blocks: 5
Enter the number of files: 4

Enter the size of the blocks:-
Block 1: 3
Block 2: 6
Block 3: 7
Block 4: 5
Block 5: 3
Enter the size of the files :-
File 1: 7
File 2: 4
File 3: 2
File 4: 5

File No File Size       Block No        Block Size      Fragment
1               7               3               7               0

2               4               4               5               1

3               2               1               3               1

4               5               2               6               1
cse@ubuntu2:~/Desktop/cseb$
```

AIM:

ALGORITHM:

Step 1: Start the process.
Step 2: Declare the size with respect to page length.
Step 3: Check the need of replacement from the page to memory.
Step 4: Check the need of replacement from old page to new page in memory.
Step 5: Form a queue to hold all pages.
Step 6: Insert the page require memory into the queue.
Step 7: Check for bad replacement and page fault.
Step 8: Get the number of processes to be inserted.
Step 9: Display the values.
Step 10: Stop the process.

PROGRAM:

```
#include<stdio.h>
int main() {
int i, j, n, a[50], frame[10], no, k, avail, count = 0;
printf("\n ENTER THE NUMBER OF PAGES:\n");
scanf("%d", &n);
printf("\n ENTER THE PAGE NUMBERS:\n");
for (i = 1; i <= n; i++) {
scanf("%d", &a[i]);
}
printf("\n ENTER THE NUMBER OF FRAMES: ");
scanf("%d", &no);
for (i = 0; i < no; i++) {
frame[i] = -1;
}
j = 0;
printf("\nReference String\tPage Frames\n");
for (i = 1; i <= n; i++) {
```

```
printf("%d\t\t", a[i]);
avail = 0;
for (k = 0; k < no; k++) {
if (frame[k] == a[i]) {
avail = 1;
break;
}
}
if (avail == 0) {
frame[j] = a[i];
j = (j + 1) % no;
count++;
}
for (k = 0; k < no; k++) {
if (frame[k] == -1) {
printf("-\t");
} else {
printf("%d\t", frame[k]);
}}
printf("\n");
}
printf("Page Faults: %d\n", count);
return 0;
}
```

OUTPUT:

```
cse@ubuntu2:~/Desktop/cseb$ gcc fifo.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out

 ENTER THE NUMBER OF PAGES:
10

 ENTER THE PAGE NUMBERS:
7 8 4 5 3 7 4 5 2 7

 ENTER THE NUMBER OF FRAMES: 3

Reference String        Page Frames
7               7       -       -
8               7       8       -
4               7       8       4
5               5       8       4
3               5       3       4
7               5       3       7
4               4       3       7
5               4       5       7
2               4       5       2
7               7       5       2
Page Faults: 10
cse@ubuntu2:~/Desktop/cseb$
```

AIM:

ALGORITHM:

Step 1: Start the process.
Step 2: Declare the size with respect to page length.
Step 3: Check the need of replacement from the page to memory.
Step 4: Check the need of replacement from old page to new page in memory.
Step 5: Form a queue to hold all pages.
Step 6: Insert the page require memory into the queue.
Step 7: Check for bad replacement and page fault.
Step 8: Get the number of processes to be inserted.
Step 9: Display the values.
Step 10: Stop the process.

PROGRAM:

```
int main() {
int n, pg[30], fr[10];
int count[10], i, j, k, fault, f, flag, temp, current, c, dist, max, m, cnt, p, x;
fault = 0;
dist = 0;
k = 0;
printf("Enter the total number of pages: ");
scanf("%d", &n);
printf("Enter the page reference sequence:\n");
for(i = 0; i < n; i++) {
scanf("%d", &pg[i]);
}
printf("\nEnter frame size: ");
scanf("%d", &f);
for(i = 0; i < f; i++) {
count[i] = 0;
```

```
fr[i] = -1;
}
printf("\nPage reference sequence\tPage frames\n");
for(i = 0; i < n; i++) {
flag = 0;
temp = pg[i];
for(j = 0; j < f; j++) {
if(temp == fr[j]) {
flag = 1;
break;
}
}
if(flag == 0 && k < f) {
fault++;
fr[k] = temp;
k++;
} else if(flag == 0 && k == f) {
fault++;
for(cnt = 0; cnt < f; cnt++) {
current = fr[cnt];
for(c = i + 1; c < n; c++) {
if(current != pg[c]) {
count[cnt]++;
} else {
break;
}
}
}
max = -1;
for(m = 0; m < f; m++) {
if(count[m] > max) {
max = count[m];
p = m;
}
}
fr[p] = temp;
}
printf("\nPage %d\t\t", pg[i]);
for(x = 0; x < f; x++) {
printf("%d\t", fr[x]);
}
```

```
for(x = 0; x < f; x++) {
count[x] = 0;
}
}
printf("\nTotal number of page faults: %d\n", fault);
return 0;
}
```

OUTPUT:

```
cse@ubuntu2:~/Desktop/cseb$ gcc optimal.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out
Enter the total number of pages: 10
Enter the page reference sequence:
6 5 4 3 7 5 4 6 7 4

