K.S.RANGASAMY COLLEGE OF TECHNOLOGY

(Autonomous Institution)

TIRUCHENGODE – 637 215

RECORD NOTEBOOK

60 CB 4P1 – OPERATING SYSTEMS LAB

Semester-VI ,Year – II

Batch-2022-2026

BACHELOR OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND BUSINESS SYSTEMS

K.S.RANGASAMY COLLEGE OF TECHNOLOGY

(An Autonomous Institution, affiliated to Anna University Chennai and Approved by AICTE, New Delhi)

TIRUCHENGODE - 637 215

K.S.RANGASAMY COLLEGE OF TECHNOLOGY

(Autonomous Institution)

TIRUCHENGODE – 637 215

CERTIFICATE

Register Number: **73772227109**

Certified that this is the bonafide record of work done by ~~Selvan~~/Selvi **DHARSHINI.G** of the Fourth Semester **B.Tech.Computer Science and Business Systems** branch during the academic year **2023-2024** in the '60 CB 4P1 – Operating Systems Lab'.

Staff in-charge   Head of the Department

Submitted for the End Semester Practical Examination on …………………….

Internal Examiner I   Internal Examiner II

CONTENTS

60 CB 4P1 – Operating Systems Lab

Ex.No	Date	Name of the Experiment	Page No	Marks	Sign
1	21/02/2024	Analysis and Synthesis of Basic Linux Commands			
2	28/02/2024	Programs using Shell Programming			
3	06/03/2024	Implementation of UNIX System Calls			
4	13/03/2024	Installation of Linux Operating System			
5	03/04/2024	Implementation of POSIX Thread Functions for create, join and exit			
6	10/04/2024	Simulation and Analysis of Non pre-emptive and Pre-emptive CPU Scheduling Algorithms			
7	17/04/2024	Simulation of Producer – Consumer Problem using Semaphores and Implementation of Dining Philosopher’s Problem to demonstrate Process Synchronization			
8	24/04/2024	Simulation of Banker’s Algorithm for Deadlock Avoidance			
9	22/05/2024	Analysis and Simulation of Memory Allocation and Management Techniques			
10	29/05/2024	Implementation of Page Replacement Techniques			
11	30/05/2024	Simulation of Disk Scheduling Algorithms			
12	04/06/2024	Implementation of File organization Techniques			

K.S.RANGASAMY COLLEGE OF TECHNOLOGY, TIRUCHENGODE - 637215
DEPARTMENT OF COMPUTER SCIENCE AND BUSINESS SYSTEMS

VISION

- To produce skilled professionals to the dynamic needs of the industry with innovative computer science Professionals associate with managerial services

MISSION

- To promote student's ability through innovative teaching in computer science to compete globally as an engineer
- To inculcate management skills to meet the industry standards and augment human values and life skills to serve the society

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

PEO1: Graduates will provide effective solutions for software and hardware industries by applying the Concepts of basic science and engineering fundamentals.

PEO2: Graduates will be professionally competent and successful in their career through life-long Learning.

PEO3: Graduates will contribute individually or as member of a team in handling projects and demonstrate Social responsibility and professional ethics.

PROGRAMME OUTCOMES (POs)

Engineering Graduates will be able to:

PO1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems. Graduates will contribute individually or as member of a team in handling projects and demonstrate social responsibility and professional ethics.

PO2: Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principlesof mathematics, natural sciences, and engineering sciences. Graduates will contribute individually or as member of a team in handling projects and demonstrate social responsibility and professional ethics.

PO3: Design /development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriateconsideration for the public health and safety, and the cultural, societal, and environmental considerations. Graduates will contribute individually or as member of a team in handling projects and demonstrate social responsibility and professional ethics.

PO4: Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions. Graduates will contribute individually or as member of a team in handling projects and demonstrate social responsibility and professional ethics

PO5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO6: The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7: Environment and sustainability: Understand the impact of the professional engineering solution sin societal and environmental contexts, and demonstrate the knowledge of, and need for unstainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice. Graduates will contribute individually or as member of a team in handling projects and demonstrate social responsibility and professional ethics.

PO9: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings. Graduates will contribute individually or as member of a team in handling projects and demonstrate social responsibility and professional ethics.

PO10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clearinstructions.

PO11: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one 's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAMME SPECIFIC OUTCOMES (PSOs):

Engineering Graduates will be able to:

PSO1: Apply analytical and technical skill of computer science to provide justifiable solution for real world applications

PSO2: Analyze various managerial skills and business disciplines to improve the industry growth and development

EX. NO: 01	**SIMPLE COMMANDS**
DATE:21/02/2024	

Aim:
To study the basic unix commands and execute those commands.

Commands:

1. Who am i

Syntax: $ who am i

Description: This command is used to display user name, terminal name, date and time of login.

Output:
```
$ who am i
CSE10 pts/18 JUL/16 14.00 <192.50.50.112>
```

2. Who

Syntax: $ who

Description: This command is used to list the users who are currently logged to the system.

Output:
```
$ who
CSE09 pts/16 JUL/16 14.02  <192.50.50.113>
CSE11 pts/19 JUL/16 14.02  <192.50.50.115>
```

3. tty

Syntax: $ tty

Description: This command is used to display the terminal name.

Output:
```
$ tty
/dev/pts/18
```

4. uname

Syntax: $ uname

Description: This command is used display the operating system in which we are working.

Output:
```
$ uname Linux
```

Directory commands

5. pwd

Syntax: $ pwd

Description: This command is used to display the current working directory.

Output:
```
$ pwd
/home/CSE10/kalai
```

6. ls

Syntax: $ ls

Description: This command is used to list the files stored in the directory.

Output:

```
$ ls
x.out    y.out    z.out    kalai
```

7. mkdir

Syntax: $ mkdir dir name

Description: This command is used to create new directory.

Output:

```
$ mkdir kalai
```

8. rmdir

Syntax: $ rmdir dir name

Description: This command is used to delete / remove a directory.

Output:

```
$ rmdir kalai
```

9. cd

Syntax: $ cd dir name

Description: This command is used to change the directory.

Output:

```
$ cd kalaii
```

File manipulation commands

10. cat

a. **Syntax:** $cat filename

Description: This command is used to view the content of the file.

Output:

```
 $ cat x.outWin
  Dos Polaris
  Linux
```

b. **Syntax:** $cat >filename

Description: This command is used to change the content of the file.

Output:

```
$ cat
>x.out
Goat
<ctrl+D>
```

c. **Syntax:** $cat >>filename

Description: This command is used to modify the content of the file.

Output:

```
$ cat
>>x.
out boat
<ctrl+D>
```

d. **Syntax:** $cat filename1 filename2 > new file

Description: This command is used to concatenate 2 files and save it in a new file.

Output:

```
$ cat  x.out y.out > z.out
```

Content of x.out	**Content of y.out**	**Content of z.out**
Goat	sun	Goat
Boat	moon	boat
		Sun
		Moon

11. cp

Syntax: $cp source filename destination filename

Description: This command is used to copy the content of the one file to another file.

Output:

```
$ cp x.out a.out
$ cat a.out
Goat
boat
```

12. mv

Syntax: $mv current filename new filename

Description: This command is used to move the content of the one file to another file.

Output:

```
$ cp a.out b.out
$ cat b.out
Goat
Boat
```

13. rm

Syntax: $rm filename

Description: This command is used to delete a file.

Output:

```
$ rm b.out
$ cat b.out
No such file or directory
```

14.wc

a. **Syntax:** $ wc –c filename

Description: This command is used to count the number of characters in a file.

Output:

```
$   wc   – c
x.out10
x.out
```

b. **Syntax:** $ wc –w filename

Description: This command is used to count the number of words in a file.

