EXPERIMENT NO. 1 **DATE:**

FAMILIARIZATION OF BASIC LINUX COMMANDS

AIM

To familiarize the Basic Linux Commands.

COMMANDS:

1. ***Date Command***: This command is used to display the current data and time.

   Syntax: $date

   $date +%ch

   Options:

   a = Abbreviated weekday.

   A = Full weekday.

   b = Abbreviated month.

   B = Full month.

   c = Current day and time.

   C = Display the century as a decimal number.

   d = Day of the month.

   D = Day in “mm/dd/yy” format

   m = Month of the year.

   M = Minute.

   P = Display AM or PM

   S = Seconds

   T = HH:MM:SS format

   u = Week of the year.

2. ***Calendar Command***: This command is used to display the calendar of the year or the particular month of calendar year.

   Syntax:

   $cal <year>

   $cal <month> <year>

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

3. ***Echo Command***: This command is used to print the arguments on the screen.

   Syntax: $echo <text>

4. ***who Command***: It is used to display who are the users connected to our computer currently.

   Syntax: $who – options

   Options: -

   H–Display the output with headers.

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

5. ***CLEAR Command***: It is used to clear the screen.

   Syntax: $clear

6. ***LIST Command***: It is used to list all the contents in the current working directory.

   Syntax: $ ls – options <arguments>

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

   Options:

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

   c– list all the files column-wise.

   d- list all the directories.

m- list the files separated by commas.

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

r- list the files in reverse alphabetical order.

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

x-list in column wise sorted order.

DIRECTORY RELATED COMMANDS:

7. ***Present Working Directory Command***: To print the complete path of the current working directory.

   Syntax: $pwd

8. ***MKDIR Command***: To create or make a new directory in a current directory.

   Syntax: $mkdir <directory name>

9. ***CD Command***: To change or move the directory to the mentioned directory.

   Syntax: $cd <directory name.

10. ***RMDIR Command***: To remove a directory in the current directory & not the current directory itself.

    Syntax: $rmdir <directory name>

FILE RELATED COMMANDS:

11. ***CREATE A FILE***: To create a new file in the current directory we use CAT command.

    Syntax: $cat > <filename.

    The > symbol is redirectory we use cat command.

12. ***DISPLAY A FILE***: To display the content of file mentioned we use CAT command without ">" operator.

    Syntax: $cat <filename.

    Options –s = to neglect the warning /error message.

13. ***COPYING CONTENTS***: To copy the content of one file with another. If file does not exist, a new file is created and if the file exists with some data then it is overwritten.

    Syntax: $ cat <filename source> >> <destination filename>

    $ cat <source filename> >> <destination filename> it avoids overwriting.

    Options:

        -n content of file with numbers included with blank lines.

    Syntax:

        $cat –n <filename>

14. ***SORTING A FILE***: To sort the contents in alphabetical order in reverse order.

    Syntax:

    $sort <filename >

    Option: $ sort –r <filename>

15. ***COPYING CONTENTS FROM ONE FILE TO ANOTHER***: To copy the contents from source to destination file, so that both contents are same.

    Syntax:

    $cp <source filename> <destination filename>

    $cp <source filename path > <destination filename path>

16. ***MOVE Command***: To completely move the contents from source file to destination file and to remove the source file.

    Syntax:

    $ mv <source filename> <destination filename>

17. ***REMOVE Command***: To permanently remove the file we use this command.

    Syntax:

    $rm <filename>

18. ***WORD Command***: To list the content count of no of lines, words, characters.

    Syntax:

    $wc<filename>

    Options:

        -c – to display no of characters.

        -l – to display only the lines.

        -w – to display the no of words.

FILTERS AND PIPES

19. ***HEAD***: It is used to display the top ten lines of file.

    Syntax: $head<filename>

20. ***TAIL***: This command is used to display the last ten lines of file.

    Syntax: $tail<filename>

21. ***PIPE***: It is a mechanism by which the output of one command can be channelled into the input of another command.

    Syntax: $who | wc-l

22. ***TR***: The tr filter is used to translate one set of characters from the standard inputs to another.

    Syntax: $tr "[a-z]" "[A-Z]"

RESULT:

Familiarized the Basic Unix commands successfully.

EXPERIMENT NO. 2 **DATE:**

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()**

   The fork() system call is used to create processes. It returns a process id. After calling fork() system call, it becomes a child process of the caller. After creation of child call, the instruction followed by fork() will be executed after a new child process is created. Fork() returns a negative value, if the child process creation is unsuccessful, zero if the new child process is created, returns a positive value which is the 'process id', to the parent.

2. **exec()**

   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. Using exec system call will replace the old file or program from the process with a new file or program. The entire content of the process is replaced with a new program. The user data segment which executes the exec() system call is replaced with the data file whose name is provided in the argument while calling exec().

3. **getpid()**

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

4. **exit()**

   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.

5. **wait()**

   As in the case of a fork, child processes are created and get executed but the parent process is suspended until the child process executes. In this case, a wait() system call is activated automatically due to the suspension of the parent process. After the child process ends the execution, the parent process gains control again.

6. **close()**

   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.

7. **stat()**

   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.

8. **opendir()**

   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.

9. **readdir()**

   The readdir() function returns a pointer to a dirent structure describing the next directory entry in the directory stream associated with dirp. A call to readdir() overwrites data produced by a previous call to readdir() on the same directory stream. Calls for different directory streams do not overwrite the data of each other. If the call to readdir() actually reads the directory, the access time of the directory is updated.

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 :

```
opendir.c
..
masm
a.out
```

.**EXPERIMENT NO. 3** **DATE:**

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 hold 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:
    1. Use readdir(dir) to read the next directory entry.
    2. Check if the returned entry pointer is not NULL.
    3. If not NULL, print the name of the directory entry.
    4. 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 = 4080

Child process: PID = 4081

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 28
-rwxrwxr-x 1 user user 8384 Mar 21 11:18 a.out
-rw-rw-r-- 1 user user 202 Mar 21 11:18 exec.c
-rw-rw-r-- 1 user user 380 Mar 21 11:17 getpid.c
drwxr-xr-x 2 user user 4096 Dec 17 2021 masm
-rw-rw-r-- 1 user user 375 Mar 21 11:15 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");
```

ALGORITHM: FORK, WAIT

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:
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.