Enter frame size: 2

Page reference sequence Page frames

Page 6          6       -1
Page 5          6       5
Page 4          4       5
Page 3          3       5
Page 7          7       5
Page 5          7       5
Page 4          7       4
Page 6          7       6
Page 7          7       6
Page 4          4       6
Total number of page faults: 8
cse@ubuntu2:~/Desktop/cseb$
```

AIM:

ALGORITHM:

Step 1: Start the process.
Step 2: Declare the size with respect to page length.
Step 3: Check the need of replacement from the page to memory.
Step 4: Check the need of replacement from old page to new page in memory.
Step 5: Form a queue to hold all pages.
Step 6: Insert the page require memory into the queue.
Step 7: Check for bad replacement and page fault.
Step 8: Get the number of processes to be inserted.
Step 9: Display the values.
Step 10: Stop the process.

PROGRAM:

```
#include <stdio.h>
int findLRU(int time[], int n) {
int i, minimum = time[0], pos = 0;
for(i = 1; i < n; ++i) {
if(time[i] < minimum) {
minimum = time[i];
pos = i;
}
}
return pos;
}
int main() {
int no_of_frames, no_of_pages, frames[10], pages[30], counter = 0, time[10], flag1, flag2,
i, j, pos, faults = 0;
printf("Enter number of frames: ");
scanf("%d", &no_of_frames);
```

```
printf("Enter number of pages: ");
scanf("%d", &no_of_pages);
printf("Enter reference string: ");
for(i = 0; i < no_of_pages; ++i) {
scanf("%d", &pages[i]);
}
for(i = 0; i < no_of_frames; ++i) {
frames[i] = -1;
}
printf("\nPage reference sequence:\tPage frames\n");
for(i = 0; i < no_of_pages; ++i) {
flag1 = flag2 = 0;
for(j = 0; j < no_of_frames; ++j) {
if(frames[j] == pages[i]) {
counter++;cl
time[j] = counter;
flag1 = flag2 = 1;
break;
}
}
if(flag1 == 0) {
for(j = 0; j < no_of_frames; ++j) {
if(frames[j] == -1) {
counter++;
faults++;
frames[j] = pages[i];
time[j] = counter;
flag2 = 1;
break;
}}}
if(flag2 == 0) {
pos = findLRU(time, no_of_frames);
counter++;
faults++;
frames[pos] = pages[i];
time[pos] = counter;
}
printf("%d\t\t", pages[i]);
for(j = 0; j < no_of_frames; ++j) {
printf("%d\t", frames[j]);
}
```

```
printf("\n");
}
printf("\nTotal Page Faults = %d\n", faults);
return 0;
}
```

OUTPUT:

```
cse@ubuntu2:~/Desktop/cseb$ gcc lru.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out
Enter number of frames: 3
Enter number of pages: 12
Enter reference string: 3 2 4 10 5 3 10 5 6 7 8 4

Page reference sequence:        Page frames
3                3       -1      -1
2                3       2       -1
4                3       2       4
10               10      2       4
5                10      5       4
3                10      5       3
10               10      5       3
5                10      5       3
6                10      5       6
7                7       5       6
8                7       8       6
4                7       8       4

Total Page Faults = 10
cse@ubuntu2:~/Desktop/cseb$
```

AIM:

ALGORITHM:

Step 1: Start the program.
Step 2: Get the number of directories.
Step 3: Get the names of directories.
Step 4: Get the number of files in each directory.
Step 5: Read the names of files in each directory.
Step 6: If the entered name exists read a new name.
Step 7: Display directories along with its files.
Step 8: Stop the program.

PROGRAM:

```
#include<stdio.h>
#include<string.h>
void main ()
{
int nf=0,i=0,j=0,ch;
char mdname[10],fname[10][10],name[10];
printf("Enter the directory name:");
scanf("%s",mdname);
printf("Enter the number of files:");
scanf("%d",&nf);
do
{
printf("Enter file name to be created:");
scanf("%s",name);
for(i=0;i<nf;i++)
{
if(!strcmp(name,fname[i]))
break;
}
```

```
if(i==nf)
{
strcpy(fname[j++],name);
nf++;
}
else
printf("There is already %s\n",name);
printf("Do you want to enter another file(yes - 1 or no - 0):");
scanf("%d",&ch);
}
while(ch==1);
printf("Directory name is:%s\n",mdname);
printf("Files names are:");
for(i=0;i<j;i++)
printf("\n%s",fname[i]);
printf("\n");
}
```

OUTPUT:

```
cse@ubuntu2:~/Desktop/cseb$ gcc singleleveldir.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out
Enter the directory name:os
Enter the number of files:2
Enter file name to be created:aaa
Do you want to enter another file(yes - 1 or no - 0):1
Enter file name to be created:bbb
Do you want to enter another file(yes - 1 or no - 0):0
Directory name is:os
Files names are:
aaa
bbb
cse@ubuntu2:~/Desktop/cseb$
```

AIM:

ALGORITHM:

Step 1: Start the program.
Step 2: Get the number of master file directories.
Step 3: Get the names of master file directories.
Step 4: Get the number of user file directories.
Step 5: Read the names of files in each user file directory.
Step 6: If the entered name exists read a new name.
Step 7: Display master file directories along with user file directories and the files inside each user file directory.
Step 8: Stop the program.

PROGRAM:

