ವಿಶ್ವೇಶ್ವರಯ್ಯ ತಾಂತ್ರಿಕ ವಿಶ್ವವಿದ್ಯಾಲಯ, ಬೆಳಗಾವಿ
VISVESVARAYA TECHNOLOGICAL UNIVERSITY - BELAGAVI

BANGALORE INSTITUTE OF TECHNOLOGY

K.R.ROAD, V.V.PURA, BENGALURU -560 004

Department of Computer Science and Engineering (Data Science)

Operating Systems Laboratory Manual (BCS303)

III- Semester

Prepared by

Prof. Prathik K

BANGALORE INSTITUTE OF TECHNOLOGY
(*Affiliated to Visvesvaraya Technological University – VTU*)
K.R. Road, V.V. Puram, Bengaluru – 560 004. Phone: 26613237/26615865
Fax: 22426796

DEPARTMENT OF CSE (DATA SCIENCE)

Vision

Establish and develop the Institute as the Centre of higher learning, ever abreast with expanding horizon of knowledge in the field of Engineering and Technology with entrepreneurial thinking, leadership excellence for life-long success and solve societal problems.

Mission

1. Provide high quality education in the Engineering disciplines from the undergraduate through doctoral levels with creative academic and professional programs.
2. Develop the Institute as a leader in Science, Engineering, Technology, Management and Research and apply knowledge for the benefit of society.
3. Establish mutual beneficial partnerships with Industry, Alumni, Local, State and Central Governments by Public Service Assistance and Collaborative Research.
4. Inculcate personality development through sports, cultural and extracurricular activities and engage in social, economic and professional challenges.

BANGALORE INSTITUTE OF TECHNOLOGY
(*Affiliated to Visvesvaraya Technological University – VTU*)
K.R. Road, V.V. Puram, Bengaluru – 560 004. Phone: 26613237/26615865
Fax: 22426796

DEPARTMENT OF CSE (DATA SCIENCE)

VISION:

To be a center of excellence in computer engineering education, empowering graduates as highly skilled professionals.

MISSION:

1. To provide a platform for effective learning with emphasis on technical excellence.
2. To train the students to meet current industrial standards and adapt to emerging technologies.
3. To instill the drive for higher learning and research initiatives.
4. To inculcate the qualities of leadership and Entrepreneurship.

PROGRAM EDUCATIONAL OBJECTIVES (PEO)

1. Graduates will apply fundamental and advanced concepts of Computer Science and Engineering for solving real world problems.
2. Graduates will build successful professional careers in various sectors to facilitate societal needs.
3. Graduates will have the ability to strengthen the level of expertise through higher studies and research.
4. Graduates will adhere to professional ethics and exhibit leadership qualities to become an Entrepreneur.

PROGRAM SPECIFIC OUTCOMES (PSOs)

1. The graduates of the program will have the ability to build software products by applying theoretical concepts and programming skills.
2. The graduates of the program will have the ability to pursue higher

Program Outcomes

1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

COURSE LEARNING OBJECTIVES (CLO)

CLO1. To Demonstrate the need for OS and different types of OS.

CLO2. To discuss suitable techniques for management of different resources.

CLO3. To demonstrate different APIs/Commands related to processor, memory, storage and file system management.

COURSE OUTCOMES (CO)

At the end of the course the student will be able to:

CO1. Explain the structure and functionality of Operating Systems.

CO2. Apply appropriate CPU Scheduling algorithm for the given Problem.

CO3. Analyze the various Technics for process for process Synchronization and Deadlock Handling.

CO4. Apply various technics for memory management.

CO5. Explain file and secondary management Strategies.

CO6. Describe the need for information Protection mechanism.

CO TO PO & PSO MAPPING

		PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2
BCS303	**CO1**	**3**		**2**		**1**									
	CO2	**3**	**2**	**2**		**1**									
	CO3	**3**	**2**	**2**		**1**									
	CO4	**3**	**2**	**2**		**1**									
	CO5	**3**		**2**		**1**									
	CO6	**3**					**1**								

Operating Systems Laboratory

Subject Code : BCS303 **CIE Marks : 50**
Hours/Week : 0:0:2 **Total Hours : 20**

LIST OF PROGRAMS

Sl. No.	Name of Experiment
1.	Develop a C Program to Implement the process System call(fork(), exec(), wait(), create process terminate process)
2.	Simulate the following CPU scheduling algorithms to find turnaround time and waiting time: a) FCFS b) SJF c) Round Robin d) Priority.
3.	Develop a C program to simulate producer-consumer problem using semaphores.
4.	Develop a C Program which demonstrates inter-process communication between a reader process and a writer process. Use mkfifo, open, read, write and close APIs in your program.
5.	Develop a C program to simulate Bankers Algorithm for DeadLock Avoidance.
6.	Develop a C program to simulate the following contiguous memory allocation Techniques: a) Worst fit b) Best fit c) First fit.
7.	Develop C program to simulate page replacement algorithms: a) FIFO b) LRU
8.	Simulate following File Organization Techniques a) Single level directory b) Two level directory
9.	Develop a C program to simulate the Linked file allocation strategies.
10.	Develop a C program to simulate SCAN disk sheduling algorithm.

Operating Systems Laboratory

Subject Code : BCS303 **CIE Marks: 50**
Hours/Week : 0:0:2 **Total Hours: 20**

SCHEDULE OF EXPERIMENTS

Sl. No	Name of Experiment	WEEK
1	Sample programs	Week1
2	Develop a C Program to Implement the process System call(fork(), exec(), wait(), create process, terminate process)	Week2
3	Simulate the following CPU scheduling algorithms to find turnaround time and waiting time: a) FCFS b) SJF c) Round Robin d) Priority	Week3
4	Develop a C program to simulate procedure-consumer problem using semaphores.	Week4
5	Develop a C Program which demonstrates interprocess communication between a reader process and a write process. Use mkfifo, open, read, write and close APIs in your program.	Week5
6	Develop a C program to simulate Bankers Algorithm for DeadLock Avoidance.	Week6
7	Develop a C program to simulate the following contiguous memory allocation Techniques: a) Worst fit b) Best fit c) First fit.	Week7
8	Develop C program to simulate page replacement algorithms: a) FIFO b) LRU	Week8
9	Simulate following File Organization Techniques a) Single level directory b) Two level directory	Week9
10	Develop a C program to simulate the Linked file allocation strategies.	Week10
11	Develop a C program to simulate SCAN disk sheduling algorithm.	Week11
12	Final Lab Test	Week12

LABORATORY

General Lab Guidelines:

1. Conduct yourself in a responsible manner at all times in the laboratory. Intentional misconduct will lead to exclusion from the lab.
2. Do not wander around, or distract other students, or interfere with the laboratory experiments of other students.
3. Read the handout and procedures before starting the experiments. Follow all written and verbal instructions carefully.
4. If you do not understand the procedures, ask the instructor or teaching assistant. Attendance in all the labs is mandatory, absence permitted only with prior permission from the Class teacher.
5. The workplace has to be tidy before, during and after the experiment.
6. Do not eat food, drink beverages or chew gum in the laboratory.
7. Every student should know the location and operating procedures of all Safety equipment including First Aid Kit and Fire extinguisher.