```
        int status;
        wait(&status);
        printf("Parent process: Child exited with status %d\n", status);
    }
    return 0;
}
```

OUTPUT:

```
Parent process: PID = 4197
Parent process: Waiting for child to exit...
Child process: PID = 4198
Child process: Parent PID = 4197
Child process: Exiting
Parent process: Child exited with status 0
```

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 output verified.

PROGRAM:

```
echo "Enter a number:"
read num
if (( $num % 2 == 0 )); then
   echo "$num is even."
else
   echo "$num is odd."
Fi
```

OUTPUT:

```
Enter a number:
10
10 is even.
Enter a number:
9
9 is odd.
```

EXPERIMENT NO. 4 **DATE:**

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:

```
echo "Enter the first number:"
read num1
echo "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:
15
Enter the second number:
27
27 is the largest.
Enter the first number:
34
Enter the second number:
18
34 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:

```
echo "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:

```
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:
2024
2024 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 outputs verified.

PROGRAM: 5.1

```
#include<stdio.h>
int main() {
int num_processes, i;
printf("Enter the number of processes: ");
scanf("%d", &num_processes);
if (num_processes <= 0) {
printf("Invalid number of processes.\n");
return 1;
}
int arrival_time[num_processes], burst_time[num_processes];
int waiting_time[num_processes], turnaround_time[num_processes];
float avg_waiting_time = 0, avg_turnaround_time = 0;
printf("Enter arrival time and burst time for each process:\n");
for (i = 0; i < num_processes; i++) {
printf("Process %d: ", i + 1);
scanf("%d %d", &arrival_time[i], &burst_time[i]);
}
waiting_time[0] = 0;
for (i = 1; i < num_processes; i++) {
waiting_time[i] = waiting_time[i-1] + burst_time[i-1];
}
for (i = 0; i < num_processes; i++) {
turnaround_time[i] = waiting_time[i] + burst_time[i];
```

EXPERIMENT NO. 5 **DATE:**

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. Check if num_processes is less than or equal to 0. If true, print "Invalid number of processes." and terminate the program with an error code 1.
5. Read arrival time and burst time for each process.
6. Set waiting_time[0] to 0.
7. Calculate waiting_time[i] as waiting_time[i-1] + burst_time[i-1].
8. Calculate turnaround_time[i] as waiting_time[i] + burst_time[i].
9. Add waiting_time[i] to avg_waiting_time.
10. Add turnaround_time[i] to avg_turnaround_time.
11. Divide avg_waiting_time and avg_turnaround_time by num_processes.
12. Display a table showing the process number, arrival time, burst time, waiting time, and turnaround time for each process.
13. Display the average waiting time and average turnaround time.
14. End.

```
}
for (i = 0; i < num_processes; i++) {
avg_waiting_time += waiting_time[i];
avg_turnaround_time += turnaround_time[i];
}
avg_waiting_time /= num_processes;
avg_turnaround_time /= num_processes;
printf("\nProcess\t Arrival Time\t Burst Time\t Waiting Time\t Turnaround Time\n");
for (i = 0; i < num_processes; i++) {
printf("%d\t\t %d\t\t %d\t\t %d\t\t %d\n", i + 1, arrival_time[i], burst_time[i],
waiting_time[i], turnaround_time[i]);
}
printf("\nAverage Waiting Time: %.2f\n", avg_waiting_time);
printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);
return 0;
}
```

OUTPUT :

Enter the number of processes: 3

Enter arrival time and burst time for each process:

Process 1: 0 5

Process 2: 2 3

Process 3: 4 6

Process Arrival Time Burst Time Waiting Time Turnaround Time

1 0 5 0 5

2 2 3 5 8

3 4 6 8 14

Average Waiting Time: 4.33

Average Turnaround Time: 9.00

PROGRAM: 5.2

```
#include <stdio.h>
int main()
{
    int n, i, j, temp, total_waiting_time = 0, total_turnaround_time = 0;
    float avg_waiting_time, avg_turnaround_time;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int burst_time[n], arrival_time[n], waiting_time[n], turnaround_time[n];
    printf("Enter the burst time and arrival time for each process:\n");
    for (i = 0; i < n; i++) {
        printf("Burst time for process %d: ", i + 1);
        scanf("%d", &burst_time[i]);
        printf("Arrival time for process %d: ", i + 1);
        scanf("%d", &arrival_time[i]);
    }
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (arrival_time[j] > arrival_time[j + 1]) {
                temp = arrival_time[j];
                arrival_time[j] = arrival_time[j + 1];
                arrival_time[j + 1] = temp;
                temp = burst_time[j];
                burst_time[j] = burst_time[j + 1];
```

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. Calculate waiting time for each process:
7. Initialize waiting_time[0] to 0.
8. Add burst_time[j] to min_burst_time.
9. Calculate waiting_time[i] as min_burst_time - arrival_time[i].
10. Calculate turnaround_time[i] as burst_time[i] + waiting_time[i].
11. Calculate total waiting time and total turnaround time.
12. Calculate average waiting time and average turnaround time.
13. Display the process number, burst time, arrival time, waiting time, and turnaround time for each process.
14. Display the average waiting time and average turnaround time.
15. Stop.

```
                    burst_time[j + 1] = temp;
                }
            }
        }
        waiting_time[0] = 0;
        for (i = 1; i < n; i++) {
            int min_burst_time = burst_time[i];
            for (j = 0; j < i; j++) {
                min_burst_time += burst_time[j];
            }
            waiting_time[i] = min_burst_time - arrival_time[i];
            total_waiting_time += waiting_time[i];
        }
        for (i = 0; i < n; i++) {
            turnaround_time[i] = burst_time[i] + waiting_time[i];
            total_turnaround_time += turnaround_time[i];
        }
        avg_waiting_time = (float)total_waiting_time / n;
        avg_turnaround_time = (float)total_turnaround_time / n;
        printf("\nProcess\tBurst Time\tArrival Time\tWaiting Time\tTurnaround Time\n");
        for (i = 0; i < n; i++) {
            printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", i + 1, burst_time[i], arrival_time[i],
            waiting_time[i], turnaround_time[i]);
        }
        printf("\nAverage Waiting Time: %.2f\n", avg_waiting_time);
```

```
    printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);
    return 0;
}
```

OUTPUT:

Enter the number of processes: 4

Enter the burst time and arrival time for each process:

