Date: - 01-01-2025

EXPERIMENT NO - 1

FAMILIARIZATION OF BASIC LINUX COMMANDS

AIM

To familiarize the Basic Linux Commands.

COMMANDS

1. DATE

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

SYNOPSIS

date [OPTION]... [+FORMAT]

OPTION

-d, --date=STRING
display time described by STRING, not 'now'
-r, --reference=FILE
display the last modification time of FILE
-s, --set=STRING
set time described by STRING

It displays the current time in the given FORMAT, or set the system date.

2. CALENDAR

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

SYNOPSIS

calendar [-abw] [-A num] [-B num] [-l num] [-e num]
[-f calendarfile] [-t [[[cc]yy]mm]dd]

OPTIONS

-A num

Print lines from today and next num days (forward, future). Defaults to one. (same as -l)

-a

Process the "calendar" files of all users and mail the results to them. This requires superuser privileges.

-b

Enforce special date calculation mode for Cyrillic calendars.

3. ECHO

This command is used to display a line of text.

SYNOPSIS

echo [SHORT-OPTION]... [STRING]...
echo LONG-OPTION

OPTIONS

-n	do not output the trailing newline
-e	enable interpretation of backslash escapes
-E	disable interpretation of backslash escapes (default)

4. WHO

This command shows who is logged on.
It prints information about users who are currently logged in.

SYNOPSIS

who [OPTION]... [FILE | ARG1 ARG2]

OPTIONS

-b, --boot	time of last system boot
-d, --dead	print dead processes
-H, --heading	print line of column heading

5. LS

This command list directory contents.
It displays list information about the FILEs

SYNOPSIS

ls [OPTION]... [FILE]...

OPTIONS

-a, --all	do not ignore entries starting with .
-C	list entries by columns
-l	use a long listing format

6. PWD

This command prints name of current/working directory.
It prints the full filename of the current working directory.

SYNOPSIS

pwd [OPTION]...

OPTIONS

-L, --logical	use PWD from environment, even if it contains symlinks
-P, --physical	avoid all symlinks

7. MKDIR

This command makes directories.
It creates the DIRECTORY (ies), if they do not already exist.

SYNOPSIS

mkdir [OPTION]... DIRECTORY...

OPTIONS

-m, --mode=MODE set file mode (as in chmod), not a=rwx – umask

-p, --parents no error if existing, make parent directories as needed

-v, --verbose print a message for each created directory

8. RMDIR

This command removes empty directories.
It removes the DIRECTORY (ies), if they are empty.

SYNOPSIS

rmdir [OPTION]... DIRECTORY...

OPTIONS

-p, --parents remove DIRECTORY and its ancestors;

-v, --verbose output a diagnostic for every directory processed

9. MV

This command moves (rename) files.
It renames SOURCE to DEST, or move SOURCE(s) to DIRECTORY.

SYNOPSIS

mv [OPTION]... [-T] SOURCE DEST
mv [OPTION]... SOURCE... DIRECTORY
mv [OPTION]... -t DIRECTORY SOURCE...

OPTIONS

--backup[=CONTROL] make a backup of each existing destination file

-b like --backup but does not accept an argument

-f, --force do not prompt before overwriting

10. RM

This command removes files or directories.
It removes each specified file. By default, it does not remove directories.

SYNOPSIS

rm [OPTION]... [FILE]...

OPTIONS

-i	prompt before every removal
-d, --dir	remove empty directories
-f, --force	ignore nonexistent files and arguments, never prompt

11. CP

This command copies files and directories.
It copies SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.

SYNOPSIS

cp [OPTION]... [-T] SOURCE DEST
cp [OPTION]... SOURCE... DIRECTORY

OPTIONS

-b	like --backup but does not accept an argument
-H	follow command-line symbolic links in SOURCE
-l, --link	hard link files instead of copying

12. WC

This command print newline, word, and byte counts for each file.
It prints newline, word, and byte counts for each FILE, and a total line if more than one FILE is specified.

SYNOPSIS

wc [OPTION]... [FILE]...
wc [OPTION]... --files0-from=F

OPTIONS

-c, --bytes	print the byte counts
-m, --chars	print the character counts
-l, --lines	print the newline counts

13.HEAD

This command output the first part of files.
It prints the first 10 lines of each FILE to standard output.

SYNOPSIS

head [OPTION]... [FILE]...

OPTIONS

-q, --quiet, --silent	never print headers giving file names
-z, --zero-terminated	line delimiter is NUL, not newline

14.TAIL

This command output the last part of files.
It prints the last 10 lines of each FILE to standard output. With more than one FILE, precede each with a header giving the file name.

SYNOPSIS

tail [OPTION]... [FILE]...

OPTIONS

-c, --bytes=[+]NUM	output the last NUM bytes; or use -c +NUM to output starting with byte NUM of each file
-F s	ame as --follow=name –retry

15.PIPE

pipe() creates a pipe ,a unidirectional data channel that can be used for inter process communication. The array pipefd is used to return two file descriptors referring to the ends of pipe, pipefd[0] refers to the read end ,pipefd[1]refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernal until it is read from the read end of the pipe.

SYNOPSIS

int pipe (int pipefd[2]);

16. CAT

This command concatenate files and print on the standard output.
It concatenates FILE(s) to standard output.

SYNOPSIS

cat [OPTION]... [FILE]...

OPTIONS

-e	equivalent to -vE
-E, --show-ends	display $ at end of each line

17. GREP

This command (grep, egrep, fgrep, rgrep) prints lines matching a pattern.
grep searches for PATTERN in each FILE. A FILE of "-" stands for standard input. If no FILE is given, recursive searches examine the working directory, and non-recursive searches read standard input. By default, grep prints the matching lines.

SYNOPSIS

grep [OPTIONS] PATTERN [FILE...]
grep [OPTIONS] -e PATTERN ... [FILE...]

OPTIONS

-E, --extended-regexp
Interpret PATTERN as an extended regular expression
-G, --basic-regexp
Interpret PATTERN as a basic regular expression (BRE, see below). This is the default.

18.SUDO

This command (sudo, sudoedit) executes a command as another user. sudo allows a permitted user to execute a command as the super user or another user, as specified by the security policy.

SYNOPSIS

sudo -h | -K | -k | -V

sudo -v [-AknS] [-a type] [-g group] [-h host] [-p prompt][-u user]

OPTIONS

-b, --background

Run the given command in the background.

-H, --set-home

Request that the security policy set the HOME environment variable to the home directory specified by the target user's password database entry.

19.CHMOD

This command changes file mode bits. The manual page documents the GNU version of chmod. chmod changes the file mode bits of each given file according to mode.

SYNOPSIS

chmod [OPTION]... MODE[,MODE]... FILE...

chmod [OPTION]... OCTAL-MODE FILE...

OPTIONS

-c, --changes like verbose but report only when a change is made

-f, --silent, --quiet suppress most error messages

-v, --verbose output a diagnostic for every file processed

20.CHOWN

This command changes file owner and group. chown changes the user and/or group ownership of each given file.

SYNOPSIS

chown [OPTION]... [OWNER][:[GROUP]] FILE...
chown [OPTION]... --reference=RFILE FILE...

OPTION

-c, --changes like verbose but report only when a change is made
-f, --silent, --quiet suppress most error messages

21. PING

This command sends ICMP ECHO_REQUEST to network hosts.
ping uses the ICMP protocol's mandatory ECHO_REQUEST datagram to elicit an ICMP ECHO_RESPONSE from a host or gateway.
ECHO_REQUEST datagrams (``pings'') have an IP and ICMP header, followed by a struct timeval and then an arbitrary number of ``pad'' bytes used to fill out the packet.

SYNOPSIS
ping [-aAbBdDfhLnOqrRUvV46] [-c count] [-F flowlabel] [-i interval] [-I interface] [-l preload] [-m mark] [-M pmtud-isc_option] [-N nodeinfo_option] [-w deadline] [-W timeout] [-p pattern] [-Q tos] [-s packetsize] [-S sndbuf] [-t ttl] [-T timestamp option] [hop ...] destination

OPTIONS

-4 Use IPv4 only.
-6 Use IPv6 only.
-a Audible ping.
-b Allow pinging a broadcast address.

22. NETSTAT

This command prints network connections, routing tables, interface statistics, masquerade connections, and multicast memberships. Netstat prints information about the Linux networking sub-system.

SYNOPSIS

netstat [address_family_options] [--tcp|-t] [--udp|-u]
[--udplite|-U] [--sctp|-S] [--raw|-w] [--l2cap|-2]
[--rfcomm|-f] [--listening|-l] [--all|-a] [--numeric|-n]
[--numeric-hosts] [--numeric-ports] [--numeric-users] [--sym-
bolic|-N] [--extend|-e[--extend|-e]] [--timers|-o] [--pro-
gram|-p] [--verbose|-v] [--continuous|-c] [--wide|-W]

OPTIONS

--statistics, -s Display summary statistics for each protocol.

-e, --extend Display additional information. Use this option twice for maximum detail.

-o, --timers Include information related to networking timers.

RESULT

Familiarized the Basic Linux commands successfully.

Date: - 09-01-2025

EXPERIMENT NO - 2

FAMILIARIZATION OF LINUX SYSTEM CALLS

AIM

To familiarize the various System Calls of Linux Operating System (fork, exec, getpid, exit, wait, close, stat, opendir, readdir).