DO'S:-

1. An ID card is a must.
2. Keep your belongings in a designated area.
3. Sign the log book when you enter/leave the laboratory.
4. Records have to be submitted every week for evaluation.
5. The program to be executed in the respective lab session has to be written in the lab observation copy beforehand.
6. After the lab session, shut down the computers.
7. Report any problem in system (if any) to the person in-charge

DON'TS:-

1. Do not insert metal objects such as clips, pins and needles into the computer casings(They may cause fire) and should not attempt to repair, open, tamper or interfere with any of the computer, printing, cabling, or other equipment in the laboratory.
2. Do not change the system settings and keyboard keys.
3. Do not upload, delete or alter any software/ system files on laboratory computers.
4. No additional material should be carried by the students during regular labs.
5. Do not open any irrelevant websites in labs.

6. Do not use a flash drive on lab computers without the consent of the lab instructor.
7. Students are not allowed to work in the Laboratory alone or without the presence of the instructor/teaching assistant.

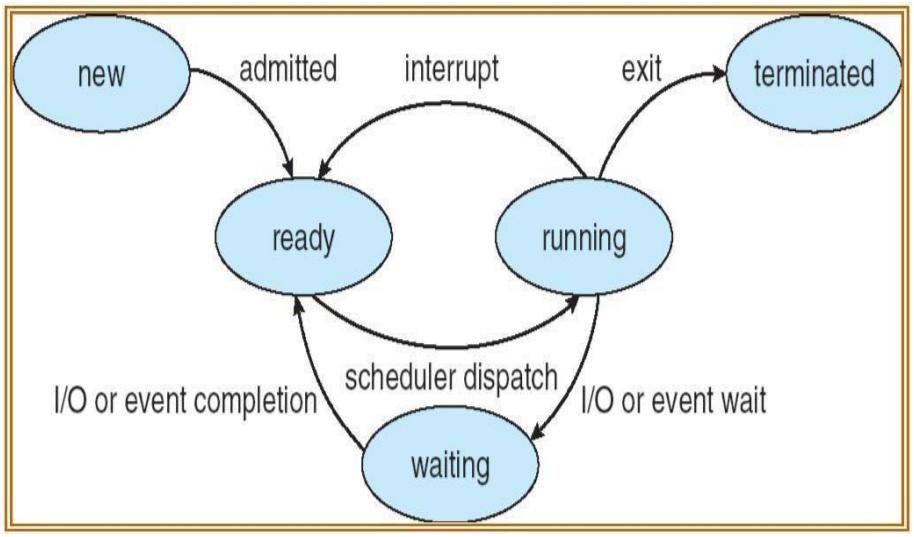
INTRODUCTION TO OPERATING SYSTEMS

Introduction

An Operating System is a program that manages the Computer hardware. It controls and coordinates the use of the hardware among the various application programs for the various users.

A Process is a program in execution. As a process executes, it changes *state*

- New: The process is being created
- Running: Instructions are being executed
- Waiting: The process is waiting for some event to occur
- Ready: The process is waiting to be assigned to a process
- Terminated : The process has finished execution



Apart from the program code, it includes the current activity represented by

- Program Counter
- Contents of Processor registers,
- Process Stack which contains temporary data like function parameters, return addresses and local variables.
- Data section which contains global variables.
- Heap for dynamic memory allocation

A Multi-programmed system can have many processes running simultaneously with the CPU multiplexed among them. By switching the CPU between the processes, the OS can make the computer more productive. There is Process Scheduler which selects the process among many processes that are ready, for program execution on the CPU. Switching the CPU to another process requires performing a state save of the current process and a state

restore of new process, this is Context Switch.

Advantages of OS

- The GUI includes a variety of menus, buttons, and symbols that all work together to make any interaction easier to understand and utilize.
- In addition, the GUI is intended to be user-friendly, so it doesn't require additional skills for a beginner to use the system.
- The OS is responsible for controlling the PC's hardware capabilities with the help of device drivers that assist in controlling a specific hardware device.
- Using a distributed operating system (DOS), data processing tasks can be divided among many processors.
- With the aid of an OS, the same information may be given to several users at once.
- The most remarkable feature of an OS is the plug-and-play feature which eliminates the need for any additional drivers to use devices like a mouse or keyboard.
- The OS is also responsible for controlling the memory available in the computer system. For the same, it performs memory division, scheduling algorithms, etc. to manage the memory space.
- In order to schedule various processes, OS also employs schedulers. Schedulers are special system software that selects a job (application program) to be entered into the system and determine which process (application or system program) should be executed in an instant. Schedulers can either be long-term, short-term, or medium-term.
- Several scheduling algorithms are also used to select a process for execution. Some of them are First-come-first-serve (FCFS), Shortest job first (SJF), Round robin (RR), Priority scheduling, etc.
- Operating systems perform context-switching between a number of processes. Before switching to another process, it saves the state of the current process and then moves to the other one. The state of a process is saved in a Process Control Block (PCB). PCB is a data structure to save the information of a process

List of OS available

LINUX: Linux is a family of open-source Unix-like operating systems based on the Linux kernel, an operating system kernel first released on September 17, 1991, by Linus Torvalds.

macOS: macOS is an operating system developed and marketed by Apple Inc. since 2001. It is the primary operating system for Apple's Mac computers.

Windows: Microsoft Windows is a group of several proprietary graphical operating

system families developed and marketed by Microsoft.

Ubuntu: Ubuntu is a Linux distribution based on Debian and composed mostly of free and open-source software.

Android: Android is a mobile operating system based on a modified version of the Linux kernel and other open-source software, designed primarily for touchscreen mobile devices such as smartphones and tablets.