Burst time for process 1: 6

Arrival time for process 1: 0

Burst time for process 2: 8

Arrival time for process 2: 1

Burst time for process 3: 3

Arrival time for process 3: 2

Burst time for process 4: 4

Arrival time for process 4: 3

Process	Burst Time	Arrival Time	Waiting Time	Turnaround Time
1	6	0	0	6
2	8	1	6	14
3	3	2	14	17
4	4	3	17	21

Average Waiting Time: 9.25

Average Turnaround Time: 14.50

PROGRAM: 5.3

```
#include <stdio.h>

int main()

{

        int n, i, j, time_quantum, total_waiting_time = 0, total_turnaround_time = 0, total_processes_completed = 0;

        float avg_turnaround_time, avg_waiting_time;

        printf("Enter the number of processes: ");

        scanf("%d", &n);

        int burst_time[n], arrival_time[n], remaining_time[n], waiting_time[n], turnaround_time[n];

        printf("Enter the burst time and arrival time for each process:\n");

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

                printf("Burst time for process %d: ", i + 1);

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

                printf("Arrival time for process %d: ", i + 1);

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

                remaining_time[i] = burst_time[i];

        }

        printf("Enter the time quantum: ");

        scanf("%d", &time_quantum);

        int current_time = 0;

        while (total_processes_completed < n) {

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

                        if (remaining_time[i] > 0 && arrival_time[i] <= current_time) {
```

ALGORITHM: ROUND ROBIN

1. Start
2. Read the number of processes into variable n.
3. Declare arrays burst_time[], arrival_time[], remaining_time[], waiting_time[], and turnaround_time[], each of size n.
4. Read the burst time and arrival time for each process into their respective arrays.
5. Initialize the remaining time for each process to its burst time.
6. Read the time quantum into the variable time_quantum.
7. Initialize current_time to 0.
8. While the total number of processes completed is less than n, do the following:
    a. Loop through each process from i = 0 to i = n - 1.
    b. If the remaining time for process i is greater than 0 and its arrival time is less than or equal to current_time, do the following:

        i. If the remaining time for process i is less than or equal to the time quantum:

        - Increment current_time by the remaining time for process i.

        - Calculate the turnaround time for process i as current_time - arrival_time[i].

        - Calculate the waiting time for process i as turnaround_time[i] - burst_time[i].

        - Increment the total number of processes completed.

        ii. Otherwise (if the remaining time for process i is greater than the time quantum):

        - Increment current_time by the time quantum.

        - Reduce the remaining time for process i by the time quantum.
9. Calculate the average turnaround time as the total turnaround time divided by n.
10. Calculate the average waiting time as the total waiting time divided by n.
11. Display the process number, burst time, arrival time, turnaround time, and waiting time for each process in a formatted table.
12. Display the average turnaround time and average waiting time.
13. Stop

```
                if (remaining_time[i] <= time_quantum) {
                    current_time += remaining_time[i];
                    turnaround_time[i] = current_time - arrival_time[i];
                    total_turnaround_time += turnaround_time[i];
                    remaining_time[i] = 0;
                    total_waiting_time += turnaround_time[i] - burst_time[i];
                    total_processes_completed++;
                } else {
                    current_time += time_quantum;
                    remaining_time[i] -= time_quantum;
                }
            }
        }
    }
    avg_turnaround_time = (float)total_turnaround_time / n;
    avg_waiting_time = (float)total_waiting_time / n;
    printf("\nProcess\tBurst Time\tArrival Time\tTurnaround Time\tWaiting Time\n");
    for (i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", i + 1, burst_time[i], arrival_time[i],
        turnaround_time[i], turnaround_time[i] - burst_time[i]);
    }
    printf("\nAverage Turnaround Time: %.2f\n", avg_turnaround_time);
    printf("Average Waiting Time: %.2f\n", avg_waiting_time);
    return 0;
}
```

OUTPUT:

Enter the number of processes: 3

Enter the burst time and arrival time for each process:

Burst time for process 1: 5

Arrival time for process 1: 0

Burst time for process 2: 3

Arrival time for process 2: 1

Burst time for process 3: 7

Arrival time for process 3: 2

Enter the time quantum: 2

Process	Burst Time	Arrival Time	Turnaround Time	Waiting Time
1	5	0	12	7
2	3	1	6	3
3	7	2	17	10

Average Turnaround Time: 11.67

Average Waiting Time: 6.67

PROGRAM: 5.4

```
#include <stdio.h>
int main()
{
    int n, i, j, total_waiting_time = 0, total_turnaround_time = 0;
    float avg_turnaround_time, avg_waiting_time;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int burst_time[n], arrival_time[n], priority[n], waiting_time[n], turnaround_time[n], completion_time[n];
    printf("Enter the burst time, arrival time, and priority for each process:\n");
    for (i = 0; i < n; i++) {
        printf("Burst time for process %d: ", i + 1);
        scanf("%d", &burst_time[i]);
        printf("Arrival time for process %d: ", i + 1);
        scanf("%d", &arrival_time[i]);
        printf("Priority for process %d: ", i + 1);
        scanf("%d", &priority[i]);
    }
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (arrival_time[j] > arrival_time[j + 1]) {
                int temp = arrival_time[j];
                arrival_time[j] = arrival_time[j + 1];
                arrival_time[j + 1] = temp;
```