SYSTEM CALLS

1. FORK

   Syntax : pid_t fork(void);

   fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.

2. EXEC

   Syntax : extern char **environ;
   int execl(const char *path, const char *arg, ... /* (char *) NULL */);
   int execlp(const char *file, const char *arg, .../* (char *) NULL */);

   The exec system call is used to execute a file which is residing in an active process. When exec is called the previous executable file is replaced and new file is executed.

3. GETPID

   Syntax : pid_t getpid(void);

   getpid() returns the process ID of the calling process. This is often used by routines that generate unique temporary filenames.

4. GETPPID

   Syntax : pid_t getppid(void);

getppid() returns the process ID of the parent of the calling process.

5. EXIT

Syntax : void exit(int status);

The exit() is such a function or one of the system calls that is used to terminate the process.This system call defines that the thread execution is completed especially in the case of a multi-threaded environment. For future reference, the status of the process is captured. After the use of exit() system call, all the resources used in the process are retrieved by the operating system and then terminate the process.

6. WAIT

Syntax : pid_t wait(int *wstatus);

All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal.

7. CLOSE

Syntax : int close(int fd);

close() closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks held on the file it was associated with, and owned by the process, are removed regardless of the file descriptor that was used to obtain the lock.

8. STAT

Syntax : stat [OPTION]... FILE...

Display file or file system status. Stat system call is a system call in Linux to check the status of a file such as to check when the file was accessed. The stat() system call actually returns file attributes. The file attributes of an

inode are basically returned by stat() function. An inode contains the metadata of the file.

9. OPENDIR

Syntax : DIR *opendir(const char *name);
DIR *fdopendir(int fd);

The opendir() function opens a directory stream corresponding to the directory name, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

10. READDIR

Syntax : struct dirent *readdir(DIR *dirp);

The readdir() function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by dirp. It returns NULL on reaching the end of the directory stream or if an error occurred

RESULT

Familiarized the various System Calls of Linux Operating System (fork, exec, getpid, exit, wait, close, stat, opendir, readdir) successfully.

PROGRAM: OPENDIR, READDIR

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

int main()
{
   DIR *dir;
   struct dirent *entry;
   dir = opendir(".");
   if (dir == NULL)
     {
      perror("Error opening directory");
      exit(EXIT_FAILURE);
      }
   while ((entry = readdir(dir)) != NULL)
       {
        printf("%s\n", entry->d_name);
        }

closedir(dir);
return 0;
}
```

OUTPUT

```
.
..
.vs code
dir.c
dir.exe
```

Date: - 22-01-2025

EXPERIMENT NO - 3

PROGRAMS USING THE I/O SYSTEM CALLS OF LINUX OPERATING SYSTEM

AIM

To write C programs using the I/O system calls of Linux operating system (fork, exec, getpid, exit, wait, close, stat, opendir, readdir).

ALGORITHM: OPENDIR, READDIR

1. Start
2. Include necessary header files: <stdio.h>, <stdlib.h>, <sys/types.h> and <dirent.h>
3. Declare variables:
    a. DIR * dir to gold a pointer to the directory structure.
    b. struct dirent *entry to hold a pointer to the directory entry structure
4. Open the current directory using opendir(“.”).
5. Check if the directory opening was successful:
    a. If successful,proceed.
    b. If unsuccessful, print an error message using perror() and exit the program with failure status using exit(EXIT_FAILURE).
6. While there are still directory entries to read:
    a. Use readdir(dir) to read the next directory entry.
    b. Check if the returned entry pointer is not NULL.
    c. If not NULL, print the name of the directory entry.
    d. If NULL, exit the loop.
7. Close the directory using closedir(dir).
8. Return 0 to indicate successful program execution.
9. End.

PROGRAM: FORK, GETPID

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
  pid_t pid;
  pid = fork();
if (pid < 0)
{
   fprintf(stderr, "Fork failed\n");
    exit(EXIT_FAILURE);
}
else if (pid == 0)
{
    printf("Child process: PID = %d\n", getpid());
}
else
{
    printf("Parent process: PID = %d\n", getpid());
}
return 0;
}
```

OUTPUT

```
Parent process: PID = 59820
Child process: PID = 59821
```

ALGORITHM: FORK, GETPID

1. Start
2. Include necessary header files: <stdio.h>, <stdlib.h> and <unistd.h>.
3. Declare a variable pid of type pid_t to store the process ID.
4. Call fork() to create a new process.
5. Check the return value of fork():
   a. If fork() returns a negative value, print "Fork failed" and exit the program with failure status using exit(EXIT_FAILURE).
   b. If fork() returns 0, it means the current process is the child process.
6. Print Child process ID obtained from getpid().
7. If fork() returns a positive value, it means the current process is the parent process:
   a. Print Parent process ID obtained from getpid().
8. Return 0 to indicate successful program execution.
9. End.

PROGRAM: EXEC

```
#include <stdio.h>
#include <unistd.h>
int main()
{
  printf("Executing ls command using exec system call...\n");
  execlp("ls", "ls", "-l", NULL);
  perror("Error executing command");
return 1;
}
```

OUTPUT

```
Executing ls command using exec system call...
total 20
-rwxrwxr-x 1 user user 8384 Jan 22 09:18 a.out
 -rw-rw-r-- 1 user user 202 Jan 22 09:18 Exec.c
-rw-rw-r-- 1 user user 380 Jan 22 09:17 getpid.c
-rw-rw-r-- 1 user user 375 Jan 22 09:16 opendir.c
```

ALGORITHM: EXEC

1. Start
2. Include necessary header files: <stdio.h> and <unistd.h>.
3. Print a message indicating that the program is executing the "ls" command using the exec system call.
4. Call the execlp function to execute the "ls" command:
   a. The first argument is the command to execute ("ls").
   b. The subsequent arguments are the command and its arguments ("-l").
   c. The last argument must be NULL.
5. If execlp returns, it indicates an error occurred:
   a. Print an error message using perror.
6. Return 1 to indicate an error occurred during program execution.
7. End.

PROGRAM: FORK, WAIT

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main(){
  pid_t pid;
  pid = fork();
   if (pid < 0) {
     perror("Fork failed");
     exit(EXIT_FAILURE); }
   else if (pid == 0) {
    printf("Child process: PID = %d\n", getpid());
    printf("Child process: Parent PID = %d\n", getppid());
    sleep(2);
    printf("Child process: Exiting\n");
    exit(EXIT_SUCCESS);
   }
  else{
   printf("Parent process: PID = %d\n", getpid());
   printf("Parent process: Waiting for child to exit...\n");
   int status;
   wait(&status);
   printf("Parent process: Child exited with status %d\n", status); }
 return 0;
}
```

OUTPUT

Parent process: PID = 60160

Parent process: Waiting for child to exit...

Child process: PID = 60161

Child process: Parent PID = 60160

Child process: Exiting

Parent process: Child exited with status 0

ALGORITHM: FORK, WAIT

1. Start
2. Include necessary header file: <stdio.h> and <unistd.h>.
3. Print a message indicating that the program is executing the "ls" command using the exec system call.
4. Call the execlp function to execute the "ls" command:
5. The first argument is the command to execute ("ls").
6. The subsequent arguments are the command and its arguments ("-l").
7. The last argument must be NULL.
8. If execlp returns, it indicates an error occurred:
9. Print an error message using perror.
10. Return 1 to indicate an error occurred during program execution.
11. End.

RESULT

C programs using the I/O system calls of Linux operating system (fork, exec, getpid, exit, wait, close, stat, opendir, readdir) has been executed successfully and verified the output.

PROGRAM

```
#!/bin/bash
echo -n "Enter the number:"
read num
if [ $((num % 2)) -eq 0 ]; then
   echo "$num is even"
else
  echo "$num is odd"
fi
```

OUTPUT

```
Enter a number: 18
18 is even.
Enter a number: 9
9 is odd.
```

Date: - 22-01-2025

PROGRAM - 4

SHELL PROGRAMMING

AIM

To write simple programs using Shell programming.

ALGORITHM: ODD OR EVEN

1. Start
2. Print "Enter a number:"
3. Read input number and store it in a variable 'num'.
4. Check if the number is divisible by 2:
    a. If the remainder of dividing 'num' by 2 is equal to 0, then the number is even.

       Print "$num is even."

    b. If the remainder of dividing 'num' by 2 is not equal to 0, then the number is odd. Print "$num is odd."
5. End

PROGRAM

```
#!/bin/bash
echo -n "Enter the first number:"
read num1
echo -n "Enter the second number:"
read num2
if [ $num1 -gt $num2 ]; then
     echo "$num1 is the largest."
elif [ $num1 -lt $num2 ]; then
     echo "$num2 is the largest."
else
    echo "Both numbers are equal."
fi
```

OUTPUT

```
Enter the first number: 18
Enter the second number: 9
18 is the largest.