```
#include<stdio.h>
struct st{
char dname[20],sdname[20][20],fname[20][20][20];
int ds,sds[20];
}dir[20];
int main(){
int i,j,k,n;
printf("Enter number of master file directories :");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Enter Name of directory %d :",i+1);
122;
scanf("%s",dir[i].dname);
printf("Enter number of user file directories :");
scanf("%d",&dir[i].ds);
for(j=0;j<dir[i].ds;j++)
{
```

```
printf("Enter user file directory name and size : ");
scanf("%s",dir[i].sdname[j]);
scanf("%d",&dir[i].sds[j]);
for(k=0;k<dir[i].sds[j];k++)
{
printf("Enter file name :");
scanf("%s",dir[i].fname[j][k]);
}}}
printf("\n Master dir name\tsize\t sub dir name\t size\t files\n");
printf("\n****************************************\n");
for(i=0;i<n;i++)
{
printf("%s\t\t%d",dir[i].dname,dir[i].ds);
for(j=0;j<dir[i].ds;j++)
{
printf("\t%s\t\t%d\t",dir[i].sdname[j],dir[i].sds[j]);
for(k=0;k<dir[i].sds[j];k++)
printf("%s\t",dir[i].fname[j][k]);
printf("\n\t\t");
}
printf("\n");
}}
```

OUTPUT:

```
cse@ubuntu2:~/Desktop/cseb$ gcc twoleveldir.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out
Enter number of master file directories :2
Enter Name of directory 1 :dir1
Enter number of user file directories :2
Enter user file directory name and size : uf1 2
Enter file name :file1
Enter file name :file2
Enter user file directory name and size : uf2 2
Enter file name :file3
Enter file name :file4
Enter Name of directory 2 :dir2
Enter number of user file directories :3
Enter user file directory name and size : uf3 2
Enter file name :file5
Enter file name :file6
Enter user file directory name and size : uf4 2
Enter file name :file7
Enter file name :file8
Enter user file directory name and size : uf5 2
Enter file name :file9
Enter file name :file10

 Master dir name        size     sub dir name    size    files

****************************************
dir1            2       uf1             2       file1   file2
                        uf2             2       file3   file4

dir2            3       uf3             2       file5   file6
                        uf4             2       file7   file8
                        uf5             2       file9   file10

cse@ubuntu2:~/Desktop/cseb$
```

AIM:

ALGORITHM:

Step 1: Start the program.
Step 2: Get the number of memory partition and their sizes.
Step 3: Get the number of processes and values of block size for each process.
Step 4: First fit algorithm searches all the entire memory block until a hole which is big enough is encountered. It allocates that memory block for the requesting process.
Step 5: Best-fit algorithm searches the memory blocks for the smallest hole which can be allocated to requesting process and allocates it.
Step 6: Worst fit algorithm searches the memory blocks for the largest hole and allocates it to the process.
Step 7: Analyses all the three memory management techniques and display the best algorithm which utilizes the memory resources effectively and efficiently.
Step 8: Stop the program.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
int f[50], i, st, len, j, c, k, count = 0;
for (i = 0; i < 50; i++)
f[i] = 0;
printf("Files Allocated are:\n");
x:
count = 0;
printf("Enter starting block and length of files: ");
scanf("%d %d", &st, &len);
for (k = st; k < (st + len); k++)
{
```

```c
if (f[k] == 0)
count++;
}
if (len == count)
{
for (j = st; j < (st + len); j++)
{
if (f[j] == 0)
{
f[j] = 1;
printf("%d\t%d\n", j, f[j]);
}
}
if (j != (st + len - 1))
printf("The file is allocated to the disk\n");
}
else
{
printf("The file is not allocated\n");
}
printf("Do you want to enter more files? (Yes - 1 / No - 0): ");
scanf("%d", &c);
if (c == 1)
goto x;
}
```

OUTPUT:

```
cse@ubuntu2:~/Desktop/cseb$ gcc seqfileall.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out
Files Allocated are:
Enter starting block and length of files: 1 10
1       1
2       1
3       1
4       1
5       1
6       1
7       1
8       1
9       1
10      1
The file is allocated to the disk
Do you want to enter more files? (Yes - 1 / No - 0): 1
Enter starting block and length of files: 15 5
15      1
16      1
17      1
18      1
19      1
The file is allocated to the disk
Do you want to enter more files? (Yes - 1 / No - 0): 0
cse@ubuntu2:~/Desktop/cseb$
```

AIM:

ALGORITHM:

Step 1: Start the program
Step 2: Create a queue to hold all pages in memory
Step 3: When the page is required replace the page at the head of the queue
Step 4: Now the new page is inserted at the tail of the queue
Step 5: Create a stack
Step 6: When the page fault occurs replace page present at the bottom of the stack
Step 7: Stop the allocation.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
int f[50], p,i, st, len, j, c, k, a;
for(i=0;i<50;i++)
f[i]=0;
printf("Enter how many blocks already allocated: ");
scanf("%d",&p);
printf("Enter blocks already allocated: ");
for(i=0;i<p;i++)
{
scanf("%d",&a);
f[a]=1;
}
x: printf("Enter index starting block and length: ");
scanf("%d%d", &st,&len);
k=len;
if(f[st]==0){
```

```
for(j=st;j<(st+k);j++)
{
if(f[j]==0)
{
f[j]=1;
printf("%d-------->%d\n",j,f[j]);
}
else{
printf("%d Block is already allocated \n",j);
k++;
}}}
else
printf("%d starting block is already allocated \n",st);
printf("Do you want to enter more file(Yes - 1/No - 0)");
scanf("%d", &c);
if(c==1)
goto x;
else
exit(0);
}
```

OUTPUT:

```
cse@ubuntu2:~/Desktop/cseb$ gcc linkedfileall.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out
Enter how many blocks already allocated: 5
Enter blocks already allocated: 1 2 3 4 5
Enter index starting block and length: 0 5
0-------->1
1 Block is already allocated
2 Block is already allocated
3 Block is already allocated
4 Block is already allocated
5 Block is already allocated
6-------->1
7-------->1
8-------->1
9-------->1
Do you want to enter more file(Yes - 1/No - 0)0
cse@ubuntu2:~/Desktop/cseb$
```

AIM:

ALGORITHM:

Step 1: Start
Step 2: Let n be the size of the buffer
Step 3: check if there are any producer
Step 4: if yes check whether the buffer is full
Step 5: If no the producer item is stored in the buffer
Step 6: If the buffer is full the producer has to wait
Step 7: Check there is any consumer.If yes check whether the buffer is empty
Step 8: If no the consumer consumes them from the buffer
Step 9: If the buffer is empty, the consumer has to wait.
Step 10: Repeat checking for the producer and consumer till required
Step 11: Terminate the process.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
int f[50], index[50],i, n, st, len, j, c, k, ind,count=0;
for(i=0;i<50;i++)
f[i]=0;
x:printf("Enter the index block: ");
scanf("%d",&ind);
if(f[ind]!=1)
{
printf("Enter no of blocks needed and no of files for the index %d on the disk : \n", ind);
scanf("%d",&n);
}
else
```

```
{
printf("%d index is already allocated \n",ind);
goto x;
}
y: count=0;
for(i=0;i<n;i++)
{
scanf("%d", &index[i]);
if(f[index[i]]==0)
count++;
}
if(count==n)
{
for(j=0;j<n;j++)
f[index[j]]=1;
printf("Allocated\n");
printf("File Indexed\n");
for(k=0;k<n;k++)
printf("%d-------->%d : %d\n",ind,index[k],f[index[k]]);
}
else
{
printf("File in the index is already allocated \n");
printf("Enter another file indexed");
goto y;
}
printf("Do you want to enter more file(Yes - 1/No - 0)");
scanf("%d", &c);
if(c==1)
goto x;
else
exit(0);
}
```

OUTPUT:

```
cse@ubuntu2:~/Desktop/cseb$ gcc indexfileall.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out
Enter the index block: 0
Enter no of blocks needed and no of files for the index 0 on the disk :
5 1 2 3 4 5
Allocated
File Indexed
0-------->1 : 1
0-------->2 : 1
0-------->3 : 1
0-------->4 : 1
0-------->5 : 1
Do you want to enter more file(Yes - 1/No - 0)1
Enter the index block: 2
2 index is already allocated
Enter the index block: 6
Enter no of blocks needed and no of files for the index 6 on the disk :
2 7 10
Allocated
File Indexed
6-------->7 : 1
6-------->10 : 1
Do you want to enter more file(Yes - 1/No - 0)0
cse@ubuntu2:~/Desktop/cseb$
```


AIM:

ALGORITHM:

Step 1: Start
Step 2: Define function FCFS (First-come, First-Served)
Step 2.1: Traverse the disk request array in the order of arrival.
Step 2.2: For each request, calculate the seek time as the absolute difference between the current request and previous request
Step 2.3: Update the current head position to the current request
Step 2.4: Repeat until all requests are processed
Step 2.5: Display the sequence of disk access and the total seek time.
Step 3: Call the function fcfs ().
Step 4: Stop

PROGARM:

```
#include <stdio.h>
#include <stdlib.h>
#include<limits.h>
void fcfs(int disk[], int n, int head) {
int i , seek_time = 0;
printf("\nFCFS Scheduling:\n");
printf("Sequence of disk access:");
for (i = 0; i < n; i++) {
printf(" %d", disk[i]);
seek_time += abs(disk[i] - head);
head = disk[i];
}
printf("\nTotal Seek Time: %d\n", seek_time);
}
int main() {
int disk[] = { 98, 183, 37, 122, 14, 124, 65, 67 };
```

```
int n = sizeof(disk) / sizeof(disk[0]);
int head;
printf("Enter the initial position of the disk head: ");
scanf ("%d", &head);
fcfs(disk, n, head);
return 0;
}
```