General Commands

Command	Function
date	Used to display the current system date and time.
date +%D	Displays date only
date +%T	Displays time only
date +% Y	Displays the year part of date
date +% H	Displays the hour part of time
cal	Calendar of the current month
calyear	Displays calendar for all months of the specified year
calmonth year	Displays calendar for the specified month of the year
who	Login details of all users such as their IP, Terminal No, User name,
who am i	Used to display the login details of the user
tty	Used to display the terminal name
uname	Displays the Operating System
uname –r	Shows version number of the OS (kernel).
uname –n	Displays domain name of the server
echo "txt"	Displays the given text on the screen
echo $HOME	Displays the user's home directory
bc	Basic calculator. Press **Ctrl+d**to quit
lpfile	Allows the user to spool a job along with others in a print queue.
man cmdname	Manual for the given command. Press **q** to exit
history	To display the commands used by the user since log on.
exit	Exit from a process. If shell is the only process then logs out

Directory Commands

Command	Function
pwd	Path of the present working directory
mkdirdir	A directory is created in the given name under the current directory
mkdirdir1 dir2	A number of sub-directories can be created under one stroke
cd subdir	Change Directory. If the subdirstarts with / then path starts from **root** (absolute) otherwise from current working directory.
Cd	To switch to the home directory.
cd /	To switch to the root directory.
cd..	To move back to the parent directory
rmdirsubdir	Removes an empty sub-directory.

File Commands

Command	Function
cat >filename	To create a file with some contents. To end typing press **Ctrl+d**. The >symbol means redirecting output to a file. (<for input)
cat filename	Displays the file contents.
cat >>filename	Used to append contents to a file
cpsrc des	Copy files to given location. If already exists, it will be overwritten
cp –i src des	Warns the user prior to overwriting the destination file
cp –r src des	Copies the entire directory, all its sub-directories and files.
mv old new	To rename an existing file or directory. –i option can also be used
mv f1 f2 f3 dir	To move a group of files to a directory.
mv –v old new	Display name of each file as it is moved.
Rmfile	Used to delete a file or group of files. –i option can also be used
rm *	To delete all the files in the directory.
rm –r *	Deletes all files and sub-directories
rm –f *	To forcibly remove even write-protected files
Ls	Lists all files and subdirectories (blue colored) in sorted manner.
Lsname	To check whether a file or directory exists.
lsname*	Short-hand notation to list out filenames of a specific pattern.
ls –a	Lists all files including hidden files (files beginning with .)
ls –x dirname	To have specific listing of a directory.
ls –R	Recursive listing of all files in the subdirectories

ls –l	Long listing showing file access rights (read/write/execute-**rwx** for user/group/others-**ugo**).
cmpfile1 file2	Used to compare two files. Displays nothing if files are identical.
Wcfile	It produces a statistics of lines (**l**), words(**w**), and characters(**c**).
chmodperm file	Changes permission for the specified file. (r=4, w=2, x=1) chmod 740 file sets all rights for user, read only for groups and no rights for others

SAMPLE PROGRAM QUESTIONS

1. Write a C program to sort the elements using bubble sort and selection sort technique.
2. Write a C program to search an element using Binary search and Linear Search.
3. Write a program in C to add numbers using call by reference.
4. Write a program in C to find the factorial of a given number using pointers.
5. Write a program in C to print the first 50 natural numbers using recursion.
6. Write a C program to store and display the details of n students using structure.
7. Write a program in C to merge two arrays of the same size sorted in descending order.
8. Write a C program to display a pattern like a right-angle triangle with a number.

   The pattern like:
   1
   01
   101
   0101
   10101
9. Write a program in C to display the sum of the series [9 + 99 + 999 + 9999 ...].
10. Write a program in C to print Floyd's Triangle.

LAB PROGRAMS

PROGRAM 1

1. Develop a program to implement the Process system calls(fork(),exec(),wait(),create process, terminate process)

DESCRIPTION:

i. fork ( ) :

Used to create new process. The new process consists of a copy of the address space of the original process. The value of process id for the child process is zero, whereas the value of process id for the parent is an integer value greater than zero.

Syntax: fork ( );

ii. wait ( ) :

The parent waits for the child process to complete using the wait system call. The wait system call returns the process identifier of a terminated child, so that the parent can tell which of its possibly many children has terminated.

Syntax: wait (NULL);

iii. exit ( ):

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit system call. At that point, the process may return data (output) to its parent process (via the wait system call).

Syntax: exit (0);

PROGRAM:

```
#include<stdio.h> // printf()
#include<stdlib.h> // exit()
#include<sys/types.h> // pid_t
#include<sys/wait.h> // wait()
#include<unistd.h> // fork
int main(int argc, char **argv)
{
        pid_t pid;
        pid = fork();
        if(pid==0)
        {
                printf("It is the child process and pid is %d\n",getpid());
```

```
                int i=0;
                for(i=0;i<8;i++)
                {
                        printf("%d\n",i);
                }
                exit(0);
                }
else if(pid > 0)
        {
        printf("It is the parent process and pid is %d\n",getpid());
        int status;
        wait(&status);
        printf("Child is reaped\n");
        }
        else
        {
                printf("Error in forking..\n");
                exit(EXIT_FAILURE);
        }
return 0;
}
```

Explanation:

1. **Header Files:**
   - **#include<stdio.h>**: Standard input-output functions.
   - **#include<stdlib.h>**: Standard library functions, including **exit**.
   - **#include<sys/types.h>**: Data types used in system calls.
   - **#include<sys/wait.h>**: Header file for the **wait** function.
   - **#include<unistd.h>**: Provides access to POSIX operating system API, including the **fork** function.
2. **main Function:**
   - **int main(int argc, char **argv)**: The main function takes command-line arguments, but it doesn't use them in this program.
3. **Forking:**

- **pid_t pid;**: Declaration of a variable to store the process ID (PID).
- **pid = fork();**: The **fork** system call is used to create a new process by duplicating the existing process.
    - If **fork** returns 0, it means the process is the child.
    - If **fork** returns a positive value, it is the PID of the parent process.
    - If **fork** returns -1, an error occurred.

4. **Child Process:**
    - **if (pid == 0)**: This condition checks if the process is the child.
        - The child process prints its own PID using **getpid()**.
        - It then runs a loop to print numbers from 0 to 7.
        - After printing, it calls **exit(0)** to terminate the child process.
5. **Parent Process:**
    - **else if (pid > 0)**: This condition checks if the process is the parent.
        - The parent process prints its own PID using **getpid()**.
        - It declares a variable **status** to store the exit status of the child.
        - It waits for the child process to complete using **wait(&status)**.
        - After the child process completes, it prints "Child is reaped."
6. **Error Handling:**
    - **else**: This block is executed if **fork** returns -1, indicating an error in process creation.
        - It prints an error message and exits the program with **EXIT_FAILURE**.
7. **Returning from main:**
    - **return 0;**: The **main** function returns 0 to indicate successful execution.