Enter the first number: 50
Enter the second number: 60
60 is the largest.
```

ALGORITHM: LARGEST OF TWO NUMBERS

1. Start
2. Print "Enter the first number:"
3. Read input number and store it in a variable 'num1'
4. Print "Enter the second number:"
5. Read input number and store it in a variable 'num2'
6. Check if 'num1' is greater than 'num2':
    a. If 'num1' is greater than 'num2', then 'num1' is the largest.
        i. Display "$num1 is the largest."
    b. If 'num1' is less than 'num2', then 'num2' is the largest.
        i. Display "$num2 is the largest."
    c. If 'num1' is equal to 'num2', then both numbers are equal.
        i. Display "Both numbers are equal."
7. End

PROGRAM

```
#!/bin/bash
echo -n "Enter a Number:"
read n
i=`expr $n - 1`
p=1
while [ $i -ge 1 ]
  do
   n=`expr $n \* $i`
   i=`expr $i - 1`
 done
echo "The Factorial of the given Number is $n"
```

OUTPUT

```
Enter a number: 5
The Factorial of the given Number is 120
```

ALGORITHM: FACTORIAL OF A NUMBER

1. Start
2. Read the input number and store it in a variable n.
3. Initialize a variable i to n - 1.
4. Initialize a variable p to 1.
5. Start a while loop with the condition [$i -ge 1] to iterate until i is greater than or equal to 1.
6. Inside the loop:
    a. Multiply n with i and store the result back into n.
    b. Decrement i by 1.
7. After the loop, n will contain the factorial of the original input number.
8. Display the factorial of the given number using echo.
9. Stop.

PROGRAM

```
#!/bin/bash
echo "Enter a year:"
read year
if [ $((year % 4)) -eq 0 ]; then
if [ $((year % 100)) -ne 0 ] || [ $((year % 400)) -eq 0 ]; then
echo "$year is a leap year."
else
echo "$year is not a leap year."
fi
else
echo "$year is not a leap year."
fi
```

OUTPUT

```
Enter a year:2028
2028 is a leap year.

Enter a year:2017
2017 is not a leap year.
```

ALGORITHM: LEAP YEAR

1. Start
2. Read the input year and store it in a variable year.
3. Check if the year is divisible by 4 using the expression $((year % 4)) -eq 0.
4. If the year is divisible by 4, proceed to the next step; otherwise, go to step 7.
5. Check if the year is not divisible by 100 or if it is divisible by 400.
6. If the condition in step 5 is true, display "Leap year."; otherwise, display " Not a leap year."
7. If the year is not divisible by 4, display "$year is not a leap year."
8. Stop

RESULT

Using programming basics, simple shell scripts were executed successfully and verified the output.

PROGRAM: FCFS

```
#include <stdio.h>
int main() {

   int n,i;
   printf("Enter the no.of process: ");
   scanf("%d",&n);


   int at[n],bt[n],tat[n],wt[n],ct[n],p[n];

   for(i=0;i<n;i++)
   {
      p[i]=i+1;
      printf("Process %d Arrival Time:",p[i]);
      scanf("%d",&at[i]);

      printf("Process %d Burst Time:",p[i]);
      scanf("%d",&bt[i]);

   }

   int total_tat=0,total_wt=0;
   float avg_tat,avg_wt;

   for(i=0;i<n;i++)
   {
      for(int j=i+1;j<n;j++)
      {
         if(at[i]>at[j])
         {
            int temp=at[i];
            at[i]=at[j];
            at[j]=temp;

            temp=bt[i];
            bt[i]=bt[j];
            bt[j]=temp;
```

Date: - 29-01-2025

EXPERIMENT NO - 5

IMPLEMENTATION OF CPU SCHEDULING ALGORITHMS

AIM

To write C programs to implement various CPU scheduling algorithms.

5.1) FCFS 5.2) SJF 5.3) Round Robin 5.4) Priority

ALGORITHM: FCFS

1. Start.
2. Declare variables to store the number of processes, arrival time, burst time, waiting time, and turnaround time for each process.
3. Read the number of processes.
4. Read arrival time and burst time for each process.
5. Sort the arrival time and corresponding burst time using bubble sort.
6. Set completion_time [0]=at[0]+bt[0], tat[0]=ct[0]-at[0], wt[0]=tat[0]-bt[0]
7. Calculate ct[i] as ct[i-1] + burst_time[i-1].
8. Calculate turnaround_time[i] as ct[i] – arrival_time[i].
9. Calculate waiting_time[i] as tat[i] – burst_time[i].
10. Add waiting_time[i] to avg_waiting_time.
11. Add turnaround_time[i] to avg_turnaround_time.
12. Divide avg_waiting_time and avg_turnaround_time by num_processes.
13. Display a table showing the process number, arrival time, burst time, waiting time, and turnaround time for each process.
14. Display the average waiting time and average turnaround time.
15. End.

```
    temp=p[i];
            p[i]=p[j];
            p[j]=temp;


          }
       }
    }
          ct[0]=at[0]+bt[0];
          tat[0]=ct[0]-at[0];
          wt[0]=tat[0]-bt[0];

           for(i=1;i<n;i++)
           {
            ct[i]=ct[i-1]+bt[i];
            tat[i]=ct[i]-at[i];
            wt[i]=tat[i]-bt[i];

         }
           for(i=0;i<n;i++)
           {
            total_tat+=tat[i];
            total_wt+=wt[i];
           }


    avg_tat=(float)total_tat/n;
     avg_wt=(float)total_wt/n;

     printf("\nPno.\t AT\t BT\t CT\t TAT\t WT\n");
     for(i=0;i<n;i++)
     {
        printf("%d \t%d \t%d \t%d \t%d \t%d\n",p[i],at[i],bt[i],ct[i],tat[i],wt[i]);
     }

    printf("\nAvg TAT: %.2f\n",avg_tat);
     printf("Avg WT: %.2f",avg_wt);

    return 0;
}
```

OUTPUT

Enter the no.of process: 5

Process 1 Arrival Time:4

Process 1 Burst Time:3

Process 2 Arrival Time:5

Process 2 Burst Time:2

Process 3 Arrival Time:2

Process 3 Burst Time:5

Process 4 Arrival Time:1

Process 4 Burst Time:4

Process 5 Arrival Time:0

Process 5 Burst Time:3

Pno.	AT	BT	CT	TAT	WT
5	0	3	3	3	0
4	1	4	7	6	2
3	2	5	12	10	5
1	4	3	15	11	8
2	5	2	17	12	10

Avg TAT: 8.40

Avg WT: 5.00

PROGRAM: SJF

```
#include <stdio.h>
int main() {
    int n,i;
   printf("Enter the no.of process: ");
   scanf("%d",&n);
   int at[n],bt[n],tat[n],wt[n],ct[n];
   for(i=0;i<n;i++)
   {
      printf("Process %d Arrival Time:",i+1);
      scanf("%d",&at[i]);

      printf("Process %d Burst Time:",i+1);
      scanf("%d",&bt[i]);
      ct[i]=0;
   }

  int current_time=0;
  int  completed=0;
   int total_tat=0,total_wt=0;
   float avg_tat,avg_wt;
   printf("\nProcess\t AT\t BT\t TAT\t WT\n");

   while(completed<n)
   {
      int idx=-1;
      int minbt=9999;


     for(i=0;i<n;i++)
     {
        if(at[i]<=current_time && bt[i]<minbt && ct[i]==0)
        {
           idx=i;
```

ALGORITHM: SJF

1. Start.
2. Read the number of processes into variable n.
3. Declare arrays burst_time[], arrival_time[], waiting_time[], and turnaround_time[] each of size n.
4. Read the burst time and arrival time for each process.
5. Sort the processes based on arrival time using Bubble Sort:
6. Initialize current_time and completed to 0.
7. Run a loop till all process are completed
    a. Set idx=-1 and minbt to any large number.
    b. Run a loop from i=0 to n
    c. Check if arrival_time<=current_time ,burst_time<minbt and ct[i]=0
    d. Set min_burst_time to burst_time[i] and idx=i.
    e. Calculate waiting_time[idx] as current_time - arrival_time[idx].
    f. Calculate turnaround_time[idx] as burst_time[idx] + waiting_time[idx].
    g. Else increment current_time.
8. Calculate total waiting time and total turnaround time.
9. Calculate average waiting time and average turnaround time.
10. Display the process number, burst time, arrival time, waiting time, and turnaround time for each process.
11. Display the average waiting time and average turnaround time.
12. Stop.