Output:

```
$ wc –w x.out
2              x.out
```

c. **Syntax:** $ wc –l filename
Description: This command is used to count the number of lines in a file.
Output:

```
$ wc x.out2
x.out
```

15. head
Syntax: $ head filename
Description: This command is used to list the first 10 elements of the file.
Output:

```
$ head x.out
Goat
Boat
House
Joy
King
Pop
Puppy
Loose
Hello
Gun
```

16. tail
a. Syntax: $ tail filename
Description: This command is used to list the last 10 elements of the file.
Output:

```
$ tail x.outKing
Pop Puppy
Loose Hello
Gun Boy
Moon
Greater
Eagle
```

b. Syntax: $ tail -size filename
Description: This command is used to list the last size number of elements of the file.
Output:

```
$ tail -5 x.outGun
Boy Moon
Greater
Eagle
```

17. cmp
Syntax: $ cmp file1 file2
Description: This command is used to compare 2 files and list where difference occur (text or binary).
Output :

```
 $  cmp x.out y.out
x.out    y.out    differ: byte 1 line
```

Information commands

18. date
Syntax: $ date
Description: This command is used to report the current date and time.
Output:

```
$ date
Sat JUL16 15: 11: 30  IST 2005
```

19. cal
Syntax: $ cal
Description: This command is used to display the calender of current month.
Output:
```
$ cal
JULY 2005
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
```
20. mail
Syntax: $ mail
Description: This command is used to find whether any mail is avilable for our login.
Output:
```
$ mail
No mail for CSE10
```

AIM & ALGORITHM	30	
PROGRAM/EXECUTION	30	
OUTPUT	20	
VIVA	10	
RESULT	10	
TOTAL	100	

Result:
Thus, the basic commands of UNIX were studied and executed.

EX. NO: 02	**SHELL PROGRAMMING**
DATE:28/02/2024	

Description:

A set of commands can be grouped together under a single file name and executed. This is done using shell script. Shell provides us with features to enable it to be used as programming language. Some of these features are programming constructs logical and condition operators command substitution escape mechanism, positional parameters etc.

Shell Program Use:

1. Certain special characters called Meta characters in addition to other characters.
2. Valuables and variables name with assignment statements.
3. Control statements such as if...Then else, while do … done, case … esac etc.
4. input statement such as read and output statement as echo.
5. All the commands including pipelining and redirection available under UNIX.

EX. NO: 2a	FACTORIAL OF A NUMBER
DATE: 28/02/2024	

Aim:
To write a shell script to find the factorial of a given number.

Algorithm:

1. Start.
2. Print "Enter the number:".
3. Read input into variable n.
4. Initialize variable i to 1.
5. Initialize variable x to 1.
6. While i is less than or equal to n, do the following:
7. Update x to be the product of i and x.
8. Increment i by 1.
9. End While loop.
10. Print "The factorial value is : " followed by the value of x.
11. End.

Program:

```
echo “Enter the number:”
read n
i=1
x=1
while [ $i –le $n ]
do
x=`expr $i \* $x`
i= `expr $i + 1`
done
echo “The factorial value is : $x”
```

Output:

Result:

Thus, the shell script to find the factorial of the given number was written and its ouput was verified.

EX. NO: 2b	**FIBONACCI SERIES**
DATE: 28/02/2024	

Aim:
To write a shell script to find the fibonacci series of the given number.

Algorithm:
1. Start.
2. Print "Enter the value of n:".
3. Read the input value into variable n.
4. Initialize variable a to -1.
5. Initialize variable b to 1.
6. Initialize variable i to 1.
7. While i is less than or equal to n, do the following:
8. Compute s as the sum of a and b.
9. Print the value of s.
10. Update a to the value of b.
11. Update b to the value of s.
12. Increment i by 1.
13. End While loop.
14. End.

Program:
```
echo “Enter the value of n:”
a=-1
b=1
i=1
read n
while [ $i –le $n ]
do
s=`expr $a + $b`
echo $s
a=$b
b=$s
i=`expr $i + 1`
done
```
Output:

Result:
Thus, the shell script to find the fibonacci series of the given number was written and its output was verified.

EX. NO: 2c	ARMSTRONG NUMBER
DATE: 28/02/2024	

Aim:
To write a shell script to check whether the given number is armstrong or not.

Algorithm:

1. Start.
2. Print "-------- Armstrong Number".
3. Print "Enter the Number:".
4. Read the input number into variable n.
5. Assign the value of n to variable b (to store the original number).
6. Initialize variable s to 0 (this will hold the sum of the cubes of the digits).
7. While n is greater than 0, do the following:
8. Compute r as n % 10 (the last digit of n).
9. Update s to s + r * r * r.
10. Update n to n / 10.
11. End While loop.
12. If s is equal to b:
13. Print "Armstrong number".
14. Else:
15. Print "Not armstrong number".
16. End.

Program:

```
echo “---------Armstrong Number      “
echo “Enter the Number:”
read n
b=$n
s=0
while [ $n –gt 0 ]
do
r=`expr $n % 10`
s=`expr $s + $r \* $r \* $r`
n=`expr $n / 10`
done
if [ $s –eq $b ]
then
echo “Armstrong number”
else
echo “Not armstrong number”
fi
```

Output:

Result:
Thus, the shell script to check whether the given number was armstrong or not was written and verified.

EX. NO: 2d	PRIME NUMBER
DATE: 28/02/2024	

Aim:
To write a shell script to print the prime numbers.

Algorithm:

1. Start.
2. Print "------------- PRIME NUMBERS".
3. Initialize variable x to 2.
4. While x is less than 10, do the following:
5. Initialize variable c to 0.
6. Initialize variable t to 2.
7. While t is less than x, do the following:
8. Compute m as x % t.
9. If m is equal to 0, set c to 1.
10. Increment t by 1.
11. End While loop.
12. If c is equal to 0, print that x is a prime number.
13. Increment x by 1.
14. End While loop.
15. End.

Program:

```
echo “--------------PRIME NUMBERS           “
x=2
while [ $x –lt 10 ]
do
c=0
t=2
while [ $t –lt $x ]
do
m=`expr $x % $t`
if [ $m –eq 0 ]
then
c=1
fi
t=`expr $t + 1`
done
if [ $c –eq 0 ]
then
echo “$x is a prime number”
fi
x= `expr $x + 1`
done
```

Output:

Result:

Thus, the shell script to print the prime numbers was written and its output was verified.

EX. NO: 2e	CONVERSION OF NUMBERS INTO WORD
DATE: 28/02/2024	

Aim:
To write a shell script for converting the numbers into words.

Algorithm:
1. Start.
2. Print "Enter the number".
3. Read the input number into variable num.
4. Calculate the length of num and store it in variable len.
5. Initialize variable k to 1.
6. While k is less than or equal to len, do the following:
7. Extract the k-th character from num and store it in variable digit.
8. Check the value of digit:

   if digit is 0, print "zero".
   if digit is 1, print "one".
   if digit is 2, print "two".
   if digit is 3, print "three".
   if digit is 4, print "four".
   if digit is 5, print "five".
   if digit is 6, print "six".
   if digit is 7, print "seven".
   if digit is 8, print "eight".
   if digit is 9, print "nine".
9. Otherwise, print "Invalid character".
10. Increment k by 1.
11. End While loop.

Program:
```
k=1
echo “ Enter the number”
read num
len= `echo $num |wc –c`
len= `expr $len + 1`
while [ $k –lt $len ]
do
num= `echo $num |cut –c “$k”`
case $num in
    0)  echo “ZERO”;;
    1)  echo “ONE”;;
    2)  echo “TWO”;;
    3)  echo “THREE”;;
    4)  echo “FOUR”;;
    5)  echo “FIVE”;;
    6)  echo “SIX”;;
    7)  echo “SEVEN”;;
    8)  echo “EIGHT”;;
    9)  echi “NINE”;;
```

```
esac;
k= `expr $k + 1`
done
```

Output:

AIM & ALGORITHM	30	
PROGRAM/EXECUTION	30	
OUTPUT	20	
VIVA	10	
RESULT	10	
TOTAL	100	

Result:
Thus, a shell script for converting numbers to words was written and its output was verified.

EX. NO: 3a	IMPLEMENTATION OF SYSTEM CALLS (fork(), getpid(), exit(), wait()
DATE:06/03/2024	

Aim:
To write a shell script using system calls (fork(), getpid(), exit() and wait()).

Algorithm:

1. Start and include necessary header files.
2. Declare an integer variable pid.
3. Create process using fork() and assign the return value to pid.
4. If pid < 0, print "Error" and exit.
5. If pid == 0, print child process ID and parent process ID.
6. If pid > 0, call wait(pid) and execute /bin/date using execl.
7. End

Program:
```
#include<stdio.h>
main()
{
int pid;
pid=fork();
if(pid<0)
{
printf("Error");
exit(0);
}
else if(pid==0)
{
printf("\nChild process id is %d",getpid());
printf("\nParent process id is %d",getppid());
}
else
{
wait(pid);
execl"("/bin/date","date",null);
}
}
```

Output:

Result:
Thus, a shell script using fork(), getpid(), exit() and wait() system calls was written and its output was verified.