ALGORITHM: PRIORITY

1. Start
2. Read the number of processes into variable n.
3. Declare arrays burst_time[], arrival_time[], priority[], waiting_time[], turnaround_time[], and completion_time[], each of size n.
4. Read the burst time, arrival time, and priority for each process into their respective arrays.
5. Sort the processes based on arrival time.
6. Calculate completion time, turnaround time, and waiting time for each process.
    a. Initialize completion_time[0] as burst_time[0] + arrival_time[0].
    b. Calculate turnaround_time[0] as completion_time[0] - arrival_time[0].
    c. Calculate waiting_time[0] as turnaround_time[0] - burst_time[0].
    d. Update total_waiting_time by adding waiting_time[0].
    e. Update total_turnaround_time by adding turnaround_time[0].
7. Calculate average turnaround time as total_turnaround_time / n.
8. Calculate average waiting time as total_waiting_time / n.
9. Display the process number, burst time, arrival time, priority, turnaround time, and waiting time for each process in a formatted table.
10. Display the average turnaround time and average waiting time.
11. Stop

```
                temp = burst_time[j];
                burst_time[j] = burst_time[j + 1];
                burst_time[j + 1] = temp;
                temp = priority[j];
                priority[j] = priority[j + 1];
                priority[j + 1] = temp;
            }
        }
    }
    completion_time[0] = burst_time[0] + arrival_time[0];
    turnaround_time[0] = completion_time[0] - arrival_time[0];
    waiting_time[0] = turnaround_time[0] - burst_time[0];
    total_waiting_time += waiting_time[0];
    total_turnaround_time += turnaround_time[0];
    for (i = 1; i < n; i++) {
        completion_time[i] = completion_time[i - 1] + burst_time[i];
        turnaround_time[i] = completion_time[i] - arrival_time[i];
        waiting_time[i] = turnaround_time[i] - burst_time[i];
        total_waiting_time += waiting_time[i];
        total_turnaround_time += turnaround_time[i];
    }
    avg_turnaround_time = (float)total_turnaround_time / n;
    avg_waiting_time = (float)total_waiting_time / n;
    printf("\nProcess\tBurst Time\tArrival Time\tPriority\tTurnaround Time\tWaiting Time\n");
```

```
    for (i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i + 1, burst_time[i], arrival_time[i],
        priority[i], turnaround_time[i], waiting_time[i]);
    }
    printf("\nAverage Turnaround Time: %.2f\n", avg_turnaround_time);
    printf("Average Waiting Time: %.2f\n", avg_waiting_time);
    return 0;
}
```

OUTPUT:

```
Enter the number of processes: 4
Enter the burst time, arrival time, and priority for each process:
Burst time for process 1: 6
Arrival time for process 1: 0
Priority for process 1: 3
Burst time for process 2: 4
Arrival time for process 2: 1
Priority for process 2: 1
Burst time for process 3: 7
Arrival time for process 3: 2
Priority for process 3: 4
Burst time for process 4: 5
Arrival time for process 4: 3
Priority for process 4: 2
```

Process	Burst Time	Arrival Time	Priority	Turnaround Time	Waiting Time
1	6	0	3	6	0
2	4	1	1	5	1
3	7	2	4	12	5
4	5	3	2	14	9

Average Turnaround Time: 9.25

Average Waiting Time: 3.75

RESULT

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

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 : ");
        gets(str);
        printf("Data written in memory : %s\n", str);
        shmdt(str);
        return 0;
}
```

EXPERIMENT NO. 6 **DATE:**

IPC USING SHARED MEMORY

AIM

To write C programs to implement Inter Process Communication Using Shared Memory.

(Readers-Writers Problem)

ALGORITHM: Writer Process

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;
}
```

ALGORITHM: Reader Process

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

OUTPUT :

~$ gcc writer.c

~$ ./a.out

Write Data : hai

Data written in memory: hai

~$ gcc reader.c

~$ ./a.out

Data read from memory: hai

RESULT

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

PROGRAM :

```
#include<stdio.h>
int mutex=1,full=0,empty=3,x=0;
main()
{
        int n;
        void producer();
        void consumer();
        int wait(int);
        int signal(int);
        printf("\n1.PRODUCER\n2.CONSUMER\n3.EXIT\n");
        while(1)
        {
                printf("\nENTER YOUR CHOICE\n");
                scanf("%d",&n);
                switch(n)
                {
                        case 1: if((mutex==1)&&(empty!=0))
                                        producer();
                                else
                                        printf("BUFFER IS FULL");
                                break;
                        case 2: if((mutex==1)&&(full!=0))
                                consumer();
```

EXPERIMENT NO. 7 **DATE:**

PRODUCER-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.

```
                else
                        printf("BUFFER IS EMPTY");
                        break;
                case 3: exit(0);
                        break;
                }
        }
}
int wait(int s)
{
        return(--s);
}
int signal(int s)
{
        return(++s);
}
void producer()
{
        mutex=wait(mutex);
        full=signal(full);
        empty=wait(empty);
        x++;
        printf("\nproducer produces the item%d",x);
        mutex=signal(mutex);
```

```
        }
void consumer()
{
        mutex=wait(mutex);
        full=wait(full);
        empty=signal(empty);
        printf("\n consumer consumes item%d",x);
        x--;
        mutex=signal(mutex);
}
```

OUTPUT:

1.PRODUCER

2.CONSUMER

3.EXIT

ENTER YOUR CHOICE

1

producer produces the item 1

ENTER YOUR CHOICE

1

producer produces the item 2

ENTER YOUR CHOICE

2

consumer consumes item 2

ENTER YOUR CHOICE

3

RESULT

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

PROGRAM : BEST FIT

```
#include <stdio.h>
void bestfit (int f[], int b[], int nf, int nb)
{
	for(int i=0; i<nf; i++)
	{
		int flag=0;
		for(int j=0; j<nb; j++)
		{
			int temp = b[j]-f[i];
			if(temp>=0)
			{
				printf("\n File size %d is put in %d partition", f[i],b[j]);
				b[j]=0;
				flag=1;
				break;
			}
		}
		if (flag == 0)
			printf ("\n File with size %d has to wait", f[i]);
	}
}
```

EXPERIMENT NO. 8 **DATE:**

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. Define a function bestfit that takes four parameters:
   - Array f[] to store file sizes.
   - Array b[] to store partition sizes.
   - Integer nf representing the number of files.
   - Integer nb representing the number of partitions.
3. Start the function definition.
4. Iterate over each file size in the array f[]:
5. Initialize a flag variable flag to 0.
6. Iterate over each partition size in the array b[]:
7. Calculate the difference between the partition size and the file size (temp = b[j] - f[i]).
8. If the difference is non-negative, print a message indicating that the file of size f[i] is put in the partition b[j]. Update the partition size to 0 to indicate it's allocated, set flag to 1, and break out of the loop.
9. If the flag is still 0 after iterating over all partitions, print a message indicating that the file with size f[i] has to wait.
10. End the function definition.
11. Function main:Define the main function.
12. Declare integer variables nb and nf to store the number of memory blocks and files respectively.
13. Read the number of memory blocks and read the input into nb.
14. Declare an array b[] to store the sizes of memory blocks and prompt the user to enter the sizes in order. Read and store the input in the array b[].
15. Read the number of files and read the input into nf.

```
void main()
{
int nb,nf;
        printf ("Enter no of memory blocks: ");
        scanf ("%d", &nb);
        int b[nb];
        printf ("Enter no of files: ");
        scanf ("%d", &nf);
        int f[nf], i, j;
        printf("Enter the sizes of memory blocks in order: ");
        for (int i=0 ; i<nb; i++)
                scanf("%d", &b[i]);
        printf("Enter the sizes of files in order: ");
        for (i=0; i<nf; i++)
                scanf ("%d", &f[i]);
        for (i=0; i<(nb-1); i++)
                for (j=i+1; j<nb; j++)
                        if(b[j]<b[i])
                        {
                                int temp=b[j];
                                b[j]=b[i];
                                b[i] = temp;
                        }
        bestfit(f,b,nf,nb);
}
```