OUTPUT

```
It is the parent process and pid is 14998
It is the child process and pid is 14999
0
1
2
3
4
5
6
7Child is reaped
```

PROGRAM 2

2**.** Simulate the following CPU scheduling algorithms to find turnaround time and waiting time a) FCFS b) SJF c) Round Robin d) Priority

DESCRIPTION:

i. FCFS SCHEDULING ALGORITHM:

FCFS is a simplest CPU scheduling algorithm that schedules according to the arrival time of processes. The first come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first. It is implemented by using the FIFO queue.

PROGRAM:

```
#include<stdio.h>
int a[10],b[10],no[10],wt[10],ta[10];
void main()
{
        int i,j,sb=0,n,l,temp,c;
        float avgw,tw=0,tt=0,avgt;
        printf("Enter no of processes\n");
        scanf("%d",&n);
        for(i=0;i<n;i++)
{
         no[i]=i;
         printf("Enter the Arrival time and Burst time for process %d\n",i);
        scanf("%d%d",&a[i],&b[i]);
}

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

```
                b[j]=b[j+1];
                b[j+1]=temp;
                temp=no[j];
                no[j]=no[j+1];
                no[j+1]=temp;
        }
           l=a[0];

for(i=0;i<n;i++)
   {
    if(i==0)
      sb=l;
    else
    {
    sb=sb+b[i-1];
    }
    wt[i]=sb-a[i];
    ta[i]=wt[i]+b[i];
   }

for(i=0;i<n;i++)
{
  tw=tw+wt[i];
  tt=tt+ta[i];
}
avgw=tw/n;
avgt=tt/n;

printf("\tProcess\t\tAT\t\tBT\t\tWT\t\tTAT\n");
for(i=0;i<n;i++)
  {
          printf("\tP%d\t\t%d\t\t%d\t\t%d\t\t%d\n",no[i],a[i],b[i],wt[i],ta[i]);
  }
printf("\nAverage WT is %f\n",avgw);
```

```
printf("\nAverage TAT is %f\n",avgt);
}
```

Explanation:

1. **Arrays:**
   - **a[10]**: Array to store arrival times of processes.
   - **b[10]**: Array to store burst times of processes.
   - **no[10]**: Array to store process numbers.
   - **wt[10]**: Array to store waiting times of processes.
   - **ta[10]**: Array to store turnaround times of processes.
2. **Main Function:**
   - The program starts by reading the number of processes (**n**) from the user.
   - It then reads the arrival time and burst time for each process and stores them in the respective arrays.
3. **Sorting:**
   - The program sorts the processes based on their arrival times using the Bubble Sort algorithm. The arrays **a**, **b**, and **no** are rearranged accordingly.
4. **Calculating Waiting Time and Turnaround Time:**
   - The waiting time (**wt**) for each process is calculated using the formula: **waiting time = start time - arrival time**.
   - The turnaround time (**ta**) for each process is calculated using the formula: **turnaround time = waiting time + burst time**.
5. **Average Waiting Time and Turnaround Time:**
   - The program calculates the total waiting time (**tw**) and total turnaround time (**tt**) for all processes.
   - It then calculates the average waiting time (**avgw**) and average turnaround time (**avgt**).
6. **Output:**
   - The program prints a table showing the process number, arrival time, burst time, waiting time, and turnaround time for each process.
   - It also prints the average waiting time and average turnaround time for all processes.

OUTPUT

```
Enter no of processes
5
Enter the Arrival time and Burst time for process 0
2 2
Enter the Arrival time and Burst time for process 1
5 6
Enter the Arrival time and Burst time for process 2
0 4
Enter the Arrival time and Burst time for process 3
0 7
Enter the Arrival time and Burst time for process 4
7 4
```

Process	AT	BT	WT	TAT
P2	0	4	0	4
P3	0	7	4	11
P0	2	2	9	11
P1	5	6	8	14
P4	7	4	12	16

```
Average WT is 6.600000
Average TAT is 11.200000
```

DESCRIPTION:

ii. SJF SCHEDULING ALGORITHM:

SJF scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. Arrange all the jobs in order with respect to their burst times. There may be two jobs in queue with the same execution time, and then FCFS approach is to be performed. Each process will be executed according to the length of its burst time. Then calculate the waiting time and turnaround time of each of the processes accordingly.

PROGRAM:

```
#include<stdio.h>
int main()
{
        int time, burst_time[10], at[10], sum_burst_time = 0, smallest, n, i;
        int sumt = 0, sumw = 0;
        printf("enter the no of processes : ");
        scanf("%d", & n);
        for (i = 0; i < n; i++) {
```

```
        printf("the arrival time for process P%d : ", i + 1);
        scanf("%d", & at[i]);
        printf("the burst time for process P%d : ", i + 1);
        scanf("%d", & burst_time[i]);
        sum_burst_time += burst_time[i];
}
burst_time[9] = 9999;
for (time = 0; time < sum_burst_time;)
{
        smallest = 9;
        for (i = 0; i < n; i++)
        {
        if (at[i] <= time && burst_time[i] > 0 && burst_time[i] < burst_time[smallest])
        smallest = i;
        }
printf("P[%d]\t|\t%d\t|\t%d\n", smallest + 1, time + burst_time[smallest] - at[smallest], time -
at[smallest]);
sumt += time + burst_time[smallest] - at[smallest];
sumw += time - at[smallest];
time += burst_time[smallest];
burst_time[smallest] = 0;
}
printf("\n\n average waiting time = %f", sumw * 1.0 / n);
printf("\n\n average turnaround time = %f", sumt * 1.0 / n);
return 0;
}
```

Explanation:

1. **Variable Declarations:**
   - **time**: Keeps track of the current time in the scheduling algorithm.
   - **burst_time[10]**: An array to store the burst times of processes.
   - **at[10]**: An array to store the arrival times of processes.
   - **sum_burst_time**: Variable to calculate the total burst time of all processes.

- **smallest**: Index of the process with the smallest burst time.
- **n**: Number of processes.
- **i**: Loop variable.
- **sumt** and **sumw**: Variables to calculate the total turnaround time and waiting time, respectively.

2. **Input:**
   - The program takes input for the number of processes (**n**), arrival times (**at**), and burst times (**burst_time**) for each process.
3. **Initialization:**
   - The **burst_time[9]** is set to a large value (9999) to ensure that it is greater than any actual burst time. This helps in finding the smallest burst time later.
4. **Scheduling:**
   - The program uses a loop that continues until the total burst time is exhausted.
   - Within the loop, it finds the process with the smallest burst time that has arrived (**at[i] <= time**).
   - The process is then executed for its burst time, and the time is updated accordingly.
   - The burst time of the selected process is set to 0 to mark it as completed.
5. **Output:**
   - The program prints the turnaround time and waiting time for each process.
   - It also calculates and prints the average waiting time and average turnaround time.
6. **Average Waiting Time and Turnaround Time Calculation:**
   - **sumt** accumulates the turnaround time of all processes.
   - **sumw** accumulates the waiting time of all processes.
   - The averages are then calculated and printed at the end of the program.

OUTPUT