EX. NO: 3b	**IMPLEMENTATION OF SYSTEM CALLS (opendir(), readdid())**
DATE: 06/03/2024	

Aim:
To write a shell script using system calls (opendir() and readdir()).

Algorithm:
1. Include necessary header files:
2. stdio.h: For standard input/output functions.
3. stdlib.h: For the exit function.
4. sys/types.h: For data types used in dirent.h.
5. dirent.h: For directory handling functions.
6. Define the main function to accept command-line arguments argc (argument count) and argv (argument vector).
7. Check if the number of arguments (argc) is not equal to 2:
8. If true, print "Directory name is required" and exit with status 1.
9. Attempt to open the directory specified by argv[1] using opendir:
10. If the directory cannot be opened (i.e., opendir returns NULL), print "Can't open <directory_name>" and exit with status 1.
11. Initialize a loop to read directory entries using readdir:
12. While there are more entries to read (i.e., readdir returns a non-NULL pointer):
13. Print the name of the current directory entry (dirp->d_name).
14. Close the directory using closedir.
15. Return 0 to indicate successful execution.

Program:
```
#include<stdio.h>
#include<sys/types.h>
#include<dirent.h>
int main(int argc, char * argv[])
{
DIR *dp;
struct dirent *dirp;
if(argc!=2)
{
printf(“Directory name is required\n”);
exit(1);
}
if((dp=opendir(argc[1]))==0)
{
printf(“Cant open %s\n”, argv[1]);
exit(1);
while((dirp=readdir(dp))!=0)
printf(“%s\n”, dirp->d_name);
closedir(dp);
exit(0);
}
```

Output:

AIM & ALGORITHM	30	
PROGRAM/EXECUTION	30	
OUTPUT	20	
VIVA	10	
RESULT	10	
TOTAL	100	

Result:
Thus, a shell script using system calls was written and its output was verified.

EX. NO: 04	**INSTALLATION OF LINUX OPERATING SYSTEM**
DATE: 13/03/2024	

Aim:
To install linux operating system.

Algorithm :

Algorithm for Installing a Linux Operating System:

1. Download Linux ISO:
   Visit the official website of the chosen Linux distribution and download the ISO file.

2. Prepare Installation Media:
   Create a bootable USB drive or burn the ISO file to a DVD.

3. Boot from Installation Media:
   1) Insert the bootable media into the computer.
   2) Restart the computer and boot from the installation media.

4. Start Installation:
   Follow the on-screen prompts to start the installation process.

5. Configure Settings:
   1) Select language, timezone, and keyboard layout.
   2) Set up network configuration (if required).

6. Partition Disks:
   1) Choose partitioning scheme (e.g., guided or manual partitioning).
   2) Select the disk to install Linux and allocate disk space.

7. Set Up User Account:
   Create a user account and set password.

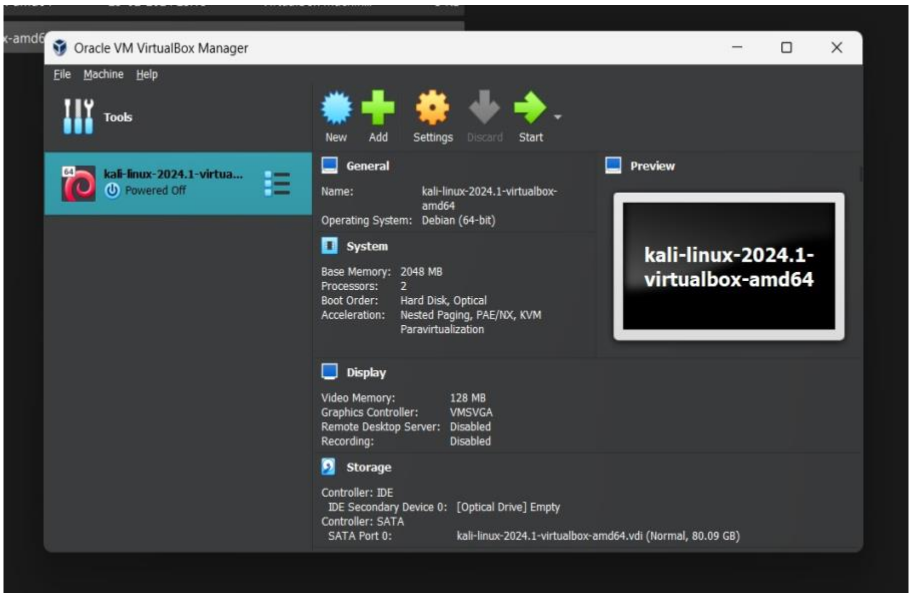
8. Install the System:
   The installer copies files and installs the base system.

9. Install Boot Loader:
   Install the boot loader (e.g., GRUB) to the master boot record.

10. Complete Installation:
    Once the installation is complete, reboot the system.

11. Post-Installation Configuration:
    Update and upgrade the system.
    Install additional software or tools as needed.

Output :


Oracle VM VirtualBox Manager
File Machine Help
Tools
New
Add
Settings
Discard
Start
kali-linux-2024.1-virtua...
Powered Off
Preview
kali-linux-2024.1-virtualbox-amd64

```
General
Name:              kali-linux-2024.1-virtualbox-
                   amd64
Operating System:  Debian (64-bit)

System
Base Memory:   2048 MB
Processors:    2
Boot Order:    Hard Disk, Optical
Acceleration:  Nested Paging, PAE/NX, KVM
               Paravirtualization

Display
Video Memory:            128 MB
Graphics Controller:     VMSVGA
Remote Desktop Server:   Disabled
Recording:               Disabled

Storage
Controller: IDE
  IDE Secondary Device 0:  [Optical Drive] Empty
Controller: SATA
  SATA Port 0:             kali-linux-2024.1-virtualbox-amd64.vdi (Normal, 80.09 GB)
```

AIM & ALGORITHM	30	
PROGRAM/EXECUTION	30	
OUTPUT	20	
VIVA	10	
RESULT	10	
TOTAL	100	

Result :

Thus, the linux operating system has been installed successfully.

EX. NO: 5a	IMPLEMENTATION OF POSIX THREAD FUNCTION (create)
DATE:03/04/2024	

Algorithm:

1. Include necessary header files:
2. #include <stdio.h>: For standard input/output functions.
3. #include <stdlib.h>: For general utility functions.
4. #include <pthread.h>: For POSIX thread functions.
5. Define the thread function:
6. Print "Thread is executing...".
7. Exit the thread using pthread_exit(NULL).
8. In the main function:
9. Declare a variable pthread_t thread_id to hold the thread identifier.
10. Create a new thread using pthread_create:
11. If the thread creation fails (pthread_create returns a non-zero value), print "Error creating thread" and exit the program with status 1.
12. If the thread creation succeeds, print "Main thread has created a new thread".
13. Wait for the created thread to terminate using pthread_join(thread_id, NULL):
14. This ensures that the main thread waits for the new thread to finish execution before continuing.
15. Return 0 to indicate successful execution of the program.

Program:

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

void *thread_function(void *arg) {
    printf("Thread is executing...\n");
    pthread_exit(NULL);
}

int main() {
    pthread_t thread_id;

    // Create a new thread
    if (pthread_create(&thread_id, NULL, thread_function, NULL)) {
        fprintf(stderr, "Error creating thread\n");
        return 1;
    }

    printf("Main thread has created a new thread\n");

    return 0;
}
```

Output:

Result:
The program to create a new thread using pthread_create() is successfully executed and verified.

EX. NO: 5b	IMPLEMENTATION OF POSIX THREAD FUNCTION (join)
DATE:03/04/2024	

Aim:
To understand the process of joining POSIX threads in a Linux environment.'

Algorithm:

```
1. Include necessary headers: stdio.h, stdlib.h, pthread.h.
2. Define the thread function to execute when the thread is created.
3. In the main function:
   - Declare a variable to hold the thread identifier.
   - Create a new thread using pthread_create.
   - If thread creation fails, print an error message and exit.
   - Wait for the created thread to finish execution using pthread_join.
   - Print a message indicating successful thread join.
4. End.
```

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *thread_function(void *arg) {
   printf("Thread is executing...\n");
   pthread_exit(NULL);
}
int main() {
   pthread_t thread_id;
   // Create a new thread
   if (pthread_create(&thread_id, NULL, thread_function, NULL)) {
      fprintf(stderr, "Error creating thread\n");
      return 1;
   }
   // Main thread waits for the created thread to finish
   printf("Main thread waiting for the created thread to finish...\n");
   pthread_join(thread_id, NULL);
   printf("Thread has joined successfully\n");
   return 0;
}
```

Output:

Result:
The program to join a thread using pthread_join() is successfully executed and verified.