16. Declare an array f[] to store the sizes of files and prompt the user to enter the sizes in order. Read and store the input in the array f[].
17. Sort the array b[] in ascending order to apply the best fit strategy.
18. Call the bestfit function with arguments f, b, nf, and nb.
19. End of the main function.
20. Stop

OUTPUT:

Enter no of memory blocks: 4

Enter no of files: 4

Enter the sizes of memory blocks in order: 200 350 100 500

Enter the sizes of files in order: 150 333 222 40

File size 150 is put in 200 partition

File size 333 is put in 350 partition

File size 222 is put in 500 partition

File size 40 is put in 100 partition

PROGRAM : WORST FIT

```
#include <stdio.h>
#define max 25
void worstfit(int b[], int f[], int nb, int nf)
{
	int i, j, k = 0, temp, highest, flag;
	for (i = 1; i <= nf; i++)
	{
		highest = 0;
		flag = 0;
		for (j = 1; j <= nb; j++)
		{
			temp = b[j] - f[i];
			if (temp >= 0) {
				if (highest < temp) {
					k = j;
					highest = temp;
				}
				flag = 1;
			}
		}
		if (flag)
		{
			printf("\nFile Size %d is put in %d partition", f[i], b[k]);
```

ALGORITHM: WORST FIT

1. Start.
2. Include necessary header files.
3. Define a constant ‘max’ to represent the maximum size of arrays.
4. Define the function ‘worstfit’.
5. Declare the required variables.
6. Iterate over each file:
   - ☐ Initialize `highest` and `flag` to 0 for each file.
   - ☐ Iterate over each block:
   - ☐ Calculate the remaining space in the block after allocating the file.
   - ☐ If the remaining space is non-negative:
   - ☐ If the remaining space is greater than the current highest remaining space (`highest`), update `highest` and `k`.
   - ☐ Set `flag` to 1 to indicate that the file is allocated.
   - ☐ If `flag` is true (file is allocated):
   - ☐ Print a message indicating the file size and the partition where it's put.
   - ☐ Update the block size with the remaining space after allocation.
   - ☐ If `flag` is false (file cannot be allocated):
   - ☐ Print a message indicating that the file must wait.
7. Define the `main()` function:
   - ☐ Declare variables.
   - ☐ Declare arrays to store block sizes and file sizes.
   - ☐ Read the number of blocks and files.
   - ☐ Read the sizes of the blocks.
   - ☐ Read the sizes of the files.
   - ☐ Call the ‘worstfit()’ function with the arrays ‘b[]’ and ‘f[]’, along with the number of blocks and files as arguments.
   - ☐ Return 0 to indicate successful execution.
8. Stop

```
                        b[k] = highest;
                }
                else
                {
                        printf("\nFile Size %d must wait", f[i]);
                }
        }
}
int main()
{
        int i, nb, nf;
        int b[max], f[max];
        printf("Memory Management Scheme - Worst Fit\n");
        printf("Enter the number of blocks: ");
        scanf("%d", &nb);
        printf("Enter the number of files: ");
        scanf("%d", &nf);
        printf("Enter the size of the blocks:\n");
        for (i = 1; i <= nb; i++) {
                printf("Block %d : ", i);
                scanf("%d", &b[i]);
        }
        printf("Enter the size of the files:\n");
        for (i = 1; i <= nf; i++) {
```

```
            printf("File %d: ", i);
            scanf("%d", &f[i]);
        }
        worstfit(b, f, nb, nf);
        return 0;
}
```

OUTPUT :

```
Memory Management Scheme - Worst Fit
Enter the number of blocks: 4
Enter the number of files: 3
Enter the size of the blocks:
Block 1 : 100
Block 2 : 200
Block 3 : 50
Block 4 : 150
Enter the size of the files:
File 1: 60
File 2: 120
File 3: 80
File Size 60 is put in 4 partition
File Size 120 is put in 2 partition
File Size 80 is put in 1 partition
```

PROGRAM: FIRST FIT

```
#include <stdio.h>
#define max 25
void firstfit(int b[], int f[], int nb, int nf)
{
	int i, j, k = 0, temp, flag;
	for (i = 1; i <= nf; i++)
	{
		flag = 0;
		for (j = 1; j <= nb; j++)
		{
			temp = b[j] - f[i];
			if (temp >= 0) {
				k = j;
				printf("\nFile Size %d is put in %d partition", f[i], b[k]);
				b[k] = temp;
				flag = 1;
				break;
			}
		}
		if (flag == 0)
		printf("\nFile Size %d must Wait", f[i]);
	}
}
```

ALGORITHM: FIRST FIT

1. Start.
2. Implement the firstfit function:
3. Iterate over each file.
    i. For each file, iterate over each block.
    ii. Check if the current block size is sufficient to accommodate the file:
4. Subtract the file size from the block size.
5. If the result is greater than or equal to zero, assign the file to this block.
6. Update the block size to reflect the remaining space.
7. Set the flag to 1 to indicate successful allocation.
    i. If no block can accommodate the file (flag remains 0), print a message indicating that the file must wait.
    ii. Reset the flag for the next file.
8. Implement the main function:
9. Declare arrays b and f to store block sizes and file sizes, respectively.
10. Read the number of blocks and files.
11. Read the sizes of blocks and files.
12. Call the firstfit function with the arrays b and f.
13. Return 0 to indicate successful execution.
14. Stop.

```
int main()
{
	int i, nb, nf;
	int b[max], f[max];
	printf("Memory Management Scheme - First Fit\n");
	printf("Enter the number of blocks: ");
	scanf("%d", &nb);
	printf("Enter the number of files: ");
	scanf("%d", &nf);
	printf("Enter the size of the blocks:\n");
	for (i = 1; i <= nb; i++)
	{
		printf("Block %d : ", i);
		scanf("%d", &b[i]);
	}
	printf("Enter the size of the files:\n");
	for (i = 1; i <= nf; i++)
	{
		printf("File %d: ", i);
		scanf("%d", &f[i]);
	}
	firstfit(b, f, nb, nf);
	return 0;
}
```

OUTPUT :

Memory Management Scheme - First Fit

Enter the number of blocks: 4

Enter the number of files: 3

Enter the size of the blocks:

Block 1 : 50

Block 2 : 70

Block 3 : 30

Block 4 : 100

Enter the size of the files:

File 1: 60

File 2: 40

File 3: 80

File Size 60 is put in 1 partition

File Size 40 is put in 2 partition

File Size 80 must Wait

RESULT

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

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");
```

EXPERIMENT NO. 9 **DATE:**

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.