```
            minbt=bt[i];
          }
       }
          if(idx!=-1)
          {
              wt[idx]=current_time-at[idx];
             tat[idx]=wt[idx]+bt[idx];

printf("%d\t%d\t%d\t%d\t%d\n",idx+1,at[idx],bt[idx],tat[idx],wt[idx]);

             total_tat+=tat[idx];
             total_wt+=wt[idx];

             current_time+=bt[idx];
             ct[idx]=1;
             completed ++;
          }

          else
          {
             current_time++;
          }
        }
    avg_tat=(float)total_tat/n;
      avg_wt=(float)total_wt/n;
    printf("\nAvg TAT: %.2f\n",avg_tat);
     printf("Avg WT: %.2f",avg_wt);
    return 0;
}
```

OUTPUT

Enter the no.of process: 4

Process 1 Arrival Time:0

Process 1 Burst Time:5

Process 2 Arrival Time:2

Process 2 Burst Time:4

Process 3 Arrival Time:3

Process 3 Burst Time:5

Process 4 Arrival Time:6

Process 4 Burst Time:2

Process	AT	BT	TAT	WT
1	0	5	5	0
2	2	4	7	3
4	6	2	5	3
3	3	5	13	8

Avg TAT: 7.50

Avg WT: 3.50

PROGRAM: ROUND ROBIN

```
#include<stdio.h>
int q[100];
int f=-1,r=-1;
void insert(int n)
{
  if (f==-1)
     f=0;
  r=r+1;
  q[r]=n;
}
int delete()
{
   int n=q[f];
   f=f+1;
   return n;
}
void main()
{
   int idx,i,j,n,t=0;
   printf("Enter the no.of process: ");
   scanf("%d",&n);
   int at[n],bt[n],tat[n],tb[n],wt[n],ct[n],exist[n],tq,p[n];
   for(i=0;i<n;i++)
   {
      p[i]=i+1;
      printf("Process %d AT: ",p[i]);
      scanf("%d",&at[i]);
      printf("Process %d BT: ",p[i]);
      scanf("%d",&bt[i]);
      exist[i]=0;
   }
   printf("Enter the time quantum: ");
   scanf("%d",&tq);
```

ALGORITHM: ROUND ROBIN

1. Start
2. Read the number of processes
3. Declare arrays at[] for Arrival Time, bt[] for Burst Time, tat[] for Turnaround Time, wt[] for Waiting Time, ct[] for Completion Time, tb[] for remaining Burst Time, and exist[] for tracking whether a process has entered the queue.
4. Declare integer variables f and r to represent the front and rear of the queue, initialized to -1.
5. Read burst time and arrival time for each process:
   a. Initialize exist[i] = 0 (mark process as not yet in the queue).
6. Read the time quantum tq.
7. Sort the processes in ascending order of at[] (Arrival Time) using a Bubble Sort.
8. Initialize tb[] for remaining burst time:
   a. Copy the values of bt[] into tb[].
9. While queue is not empty (f <= r):

   a. Dequeue the process idx = delete().

   b. If the remaining burst time tb[idx] >= tq:

   - Reduce tb[idx] -= tq (decrease the burst time by the time quantum).
   - Increment t += tq (advance the current time by the time quantum).

   c. Else:

   - Set t += tb[idx] (process finishes).
   - Set tb[idx] = 0 (process finished, burst time is 0).

   d. Check if there are any processes that have arrived by current_time = t but are not in the queue:

   - If exist[i] == 0 && at[i] <= t, insert process i into the queue and mark it as existing (exist[i] = 1).

   e. If tb[idx] == 0 (process is completed):

   - Set ct[idx] = t (set completion time)

```
int total_tat=0,total_wt=0;
   float avg_tat,avg_wt;

   for(i=0;i<n;i++)
   {
      for(j=i+1;j<n;j++)
      {
         if(at[i]>at[j])
         {
            int temp=at[i];
            at[i]=at[j];
            at[j]=temp;

            temp=bt[i];
            bt[i]=bt[j];
            bt[j]=temp;

             temp=p[i];
            p[i]=p[j];
            p[j]=temp;
         }
      }
   }
   for(i=0;i<n;i++)
   {
      tb[i]=bt[i];
   }

   insert(0);
   exist[0]=1;

   while(f<=r)
   {
      idx=delete();
      if(tb[idx]>=tq)
```

- Calculate Turnaround Time: tat[idx] = ct[idx] - at[idx].
- Calculate Waiting Time: wt[idx] = tat[idx] - bt[idx].
- Add the Turnaround Time to the total turnaround time: total_tat += tat[idx].
- Add the Waiting Time to the total waiting time: total_wt += wt[idx].

- f. Else (process is not finished):
  - Insert the process idx back into the queue for further execution.

10. Calculate the average Turnaround Time: avg_tat = total_tat / n.
11. Calculate the average Waiting Time: avg_wt = total_wt / n.
12. Display the process number, Arrival Time (AT), Burst Time (BT), Turnaround Time (TAT), and Waiting Time (WT) for each process.
13. Print the average Turnaround Time (TAT) and Waiting Time (WT).
14. End.

```
    {
       tb[idx]-=tq;
       t+=tq;
    }
    else
    {
       t+=tb[idx];
       tb[idx]=0;
    }

    for(i=0;i<n;i++)
    {
       if(exist[i]==0 && at[i]<=t)
       {
          insert(i);
          exist[i]=1;
       }
    }
      if(tb[idx]==0)
    {
       ct[idx]=t;
       tat[idx]=ct[idx]-at[idx];
       wt[idx]=tat[idx]-bt[idx];

       total_tat+=tat[idx];
       total_wt+=wt[idx];
    }
    else
    {
       insert(idx);
    }
}
  printf("\nProcess\t AT\t BT\t TAT\t WT\n");
```

```
    for(i=0;i<n;i++)
    {
       printf("%d\t%d\t%d\t%d\t%d\n",p[i],at[i],bt[i],tat[i],wt[i]);
    }
  avg_tat=(float)total_tat/n;
  avg_wt=(float)total_wt/n;
  printf("Avg TAT: %.2f\n",avg_tat);
  printf("Avg WT: %.2f",avg_wt);
}
```

OUTPUT

```
Enter the no.of process: 4
Process 1 AT: 0
Process 1 BT: 3
Process 2 AT: 1
Process 2 BT: 6
Process 3 AT: 5
Process 3 BT: 4
Process 4 AT: 3
Process 4 BT: 2
Enter the time quantum: 3

Process     AT    BT    TAT   WT
1           0     3     3     0
2           1     6     13    7
4           3     2     5     3
3           5     4     10    6
Avg TAT: 7.75
Avg WT: 4.00
```

PROGRAM: PRIORITY

```
#include <stdio.h>

int main() {
   int n,i;
   printf("Enter the no.of process: ");
   scanf("%d",&n);

   int at[n],bt[n],tat[n],wt[n],ct[n],p[i];

   for(i=0;i<n;i++)
   {
      p[i]=i+1;
      printf("Process %d Arrival Time:",p[i]);
      scanf("%d",&at[i]);

      printf("Process %d Burst Time:",p[i]);
      scanf("%d",&bt[i]);

      printf("Process %d Priority:",p[i]);
      scanf("%d",&p[i]);

      ct[i]=0;
   }

  int current_time=0;
  int  completed=0;

   int total_tat,total_wt;
   float avg_tat,avg_wt;
   while(completed<n)
   {
      int idx=-1;
      int hp=9999;
```

ALGORITHM: PRIORITY

1. Start.
2. Read the number of processes into variable n.
3. Declare arrays burst_time[], arrival_time[], waiting_time[], and turnaround_time[] each of size n.
4. Read the burst time ,priority and arrival time for each process.
5. Sort the processes based on arrival time using Bubble Sort:
6. Initialize current_time and completed to 0.
7. Run a loop till all process are completed
    a. Set idx=-1 and hp to any large number.
    b. Run a loop from i=0 to n
    c. Check if arrival_time<=current_time ,priority<hp and ct[i]=0
    d. Set hp to priority[i] and idx=i.
    e. Set current_time=current_time +burst_time[idx]
    f. Set ct[idx]=current_time;
    g. Calculate turnaround_time[idx] as ct[idx]-arrival_time[idx].
    h. Calculate waiting_time[idx] as turnaround_time[idx]-burst_time[idx]
    i. Else increment current_time.
8. Calculate total waiting time and total turnaround time.
9. Calculate average waiting time and average turnaround time.
10. Display the process number, burst time, arrival time, waiting time, and turnaround time for each process.
11. Display the average waiting time and average turnaround time.
12. Stop.

```
    for(i=0;i<n;i++)
    {
       if(at[i]<=current_time && p[i]<hp && ct[i]==0)
       {
          idx=i;
          hp=p[i];
       }
    }
       if(idx!=-1)
       {
          current_time+=bt[idx];
          ct[idx]=current_time;
          tat[idx]=ct[idx]-at[idx];
          wt[idx]=tat[idx]-bt[idx];
          total_tat+=tat[idx];
          total_wt+=wt[idx];
          completed++;
       }
      else
       {
          current_time++;
       }
    }
   avg_tat=(float)total_tat/n;
    avg_wt=(float)total_wt/n;
    printf("\nProcess\t AT\t BT\t TAT\t WT\n");
    for(i=0;i<n;i++)
    {
       printf("%d\t%d\t%d\t%d\t%d\n",p[i],at[i],bt[i],tat[i],wt[i]);
    }
   printf("\nAvg TAT: %.2f\n",avg_tat);
   printf("Avg WT: %.2f",avg_wt);

  return 0;
}
```

OUTPUT

Enter the no.of process: 4

Process 1 Arrival Time:0

Process 1 Burst Time:6

Process 1 Priority:2

Process 2 Arrival Time:5

Process 2 Burst Time:3

Process 2 Priority:5

Process 3 Arrival Time:3

Process 3 Burst Time:6

Process 3 Priority:1

Process 4 Arrival Time:4

Process 4 Burst Time:3

Process 4 Priority:6

Process	AT	BT	TAT	WT
2	0	6	6	0
5	5	3	10	7
1	3	6	9	3
6	4	3	14	11

Avg TAT: 9.75

Avg WT: 5.25

RESULT

Programs to implement various CPU scheduling algorithms (FCFS, SJF, Round Robin, Priority) has been executed successfully and verified the output.

PROGRAM: WRITER

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <string.h>

int main()
{
 key_t key = ftok("shmfile", 65);
 int shmid = shmget(key, 1024, 0666 | IPC_CREAT);
 char *str = (char *)shmat(shmid, (void *)0, 0);
 printf("Write Data : ");
 fgets(str,1024,stdin);
 printf("Data written in memory : %s\n", str);
 shmdt(str);
 return 0;
}
```