EX. NO: 5c	**IMPLEMENTATION OF POSIX THREAD FUNCTION (exit)**
DATE: 03/04/2024	

Aim:
To understand the process of joining POSIX threads in a Linux environment.’

Algorithm :

1. Include Necessary Headers: Import required header files: stdio.h, stdlib.h, and pthread.h.
2. Define the Thread Function: Implement the function to be executed by the created thread. This function prints messages and then exits using pthread_exit.
3. Main Function:
   - Declare a variable to hold the thread identifier: pthread_t thread_id.
   - Create a new thread using pthread_create:
     - If thread creation fails, print an error message to stderr and exit with status 1.
4. Wait for Thread to Finish:
   - After creating the thread, the main thread waits for the created thread to finish using pthread_join.
5. Print Success Message:
   - Once the created thread finishes execution, print a message indicating successful thread join.
6. Return Success Status:
   - Exit the program with status 0 to indicate successful execution.

Program:
```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_function(void *arg) {
    printf("Thread is executing...\n");
    printf("Thread is now exiting...\n");
    pthread_exit(NULL);
}

int main() {
    pthread_t thread_id;

    // Create a new thread
    if (pthread_create(&thread_id, NULL, thread_function, NULL)) {
        fprintf(stderr, "Error creating thread\n");
        return 1;
    }

    // Main thread waits for the created thread to finish
    printf("Main thread waiting for the created thread to finish...\n");
    pthread_join(thread_id, NULL);
    printf("Thread has joined successfully\n");

    return 0;
}
```

Output:

AIM & ALGORITHM	30	
PROGRAM/EXECUTION	30	
OUTPUT	20	
VIVA	10	
RESULT	10	
TOTAL	100	

Result:
The program to exit thread using pthread_exit() is successfully executed and verified.

EX. NO: 6a	NON-PREEMPTIVE CPU SCHEDULING ALDORITHM
DATE:10/04/2024	

Aim:
To simulate and analyze the performance of a non-preemptive CPU scheduling algorithm.

Algorithm:

1. Define a structure Process to represent a process with attributes id, arrival time, and burst time.
2. Implement a non-preemptive scheduling function that takes an array of processes and the number of processes as input.
3. Initialize total_time and average_waiting_time variables.
4. Sort the processes array based on arrival time using bubble sort.
5. Calculate total waiting time by iterating through the sorted processes and accumulating waiting times.
6. Calculate average waiting time by dividing total waiting time by the number of processes.
7. In the main function, define an array of processes and its size.
8. Call the non_preemptive_scheduling function with the array of processes and size, and print the average waiting time.

Program:
```
#include <stdio.h>

// Structure to represent a process
struct Process {
    int id;
    int arrival_time;
    int burst_time;
};

// Function to perform non-preemptive CPU scheduling
float non_preemptive_scheduling(struct Process processes[], int n) {
    int total_time = 0;
    float average_waiting_time = 0;

    // Sort processes by arrival time
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].arrival_time > processes[j + 1].arrival_time) {
                // Swap processes
                struct Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }

    // Calculate total waiting time
    for (int i = 0; i < n; i++) {
        total_time += processes[i].burst_time;
        average_waiting_time += total_time - processes[i].arrival_time;
    }

    average_waiting_time /= n;
    return average_waiting_time;
}

int main() {
    // Example usage
    struct Process processes[] = {{1, 0, 3}, {2, 1, 5}, {3, 2, 2}, {4, 3, 8}}; // Format: {id,
arrival_time, burst_time}
    int num_processes = sizeof(processes) / sizeof(processes[0]);

    float avg_waiting_time = non_preemptive_scheduling(processes, num_processes);
    printf("Average Waiting Time (Non-preemptive): %.2f\n", avg_waiting_time);

    return 0;
}
```

Output:

Result:
The program successfully simulates a non-preemptive CPU scheduling algorithm and calculates the average waiting time for a set of processes.

EX. NO: 6b	PREEMPTIVE CPU SCHEDULING ALDORITHM
DATE: 10/04/2024	

Aim:
To simulate and analyze the performance of a preemptive CPU scheduling algorithm.

Algorithm:

1. Define a structure to represent a process with attributes id, arrival time, and burst time.
2. Implement a preemptive scheduling function that takes an array of processes and the number of processes as input.
3. Initialize variables for total time, current time, and total waiting time.
4. Sort processes by burst time.
5. Iterate through processes, selecting the one with the minimum burst time and whose arrival time is less than or equal to the current time.
6. Update current time and waiting time accordingly, removing the selected process from the array.
7. Calculate average waiting time and return it.
8. In the main function, define an array of processes and its size, then call the preemptive scheduling function and print the average waiting time.

Program:

```
#include <stdio.h>

// Structure to represent a process
struct Process {
    int id;
    int arrival_time;
    int burst_time;
};

// Function to perform preemptive CPU scheduling
float preemptive_scheduling(struct Process processes[], int n) {
    int total_time = 0;
    int current_time = 0;
    float total_waiting_time = 0;

    // Sort processes by burst time
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].burst_time > processes[j + 1].burst_time) {
                // Swap processes
                struct Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
    // Perform preemptive scheduling
    while (n > 0) {
        int min_burst_index = -1;
        int min_burst_time = 99999;
        for (int i = 0; i < n; i++) {
            if (processes[i].arrival_time <= current_time && processes[i].burst_time <
min_burst_time) {
                min_burst_time = processes[i].burst_time;
                min_burst_index = i;
            }
        }

        if (min_burst_index == -1) {
            current_time++;
        } else {
            total_waiting_time += current_time - processes[min_burst_index].arrival_time;
            current_time += processes[min_burst_index].burst_time;
            for (int i = min_burst_index; i < n - 1; i++) {
                processes[i] = processes[i + 1];
            }
            n--;
        }
    }

    float average_waiting_time = total_waiting_time / n;
    return average_waiting_time;
}

int main() {
    // Example usage
    struct Process processes[] = {{1, 0, 3}, {2, 1, 5}, {3, 2, 2}, {4, 3, 8}}; // Format: {id,
arrival_time, burst_time}
    int num_processes = sizeof(processes) / sizeof(processes[0]);

    float avg_waiting_time = preemptive_scheduling(processes, num_processes);
    printf("Average Waiting Time (Preemptive): %.2f\n", avg_waiting_time);

    return 0;
}
```

Output:

AIM & ALGORITHM	30	
PROGRAM/EXECUTION	30	
OUTPUT	20	
VIVA	10	
RESULT	10	
TOTAL	100	

Result:
The program successfully simulates a preemptive CPU scheduling algorithm and calculates the average waiting time for a set of processes.

EX. NO: 7a	PRODUCER-CONSUMER PROBLEM USING SEMAPHORES
DATE: 17/04/2024	

Aim:
To demonstrate process synchronization using semaphores in the Producer-Consumer Problem.

Algorithm:
1. Define buffer size, number of items, and buffer array, along with in and out variables.
2. Initialize semaphores for mutex, full, and empty.
3. Define producer and consumer functions.
4. Implement the producer function to produce items, using semaphores to control access to the buffer.
5. Implement the consumer function to consume items, also using semaphores for synchronization.
6. In the main function, initialize semaphores and create producer and consumer threads.
7. Wait for both threads to finish execution using pthread_join.
8. Destroy semaphores and exit the program.