```
    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;
                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;
        }
        printf("Frames: ");
        for (j = 0; j < num_frames; j++) {
            printf("%d ", frames[j]);
        }
        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: 1 2 3 4 1 2 5 1

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

FIFO Page Replacement Algorithm:

Page Fault: Page 1

Frames: 1 -1 -1

Page Fault: Page 2

Frames: 1 2 -1

Page Fault: Page 3

Frames: 1 2 3

Page Fault: Page 4

Frames: 4 2 3

Page Fault: Page 1

Frames: 4 1 3

Page Fault: Page 2

Frames: 4 1 2

Page Fault: Page 5

Frames: 4 5 2

Page Fault: Page 1

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");
```

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.

```
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;
                }
        }
        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;
        }
        printf("Frames: ");
```

```
            for (j = 0; j < num_frames; j++) {
                printf("%d ", frames[j]);
            }
            printf("\n");
        }
        printf("\nTotal Page Faults: %d\n", page_faults);
        return 0;
}
```

OUTPUT

Enter the number of frames: 3

Enter the number of pages: 10

Enter the page reference string: 7 0 1 2 0 3 0 4 2 3

Page Reference String: 7 0 1 2 0 3 0 4 2 3

LRU Page Replacement Algorithm:

Page Fault: Page 7

Frames: 7 -1 -1

Page Fault: Page 0

Frames: 7 0 -1

Page Fault: Page 1

Frames: 7 0 1

Page Fault: Page 2

Frames: 2 0 1

Page Fault: Page 3

Frames: 2 3 1

Page Fault: Page 0

Frames: 2 3 0

Page Fault: Page 4

Frames: 4 3 0

Page Fault: Page 2

Frames: 4 2 0

Page Fault: Page 3

Frames: 4 2 3

Total Page Faults: 9

PROGRAM: LFU

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

ALGORITHM: LFU

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.
3. Read the value of number of frames from the user.
4. Read the value of number of pages from the user.
5. Read the page reference string (pages) from the user and initialize the page_frequency array to all 0s.
6. Initialize all frames in frames to -1 (indicating empty).
7. Display the page reference string.
8. Perform the LFU page replacement algorithm:
    - ☐ Iterate through each page in the page reference string:
    - ☐ Check if the page is already present in any frame:
    - ☐ If yes, mark page_found as 1 and increment the page frequency.
    - ☐ If the page is not found in any frame:
    - ☐ Increment page_faults.
    - ☐ Find the least frequently used frame:
    - ☐ Iterate through frames and find the frame with the lowest page frequency.
    - ☐ Replace the page in the least frequently used frame with the current page.
    - ☐ Set the frequency of the new page to 1.
9. Display the contents of frames after the page replacement.
10. Display the total number of page faults.
11. End.

```
for (i = 0; i < num_pages; i++) {
        int page_found = 0;
        int least_frequency_frame = 0;
        int least_frequency = page_frequency[frames[0]];
        for (j = 0; j < num_frames; j++) {
                if (frames[j] == pages[i]) {
                        page_found = 1;
                        page_frequency[pages[i]]++;
                        break;
                }
                if (page_frequency[frames[j]] < least_frequency) {
                        least_frequency = page_frequency[frames[j]];
                        least_frequency_frame = j;
                }
        }
        if (!page_found) {
                printf("\nPage Fault: Page %d\n", pages[i]);
                page_faults++;
                frames[least_frequency_frame] = pages[i];
                page_frequency[pages[i]] = 1;
        }
        printf("Frames: ");
        for (j = 0; j < num_frames; j++) {
                printf("%d ", frames[j]);
```

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

OUTPUT

Enter the number of frames: 3

Enter the number of pages: 10

Enter the page reference string: 7 0 1 2 0 3 0 4 2 3

Page Reference String: 7 0 1 2 0 3 0 4 2 3

LFU Page Replacement Algorithm:

Page Fault: Page 7

Frames: 7 -1 -1

Page Fault: Page 0

Frames: 7 0 -1

Page Fault: Page 1

Frames: 7 0 1

Page Fault: Page 2

Frames: 2 0 1

Page Fault: Page 0

Frames: 2 0 1

Page Fault: Page 3

Frames: 2 0 3

Page Fault: Page 0

Frames: 2 0 3

Page Fault: Page 4

Frames: 4 0 3

Page Fault: Page 2

Frames: 4 2 3

Page Fault: Page 3

Frames: 4 2 3

Total Page Faults: 9

RESULT

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

PROGRAM :

```
#include <stdio.h>
int main()
{
        int n, m, i, j, k;
        printf("Enter number of processes: ");
        scanf("%d", &n);
        printf("Enter number of resources: ");
        scanf("%d", &m);
        int available[m];
        printf("Enter available resources: ");
        for (i = 0; i < m; i++)
                scanf("%d", &available[i]);
        int allocation[n][m], max[n][m], need[n][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 max matrix:\n");
        for (i = 0; i < n; i++) {
                printf("For process %d: ", i);
                for (j = 0; j < m; j++)
```

EXPERIMENT NO. 10 **DATE:**

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: Subtract allocation matrix from the maximum matrix to get the need matrix.
9. Initialize the finish array to track the finish status of each process (0 for unfinished, 1 for finished).
10. Initialize the work array to represent the available resources.
11. Initialize the safe sequence array and count to keep track of the safe sequence.
12. 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.
13. Loop through each process:
14. If the process is not finished: Check if the resources needed by the process can be satisfied by available resources.
15. If resources are sufficient, mark the process as finished, update available resources, and add the process to the safe sequence.
16. Set the 'found' flag to 1 if a safe process is found in this iteration.
17. If no safe process is found in the current iteration, break the loop and print "System is in unsafe state".
18. If the system is in a safe state, print "System is in safe state".
19. Display the safe sequence of processes.
20. Stop.