OUTPUT

```
Write Data : Hi
Data written in memory : Hi
```

Date: - 19-02-2025

PROGRAM - 6

IPC USING SHARED MEMORY

AIM

To write C programs to implement Inter Process Communication Using Shared Memory. (Readers-Writers Problem)

ALGORITHM: WRITER

1. Start
2. Generate a unique key using function ftok().
3. Create a shared memory segment using shmget and get an identifier for the shared memory segment.
4. Attach to the shared memory segment created using shmat.
5. Write the data to the reader to the shared memory segment created.
6. Detach from the shared memory segment using shmdt().
7. End

PROGRAM: READER

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
int main()
{
 key_t key = ftok("shmfile", 65);
 int shmid = shmget(key, 1024, 0666 | IPC_CREAT);
 char *str = (char *)shmat(shmid, (void *)0, 0);
 printf("Data read from memory: %s\n", str);
 shmdt(str);
 shmctl(shmid, IPC_RMID, NULL);
 return 0;
}
```

OUTPUT

Data read from memory: Hi

ALGORITHM: READER

1. Start
2. Generate a unique key using function ftok().
3. Create a shared memory segment using shmget and get an identifier for the shared memory segment.
4. Attach to the shared memory segment created using shmat.
5. Enter the data to the reader to the shared memory segment created.
6. Detach from the shared memory segment using shmdt().
7. To destroy the shared memory call shmctl().
8. End

RESULT

Programs to implement the Inter Process Communication Using Shared Memory (Readers Writers Problem) has been executed successfully and verified the output.

PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
int x=0,full=0,empty,mutex=1,ch;
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("Producer produced %d\n",x);
   mutex=signal(mutex);
}

void consumer()
{
   mutex=wait(mutex);
   full=wait(full);
   empty=signal(empty);
   printf("Consumer consumed %d\n",x);
     x--;
   mutex=signal(mutex);
}
void main() {
         printf("Enter the size of Buffer:");
```

Date: - 19-02-2025

PROGRAM - 7

PROODUCER – CONSUMER PROBLEM USING SEMAPHORES

AIM

To write a C-program to implement the Producer – Consumer problem using semaphores.

ALGORITHM

1. Start.
2. Declare the required variables.
3. Initialize the buffer size and get maximum item you want to produce.
4. Get the option, which you want to do either producer, consumer or exit from the operation.
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.
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.
7. If you select exit come out of the program.
8. Stop.

```
    scanf("%d",&empty);
  while(1)
  {
    printf("\n1.PRODUCER\n2.CONSUMER\n3.EXIT\n");
    printf("Enter the choice: ");
    scanf("%d",&ch);

    switch(ch)
    {
      case 1:
        if((mutex==1) && (empty!=0))
          {
            producer();
          }
          else
          {
            printf("BUFFER FULL\n");
          }
           break;
      case 2:
        if((mutex==1) && (full!=0))
        {
          consumer();
        }
        else
          {
            printf("BUFFER EMPTY\n");
           }
          break;
       case 3:
        exit(0);
        break;
    }
  }
}
```

OUTPUT

```
Enter the size of Buffer:2

1.PRODUCER
2.CONSUMER
3.EXIT
Enter the choice: 1
Producer produced 1

1.PRODUCER
2.CONSUMER
3.EXIT
Enter the choice: 1
Producer produced 2

1.PRODUCER
2.CONSUMER
3.EXIT
Enter the choice: 1
BUFFER FULL

1.PRODUCER
2.CONSUMER
3.EXIT
Enter the choice: 2
Consumer consumed 2

1.PRODUCER
2.CONSUMER
3.EXIT
Enter the choice: 3
```

RESULT

Program to implement the Producer–Consumer problem using semaphores has been executed successfully and verified the output.

PROGRAM: BEST FIT

```
#include <stdio.h>
int main() {
    int p,m,i,j;
     printf("Enter the no. of blocks:");
    scanf("%d",&m);
     printf("Enter the no. of files:");
    scanf("%d",&p);
    int a[p],b[m];
 printf("Enter blocks:\n");
    for(i=0;i<m;i++)
       {
           printf("Block %d: ",i+1);
          scanf("%d",&b[i]);
       }
        printf("Enter files:\n");
    for(i=0;i<p;i++)
       {
          printf("File %d: ",i+1);
          scanf("%d",&a[i]);
       }
      for(i=0;i<m;i++)
      {
         for(j=i+1;j<m;j++)
         {
            if(b[i]>b[j])
            {
              int temp=b[i];
              b[i]=b[j];
              b[j]=temp;
            }
         }
      }
```

Date: - 05-03-2025

PROGRAM - 8

MEMORY ALLOCATION METHODS FOR FIXED PARTITION

AIM

To write C-programs to implement Memory Allocation Methods for fixed partition.

a) Best Fit b) Worst Fit c) First Fit.

ALGORITHM: BEST FIT

1. Start.
2. Read the number of blocks (m):
3. Read the number of files (p):
4. Declare arrays for files and blocks
5. Read the files and blocks
6. Sort the block sizes in ascending order:
7. Allocate memory using Best Fit:
8. For each file i from 1 to p, do the following:
    a. Set a flag ,flag = 0 (indicating no allocation).
    b. For each block j from 1 to m, do the following:
        i. If b[j] >= a[i] (the block is large enough to accommodate the file):
1. Print the allocation: "File i -> Block j".
2. Set b[j] = 0 (mark the block as allocated).
3. Set flag = 1 (indicating that the file has been allocated).
4. Break the loop (no need to check further blocks).
    c. If no block was found (i.e., flag == 0), print: "File i has to wait...".
9. End.

```
for(i=0;i<p;i++){
        int flag=0;
        for(j=0;j<m;j++) {
           if(b[j]>=a[i])
           {
             printf("File :%d -> Block:%d\n",a[i],b[j]);
             b[j]=0;
             flag=1;
             break;
           } }
        if(!flag){
           printf("File :%d has to wait...\n",a[i]);
        } }
   return 0;
}
```

OUTPUT

```
Enter the no. of blocks:3
Enter the no. of files:3
Enter blocks:
Block 1: 100
Block 2: 200
Block 3: 300
Enter files:
File 1: 210
File 2: 110
File 3: 400
File :210 -> Block:300
File :110 -> Block:200
File :400 has to wait...
```

PROGRAM: WORST FIT

```
#include <stdio.h>
int main() {
   int p,m,i,j;

    printf("Enter the no. of blocks:");
   scanf("%d",&m);
    printf("Enter the no. of files:");
   scanf("%d",&p);

   int a[p],b[m];

 printf("Enter blocks:\n");
   for(i=0;i<m;i++)
      {
          printf("Block %d: ",i+1);
         scanf("%d",&b[i]);
      }

       printf("Enter files:\n");
   for(i=0;i<p;i++)
      {
         printf("File %d: ",i+1);
         scanf("%d",&a[i]);
      }

     for(i=0;i<m;i++)
     {
        for(j=i+1;j<m;j++)
        {
           if(b[i]<b[j])
           {
              int temp=b[i];
              b[i]=b[j];
              b[j]=temp;
           }
        }
     }
```

ALGORITHM: WORST FIT

1. Start.
2. Read the number of blocks (m):
3. Read the number of files (p):
4. Declare arrays for files and blocks
5. Read the files and blocks
6. Sort the block sizes in descending order:
7. Allocate memory using Worst Fit:
8. For each file i from 1 to p, do the following:
    a. Set a flag ,flag = 0 (indicating no allocation).
    b. For each block j from 1 to m, do the following:
        i. If b[j] >= a[i] (the block is large enough to accommodate the file):
            1. Print the allocation: "File i -> Block j".
            2. Set b[j] = 0 (mark the block as allocated).
            3. Set flag = 1 (indicating that the file has been allocated).
            4. Break the loop (no need to check further blocks).
    c. If no block was found (i.e., flag == 0), print: "File i has to wait...".
9. End.

```
   for(i=0;i<p;i++) {
        int flag=0;
        for(j=0;j<m;j++) {
           if(b[j]>=a[i]) {
              printf("File :%d -> Block:%d\n",a[i],b[j]);
              b[j]=0;
              flag=1;
              break;
           }
        }
        if(!flag) {
           printf("File :%d has to wait...\n",a[i]);
        }
     }
   return 0;
}
```

OUTPUT

```
Enter the no. of blocks:3

Enter the no. of files:3

Enter blocks:

Block 1: 100

Block 2: 200

Block 3: 300

Enter files:

File 1: 210

File 2: 110

File 3: 400

File :210 -> Block:300

File :110 -> Block:200

File :400 has to wait...
```