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

#define BUFFER_SIZE 5
#define NUM_ITEMS 10

int buffer[BUFFER_SIZE];
int in = 0, out = 0;

sem_t mutex, full, empty;

void *producer(void *arg) {
   for (int i = 0; i < NUM_ITEMS; i++) {
      sem_wait(&empty);
      sem_wait(&mutex);

      buffer[in] = i;
      printf("Produced item: %d\n", i);
      in = (in + 1) % BUFFER_SIZE;

      sem_post(&mutex);
      sem_post(&full);
   }
   pthread_exit(NULL);
}

void *consumer(void *arg) {
   int item;
```

```
    for (int i = 0; i < NUM_ITEMS; i++) {
        sem_wait(&full);
        sem_wait(&mutex);

        item = buffer[out];
        printf("Consumed item: %d\n", item);
        out = (out + 1) % BUFFER_SIZE;

        sem_post(&mutex);
        sem_post(&empty);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t producer_thread, consumer_thread;

    sem_init(&mutex, 0, 1);
    sem_init(&full, 0, 0);
    sem_init(&empty, 0, BUFFER_SIZE);

    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    sem_destroy(&mutex);
    sem_destroy(&full);
    sem_destroy(&empty);

    return 0;
}
```

Output:

Result:
The program successfully simulates the Producer-Consumer Problem using semaphores, demonstrating process synchronization between the producer and consumer threads.

EX. NO: 7b	IMPLEMENTATION OF DINING PHILOSOPHER PROBLEM
DATE:17/04/2024	

Aim:
To demonstrate process synchronization using mutex locks in the Dining Philosophers Problem.

Algorithm:
1. Define the number of philosophers and the states they can be in.
2. Initialize semaphores for mutex and forks.
3. Implement the test function to check if a philosopher can start eating.
4. Implement functions to take and put forks, managing the states of philosophers.
5. Define the philosopher function to represent the behavior of a philosopher.
6. In the philosopher function, have philosophers alternate between thinking, taking forks, eating, and putting forks.
7. In the main function, initialize semaphores and create philosopher threads.
8. Wait for philosopher threads to finish execution and clean up resources before exiting.

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

#define NUM_PHILOSOPHERS 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2

sem_t mutex;
sem_t forks[NUM_PHILOSOPHERS];

int state[NUM_PHILOSOPHERS];

void test(int philosopher_id) {
    if (state[philosopher_id] == HUNGRY && state[(philosopher_id + 1) %
NUM_PHILOSOPHERS] != EATING
        && state[(philosopher_id + NUM_PHILOSOPHERS - 1) % NUM_PHILOSOPHERS]
!= EATING) {
        state[philosopher_id] = EATING;
        printf("Philosopher %d is eating\n", philosopher_id);
        sem_post(&forks[philosopher_id]);
    }
}

void take_forks(int philosopher_id) {
    sem_wait(&mutex);
    state[philosopher_id] = HUNGRY;
    printf("Philosopher %d is hungry\n", philosopher_id);
    test(philosopher_id);
    sem_post(&mutex);
    sem_wait(&forks[philosopher_id]);
}

void put_forks(int philosopher_id) {
    sem_wait(&mutex);
    state[philosopher_id] = THINKING;
    printf("Philosopher %d puts down forks and starts thinking\n", philosopher_id);
    test((philosopher_id + 1) % NUM_PHILOSOPHERS);
    test((philosopher_id + NUM_PHILOSOPHERS - 1) % NUM_PHILOSOPHERS);
    sem_post(&mutex);
}

void *philosopher(void *arg) {
    int philosopher_id = *((int *)arg);

    while (1) {
        // Philosopher thinks for a while
        printf("Philosopher %d is thinking\n", philosopher_id);
        sleep(1);

        // Philosopher gets hungry and wants to eat
        take_forks(philosopher_id);

        // Philosopher eats for a while
        sleep(2);

        // Philosopher puts down forks and starts thinking again
        put_forks(philosopher_id);
    }
}

int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS];
    int philosopher_id[NUM_PHILOSOPHERS];

    sem_init(&mutex, 0, 1);
    for (int i = 0; i < NUM_PHILOSOPHERS; i++)
        sem_init(&forks[i], 0, 0);

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        philosopher_id[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, &philosopher_id[i]);
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; i++)
        pthread_join(philosophers[i], NULL);

    for (int i = 0; i < NUM_PHILOSOPHERS; i++)
```

```
        sem_destroy(&forks[i]);
    sem_destroy(&mutex);

    return 0;
}
```

Output:

AIM & ALGORITHM	30	
PROGRAM/EXECUTION	30	
OUTPUT	20	
VIVA	10	
RESULT	10	
TOTAL	100	

Result:
The program successfully simulates the Dining Philosophers Problem using mutex locks, demonstrating process synchronization among the philosophers while avoiding deadlock and starvation.

EX. NO: 8a	**BANKER’S ALGORITHM INITIALIZATION AND SAFETY CHECK**
DATE: 24/04/2024	

Aim:
To initialize the Banker's Algorithm and perform a safety check to ensure that the system is in a safe state.

Algorithm:
1. Define constants for the maximum number of processes and resources.
2. Declare arrays for available, maximum, allocation, need, and finished resources.
3. Implement an initialization function to input available, maximum, and allocation resources for each process.
4. Implement a safety_check function to check if the system is in a safe state.
5. In the safety_check function, simulate resource allocation to determine if the system is safe or not.
6. Initialize variables and arrays in the main function.
7. Call the initialization function to input process and resource information.
8. Check if the system is in a safe state using the safety_check function and print the result.

Program:
```
#include <stdio.h>
#include <stdbool.h>
#define MAX_PROCESSES 5
#define MAX_RESOURCES 3
int available[MAX_RESOURCES];
int maximum[MAX_PROCESSES][MAX_RESOURCES];
int allocation[MAX_PROCESSES][MAX_RESOURCES];
int need[MAX_PROCESSES][MAX_RESOURCES];
bool finished[MAX_PROCESSES];
void initialize(int processes, int resources) {
    // Initialize available resources
    printf("Enter available resources:\n");
    for (int i = 0; i < resources; i++) {
        scanf("%d", &available[i]);
    }
    // Initialize maximum resources needed by each process
    printf("Enter maximum resources needed by each process:\n");
    for (int i = 0; i < processes; i++) {
        printf("Process %d:\n", i);
        for (int j = 0; j < resources; j++) {
            scanf("%d", &maximum[i][j]);
            need[i][j] = maximum[i][j];
        }
    }
    // Initialize allocation matrix
    printf("Enter allocation of resources to each process:\n");
    for (int i = 0; i < processes; i++) {
        printf("Process %d:\n", i);
        for (int j = 0; j < resources; j++) {
            scanf("%d", &allocation[i][j]);
            need[i][j] -= allocation[i][j];
        }
    }
    // Initialize finished array
    for (int i = 0; i < processes; i++) {
        finished[i] = false;
    }
}
bool safety_check(int processes, int resources) {
    int work[MAX_RESOURCES];
    bool finish[MAX_PROCESSES];

    // Initialize work and finish arrays
    for (int i = 0; i < resources; i++) {
        work[i] = available[i];
    }
    for (int i = 0; i < processes; i++) {
        finish[i] = finished[i];
    }
    int count = 0;
    while (count < processes) {
        bool found = false;
        for (int i = 0; i < processes; i++) {
            if (!finish[i]) {
                bool possible = true;
                for (int j = 0; j < resources; j++) {
                    if (need[i][j] > work[j]) {
                        possible = false;
                        break;
                    }
                }
                if (possible) {
                    found = true;
                    for (int j = 0; j < resources; j++) {
                        work[j] += allocation[i][j];
                    }
                    finish[i] = true;
                    count++;
                }
            }
        }
        if (!found) {
            return false; // Unsafe state
        }
    }
    return true; // Safe state
}
```

```
int main() {
    int processes, resources;

    printf("Enter number of processes: ");
    scanf("%d", &processes);
    printf("Enter number of resources: ");
    scanf("%d", &resources);

    initialize(processes, resources);

    if (safety_check(processes, resources)) {
        printf("System is in a safe state\n");
    } else {
        printf("System is in an unsafe state\n");
    }

    return 0;
}
```
Output:

Result:
The program successfully initializes the Banker's Algorithm and performs a safety check to determine whether the system is in a safe state.

EX. NO: 8b	SIMULATION OF RESOURCE ALLOCATION AND DEADLOCK AVOIDANCE USING BANKER’S ALGORITHM
DATE:24/04/2024	

Aim:
To simulate resource allocation and demonstrate deadlock avoidance using the Banker's Algorithm.