```
            scanf("%d", &max[i][j]);
    }
    // Calculate 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];
    for (i = 0; i < n; i++)
            finish[i] = 0;
            int work[m];
    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];
                safeSeq[count++] = i;
                finish[i] = 1;
                found = 1;
            }
        }
    }
    if (!found) {
        printf("System is in unsafe state\n");
        return -1;
    }
}
printf("System is in safe state\n");
printf("Safe sequence: ");
for (i = 0; i < n; i++)
    printf("%d ", safeSeq[i]);
    printf("\n");
    return 0;
}
```

OUTPUT

Enter number of processes: 3

Enter number of resources: 4

Enter available resources: 2 1 0 0

Enter allocation matrix:

For process 0: 0 1 0 0

For process 1: 2 0 0 1

For process 2: 3 0 2 0

Enter max matrix:

For process 0: 7 5 3 2

For process 1: 3 2 2 2

For process 2: 9 0 2 2

System is in safe state

Safe sequence: 1 2 0

RESULT

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

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: ");
```

EXPERIMENT NO. 11 **DATE:**

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 arrays for allocation, request, and available resources.
9. Initialize an array 'finish' to keep track of the finish status of each process. Initially, all processes are unfinished.
10. Check if a Process can be Executed:
11. Iterate through each process:
    - If the process is not finished:
    - Check if the resources requested by the process can be satisfied by available resources.
    - If resources are sufficient, mark the process as finished, release allocated resources, and reset the loop to check all processes again.
    - Check for Unfinished Processes:
12. Iterate through each process:
    - If any process is still unfinished, set the 'deadlock' flag to indicate deadlock.
13. If the 'deadlock' flag is set, print "Deadlock detected".
14. Otherwise, print "No deadlock detected".
15. Stop.

```
for (i = 0; i < m; i++)
scanf("%d", &available[i]);
int finish[n];
for (i = 0; i < n; i++)
finish[i] = 0;
// Check if a process can be executed
int deadlock = 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;
                        }
                }
                if (canExecute) {
                        // Process can be executed, release resources
                        for (j = 0; j < m; j++)
                                available[j] += allocation[i][j];
                                finish[i] = 1;
                                i = -1; // Reset i to check all processes again
                }
        }
```

```
        }
        // Check for any unfinished process
        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: 4

Enter allocation matrix:

For process 0: 0 1 0 0

For process 1: 2 0 0 1

For process 2: 3 0 2 0

Enter request matrix:

For process 0: 7 5 3 2

For process 1: 3 2 2 2

For process 2: 9 0 2 2

Enter available resources: 2 1 0 0

Deadlock detected

RESULT

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

PROGRAM : FCFS

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
	int n, i, total_movement = 0, previous, current;
	printf("Enter the number of disk requests: ");
	scanf("%d", &n);
	if (n <= 0) {
		printf("Invalid number of disk requests.\n");
		return 1;
	}
	printf("Enter the disk request sequence:\n");
	scanf("%d", &previous);
	for (i = 1; i < n; i++) {
		scanf("%d", &current);
		total_movement += abs(current - previous);
		previous = current;
	}
	printf("Total head movement: %d\n", total_movement);
	printf("Average head movement: %.2f\n", (float)total_movement / n);
	return 0;
}
```

EXPERIMENT NO. 12 **DATE:**

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 n, i, total_movement, previous, and current of type integer.
3. Read the value of n from the user.
4. If n is less than or equal to 0, print "Invalid number of disk requests." and return 1, indicating an error.
5. Read the first disk request value into the variable previous.
6. Loop from i = 1 to n-1:
    i. Read the next disk request value into the variable current.
    ii. Calculate the absolute difference between current and previous and add it to total_movement.
    iii. Update the value of previous to current.
7. Display the value of total_movement.
8. Display Average head movement.
9. Stop.

OUTPUT

Enter the number of disk requests: 5

Enter the disk request sequence:

98

183

37

122

14

Total head movement: 292

Average head movement: 58.40

PROGRAM: SCAN

```
#include <stdio.h>
#include <stdlib.h>
#define DIRECTION_LEFT 0
#define DIRECTION_RIGHT 1
int main()
{
    int n, head, direction;
    printf("Enter the number of disk requests: ");
    scanf("%d", &n);
    if (n <= 0) {
        printf("Invalid number of disk requests.\n");
        return 1;
    }
    int *requests = (int *)malloc(n * sizeof(int));
    printf("Enter the disk request sequence:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &requests[i]);
    }
    printf("Enter the initial head position: ");
    scanf("%d", &head);
    printf("Enter the direction (0 for left, 1 for right): ");
    scanf("%d", &direction);
```

ALGORITHM: SCAN

1. Start
2. Declare variables n, head, and direction of type integer.
3. Read the value of n from the user.
4. If n is less than or equal to 0, print "Invalid number of disk requests." and return 1, indicating an error.
5. Dynamically allocate memory for an array requests of size n to store disk requests.
6. Read n disk request values from the user and store them in the array requests.
7. Read the value of head from the user.
8. Read the value of direction from the user.
9. If direction is not 0 or 1, print "Invalid direction." and return 1, indicating an error.
10. Sort the requests array in ascending order using the Bubble Sort algorithm.
11. Initialize total_movement to 0.
12. Declare variable current_index of type integer.
13. If direction is DIRECTION_LEFT, add the initial head position to total_movement, print it, and set head to 0.
14. Else, if direction is DIRECTION_RIGHT, add the absolute difference between the initial head position and the last request to total_movement, print it, and set head to the last request.
15. Begin scanning in the specified direction:
    a. If direction is DIRECTION_LEFT, search for the index of the closest request less than or equal to the current head position from right to left in the requests array.
    b. If direction is DIRECTION_RIGHT, search for the index of the closest request greater than or equal to the current head position from left to right in the requests array.
16. If no such request index is found, break out of the loop.
17. Add the absolute difference between the current head position and the request at current_index to total_movement, print the request, and update head to the position of the request.
18. Repeat steps 15-17 until all requests have been scanned.
19. Display the Total head movement.
20. Free the memory allocated for the requests array.
21. End.

```
if (direction != 0 && direction != 1) {
        printf("Invalid direction.\n");
        return 1;
}
// Sorting the requests array in ascending order using Bubble Sort
for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
                if (requests[j] > requests[j + 1]) {
                        int temp = requests[j];
                        requests[j] = requests[j + 1];
                        requests[j + 1] = temp;
                }
        }
}
int total_movement = 0;
int current_index;
if (direction == DIRECTION_LEFT) {
        // Move the head to the leftmost track
        total_movement += head;
        printf("%d -> ", head);
        head = 0;
} else if (direction == DIRECTION_RIGHT) {
        // Move the head to the rightmost track
        total_movement += abs(head - requests[n - 1]);
```

```
            printf("%d -> ", head);
            head = requests[n - 1];
        }
        // Continue scanning in the specified direction
        while (1) {
            if (direction == DIRECTION_LEFT) {
                current_index = -1;
                for (int i = n - 1; i >= 0; i--) {
                    if (requests[i] <= head) {
                        current_index = i;
                        break;
                    }
                }
                if (current_index == -1)
                    break;
                total_movement += abs(head - requests[current_index]);
                printf("%d -> ", requests[current_index]);
                head = requests[current_index];
            } else if (direction == DIRECTION_RIGHT) {
                current_index = -1;
                for (int i = 0; i < n; i++) {
                    if (requests[i] >= head) {
                        current_index = i;
                        break;
```

```
                }
            }
            if (current_index == -1)
                break;
            total_movement += abs(head - requests[current_index]);
            printf("%d -> ", requests[current_index]);
            head = requests[current_index];
        }
    }
    printf("Total head movement: %d\n", total_movement);
    free(requests);
    return 0;
}
```