```
enter the no of processes : 5
the arrival time and burst time for process P0 : 2 6
the arrival time and burst time for process P1 : 5 2
the arrival time and burst time for process P2 : 1 8
the arrival time and burst time for process P3 : 0 3
the arrival time and burst time for process P4 : 4 4
```

Process	AT	BT	WT	TAT
P[3]	0	3	3	0
P[0]	2	6	7	1
P[1]	5	2	6	4
P[4]	4	4	11	7
P[2]	1	8	22	14

average waiting time = 5.200000

average turnaround time = 9.800000

DESCRIPTION:

iii. ROUND ROBIN SCHEDULING ALGORITHM:

For round robin scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the size of the time slice. Time slices are assigned to each process in equal portions and in circular order, handling all processes execution. This allows every process to get an equal chance. Calculate the waiting time and turnaround time of each of the processes accordingly.

PROGRAM:

```
#include<stdio.h>

int main()
{
   //Input no of processed
   int  n;
   printf("Enter Total Number of Processes:");
   scanf("%d", &n);
   int wait_time = 0, ta_time = 0, arr_time[n], burst_time[n], temp_burst_time[n];
   int x = n;

   //Input details of processes
   for(int i = 0; i < n; i++)
   {
      printf("Enter Details of Process %d \n", i + 1);
      printf("Arrival Time:  ");
      scanf("%d", &arr_time[i]);
```

```
    printf("Burst Time:   ");
    scanf("%d", &burst_time[i]);
    temp_burst_time[i] = burst_time[i];
}

//Input time slot
int time_slot;
printf("Enter Time quantum:");
scanf("%d", &time_slot);

//Total indicates total time
//counter indicates which process is executed
int total = 0,  counter = 0,i;
printf("Process ID      Burst Time      Turnaround Time      Waiting Time\n");
for(total=0, i = 0; x!=0; )
{
    // define the conditions
    if(temp_burst_time[i] <= time_slot && temp_burst_time[i] > 0)
    {
        total = total + temp_burst_time[i];
        temp_burst_time[i] = 0;
        counter=1;
    }
    else if(temp_burst_time[i] > 0)
    {
        temp_burst_time[i] = temp_burst_time[i] - time_slot;
        total  += time_slot;
    }
    if(temp_burst_time[i]==0 && counter==1)
    {
        x--; //decrement the process no.
        printf("\nProcess No %d  \t\t %d\t\t\t\t %d\t\t\t %d", i+1, burst_time[i],
              total-arr_time[i], total-arr_time[i]-burst_time[i]);
        wait_time = wait_time+total-arr_time[i]-burst_time[i];
```

```
            ta_time += total -arr_time[i];
            counter =0;
        }
        if(i==n-1)
        {
            i=0;
        }
        else if(arr_time[i+1]<=total)
        {
            i++;
        }
        else
        {
            i=0;
        }
    }
    float average_wait_time = wait_time * 1.0 / n;
    float average_turnaround_time = ta_time * 1.0 / n;
    printf("\nAverage Waiting Time:%f", average_wait_time);
    printf("\nAvg Turnaround Time:%f", average_turnaround_time);
    return 0;
}
```

Explanation :

1. **Variable Declarations:**
    - **n**: Number of processes.
    - **wait_time**: Total waiting time for all processes.
    - **ta_time**: Total turnaround time for all processes.
    - **arr_time[n]**: Array to store arrival times of processes.
    - **burst_time[n]**: Array to store burst times of processes.
    - **temp_burst_time[n]**: Temporary array to store remaining burst times of processes after each time quantum.
    - **x**: Variable to track the number of remaining processes.

- **time_slot**: Time quantum or time slice.

2. **Input:**
   - The program takes input for the number of processes (**n**), arrival times (**arr_time**), and burst times (**burst_time**) for each process.
   - It also takes input for the time quantum or time slice (**time_slot**).
3. **Initialization:**
   - **temp_burst_time** is initialized with the original burst times of processes.
4. **Round Robin Scheduling:**
   - The program uses a **for** loop to iterate until all processes are completed (**x** becomes 0).
   - It checks each process and executes it for the time quantum (**time_slot**) if its remaining burst time is greater than 0.
   - After each execution, it updates the total time (**total**), remaining burst time, and checks if the process is completed.
5. **Output:**
   - The program prints a table showing the process number, burst time, turnaround time, and waiting time for each process.
   - It also calculates and prints the average waiting time and average turnaround time for all processes.

OUTPUT

Enter Total Number of Processes:3
Enter Details of Process 1
Arrival Time: 0
Burst Time: 10
Enter Details of Process 2
Arrival Time: 1
Burst Time: 8
Enter Details of Process 3
Arrival Time: 2
Burst Time: 7
Enter Time quantum:5

Process ID	Burst Time	Turnaround Time	Waiting Time
Process No 1	10	20	10
Process No 2	8	22	14
Process No 3	7	23	16

Average Waiting Time: 13.333333

Avg Turnaround Time: 21.666666

DESCRIPTION:

iv. PRIORITY SCHEDULING ALGORITHM

For priority scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the priorities. Arrange all the jobs in order with respect to their priorities. There may be two jobs in queue with the same priority, and then FCFS approach is to be performed. Each process will be executed according to its priority. Calculate the waiting time and turnaround time of each of the processes accordingly.

PROGRAM:

```
#include<stdio.h>

struct process
{
    char process_name;
    int arr_time, bst_time, ct, waiting_time, turn_around_time, priority;
    int status;
}process_queue[10];

int limit;

void Arr_time_Sorting()
{
    struct process temp;
    int i, j;
```

```
    for(i = 0; i < limit - 1; i++)
    {
        for(j = i + 1; j < limit; j++)
        {
            if(process_queue[i].arr_time > process_queue[j].arr_time)
            {
                temp = process_queue[i];
                process_queue[i] = process_queue[j];
                process_queue[j] = temp;
            }
        }
    }
}
void main()
{
    int i, time = 0, bst_time = 0, largest;
    char c;
 float wait_time = 0, turn_around_time = 0, average_waiting_time,
average_turn_around_time;
    printf("\nEnter Total Number of Processes:\t");
    scanf("%d", &limit);
    for(i = 0, c = 'A'; i < limit; i++, c++)
    {
        process_queue[i].process_name = c;
        printf("\nEnter Details For Process[%C]:\n", process_queue[i].process_name);
        printf("Enter Arrival Time:\t");
        scanf("%d", &process_queue[i].arr_time );
        printf("Enter Burst Time:\t");
        scanf("%d", &process_queue[i].bst_time);
        printf("Enter Priority:\t");
        scanf("%d", &process_queue[i].priority);
        process_queue[i].status = 0;
        bst_time = bst_time + process_queue[i].bst_time;
    }
```

```
    Arr_time_Sorting();
    process_queue[9].priority = 9999;
    printf("\nProcess Name\tArrival Time\tBurst Time\tPriority\tWaiting Time\tTurnAround Time");
    for(time = process_queue[0].arr_time; time < bst_time;)
    {
        largest = 9;
        for(i = 0; i < limit; i++)
        {
            if(process_queue[i].arr_time <= time && process_queue[i].status != 1 && process_queue[i].priority < process_queue[largest].priority)
            {
                largest = i;
            }
        }
        time = time + process_queue[largest].bst_time;
        process_queue[largest].ct = time;
        process_queue[largest].waiting_time = process_queue[largest].ct - process_queue[largest].arr_time - process_queue[largest].bst_time;
        process_queue[largest].turn_around_time = process_queue[largest].ct - process_queue[largest].arr_time;
        process_queue[largest].status = 1;
        wait_time = wait_time + process_queue[largest].waiting_time;
        turn_around_time = turn_around_time + process_queue[largest].turn_around_time;
        printf("\n%c\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d", process_queue[largest].process_name, process_queue[largest].arr_time, process_queue[largest].bst_time, process_queue[largest].priority, process_queue[largest].waiting_time, process_queue[largest].turn_around_time);
    }
    average_waiting_time = wait_time / limit;
    average_turn_around_time = turn_around_time / limit;
    printf("\n\nAverage waiting time:\t%f\n", average_waiting_time);
    printf("Average Turnaround Time:\t%f\n", average_turn_around_time);
}
```

Explanation:

1. **Struct Definition:**
   - **struct process**: Defines a structure to store information about each process, including its name, arrival time, burst time, completion time, waiting time, turnaround time, priority, and status.
2. **Function Arr_time_Sorting():**
   - Sorts the processes based on arrival time in ascending order using the Bubble Sort algorithm.
3. **Main Function:**
   - Declares variables for process execution (**i**, **time**, **bst_time**, **largest**, **c**) and average calculations (**wait_time**, **turn_around_time**, **average_waiting_time**, **average_turn_around_time**).
   - Takes input for the number of processes (**limit**) and details of each process, including arrival time, burst time, and priority.
   - Calls the **Arr_time_Sorting()** function to sort processes based on arrival time.
   - Initializes a dummy process with the highest priority to ensure correct priority comparisons.
   - Executes processes based on priority until the total burst time is reached.
   - Calculates and prints waiting time and turnaround time for each process.
   - Calculates and prints average waiting time and average turnaround time for all processes.

OUTPUT:

```
Enter Total Number of Processes:        7
Enter Details For Process[A]:
Enter Arrival Time:     0
Enter Burst Time:       3
Enter Priority: 2

Enter Details For Process[B]:
Enter Arrival Time:     2
Enter Burst Time:       5
Enter Priority: 6

Enter Details For Process[C]:
Enter Arrival Time:     1
Enter Burst Time:       4
```

Enter Priority: 3

Enter Details For Process[D]:
Enter Arrival Time: 4
Enter Burst Time: 2
Enter Priority: 5

Enter Details For Process[E]:
Enter Arrival Time: 6
Enter Burst Time: 9
Enter Priority: 7

Enter Details For Process[F]:
Enter Arrival Time: 5
Enter Burst Time: 4
Enter Priority: 4

Enter Details For Process[G]:
Enter Arrival Time: 7
Enter Burst Time: 10
Enter Priority: 10

Process Name	Arrival Time	Burst Time	Priority	Waiting Time	TurnAroundtime
A	0	3	2	0	3
C	1	4	3	2	6
F	5	4	4	2	6
D	4	2	5	7	9
B	2	5	6	11	16
E	6	9	7	12	21
G	7	10	10	20	30

Average waiting time: 7.714286
Average Turnaround Time: 13.000000

PROGRAM 3

3. Develop a C program to simulate Producer-Consumer Problem using semaphores.

DESCRIPTION:

Producer-consumer problem, is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process. One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, there must be available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

PROGRAM :

```
#include <stdio.h>
#include <stdlib.h>
int mutex = 1;
int full = 0;
int empty = 3, x = 0,y=0;
void producer()
{
        --mutex;
        ++full;
        --empty;
        x++;
        printf("\nProducer produces""item %d",x);
        ++mutex;
}
void consumer()
{
        --mutex;
        --full;
        ++empty;
```

```
        y++;
        printf("\nConsumer consumes ""item %d", y);
        ++mutex;
}
int main()
{
        int n, i;
        printf("\n1. Press 1 for Producer"
                "\n2. Press 2 for Consumer"
                "\n3. Press 3 for Exit");
        #pragma omp critical
        for (i = 1; i > 0; i++) {
        printf("\nEnter your choice:");
        scanf("%d", &n);
        switch (n)
        {
                case 1:
                if ((mutex == 1) && (empty != 0)) {
                producer();
        }
        else
        {
                printf("Buffer is full!");
        }
        break;
        case 2:
        if ((mutex == 1) && (full != 0))
        {
                consumer();
        }
        else
        {
                printf("Buffer is empty!");
        }
```