OUTPUT:

```
cse@ubuntu2:~/Desktop/cseb$ gcc fcfs.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out
Enter the initial position of the disk head: 20

FCFS Scheduling:
Sequence of disk access: 98 183 37 122 14 124 65 67
Total Seek Time: 673
cse@ubuntu2:~/Desktop/cseb$ 
```

AIM:

ALGORITHM:

Step 1 : Start.
Step 2: Define function SSTF (Shortest Seek Time First)
Step 3: Initialize a visited array to keep track of visited disk requests
Step 4: Traverse the disk request array
Step 5: Find the request with the shortest seek time for the current head position that has not been visited yet
Step 6: Mark the selected request as visited
Step 7: Calculate the seek time as the absolute difference between the current request and the previous request
Step 8: Update the current head position to the current request
Step 9: Display the sequence of disk access and the total seek time.
Step 10: Call the function sstf()
Step 11: Stop

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include<limits.h>
void sstf(int disk[], int n, int head) {
int i, j, seek_time = 0;
int visited[n];
printf("\nSSTF Scheduling:\n");
printf("Sequence of disk access:");
for (i = 0; i < n; i++) {
int min_dist = INT_MAX;
int min_index = -1;
for (j = 0; j < n; j++) {
```

```
visited[j] = 0;
}

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

if (!visited[j] && abs(disk[j] - head) < min_dist) {
min_dist = abs(disk[j] - head);
min_index = j;
}}
visited[min_index] = 1;
printf(" %d", disk[min_index]);
seek_time += min_dist;
head = disk[min_index];
}
printf("\nTotal Seek Time: %d\n", seek_time);
}
int main() {
int disk[] = { 98, 183, 37, 122, 14, 124, 65, 67 };
int n = sizeof(disk) / sizeof(disk[0]);
int head;
printf("Enter the initial position of the disk head: ");
scanf ("%d", &head);
sstf(disk, n, head);
return 0;
}
```

<u>OUTPUT:</u>

```
cse@ubuntu2:~/Desktop/cseb$ gcc sstf.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out
Enter the initial position of the disk head: 25

SSTF Scheduling:
Sequence of disk access: 14 14 14 14 14 14 14 14
Total Seek Time: 11
cse@ubuntu2:~/Desktop/cseb$
```

AIM:

ALGORITHM:

Step 1: Start.
Step 2: Define the function SCAN
Step 3: Sort the disk request array in ascending order
Step 4: Find the index where the head is located
Step 5: Traverse the disk request array from the head position towards higher cylinder number
Step 6: Calculate the seek time as the absolute difference between the current request and the previous request
Step 7: Update the current head position to the current request
Step 8: Repeat until reaching the end of the disk
Step 9: Jump to the lower end of the disk
Step 10: Traverse the disk request array from the lower end towards the start
Step 11: Calculate the seek time as the absolute difference between the current request and the previous request
Step 12: Update the current head position to the current request
Step 13: Repeat until reaching the start of the disk
Step 14 : Display the sequence of disk access and the total seek time.
Step 15: Call the function scan().
Step 16: stop.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include<limits.h>
void scan(int disk[], int n, int head, int direction) {
int i, seek_time = 0;
int start = 0, end = n - 1;
```

```
printf("\nSCAN Scheduling:\n");
printf("Sequence of disk access:");
for (i = 0; i < n - 1; i++) {
int j;
for (j = 0; j < n - i - 1; j++) {
if (disk[j] > disk[j + 1]) {
int temp = disk[j];
disk[j] = disk[j + 1];
disk[j + 1] = temp;
}
}
}
for (i = 0; i < n; i++) {
if (disk[i] > head) {
start = i - 1;
break;
}
}
if (direction == 1) {
for (i = start; i <= end; i++) {
printf(" %d", disk[i]);
seek_time += abs(disk[i] - head);
head = disk[i];
}
}
else {
for (i = start; i >= 0; i--) {
printf(" %d", disk[i]);
seek_time += abs(disk[i] - head);
head = disk[i];
}
}
printf("\nTotal Seek Time: %d\n", seek_time);
}
int main() {
int disk[] = { 98, 183, 37, 122, 14, 124, 65, 67 };
int n = sizeof(disk) / sizeof(disk[0]);
int head;
printf("Enter the initial position of the disk head: ");
scanf ("%d", &head);
scan(disk, n,head, 1);
```

```
scan(disk, n,head, 0);
return 0;
}
```

OUTPUT:

```
cse@ubuntu2:~/Desktop/cseb$ gcc scan.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out
Enter the initial position of the disk head: 30

SCAN Scheduling:
Sequence of disk access: 14 37 65 67 98 122 124 183
Total Seek Time: 185

SCAN Scheduling:
Sequence of disk access: 14
Total Seek Time: 16
cse@ubuntu2:~/Desktop/cseb$
```