Algorithm:
1. Initialize data structures for resources and processes.
2. Define request_resources function to handle resource requests.
3. Check request against need to ensure it does not exceed what the process requires.
4. Check request against available resources to ensure resources are sufficient.
5. Simulate resource allocation by temporarily allocating the requested resources.
6. Perform safety check using a predefined function to ensure system stability.
7. Define safety_check function assuming its implementation is provided elsewhere.
8. In the main function, initialize arrays, simulate a resource request, call request_resources, and print the result.

Program:

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

#define MAX_PROCESSES 5
#define MAX_RESOURCES 3

int available[MAX_RESOURCES];
int maximum[MAX_PROCESSES][MAX_RESOURCES];
int allocation[MAX_PROCESSES][MAX_RESOURCES];
int need[MAX_PROCESSES][MAX_RESOURCES];
bool finished[MAX_PROCESSES];

bool request_resources(int process_id, int request[]) {
    // Check if the request exceeds the need
    for (int i = 0; i < MAX_RESOURCES; i++) {
        if (request[i] > need[process_id][i]) {
            return false;
        }
    }

    // Check if the request exceeds the available resources
    for (int i = 0; i < MAX_RESOURCES; i++) {
        if (request[i] > available[i]) {
            return false;
        }
    }

    // Simulate resource allocation
    for (int i = 0; i < MAX_RESOURCES; i++) {
        available[i] -= request[i];
        allocation[process_id][i] += request[i];
        need[process_id][i] -= request[i];
    }

    // Perform safety check
    if (safety_check()) {
        return true; // Request granted
    } else {
        // Rollback resource allocation
        for (int i = 0; i < MAX_RESOURCES; i++) {
            available[i] += request[i];
            allocation[process_id][i] -= request[i];
            need[process_id][i] += request[i];
        }
        return false; // Request denied
    }
}

bool safety_check() {
    // Safety check implementation is same as in Experiment 1
    // You can reuse the safety_check function defined in Experiment 1
    // Here, we assume that safety_check function is defined elsewhere in the code
}

int main() {
    // Initialize available, maximum, allocation, need, and finished arrays
    // Initialization code is same as in Experiment 1

    // Simulate resource requests and allocations
    int request[MAX_RESOURCES] = {1, 0, 2}; // Sample request
    int process_id = 0; // Sample process id
    if (request_resources(process_id, request)) {
        printf("Request granted\n");
    } else {
        printf("Request denied\n");
    }

    return 0;
}
```

Output:

AIM & ALGORITHM	30	
PROGRAM/EXECUTION	30	
OUTPUT	20	
VIVA	10	
RESULT	10	
TOTAL	100	

Result:
The program successfully simulates resource allocation and demonstrates deadlock avoidance using the Banker's Algorithm. It grants or denies resource requests based on whether they result in a safe state.

EX. NO: 9a	SIMULATION OF CONTIGOUS MEMORY ALLOCATION
DATE:22/05/2024	

Aim:
To simulate contiguous memory allocation and analyze its advantages and limitations.

Algorithm:

1. Initialize constants and structures for memory management.
2. Define allocate_contiguous_memory function to allocate memory blocks.
3. Define free_contiguous_memory function to free memory blocks.
4. Define display_memory_map function to display the memory map.
5. Initialize main function and declare necessary pointers.
6. Allocate memory blocks and store pointers.
7. Link memory blocks to form a linked list.
8. Display memory map and free memory blocks.

Program:

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

#define MEMORY_SIZE 1000

// Structure to represent a memory block
struct MemoryBlock {
    int start_address;
    int size;
    struct MemoryBlock* next;
};

struct MemoryBlock* allocate_contiguous_memory(int size) {
    static int last_allocated_address = 0;

    if (last_allocated_address + size > MEMORY_SIZE) {
        printf("Error: Out of memory\n");
        return NULL;
    }

    struct MemoryBlock* block = (struct MemoryBlock*)malloc(sizeof(struct
MemoryBlock));
    block->start_address = last_allocated_address;
    block->size = size;
    block->next = NULL;

    last_allocated_address += size;
    return block;
}

void free_contiguous_memory(struct MemoryBlock* block) {
    free(block);
}

void display_memory_map(struct MemoryBlock* head) {
    printf("Memory Map:\n");
    while (head != NULL) {
        printf("Start Address: %d, Size: %d\n", head->start_address, head->size);
        head = head->next;
    }
}

int main() {
    struct MemoryBlock* memory = NULL;

    // Allocate memory blocks
    struct MemoryBlock* block1 = allocate_contiguous_memory(200);
    struct MemoryBlock* block2 = allocate_contiguous_memory(300);
    struct MemoryBlock* block3 = allocate_contiguous_memory(150);

    // Link memory blocks
    block1->next = block2;
    block2->next = block3;

    // Display memory map
    display_memory_map(block1);

    // Free memory blocks
    free_contiguous_memory(block1);
    free_contiguous_memory(block2);
    free_contiguous_memory(block3);

    return 0;
}
```

Output:

Result:
The program successfully simulates contiguous memory allocation and displays the memory map, showing the allocated memory blocks.

EX. NO: 9b	SIMULATION OF LINKED LIST MEMORY ALLOCATION
DATE: 22/05/2024	

Aim:
To simulate linked list memory allocation and understand its benefits and drawbacks.

Algorithm:

1. Define a structure to represent a memory block with size and a pointer to the next block.
2. Implement a function to allocate memory for a linked list node, handling errors if memory allocation fails.
3. Implement a function to free memory allocated for a linked list, iterating through the list and freeing each node.
4. Implement a function to display the memory map, traversing the linked list and printing the size of each block.
5. In the main function, initialize a pointer to the head of the memory block linked list.
6. Allocate memory blocks using the allocate_linked_list_memory function and store the pointers to each block.
7. Link the memory blocks by setting the next pointers accordingly.
8. Display the memory map using the display_memory_map function, then free the allocated memory using the free_linked_list_memory function.

Program:

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

// Structure to represent a memory block
struct MemoryBlock {
   int size;
   struct MemoryBlock* next;
};

struct MemoryBlock* allocate_linked_list_memory(int size) {
   struct MemoryBlock* block = (struct MemoryBlock*)malloc(sizeof(struct
MemoryBlock));
   if (block == NULL) {
      printf("Error: Out of memory\n");
      return NULL;
   }
   block->size = size;
   block->next = NULL;
   return block;
}

void free_linked_list_memory(struct MemoryBlock* head) {
   while (head != NULL) {
      struct MemoryBlock* temp = head;
      head = head->next;
```

```
        free(temp);
    }
}

void display_memory_map(struct MemoryBlock* head) {
    printf("Memory Map:\n");
    while (head != NULL) {
        printf("Size: %d\n", head->size);
        head = head->next;
    }
}

int main() {
    struct MemoryBlock* memory = NULL;

    // Allocate memory blocks
    struct MemoryBlock* block1 = allocate_linked_list_memory(200);
    struct MemoryBlock* block2 = allocate_linked_list_memory(300);
    struct MemoryBlock* block3 = allocate_linked_list_memory(150);

    // Link memory blocks
    block1->next = block2;
    block2->next = block3;

    // Display memory map
    display_memory_map(block1);

    // Free memory blocks
    free_linked_list_memory(block1);

    return 0;
}
```

Output:

Result:
The program successfully simulates linked list memory allocation and displays the memory map, showing the allocated memory blocks.

EX. NO: 9c	SIMULATION OF PAGING MEMORY ALLOCATION
DATE: 22/05/2024	

Aim:
To simulate paging memory allocation and analyze its advantages and limitations.

Algorithm:

1. Define a structure to represent a page with page number and frame number attributes.
2. Define a structure to represent a page table containing an array of page structures.
3. Implement a function to initialize the page table, allocating memory for it and setting initial values for page and frame numbers.
4. Implement a function to display the contents of the page table.
5. In the main function, initialize a pointer to the page table using the initialize_page_table function.
6. Check if memory allocation for the page table was successful.
7. Display the contents of the page table using the display_page_table function.
8. Free the allocated memory for the page table before exiting the program.

Program:

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

#define PAGE_SIZE 256
#define NUM_PAGES 4

struct Page {
    int page_number;
    int frame_number;
};

struct PageTable {
    struct Page pages[NUM_PAGES];
};

struct PageTable* initialize_page_table() {
    struct PageTable* table = (struct PageTable*)malloc(sizeof(struct PageTable));
    if (table == NULL) {
        printf("Error: Out of memory\n");
        return NULL;
    }

    // Initialize page table
    for (int i = 0; i < NUM_PAGES; i++) {
        table->pages[i].page_number = i;
        table->pages[i].frame_number = -1; // No frame assigned initially
    }
    return table;
}
```

```
void display_page_table(struct PageTable* table) {
    printf("Page Table:\n");
    for (int i = 0; i < NUM_PAGES; i++) {
        printf("Page %d -> Frame %d\n", table->pages[i].page_number, table-
>pages[i].frame_number);
    }
}

int main() {
    struct PageTable* page_table = initialize_page_table();
    if (page_table == NULL) {
        return 1;
    }

    // Display page table
    display_page_table(page_table);

    free(page_table);

    return 0;
}
```

Output:

AIM & ALGORITHM	30	
PROGRAM/EXECUTION	30	
OUTPUT	20	
VIVA	10	
RESULT	10	
TOTAL	100	

Result:
The program successfully simulates paging memory allocation and displays the page table, showing the mapping of pages to frames.