```
        break;
        case 3:
        exit(0);
        break;
        }
    }
}
```

Explanation:

1. **Global Variables:**
   - **mutex**: A binary semaphore that controls access to the critical section.
   - **full**: Represents the number of items currently in the buffer.
   - **empty**: Represents the number of empty slots in the buffer.
   - **x** and **y**: Counters for the produced and consumed items, respectively.
2. **Producer Function (producer):**
   - Decrements **mutex** to enter the critical section.
   - Increments **full** and decrements **empty** to indicate a produced item.
   - Increments **x** to keep track of the produced items.
   - Prints a message indicating that the producer has produced an item.
   - Increments **mutex** to exit the critical section.
3. **Consumer Function (consumer):**
   - Decrements **mutex** to enter the critical section.
   - Decrements **full** and increments **empty** to indicate a consumed item.
   - Increments **y** to keep track of the consumed items.
   - Prints a message indicating that the consumer has consumed an item.
   - Increments **mutex** to exit the critical section.
4. **Main Function:**
   - Presents a menu to the user with options for producing, consuming, or exiting.
   - Uses **#pragma omp critical** to ensure that only one thread at a time can execute the critical section to prevent race conditions.
   - The loop continues to prompt the user for input until the user chooses to exit.
   - Depending on the user's choice, it calls the **producer** function, **consumer** function, or exits the program.

- The program checks if the buffer is full before allowing the producer to produce and if the buffer is empty before allowing the consumer to consume.

OUTPUT:

```
1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 for Exit
Enter your choice:1

Producer produces item 1
Enter your choice:1

Producer produces item 2
Enter your choice:1

Producer produces item 3
Enter your choice:2

Consumer consumes item 1
Enter your choice:1

Producer produces item 4
Enter your choice:2

Consumer consumes item 2
Enter your choice:1

Producer produces item 5
Enter your choice:1
Buffer is full!
Enter your choice:2

Consumer consumes item 3
Enter your choice:2

Consumer consumes item 4
Enter your choice:2

Consumer consumes item 5
Enter your choice:2
Buffer is empty!
Enter your choice:3
```

PROGRAM 4

4. Develop a C program which demonstrates interprocess communication between a reader process and writer process. Use mkfifo, open, read,write and close API's in your program.

DESCRIPTION:

The file structure related system calls available in the UNIX system let you create, open, and close files, read and write files, randomly access files, alias and remove files, get information about files, check the accessibility of files, change protections, owner, and group of files, and control devices. These operations either use a character string that defines the absolute or relative path name of a file, or a small integer called a file descriptor that identifies the I/O channel. A channel is a connection between a process and a file that appears to the process as an unformatted stream of bytes. The kernel presents and accepts data from the channel as a process reads and writes that channel. To a process then, all input and output operations are synchronous and unbuffered.

SYSTEM CALLS USED:

System calls are functions that a programmer can call to perform the services of the operating system.

Open (): Open () system call to open a file. open () returns a file descriptor, an integer specifying the position of this open n file in the table of open files for the current process.

The return value is the descriptor of the file. Returns -1 if the file could not be opened. The first parameter is path name of the file to be opened and the second parameter is the opening mode

Close (): Close () system call to close a file.

It is always good practices to close files when not needed as open files do consume resources and all normal systems impose a limit on the number of files that a process can hold open.

Read (): The read () system call is used to read data from a file or other object identified by a file descriptor.

fd is the descriptor, buf is the base address of the memory area into which the data is read and MAX_BUF is the maximum amount of data to read. The return value is the actual amount of data read from the file. The pointer is incremented by the amount of data read. An attempt to read beyond the end of a file results in a return value of zero.

Write (): The write () system call is used to write data to a file or other object identified by a file descriptor.

PROGRAM:

```
/*Writer Process*/
#include <stdio.h>
```

```
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
        int fd;
        char buf[1024];
        /* create the FIFO (named pipe) */
        char * myfifo = "/tmp/myfifo";
        mkfifo(myfifo, 0666);
        printf("Run Reader process to read the FIFO File\n");
        fd = open(myfifo, O_WRONLY);
        write(fd,"Hi", sizeof("Hi"));
        /* write "Hi" to the FIFO */
        close(fd);
        unlink(myfifo); /* remove the FIFO */
        return 0;
}
```

Explanation:

Writer Process:

- It creates a named pipe (FIFO) using mkfifo.
- Opens the FIFO for writing using open and gets a file descriptor (fd).
- Writes the string "Hi" to the FIFO using the write system call.
- Closes the file descriptor and removes the FIFO using unlink.
- mkfifo(myfifo, 0666);: This line creates a named pipe (FIFO) named "/tmp/myfifo" with read and write permissions for everyone (0666).
- fd = open(myfifo, O_WRONLY);: This line opens the named pipe in write-only mode (O_WRONLY). The file descriptor (fd) is used for subsequent write operations.
- write(fd, "Hi", sizeof("Hi"));: This line writes the string "Hi" to the named pipe using the write system call. The third argument specifies the number of bytes to write.
- close(fd);: Closes the file descriptor, indicating that the writing is complete.
- unlink(myfifo);: Removes (deletes) the named pipe. This step is optional and depends on whether you want to keep the named pipe after the communication is done.

```
/* Reader Process*/
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#define MAX_BUF 1024
int main()
{
        int fd;
        /* A temp FIFO file is not created in reader */
        char *myfifo = "/tmp/myfifo";
        char buf[MAX_BUF];
        /* open, read, and display the message from the FIFO */
        fd = open(myfifo, O_RDONLY);
        read(fd, buf, MAX_BUF);
        printf("Writer: %s\n", buf);
        close(fd);
        return 0;
}
```

Explanation:

Reader Process:

- Opens the existing named pipe (FIFO) for reading using open and gets a file descriptor (fd).
- Reads the message from the FIFO using the read system call.
- Displays the received message on the console.
- Closes the file descriptor.
- fd = open(myfifo, O_RDONLY);: Opens the named pipe in read-only mode (O_RDONLY). The file descriptor (fd) is used for subsequent read operations.
- read(fd, buf, MAX_BUF);: Reads data from the named pipe into the buffer (buf). The maximum number of bytes to read is defined as MAX_BUF.
- printf("Writer: %s\n", buf);: Displays the received message on the console.
- close(fd);: Closes the file descriptor after reading is complete.

OUTPUT

cc writepro.c
./a.out
Run Reader process to read the FIFO File

cc readpro.c
./a.out
Writer: Hi

PROGRAM 5

5. Develop a C program to simulate Banker's algorithm for deadlock avoidance.

DESCRIPTION:

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock. Deadlock avoidance is one of the techniques for handling deadlocks. This approach requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

Banker's algorithm is a deadlock avoidance algorithm that is applicable to a system with multiple instances of each resource type. When the user requests a set of resources, the system must determine whether the allocation of each resource will leave the system in safe state.

PROGRAM