AIM:

ALGORITHM:

Step 1: Start.
Step 2: Display the function C-SCAN (Circular SCAN)
Step 3: Sort the disk request array in ascending order
Step 4: Find the index where the head is located
Step 5: Traverse the disk request array from head position towards higher cylinder numbers
Step 6: Calculate the seek time as the absolute difference between the current request and the previous request
Step 7: Update the current head position to the current request
Step 8: Repeat until reaching the end of the disk
Step 9: Jump to the lower end of the disk
Step 10: Traverse the disk request array from the lower end towards the start
Step 11: Calculate the seek time as the absolute difference between the current request and the previous request
Step 12: Update the current head position to the current request
Step 13: Repeat until reaching the start of the disk
Step 14: Display the sequence of disk access and the total seek time
Step 15: Call the function cscan().
Step 16:stop.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include<limits.h>
void cscan(int disk[], int n, int head) {
int i, seek_time = 0;
int start = 0, end = n - 1;
printf("\nC-SCAN Scheduling:\n");
```

```
printf("Sequence of disk access:");
for (i = 0; i < n - 1; i++) {
int j;
for (j = 0; j < n - i - 1; j++) {
if (disk[j] > disk[j + 1]) {
int temp = disk[j];
disk[j] = disk[j + 1];
disk[j + 1] = temp;
}
}
}
for (i = 0; i < n; i++) {
if (disk[i] > head) {
start = i;
break;
}
}
for (i = start; i <= end; i++) {
printf(" %d", disk[i]);
seek_time += abs(disk[i] - head);
head = disk[i];
}
printf(" 0");
for (i = 0; i < start; i++) {
printf(" %d", disk[i]);
seek_time += abs(disk[i] - head);
head = disk[i];
}
printf("\nTotal Seek Time: %d\n", seek_time);
}
int main() {
int disk[] = { 98, 183, 37, 122, 14, 124, 65, 67 };
int n = sizeof(disk) / sizeof(disk[0]);
int head;
printf("Enter the initial position of the disk head: ");
scanf ("%d", &head);
cscan(disk, n, head);
return 0;
}
```

OUTPUT:

```
cse@ubuntu2:~/Desktop/cseb$ gcc cscan.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out
Enter the initial position of the disk head: 25

C-SCAN Scheduling:
Sequence of disk access: 37 65 67 98 122 124 183 0 14
Total Seek Time: 327
cse@ubuntu2:~/Desktop/cseb$
```


AIM:

ALGORITHM:

Step 1: Start.
Step 2: Display the function LOOK
Step 3: Sort the disk request array in ascending order
Step 4: Find the index where the head is located
Step 5: Traverse the disk request array from the head position towards higher cylinder number
Step 6: Calculate the seek time as the absolute difference between the current request and the previous request
Step 7: Update the current head position to the current request
Step 8: Repeat until reaching the end of the disk or no further requests are available in that direction
Step 9: Jump to the lower end of the disk
Step 10: Traverse the disk request array from the lower end towards the start
Step 11: Calculate the seek time as the absolute difference between the current request and the previous request
Step 12: Update the current head position to the current request
Step 13: Repeat until reaching the start of the disk or no further requests are available in that direction
Step 14: Display the sequence of disk access and the total seek time
Step 15: Call the function look().
Step 16: Stop.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include<limits.h>
void look(int disk[], int n, int head, int direction)
{
```

```c
int i, seek_time = 0;
int start = 0, end = n - 1;
printf("\nLOOK Scheduling:\n");
printf("Sequence of disk access:");
for (i = 0; i < n - 1; i++) {
int j;
for (j = 0; j < n - i - 1; j++) {
if (disk[j] > disk[j + 1]) {
int temp = disk[j];
disk[j] = disk[j + 1];
disk[j + 1] = temp;
}}}
for (i = 0; i < n; i++) {
if (disk[i] > head) {
start = i - 1;
break;
}}
if (direction == 1) {
for (i = start; i <= end; i++) {
printf(" %d", disk[i]);
seek_time += abs(disk[i] - head);
head = disk[i];
}
}
else {
for (i = start; i >= 0; i--) {
printf(" %d", disk[i]);
seek_time += abs(disk[i] - head);
head = disk[i];
}}
printf("\nTotal Seek Time: %d\n", seek_time);
}

int main() {
int disk[] = { 98, 183, 37, 122, 14, 124, 65, 67 };
int n = sizeof(disk) / sizeof(disk[0]);
int head;
printf("Enter the initial position of the disk head: ");
scanf ("%d", &head);
look(disk, n, head,1);
look(disk, n, head,0);
```

```
return 0;
}
```

OUTPUT:

```
cse@ubuntu2:~/Desktop/cseb$ gcc look.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out
Enter the initial position of the disk head: 30

LOOK Scheduling:
Sequence of disk access: 14 37 65 67 98 122 124 183
Total Seek Time: 185

LOOK Scheduling:
Sequence of disk access: 14
Total Seek Time: 16
cse@ubuntu2:~/Desktop/cseb$
```