PROGRAM: FIRST FIT

```
#include <stdio.h>
int main() {
    int p,m,i,j;
     printf("Enter the no. of blocks:");
    scanf("%d",&m);
     printf("Enter the no. of files:");
    scanf("%d",&p);
     int a[p],b[m];
 printf("Enter blocks:\n");
    for(i=0;i<m;i++)
       {
           printf("Block %d: ",i+1);
          scanf("%d",&b[i]);
       }
       printf("Enter files:\n");
    for(i=0;i<p;i++)
       {
          printf("File %d: ",i+1);
          scanf("%d",&a[i]);
       }
//First Fit
 for(i=0;i<p;i++)
      {
         int flag=0;
         for(j=0;j<m;j++)
    {
            if(b[j]>=a[i])
            {
              printf("File :%d -> Block:%d\n",a[i],b[j]);
              b[j]=0;
              flag=1;
              break;
            }
```

ALGORITHM: FIRST FIT

1. Start.
2. Read the number of blocks (m):
3. Read the number of files (p):
4. Declare arrays for files and blocks
5. Read the files and blocks
6. Allocate memory using First Fit:
7. For each file i from 1 to p, do the following:
    a. Set a flag ,flag = 0 (indicating no allocation).
    b. For each block j from 1 to m, do the following:
        i. If b[j] >= a[i] (the block is large enough to accommodate the file):

1. Print the allocation: "File i -> Block j".
2. Set b[j] = 0 (mark the block as allocated).
3. Set flag = 1 (indicating that the file has been allocated).
4. Break the loop (no need to check further blocks).
    c. If no block was found (i.e., flag == 0), print: "File i has to wait...".

8. End.

```
}
        if(!flag)
        {
           printf("File :%d has to wait...\n",a[i]);
        }
     }
   return 0;
}
```

OUTPUT

```
Enter the no. of blocks:3
Enter the no. of files:3
Enter blocks:
Block 1: 100
Block 2: 200
Block 3: 300
Enter files:
File 1: 210
File 2: 110
File 3: 400
File :210 -> Block:300
File :110 -> Block:200
File :400 has to wait...
```

RESULT

Programs to implement Memory Allocation Methods for fixed partition (Best Fit, Worst Fit, and First Fit) has been executed successfully and verified the output.

PROGRAM: FIFO

```
#include<stdio.h>
int main()
{
  int num_frames, num_pages, i, j, page_faults = 0, current_frame = 0;
  printf("Enter the number of frames: ");
  scanf("%d", &num_frames);
  printf("Enter the number of pages: ");
  scanf("%d", &num_pages);
  int frames[num_frames], pages[num_pages], frame_in_use[num_frames];

 printf("Enter the page reference string: ");
  for (i = 0; i < num_pages; i++)
    {
     scanf("%d", &pages[i]);
    }
  for (i = 0; i < num_frames; i++)
  {
    frames[i] = -1;
    frame_in_use[i] = 0;
   }
  printf("\nPage Reference String: ");
  for (i = 0; i < num_pages; i++)
      {
       printf("%d ", pages[i]);
      }
    printf("\n");

printf("\nFIFO Page Replacement Algorithm:\n");
for (i = 0; i < num_pages; i++)
  {
   int page_found = 0;
   for (j = 0; j < num_frames; j++)
    {
```

Date: - 12-03-2025

PROGRAM - 9

PAGE REPLACEMENT ALGORITHMS

AIM

To write C-programs to implement page replacement algorithms:

a) FIFO b) LRU c) LFU

ALGORITHM: FIFO

1. Start.
2. Declare variables to store the number of frames, number of pages, number of page faults, frame contents, page references, and frame usage status.
3. Read the number of frames.
4. Read the number of pages.
5. Read the page reference string.
6. Initialize all frames in frames to -1 (indicating empty) and all elements in frame_in_use to 0.
7. Display the page reference string.
8. Perform FIFO page replacement algorithm:
    - Check if the page is already present in any frame:
    - If yes, mark page_found as 1 and break the loop.
    - If the page is not found in any frame:
    - Increment page_faults.
9. Replace the page at current_frame with the current page.
10. Update current_frame to point to the next frame in a circular manner.
11. Display the contents of frames after the page replacement.
12. Display the total number of page faults.
13. End

```
      if (frames[j] == pages[i])
         {
          page_found = 1;
           break;
          }
     }
    if (!page_found)
       {
        printf("\nPage Fault: Page %d\n", pages[i]);
        page_faults++;
        frames[current_frame] = pages[i];
        frame_in_use[current_frame] = 1;
        current_frame = (current_frame + 1) % num_frames;
       }
    else
    {
      printf("\nPage Hit: Page %d\n",pages[i]);
    }
printf("Frames: ");
  for (j = 0; j < num_frames; j++)
   {
   if(frames[j]!=-1)
     {
     printf("%d ", frames[j]);
    }
    else
    {
       printf("- ");
    }
   }
   printf("\n");
  }
  printf("\nTotal Page Faults: %d\n", page_faults);
return 0;
}
```

OUTPUT

Enter the number of frames: 3
Enter the number of pages: 8
Enter the page reference string: 2 3 0 3 4 5 1 4

Page Reference String: 2 3 0 3 4 5 1 4

FIFO Page Replacement Algorithm:

Page Fault: Page 2
Frames: 2 - -

Page Fault: Page 3
Frames: 2 3 -

Page Fault: Page 0
Frames: 2 3 0

Page Hit: Page 3
Frames: 2 3 0

Page Fault: Page 4
Frames: 4 3 0

Page Fault: Page 5
Frames: 4 5 0

Page Fault: Page 1
Frames: 4 5 1

Page Hit: Page 4
Frames: 4 5 1

Total Page Faults: 6

PROGRAM: LRU

```
#include <stdio.h>
int main()
{
   int num_frames, num_pages, i, j, page_faults = 0, current_frame =
0,least_recently_used;
 printf("Enter the number of frames: ");
     scanf("%d", &num_frames);
     printf("Enter the number of pages: ");
    scanf("%d", &num_pages);
   int frames[num_frames], pages[num_pages], last_used[num_frames];
    printf("Enter the page reference string: ");
      for (i = 0; i < num_pages; i++)      {
         scanf("%d", &pages[i]);
        }
      for (i = 0; i < num_frames; i++)      {
         frames[i] = -1;
         last_used[i] = 0;
        }
  printf("\nPage Reference String: ");
   for (i = 0; i < num_pages; i++)
     {
       printf("%d ", pages[i]);
      }
    printf("\n");
 printf("\nLRU Page Replacement Algorithm:\n");
    for (i = 0; i < num_pages; i++)
     {
       int page_found = 0;
        for (j = 0; j < num_frames; j++)
          {
           if (frames[j] == pages[i])
            {
              page_found = 1;
              last_used[j] = i + 1;
              break;
            } }
```

ALGORITHM: LRU

1. Start.
2. Declare variables: num_frames (integer) to store the number of frames, num_pages (integer) to store the number of pages, page_faults (integer) to count the number of page faults, current_frame (integer) to keep track of the current frame, least_recently_used (integer) to store the index of the least recently used frame.
3. Read the value of number of frames from the user.
4. Read the value of number of pages from the user.
5. Declare arrays frames, pages, and last_used with sizes according to num_frames.
6. Read the page reference string (pages) from the user.
7. Initialize all frames in frames to -1 (indicating empty) and all elements in last_used to 0.
8. Display the page reference string.
9. Perform the LRU page replacement algorithm:
    - Check if the page is already present in any frame:
    - If yes, mark page_found as 1 and update the last used time for the corresponding frame.
    - If the page is not found in any frame:
    - Increment page_faults.
    - Display "Page Fault: Page [page number]".
10. Find the least recently used frame:
    - Iterate through frames and find the frame with the smallest last_used value.
    - Replace the least recently used page in the frame with the current page.
    - Update the last used time for the least recently used frame.
11. Display the contents of frames after the page replacement.
12. Display the total number of page faults.
13. End

```
      if (!page_found)
       {
         printf("\nPage Fault: Page %d\n", pages[i]);
         page_faults++;
         least_recently_used = 0;
        for (j = 1; j < num_frames; j++)
         {
          if (last_used[j] < last_used[least_recently_used])
           {
             least_recently_used = j;
            }
          }
        frames[least_recently_used] = pages[i];
       last_used[least_recently_used] = i + 1;
      }
   else
      {
        printf("\nPage Hit: Page %d\n", pages[i]);
       }
printf("Frames: ");
  for (j = 0; j < num_frames; j++)
   {
   if(frames[j]!=-1)
     {
     printf("%d ", frames[j]);
    }
    else
    {
       printf("- ");
    }
  }
   printf("\n");
  }
  printf("\nTotal Page Faults: %d\n", page_faults);
return 0;
}
```

OUTPUT

Enter the number of frames: 3
Enter the number of pages: 8
Enter the page reference string: 2 3 0 3 4 5 1 4

Page Reference String: 2 3 0 3 4 5 1 4

LRU Page Replacement Algorithm:

Page Fault: Page 2
Frames: 2 - -

Page Fault: Page 3
Frames: 2 3 -

Page Fault: Page 0
Frames: 2 3 0

Page Hit: Page 3
Frames: 2 3 0

Page Fault: Page 4
Frames: 4 3 0

Page Fault: Page 5
Frames: 4 3 5

Page Fault: Page 1
Frames: 4 1 5

Page Hit: Page 4
Frames: 4 1 5

Total Page Faults: 6

PROGRAM: LFU

```
#include <stdio.h>
int main()
 {
   int page_faults = 0, page_count, n, i, j, flag, pos, min_freq,
replace_index,counter = 0;

  printf("Enter the number of frames: ");
    scanf("%d", &n);

    printf("Enter the number of pages: ");
    scanf("%d", &page_count);

 int frames[n], pages[page_count], freq[n],time[n];
    printf("Enter the page reference string:\n");
    for(i = 0; i < page_count; i++) {
        scanf("%d", &pages[i]);
    }
    for(i = 0; i < n; i++) {
        frames[i] = -1;
        freq[i] = 0;
        time[i] = 0;
    }
    for(i = 0; i < page_count; i++) {
        flag = 0;
        counter++;
        for(j = 0; j < n; j++) {
            if(frames[j] == pages[i]) {
                flag = 1;
                freq[j]++;
                break;
            }
        }
        if(flag == 0) {
            page_faults++;
            pos = -1;
```

ALGORITHM: LFU

1. Start.
2. Declare variables page count, n, page_faults, min_freq,pos, replace_index
3. Read the value of number of frames from the user.
4. Read the value of number of pages from the user.
5. Declare arrays frames, pages, time and freq with sizes according to n.
6. Read the page reference string (pages) from the user.
7. Initialize all frames in frames to -1 (indicating empty) and all elements in freq and time to 0.
8. Display the page reference string.
9. Perform the LFU page replacement algorithm:
    - Check if the page is already present in any frame:
    - If yes, mark flag as 1 and update the freq for the corresponding frame.
    - If the flag=0:
    - Increment page_faults.
    - Set pos=-1,min_freq=freq[0],replace_index=0
10. If a frame is empty (frames[j] == -1), the pos is set to j, and it breaks out of the loop.
11. If the frequency of the current frame is less than min_freq, it updates min_freq and sets replace_index to the current index j.
12. If the frequency is equal to min_freq, it checks which page was accessed earlier (time[j] < time[replace_index]), and replaces the least recently used page in case of a tie.
13. If no empty frame was found (pos == -1), it uses the replace_index to replace the page with the minimum frequency or the least recently used in case of ties.
14. Update freq[pos] to 1 and time[pos] of frame to counter.
15. Display the contents of frames after the page replacement.
16. Display the total number of page faults.
17. End

```
min_freq = freq[0];
            replace_index = 0;

            for(j = 0; j < n; j++) {
               if(frames[j] == -1) {
                  pos = j;
                  break;
               } else if(freq[j] < min_freq) {
                  min_freq = freq[j];
                  replace_index = j;
               } else if(freq[j] == min_freq) {
                  if(time[j] < time[replace_index]) {
                     replace_index = j;
                  }
               }
            }

            if(pos == -1) {
               pos = replace_index;
            }
            frames[pos] = pages[i];
            freq[pos] = 1;
            time[pos] = counter;
         }
         printf("\nPage Reference: %d\nFrames: ", pages[i]);
         for(j = 0; j < n; j++) {
            if(frames[j] != -1)
               printf("%d ", frames[j]);
            else
               printf("- ");
         }
         printf("\n");
      }
      printf("\nTotal Page Faults: %d", page_faults);

      return 0;
   }
```

OUTPUT

```
Enter the number of frames: 3
Enter the number of pages: 8
Enter the page reference string:
2 3 0 3 4 5 1 4

Page Reference: 2
Frames: 2 - -

Page Reference: 3
Frames: 2 3 -

Page Reference: 0
Frames: 2 3 0

Page Reference: 3
Frames: 2 3 0

Page Reference: 4
Frames: 4 3 0

Page Reference: 5
Frames: 4 3 5

Page Reference: 1
Frames: 1 3 5

Page Reference: 4
Frames: 1 3 4

Total Page Faults: 7
```

RESULT

Programs to implement page replacement algorithms (FIFO, LRU and LFU) has been executed successfully and verified the output.

PROGRAM

```
#include <stdio.h>
int main() {
  int i,j,k,n,m;
     printf("Enter the process: ");
  scanf("%d",&n);
  printf("Enter the resource: ");
  scanf("%d",&m);

  int available[m],allocation[n][m],max[n][m],need[n][m];
  printf("Available matrix: ");
  for(i=0;i<m;i++)
  {
     scanf("%d",&available[i]);
  }
  printf("Allocation matrix\n");
  for(i=0;i<n;i++)
  {
      printf("Process %d: ",i);
       for(j=0;j<m;j++)
      {
        scanf("%d",&allocation[i][j]);
      }
 }
 printf("Max matrix\n");
  for(i=0;i<n;i++)
  {
      printf("Process %d: ",i);
       for(j=0;j<m;j++)
      {
        scanf("%d",&max[i][j]);
      }
```

Date: - 12-03-2025

PROGRAM - 10

BANKER'S ALGORITHM FOR DEADLOCK AVOIDANCE

AIM

To write a C-program to implement banker's algorithm for deadlock avoidance.

ALGORITHM

1. Start.
2. Declare variables.
3. Read the number of processes (n).
4. Read the number of resources (m).
5. Read the available resources.
6. Read the allocation matrix for each process.
7. Read the maximum matrix for each process.
8. Calculate Need Matrix:
9. Subtract allocation matrix from the maximum matrix to get the need matrix.
10. Initialize the finish array to track the finish status of each process (0 for unfinished, 1 for finished).
11. Initialize the work array to represent the available resources.
12. Initialize the safe sequence array and count to keep track of the safe sequence.
13. While count is less than the total number of processes:
    - Initialize a variable 'found' to check if a safe process is found in the current iteration.
14. Loop through each process:
    - If the process is not finished:

    a. Check if the resources needed by the process can be satisfied by available resources.

```
}
 //Need matrix
  for(i=0;i<n;i++)
  {
    for(j=0;j<m;j++)
      {
        need[i][j]=max[i][j]-allocation[i][j];
       }
 }
 int finish[n],work[m];
 for(i=0;i<n;i++)
   {
      finish[i]=0;
   }
   for(i=0;i<m;i++)
   {
      work[i]=available[i];
   }
   int Safeseq[n],count=0;
   while(count<n)
   {
      int found=0;
      for(i=0;i<n;i++)
      {
         if(finish[i]==0){
         int safe=1;
         for(j=0;j<m;j++)
         {
            if(need[i][j]>work[j])
           {
              safe=0;
              break;
           }
        }
        if(safe)
        {
          for(k=0;k<m;k++)
          {
             work[k]+=allocation[i][k];
          }
```

b. If resources are sufficient, mark the process as finished, update available resources, and add the process to the safe sequence.
c. Set the 'found' flag to 1 if a safe process is found in this iteration.

15. If no safe process is found in the current iteration, break the loop and print "System is in unsafe state".
16. If the system is in a safe state, print "System is in safe state".
17. Display the safe sequence of processes.
18. Stop

```
            Safeseq[count++]=i;
            finish[i]=1;
            found=1;
          }
        }}
         if(!found)
    {
       printf("Unsafe state");
       return -1;
    }
    }

    printf("Safe Sequence :");
    for(i=0;i<n;i++)
       printf("%d->",Safeseq[i]);
    return 0;
}
```

OUTPUT

Enter the process: 2

Enter the resource: 3

Available matrix: 3 2 2

Allocation matrix

Process 0: 0 1 0

Process 1: 7 5 3

Max matrix

Process 0: 2 0 0

Process 1: 3 2 2

Safe Sequence :0->1->

RESULT

Program to implement the Banker's algorithm for deadlock avoidance has been successfully executed and verified the output.

PROGRAM

```
#include <stdio.h>
int main( ) {
   int n, m, i, j;
   printf("Enter number of processes: ");
   scanf("%d", &n);
   printf("Enter number of resources: ");
   scanf("%d", &m);
   int allocation[n][m], request[n][m], available[m];
   printf("Enter allocation matrix:\n");
   for (i = 0; i < n; i++) {
      printf("For process %d: ", i);
      for (j = 0; j < m; j++)
         scanf("%d", &allocation[i][j]);
   }
   printf("Enter request matrix:\n");
   for (i = 0; i < n; i++) {
      printf("For process %d: ", i);
      for (j = 0; j < m; j++)
         scanf("%d", &request[i][j]);
   }
   printf("Enter available resources: ");
   for (i = 0; i < m; i++)
      scanf("%d", &available[i]);
   int finish[n];
   for (i = 0; i < n; i++)
      finish[i] = 0;
  int deadlock = 0;
  int executed = 1;
  while (executed) {
      executed = 0;
      for (i = 0; i < n; i++) {
         if (!finish[i]) {
            int canExecute = 1;
      for (j = 0; j < m; j++) {
           if (request[i][j] > available[j]) {
              canExecute = 0;
              break;
} } 
```

Date: - 26-03-2025

PROGRAM - 11

DEADLOCK DETECTION ALGORITHM

AIM

To write a C-program to implement Deadlock detection algorithm.