EX. NO: 10a	IMPLEMENTATION OF PAGE REPLACEMENT TECHNIQUE-FIFO
DATE:29/05/2024	

Aim:
To implement and simulate the FIFO (First In First Out) page replacement technique.

Algorithm:

1. Define the number of frames and pages.
2. Implement a function to perform FIFO page replacement, counting the number of page faults.
3. Initialize a boolean array to track frames and an array to act as a FIFO queue.
4. Iterate through the pages, checking if each page is already in a frame.
5. If the page is not found in any frame, count it as a page fault and replace a page using FIFO.
6. In the main function, define a reference string representing the sequence of page references.
7. Call the FIFO page replacement function with the reference string and the number of pages.
8. Print the number of page faults returned by the FIFO page replacement function.

Program:
```
#include <stdio.h>
#include <stdbool.h>

#define NUM_FRAMES 3
#define NUM_PAGES 10

int fifo_replace(int pages[], int n) {
    bool frame[NUM_FRAMES] = {false};
    int frame_queue[NUM_FRAMES];
    int page_faults = 0;
    int rear = -1;

    for (int i = 0; i < n; i++) {
        int page = pages[i];
        bool found = false;

        // Check if page is already in a frame
        for (int j = 0; j < NUM_FRAMES; j++) {
            if (frame[j] && frame_queue[j] == page) {
                found = true;
                break;
            }
        }

        if (!found) {
            page_faults++;

            // Replace the page using FIFO
            rear = (rear + 1) % NUM_FRAMES;
            frame_queue[rear] = page;
            frame[rear] = true;
        }
    }

    return page_faults;
}

int main() {
    int pages[NUM_PAGES] = {1, 2, 3, 4, 1, 5, 6, 3, 7, 8}; // Reference string

    int page_faults = fifo_replace(pages, NUM_PAGES);
    printf("Page Faults (FIFO): %d\n", page_faults);

    return 0;
}
```

Output:

Result:
The program successfully implements the FIFO page replacement technique and calculates the number of page faults.

EX. NO: 10b	**IMPLEMENTATION OF PAGE REPLACEMENT TECHNIQUE-LRU**
DATE: 22/05/2024	

Aim:
To implement and simulate the LRU (Least Recently Used) page replacement technique.

Algorithm:

1. Define the number of frames and pages.
2. Implement a function to perform FIFO page replacement, counting the number of page faults.
3. Initialize a boolean array to track frames and an array to act as a FIFO queue.
4. Iterate through the pages, checking if each page is already in a frame.
5. If the page is not found in any frame, count it as a page fault and replace a page using FIFO.
6. In the main function, define a reference string representing the sequence of page references.
7. Call the FIFO page replacement function with the reference string and the number of pages.
8. Print the number of page faults returned by the FIFO page replacement function.

Program:

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

#define NUM_FRAMES 3
#define NUM_PAGES 10

int find_lru_frame(int pages[], int frame[], int n) {
    int lru_index = 0;
    int lru_time = frame[lru_index];

    for (int i = 0; i < n; i++) {
        int page = pages[i];
        bool found = false;
        for (int j = 0; j < NUM_FRAMES; j++) {
            if (frame[j] == page) {
                found = true;
                break;
            }
        }
        if (!found) {
            lru_index = i;
            break;
        }
        if (frame[i] < lru_time) {
            lru_index = i;
            lru_time = frame[i];
        }
    }
    return lru_index;
}

int lru_replace(int pages[], int n) {
    int frame[NUM_FRAMES] = {0};
    int page_faults = 0;

    for (int i = 0; i < n; i++) {
        int page = pages[i];
        bool found = false;
        for (int j = 0; j < NUM_FRAMES; j++) {
            if (frame[j] == page) {
                found = true;
                break;
            }
        }
        if (!found) {
            int lru_index = find_lru_frame(pages, frame, i);
            frame[lru_index] = page;
            page_faults++;
        }
    }
    return page_faults;
}

int main() {
    int pages[NUM_PAGES] = {1, 2, 3, 4, 1, 5, 6, 3, 7, 8}; // Reference string

    int page_faults = lru_replace(pages, NUM_PAGES);
    printf("Page Faults (LRU): %d\n", page_faults);

    return 0;
}
```

Output:

Result:
The program successfully implements the LRU page replacement technique and calculates the number of page faults.

EX. NO: 10c	**IMPLEMENTATION OF PAGE REPLACEMENT TECHNIQUE-OPTIMAL PAGE**
DATE: 22/05/2024	

Aim:
To implement and simulate the Optimal page replacement technique.

Algorithm:
1. Define the number of frames and pages.
2. Implement a function to find the index of the optimal frame to replace.
3. Initialize an array to track frames and a variable to count page faults.
4. Iterate through the pages, checking if each page is already in a frame.
5. If the page is not found in any frame, find the index of the optimal frame to replace using the find_optimal_frame function.
6. Replace the optimal frame with the current page and increment the page fault count.
7. In the main function, define a reference string representing the sequence of page references.
8. Call the Optimal page replacement function with the reference string and the number of pages, then print the number of page faults returned.

Program:

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

#define NUM_FRAMES 3
#define NUM_PAGES 10

int find_optimal_frame(int pages[], int frame[], int n, int index) {
    int farthest = index;
    int longest = 0;

    for (int i = 0; i < NUM_FRAMES; i++) {
        bool found = false;
        for (int j = index; j < n; j++) {
            if (frame[i] == pages[j]) {
                found = true;
                if (j > farthest) {
                    farthest = j;
                    longest = i;
                }
                break;
            }
        }
        if (!found) {
            return i;
        }
    }
    return longest;
}
```

```
int optimal_replace(int pages[], int n) {
    int frame[NUM_FRAMES] = {0};
    int page_faults = 0;

    for (int i = 0; i < n; i++) {
        int page = pages[i];
        bool found = false;
        for (int j = 0; j < NUM_FRAMES; j++) {
            if (frame[j] == page) {
                found = true;
                break;
            }
        }
        if (!found) {
            int optimal_index = find_optimal_frame(pages, frame, n, i + 1);
            frame[optimal_index] = page;
            page_faults++;
        }
    }
    return page_faults;
}

int main() {
    int pages[NUM_PAGES] = {1, 2, 3, 4, 1, 5, 6, 3, 7, 8}; // Reference string

    int page_faults = optimal_replace(pages, NUM_PAGES);
    printf("Page Faults (Optimal): %d\n", page_faults);

    return 0;
}
```

Output:

AIM & ALGORITHM	30	
PROGRAM/EXECUTION	30	
OUTPUT	20	
VIVA	10	
RESULT	10	
TOTAL	100	

Result:
The program successfully implements the Optimal page replacement technique and calculates the number of page faults.

EX. NO: 11a	DISK SCHEDULING ALGORITHM (First Come First Serve)
DATE: 30/05/2024	

Aim:
To write a C Program to implement FCFS Disk scheduling algorithm.