AIM:

ALGORITHM:

Step 1: Start.
Step 2: Display the function C-LOOK
Step 3: Sort the disk request array in ascending order
Step 4: Find the index where the head is located
Step 5: Traverse the disk request array from the head position towards higher cylinder number
Step 6: Calculate the seek time as the absolute difference between the current request and the previous request
Step 7: Update the current head position to the current request 85
Step 8: Repeat until reaching the end of the disk or no further requests are available in that direction
Step 9: Traverse the disk request array from the lower end towards the start
Step 10: Calculate the seek time as the absolute difference between the current request and the previous request
Step 11: Update the current head position to the current request
Step 12: Repeat until reaching the start of the disk or no further requests are available in that direction
Step 13: Display the sequence of disk access and total seek time.
Step 14: Call the function clook().
Step 15: Stop.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include<limits.h>
void clook(int disk[], int n, int head)
{
```

```
int i, seek_time = 0;
int start = 0, end = n - 1;
printf("\nC-LOOK Scheduling:\n");
printf("Sequence of disk access:");
for (i = 0; i < n - 1; i++) {
int j;
for (j = 0; j < n - i - 1; j++) {
if (disk[j] > disk[j + 1]) {
int temp = disk[j];
disk[j] = disk[j + 1];
disk[j + 1] = temp;
}}}
for (i = 0; i < n; i++) {
if (disk[i] > head) {
start = i;
break;
}
}
for (i = start; i <= end; i++) {
printf(" %d", disk[i]);
seek_time += abs(disk[i] - head);
head = disk[i];
}
for (i = 0; i < start; i++) {
printf(" %d", disk[i]);
seek_time += abs(disk[i] - head);
head = disk[i];
}
printf("\nTotal Seek Time: %d\n", seek_time);
}

int main() {
int disk[] = { 98, 183, 37, 122, 14, 124, 65, 67 };
int n = sizeof(disk) / sizeof(disk[0]);
int head;
printf("Enter the initial position of the disk head: ");
scanf ("%d", &head);
clook(disk, n, head);
return 0;
}
```

OUTPUT:

```
cse@ubuntu2:~/Desktop/cseb$ gcc clook.c
cse@ubuntu2:~/Desktop/cseb$ ./a.out
Enter the initial position of the disk head: 35