ALGORITHM

1. Start.
2. Declare variables.
3. Read the number of processes (n).
4. Read the number of resources (m).
5. Read the allocation matrix for each process.
6. Read the request matrix for each process.
7. Read the available resources.
8. Initialize all elements of finish array to 0.
9. Initialize deadlock to 0 and executed to 1.
10. While (executed):
    - Set executed = 0 (no process executed in this round).
    - For each process i:
        - If finish[i] == 0 (process i has not finished yet):
        - Check if request[i][j] <= available[j] for all resources j:
        - If this is true for all j, then allocate resources:
            available[j] += allocation[i][j]
        - Set finish[i] = 1 (mark process i as finished).
        - Set executed = 1 (since a process has been executed).
11. For each process i:
    - If finish[i] == 0, set deadlock = 1 (indicating that deadlock has been detected).
12. If deadlock == 1,
    print "Deadlock detected."
13. Else, print "No deadlock detected."
14. End

```
            if (canExecute) {
                for (j = 0; j < m; j++)
                    available[j] += allocation[i][j];
                finish[i] = 1;
                executed = 1;
            }   }   }   }
    for (i = 0; i < n; i++) {
        if (!finish[i]) {
            deadlock = 1;
            break;
        }    }
 if (deadlock)
     printf("Deadlock detected\n");
    else
        printf("No deadlock detected\n");
    return 0;
}
```

OUTPUT

Enter number of processes: 3

Enter number of resources: 3

Enter allocation matrix:

For process 0: 0 0 1

For process 1: 0 1 0

For process 2: 3 0 1

Enter request matrix:

For process 0: 3 4 7

For process 1: 2 2 1

For process 2: 8 0 0

Enter available resources: 4 4 3

Deadlock detected

RESULT

Program to implement Deadlock detection algorithm has been successfully executed and verified the output.

PROGRAM: FCFS

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

int main() {
   int n, i, head, seek = 0;
   printf("Enter the number of disk requests: ");
   scanf("%d", &n);
   int request[n];
   printf("Enter the disk request queue: ");
   for (i = 0; i < n; i++)
      scanf("%d", &request[i]);
   printf("Enter the initial head position: ");
   scanf("%d", &head);
   printf("\nSequence of disk access:\n%d", head);
   for (i = 0; i < n; i++) {
      seek += abs(request[i] - head);
      head = request[i];
      printf(" -> %d", head);
   }
   printf("\nTotal Seek Time: %d\n", seek);
   return 0;
}
```

OUTPUT

```
Enter the number of disk requests: 8
Enter the disk request queue: 98 183 37 122 14 124 65 67
Enter the initial head position: 53

Sequence of disk access:
53 -> 98 -> 183 -> 37 -> 122 -> 14 -> 124 -> 65 -> 67
Total Seek Time: 640
```

Date: - 26-03-2025

PROGRAM - 12

DISK SCHEDULING ALGORITHMS

AIM

To write C-programs to simulate disk scheduling algorithms:

a) FCFS b) SCAN c) C-SCAN.

ALGORITHM: FCFS

1. Start.
2. Declare variables
3. Read number of disk requests n.
4. Read the disk request queue request[n].
5. Read the initial head position head.
6. For i = 0 to n-1:
    a. Calculate seek time: seek += abs(request[i] - head)
    b. Move the disk head: head = request[i]
    c. Print the current head position
7. Print Sequence of disk access
8. Print Seek time
9. End.

PROGRAM: SCAN

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

int main() {
   int n, i, j, head, size, direction, index;

   printf("Enter the number of disk requests: ");
   scanf("%d", &n);

   int request[n + 2];

   printf("Enter the disk request queue: ");
   for (i = 0; i < n; i++)
      scanf("%d", &request[i]);

   printf("Enter the initial head position: ");
   scanf("%d", &head);

   printf("Enter the total disk size: ");
   scanf("%d", &size);

   printf("Enter the direction (1 for right, 0 for left): ");
   scanf("%d", &direction);

   request[n] = 0;
   request[n + 1] = size - 1;
   n += 2;

   for (i = 0; i < n - 1; i++) {
      for (j = 0; j < n - i - 1; j++) {
         if (request[j] > request[j + 1]) {
            int temp = request[j];
            request[j] = request[j + 1];
            request[j + 1] = temp;
         }
      }
```

ALGORITHM: SCAN

1. Start.
2. Declare variables
3. Read number of disk requests n.
4. Read the disk request queue request[n].
5. Read the initial head position head and direction of movement.
6. Add boundaries in the request array.
7. Sort the request[] array in ascending order to ensure that the disk arm will process the requests in the correct sequence
8. Determine the Index of the Initial Head Position
9. Perform Disk Scheduling Based on Direction**:**
    a. If direction == 1 (right):
        i. Calculate seek time: seek += abs(request[i] - head)
        ii. Move to the right (process requests from the current head position to the end of the disk).
        iii. After reaching the end, reverse direction and process remaining requests to the left.
    b. If direction == 0 (left):
        i. Calculate seek time: seek += abs(request[i] - head)
        ii. Move to the left (process requests from the current head position to the start of the disk)
        iii. After reaching the beginning, reverse direction and process remaining requests to the right.
10. Print Sequence of disk access
11. Print Seek time
12. End.

```
    }
    for (i = 0; i < n; i++) {
        if (request[i] >= head) {
            index = i;
            break;
        }
    }
    int seek = 0;
    printf("\nSequence of disk access:\n");

    if (direction == 1) {
        for (i = index; i < n; i++) {
            seek += abs(request[i] - head);
            head = request[i];
            printf(" -> %d", head);
        }
        for (i = index - 1; i >= 0; i--) {
            seek += abs(request[i] - head);
            head = request[i];
            printf(" -> %d", head);
        }
    } else {
        for (i = index - 1; i >= 0; i--) {
            seek += abs(request[i] - head);
            head = request[i];
            printf(" -> %d", head);
        }
        for (i = index; i < n - 1; i++) {
            seek += abs(request[i] - head);
            head = request[i];
            printf(" -> %d", head);
        }
    }
    printf("\nTotal Seek Time: %d\n", seek);
    return 0;
}
```

OUTPUT

Enter the number of disk requests: 8

Enter the disk request queue: 98 183 37 122 14 124 65 67

Enter the initial head position: 53

Enter the total disk size: 199

Enter the direction (1 for right, 0 for left): 0

Sequence of disk access:

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

Total Seek Time: 236

PROGRAM: C-SCAN

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int n, i, j, head, size, direction;
    printf("Enter the number of disk requests: ");
    scanf("%d", &n);
    int request[n];
    printf("Enter the disk request queue: ");
    for (i = 0; i < n; i++)
        scanf("%d", &request[i]);
    printf("Enter the initial head position: ");
    scanf("%d", &head);
    printf("Enter the total disk size: ");
    scanf("%d", &size);
    printf("Enter the direction (1 for right, 0 for left): ");
    scanf("%d", &direction);
    request[n] = 0;
    request[n + 1] = size - 1;
    n += 2;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (request[j] > request[j + 1]) {
                int temp = request[j];
                request[j] = request[j + 1];
                request[j + 1] = temp;
            }
        }
    }
    int index;
    for (i = 0; i < n; i++) {
        if (request[i] >= head) {
            index = i;
            break;
```

ALGORITHM: C-SCAN

1. Start.
2. Declare variables
3. Read number of disk requests n.
4. Read the disk request queue request[n].
5. Read the initial head position head and direction of movement.
6. Add boundaries in the request array.
7. Sort the request[] array in ascending order to ensure that the disk arm will process the requests in the correct sequence
8. Determine the Index of the Initial Head Position
9. Perform Disk Scheduling Based on Direction**:**
    a. If direction == 1 (right):
        i. Calculate seek time: seek += abs(request[i] - head)
        ii. Service all requests from the current head position to the right end
        iii. After reaching the end, reverse direction and process remaining requests to the left.
    b. If direction == 0 (left):
        i. Calculate seek time: seek += abs(request[i] - head)
        ii. Service all requests from the current head position to the left end.
        iii. After reaching the beginning, reverse direction and process remaining requests to the right.
10. Print Sequence of disk access
11. Print Seek time
12. End.

```
    }
    }
    int seek = size-1;
    printf("\nSequence of disk access:\n%d", head);

    if (direction == 1) {
        for (i = index; i < n; i++) {
            seek += abs(request[i] - head);
            head = request[i];
            printf(" -> %d", head);
        }
        head = 0;

        for (i = 0; i < index; i++) {
            seek += abs(request[i] - head);
            head = request[i];
            printf(" -> %d", head);
        }
    } else {
        for (i = index - 1; i >= 0; i--) {
            seek += abs(request[i] - head);
            head = request[i];
            printf(" -> %d", head);
        }
        head = size - 1;
        for (i = n - 1; i >= index; i--) {
            seek += abs(request[i] - head);
            head = request[i];
            printf(" -> %d", head);
        }
    }
    printf("\nTotal Seek Time: %d\n", seek);

    return 0;
}
```

OUTPUT

Enter the number of disk requests: 8

Enter the disk request queue: 98 183 37 122 14 124 65 67

Enter the initial head position: 53

Enter the total disk size: 199

Enter the direction (1 for right, 0 for left): 0

Sequence of disk access:

53 -> 37 -> 14 -> 0 -> 198 -> 183 -> 124 -> 122 -> 98 -> 67 -> 65

Total Seek Time: 384

RESULT

Programs to simulate Disk scheduling algorithms (FCFS, SCAN and C-SCAN) has been successfully executed and verified the output.