OUTPUT:

Enter the number of disk requests: 5

Enter the disk request sequence:

98

183

37

122

14

Enter the initial head position: 53

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

53 -> 98 -> 122 -> 183 -> Total head movement: 313

PROGRAM: C-SCAN

```
#include <stdio.h>
#include <stdlib.h>
#define DIRECTION_LEFT 0
#define DIRECTION_RIGHT 1
int main()
{
    int n, head, direction;
    printf("Enter the number of disk requests: ");
    scanf("%d", &n);
    if (n <= 0) {
        printf("Invalid number of disk requests.\n");
        return 1;
    }
    int *requests = (int *)malloc(n * sizeof(int));
    printf("Enter the disk request sequence:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &requests[i]);
    }
    printf("Enter the initial head position: ");
    scanf("%d", &head);
    printf("Enter the direction (0 for left, 1 for right): ");
    scanf("%d", &direction);
```

ALGORITHM: C-SCAN

1. Start
2. Declare variables n, head, and direction of type integer.
3. Read the value of n from the user.
4. If n is less than or equal to 0, print "Invalid number of disk requests." and return 1, indicating an error.
5. Dynamically allocate memory for an array requests of size n to store disk requests.
6. Read n disk request values from the user and store them in the array requests.
7. Read the value of head from the user.
8. Read the value of direction from the user.
9. If direction is not 0 or 1, print "Invalid direction." and return 1, indicating an error.
10. Sort the requests array in ascending order using the Bubble Sort algorithm.
11. Initialize total_movement to 0.
12. If direction is DIRECTION_LEFT, add the initial head position to total_movement, print it, and set head to 0.
13. Else, if direction is DIRECTION_RIGHT, add the absolute difference between the initial head position and the last request to total_movement, print it, and set head to the last request.
14. Begin scanning in the specified direction:
    - ☐ If direction is DIRECTION_LEFT, search for the index of the closest request less than or equal to the current head position from right to left in the requests array.
    - ☐ If direction is DIRECTION_RIGHT, search for the index of the closest request greater than or equal to the current head position from left to right in the requests array.
15. If no such request index is found, handle the edge case by moving the head to the opposite end and continue scanning.
16. Add the absolute difference between the current head position and the request at current_index to total_movement, print the request, and update head to the position of the request.
17. Repeat steps 14-16 until all requests have been scanned.
18. Display the Total head movement.
19. Free the memory allocated for the requests array.
20. End.

```
if (direction != 0 && direction != 1) {
        printf("Invalid direction.\n");
    return 1;
}
// Sorting the requests array in ascending order using Bubble Sort
for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
                if (requests[j] > requests[j + 1]) {
                        int temp = requests[j];
                        requests[j] = requests[j + 1];
                        requests[j + 1] = temp;
                }
        }
}
int total_movement = 0;
if (direction == DIRECTION_LEFT) {
        // Move the head to the leftmost track
        total_movement += head;
        printf("%d -> ", head);
        head = 0;
} else if (direction == DIRECTION_RIGHT) {
        // Move the head to the rightmost track
        total_movement += abs(head - requests[n - 1]);
        printf("%d -> ", head);
```

```
            head = requests[n - 1];
        }
        // Continue scanning in the specified direction
        while (1) {
            if (direction == DIRECTION_LEFT) {
                int current_index = -1;
                for (int i = n - 1; i >= 0; i--) {
                        if (requests[i] <= head) {
                                current_index = i;
                                break;
                          }
                }
                if (current_index == -1) {
                        // Move the head to the rightmost track
                        total_movement += abs(head - requests[n - 1]);
                        printf("%d -> ", requests[n - 1]);
                        head = requests[n - 1];
                  } else {
                        total_movement += abs(head - requests[current_index]);
                        printf("%d -> ", requests[current_index]);
                        head = requests[current_index];
                }
        } else if (direction == DIRECTION_RIGHT) {
                int current_index = -1;
```

```
            for (int i = 0; i < n; i++) {
                if (requests[i] >= head) {
                    current_index = i;
                    break;
                }
            }
            if (current_index == -1) {
                // Move the head to the leftmost track
                total_movement += head;
                printf("%d -> ", head);
                head = 0;
            } else {
                total_movement += abs(head - requests[current_index]);
                printf("%d -> ", requests[current_index]);
                head = requests[current_index];
            }
        }
        if (head == 0 && direction == DIRECTION_RIGHT) {
            break;
        }
        if (head == requests[n - 1] && direction == DIRECTION_LEFT) {
            break;
        }
    }
```

```
    printf("Total head movement: %d\n", total_movement);
    free(requests);
    return 0;
}
```

OUTPUT:

```
Enter the number of disk requests: 7
Enter the disk request sequence:
98
183
37
122
14
124
65
Enter the initial head position: 53
Enter the direction (0 for left, 1 for right): 1
53 -> 65 -> 98 -> 122 -> 124 -> 183 -> Total head movement: 269
```

RESULT

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