C-LOOK Scheduling:
Sequence of disk access: 37 65 67 98 122 124 183 14
Total Seek Time: 317
cse@ubuntu2:~/Desktop/cseb$
```

AIM:

PROCEDURE:

Step 1: Install VMware: Download and install VMware Workstation or VMware Player on your host operating system (the system on which VMware will be installed).

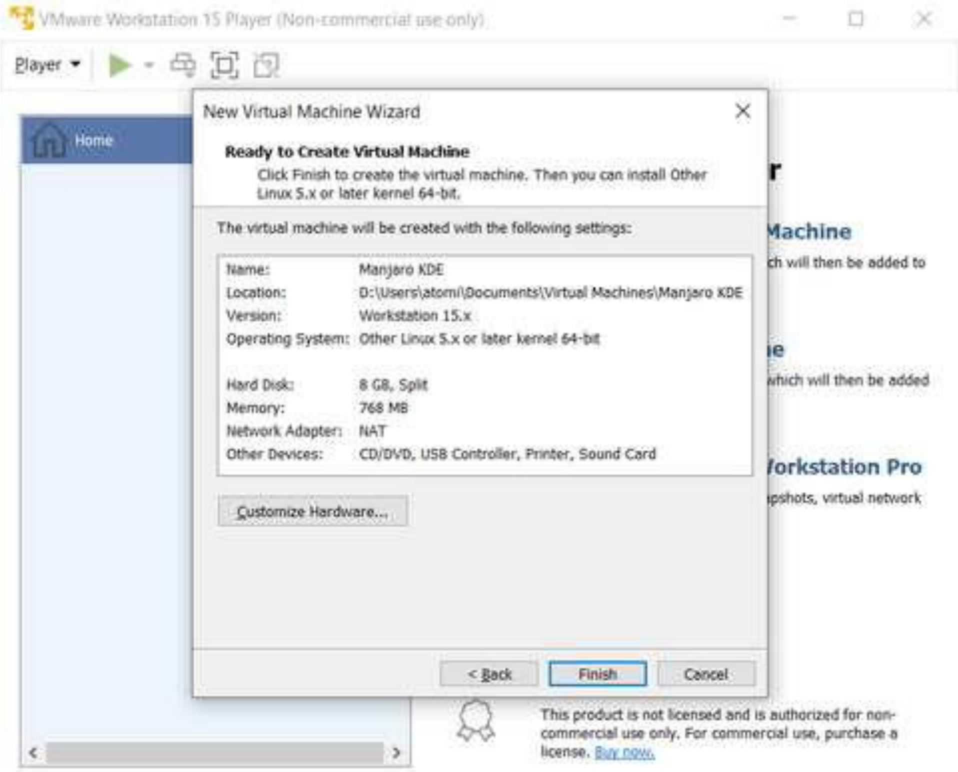
Step 2: Obtain the Linux ISO: Download the ISO file for the Linux distribution you want to install as the guest operating system. You can obtain the ISO file from the official website of the Linux distribution or from trusted sources.

Step 3: Open VMware: Launch the VMware application on your host operating system.

Step 4: Create a new virtual machine: Click on the "Create a New Virtual Machine" or "New Virtual Machine" option in the VMware application to start the virtual machine creation wizard.

Step 5: Select the installation method: Choose the option to install the guest operating system from an ISO image. Browse and select the Linux ISO file you downloaded in step 2.

Step 6: Specify guest operating system details: Select the appropriate Linux distribution and version from the list provided by VMware. If your Linux distribution is not listed, choose the closest match or select "Other Linux."



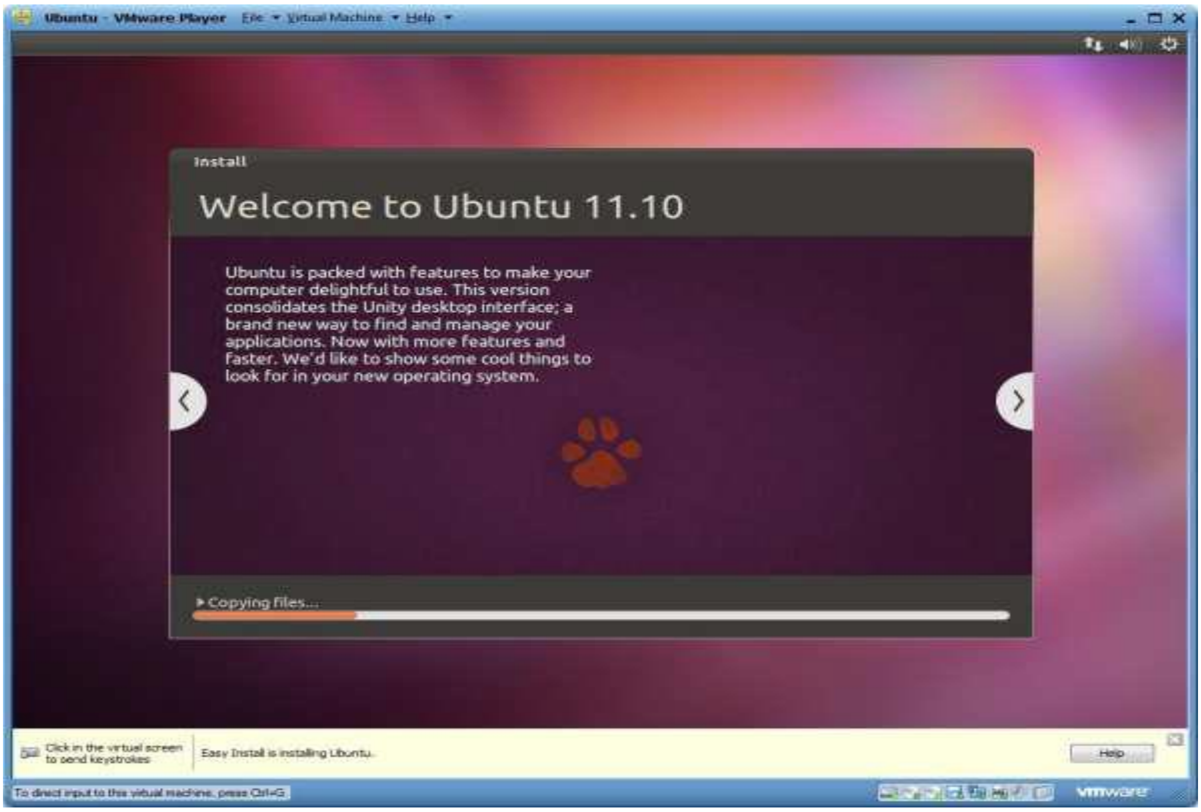
Step 7: Configure virtual machine settings: Specify the virtual machine name, location to store the virtual machine files, and allocate the desired amount of memory, CPU cores, and disk space for the virtual machine. Follow the recommended system requirements for the Linux distribution.

Step 8: Customize hardware settings (optional): You can customize additional hardware settings such as network adapters, graphics, sound, and other devices based on your requirements.

Step 9: Start the virtual machine: Once the virtual machine is created, select it from the VMware application and click on the "Play" or "Start" button to power on the virtual machine.

Step 10: Install Linux: The virtual machine will boot from the Linux ISO image. Follow the on-screen instructions to install Linux as you would on a physical machine. You may need to partition the virtual disk, select installation options, set up user accounts, and configure network settings.

Step 11: Complete the installation: Once the installation is finished, the virtual machine will restart. Log in to the Linux guest operating system using the credentials you set installation Process.



Step 12: Install VMware Tools (optional): VMware Tools provides enhanced performance and features for the guest operating system. Install VMware Tools within the Linux guest operating system to enable features like shared folders, drag and drop, and improved graphics.