Algorithm:

1. Input the maximum number of cylinders and work queue and its head starting position.
2. First Come First Serve Scheduling (FCFS) algorithm – The operations are performed in order requested
3. There is no reordering of work queue.
4. Every request is serviced, so there is no starvation
5. At the other end, the direction of head movement is reversed and servicing continues.
6. The head continuously scans back and forth across the disk.
7. The seek time is calculated.
8. Display the seek time and terminate the program.

Program:

```
#include<stdio.h>
void main()
{
int t[20], n, i, j, tohm[20], tot=0; float avhm;
int i;
printf("enter the no.of tracks");
scanf("%d",&n);
printf("enter the tracks to be traversed");
for(i=2;i<n+2;i++)
scanf("%d",&t[i]);
for(i=1;i<n+1;i++)
{
tohm[i]=t[i+1]-t[i];
if(tohm[i]<0)
tohm[i]=tohm[i]*(-1);
}
for(i=1;i<n+1;i++)
tot+=tohm[i];
avhm=(float)tot/n;
printf("tracks traversed\t difference between tracks\n");
for(i=1;i<n+1;i++)
printf("%d\t\t\t%d\n",t[i],tohm[i]);
printf("\n average header movements:%f",avhm);
}
```

Output :

Result:
Thus, the C program to implement FCFS Disk scheduling algorithm was written and its output was verified.

EX. NO: 11b	DISK SCHEDULING ALGORITHM (SCAN)
DATE: 30/05/2024	

Aim:
To write a C Program to implement SCAN Disk scheduling algorithm.

Algorithm:
1. Input the maximum number of cylinders and work queue and its head starting position.
2. SCAN Scheduling algorithm –The disk arm starts at one end of the disk, and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk.
3. At the other end, the direction of head movement is reversed and servicing continues.
4. The head continuously scans back and forth across the disk.
5. The seek time is calculated.
6. Display the seek time and terminate the program.

Program:

```
#include<stdio.h>
main()
{
int t[20], d[20], h, i, j, n, temp, k, atr[20], tot, p, sum=0; clrscr();
printf("enter the no of tracks to be traveresed");
scanf("%d'",&n);
printf("enter the position of head");
scanf("%d",&h);
t[0]=0;t[1]=h;
printf("enter total tracks");
scanf("%d",&tot);
t[2]=tot-1;
printf("enter the tracks");
for(i=3;i<=n+2;i++)
scanf("%d",&t[i]);
for(i=0;i<=n+2;i++)
for(j=0;j<=(n+2)-i-1;j++)
if(t[j]>t[j+1])
{
temp=t[j];
t[j]=t[j+1];
t[j+1]=temp;
}
for(i=0;i<=n+2;i++)
if(t[i]==h)
j=i;
break;
p=0;
while(t[j]!=tot-1)
{
```

```
atr[p]=t[j];
j++;
p++;
}

atr[p]=t[j];
p++;
i=0;
while(p!=(n+3) && t[i]!=t[h])
{
atr[p]=t[i]; i++;
}}
```

Output:

Result:
Thus, the C program to implement SCAN Disk scheduling algorithm was written and its output was verified.

EX. NO: 11c	DISK SCHEDULING ALGORITHM (Shortest Seek Time First)
DATE: 30/05/2024	

Aim:
To write a C Program to implement SSTF Disk scheduling algorithm.

Algorithm:

1. Input the maximum number of cylinders and work queue and its head starting position.
2. Shortest Seek Time First Scheduling (SSTF) algorithm – This algorithm selects the request with the minimum seek time from the current head position.
3. Since seek time increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest to the current head position.
4. The seek time is calculated.
5. At the other end, the direction of head movement is reversed and servicing continues.
6. The head continuously scans back and forth across the disk.
7. The seek time is calculated.
8. Display the seek time and terminate the program.

Program:

```
#include<stdio.h>
main()
{
int t[20], d[20], h, i, j, n, temp, k, atr[20], tot, p, sum=0; clrscr();
printf("enter the no of tracks to be traveresed");
scanf("%d'",&n);
printf("enter the position of head");
scanf("%d",&h);
t[0]=0;t[1]=h;
printf("enter the tracks");
for(i=2;i<n+2;i++)
scanf("%d",&t[i]);
for(i=0;i<n+2;i++) {
for(j=0;j<(n+2)-i-1;j++)
{ if(t[j]>t[j+1])
{
temp=t[j]; t[j]=t[j+1];
t[j+1]=temp;
} } }
for(i=0;i<n+2;i++)
if(t[i]==h)
j=i;k=i; p=0;
while(t[j]!=0)
{
atr[p]=t[j];
j--;
p++;
}
atr[p]=t[j];
for(p=k+1;p<n+2;p++,k++)
atr[p]=t[k+1];
for(j=0;j<n+1;j++)
{
if(atr[j]>atr[j+1])
d[j]=atr[j]-atr[j+1];
else
d[j]=atr[j+1]-atr[j]; sum+=d[j];
}
printf("\naverage header movements:%f",(float)sum/n);
getch();
}
```

Output:

AIM & ALGORITHM	30	
PROGRAM/EXECUTION	30	
OUTPUT	20	
VIVA	10	
RESULT	10	
TOTAL	100	

Result:
Thus, the C program to implement SSTF Disk scheduling algorithm was written and its output was verified.

EX. NO: 12a	FILE ORGANIZATION TECHNIQUE (Single level directory)
DATE: 04/06/2024	

Aim:
To write a C program to implement File Organization concept using the technique Single level directory.

Algorithm:
1. Start the Program
2. Obtain the required data through char and int data types.
3. Enter the filename, index block.
4. Print the file name index loop.
5. Fill is allocated to the unused index blocks
6. This is allocated to the unused linked allocation.
7. Stop the execution

Program:
```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<graphics.h>
void main()
{
int gd=DETECT,gm,count,i,j,mid,cir_x;
char fname[10][20];
clrscr();
initgraph(&gd,&gm,"c:\\tc\\bgi");
cleardevice();
setbkcolor(GREEN);
puts("Enter no of files do u have?");
scanf("%d",&count);
for(i=0;i<count;i++)
{
printf("Enter file %d name",i+1);
scanf("%s",fname[i]);
}
setbkcolor(GREEN);
setfillstyle(1,MAGENTA);
mid=640/count;
cir_x=mid/3;
bar3d(270,100,370,150,0,0);
settextstyle(2,0,4);
settextjustify(1,1);
outtextxy(320,125,"Root Directory");
setcolor(BLUE);
for(j=0;j<i;j++,cir_x+=mid)
{

line(320,150,cir_x,250);
fillellipse(cir_x,250,30,30);
outtextxy(cir_x,250,fname[j]);
}
getch();
}
```

Output:

Result:
Thus, the C program for single level directory file organization was written and its output was verified.

EX. NO: 12b	FILE ORGANIZATION TECHNIQUE (two level directory)
DATE: 04/06/2024	

Aim:
To write a C program to implement File Organization concept using the technique two level directory**.**
Algorithm:

1. Start the Program.
2. Initialize Variables.
3. Input Number of Directories.
4. Input Directory Names and Number of Files.
5. Input File Names for Each Directory.
6. Display Directory Structure.
7. Print File Names for Each Directory.
8. Stop the Program.

Program:

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

void main()
{
   int count_dir, count_files;

   printf("Enter number of directories: ");
   scanf("%d", &count_dir);

   char dirname[count_dir][20];
   char fname[count_dir][10][20];

   for (int i = 0; i < count_dir; i++)
   {
      printf("Enter directory %d name: ", i + 1);
      scanf("%s", dirname[i]);

      printf("Enter number of files in directory '%s': ", dirname[i]);
      scanf("%d", &count_files);

      for (int j = 0; j < count_files; j++)
      {
         printf("Enter file %d name for directory '%s': ", j + 1, dirname[i]);
         scanf("%s", fname[i][j]);
      }
   }
```

```
    // Displaying directory and files
    printf("\nDirectory Structure:\n");
    for (int i = 0; i < count_dir; i++)
    {
        printf("\n%s:\n", dirname[i]);
        for (int j = 0; j < count_files; j++)
        {
            printf(" - %s\n", fname[i][j]);
        }
    }
}
```

Input:

Output:

AIM & ALGORITHM	30	
PROGRAM/EXECUTION	30	
OUTPUT	20	
VIVA	10	
RESULT	10	
TOTAL	100	

Result:
Thus, the C program for two level directory file organization was written and its output was verified.