```
#include<stdio.h>
struct process
{
int all[6],max[6],need[6],finished,request[6];
}p[10];

int avail[6],sseq[10],ss=0,check1=0,check2=0,n,pid,nor,nori,work[6];
int main()
{
        int safeseq(void);
        int ch,k,i=0,j=0,pid,ch1;
        int violationcheck=0,waitcheck=0;
        do
```

```
{
printf("\n1.Input\n2.New Request\n3.Safe State or Not\n4.Print\n5.Exit\nEnter your choice:");
scanf("%d",&ch);
switch(ch)
{
case 1: printf("\nEnter the number of processes:");
scanf("%d",&n);
printf("\nEnter the number of resources:");
scanf("%d",&nor);
printf("\nEnter the available resources:");
for(j=0;j<n;j++)
{
for(k=0;k<nor;k++)
{
if(j==0)
{
printf("\nFor Resource Type %d:",k);
scanf("%d",&avail[k]);
}
p[j].max[k]=0;
p[j].all[k]=0;
p[j].need[k]=0;
p[j].finished=0;
p[j].request[k]=0;
}
}
for(i=0;i<n;i++)
{
printf("\nEnter Max and Allocated Resources for P %d :",i);
for(j=0;j<nor;j++)
{
printf("\nEnter the Max of Resources %d:",j);
scanf("%d",&p[i].max[j]);
```

```
                printf("\nAllocation of Resources %d:",j);
                scanf("%d",&p[i].all[j]);
                if(p[i].all[j]>p[i].max[j])
                {
                        printf("\nAllocation should be less than or equal to Max\n");
                        j--;
                }
                else
                        p[i].need[j]=p[i].max[j]-p[i].all[j];
                }
}
        break;

                case 2:
        violationcheck=0;
        waitcheck=0;
        printf("\nRequesting Process ID:\n");
        scanf("%d",&pid);
        for(j=0;j<nor;j++)
        {
                printf("\nNumber of Request for Resource %d:",j);
                scanf("%d",&p[pid].request[j]);
                if(p[pid].request[j]>p[pid].need[j])
                violationcheck=1;
                if(p[pid].request[j]>avail[j])
                waitcheck=1;
        }
                if(violationcheck==1)
                printf("\nThe Process Exceeds its Max needs: Terminated\n");
        else if(waitcheck==1)
                printf("\nLack of Resources: Process State - Wait\n");
        else
        {
                for(j=0;j<nor;j++)
```

```
                        {
                        avail[j]=avail[j]-p[pid].request[j];
                        p[pid].all[j]=p[pid].all[j]+p[pid].request[j];
                        p[pid].need[j]=p[pid].need[j]-p[pid].request[j];
                        }
                        ch1=safeseq();
                        if(ch1==0)
                        {
                        for(j=0;j<nor;j++)
                        {
                                avail[j]=avail[j]+p[pid].request[j];
                                p[pid].all[j]=p[pid].all[j]-p[pid].request[j];
                                p[pid].need[j]=p[pid].need[j]+p[pid].request[j];
                        }
                        }
                        else if(ch1==1)
                        printf("\nRequest committed.\n");
                }
                break;

                        case 3:
                if(safeseq()==1)
                        printf("\nThe System is in Safe State\n");
                else
                        printf("\nThe System is not in Safe State\n");
                break;
case 4:
                printf("\nNumber of Process:%d\n",n);
                printf("\nNumber of Resources:%d\n",nor);
                printf("\nPid\tMax\tAllocated\tNeed\n");
        for(i=0;i<n;i++)
                {
                        printf(" P%d :",i);
                        for(j=0;j<nor;j++)
```

```
                        printf(" %d ",p[i].max[j]);
                        printf("\t");
                        for(j=0;j<nor;j++)
                                printf(" %d ",p[i].all[j]);
                        printf("\t");
                        for(j=0;j<nor;j++)
                        printf(" %d ",p[i].need[j]);
                        printf("\n");
                        }
        printf("\nAvailable:\n");
        for(i=0;i<nor;i++)
                        printf(" %d ",avail[i]);
        break;
                        case 5:  break;
        }
        }while(ch!=5);
        return 0;
}
int safeseq()
{
        int tj,tk,i,j,k;
        ss=0;
        for(j=0;j<nor;j++)
        work[j]=avail[j];
        for(j=0;j<n;j++)
        p[j].finished=0;
        for(tk=0;tk<nor;tk++)
        {
        for(j=0;j<n;j++)
        {
                if(p[j].finished==0)
                {
                check1=0;
                for(k=0;k<nor;k++)
```

```
                if(p[j].need[k]<=work[k])
                check1++;
                if(check1==nor)
                {
                        for(k=0;k<nor;k++)
                        {
                        work[k]=work[k]+p[j].all[k];
p[j].finished=1;
                        }
                        sseq[ss]=j;
                        ss++;
                }
                }
        }
        }
        check2=0;
        for(i=0;i<n;i++)
if(p[i].finished==1)
        check2++;
        printf("\n");
        if(check2>=n)
        {
        for(tj=0;tj<n;tj++)
                printf("p%d",sseq[tj]);
        return 1;
        }
        else
        printf("\nThe System is not in Safe State\n");
        return 0;
}
```

Explanation

1. **Struct Definition:**
   - The program defines a structure named **process** to hold information about each process, including arrays for maximum resource needs (**max**), allocated resources (**all**), remaining resource needs (**need**), finished status, and requested resources (**request**).
2. **Global Variables:**
   - **p[10]**: An array of **process** structures to represent up to 10 processes.
   - **avail[6]**: An array representing the available resources.
   - **sseq[10]**: An array to store the safe sequence.
   - **ss**: An index to keep track of the safe sequence length.
   - **check1** and **check2**: Variables to check conditions during the safe state check.
3. **Main Function:**
   - The **main** function presents a menu-driven interface for the user.
   - Options include:
     - Inputting information about the number of processes, resources, and the initial state.
     - Handling new resource requests.
     - Checking if the system is in a safe state.
     - Printing the current state of the system.
     - Exiting the program.
4. **Functions:**
   - **safeseq**: Checks if the current system state is safe and determines the safe sequence.
   - The function uses the Banker's Algorithm to find a safe sequence based on available resources and process needs.
5. **Input Handling:**
   - The program allows the user to input information about processes, maximum resource needs, and allocated resources.
   - It checks for violations such as the allocation exceeding the maximum allowed.
6. **Resource Request Handling:**
   - The program allows users to request additional resources for a specific process.

- It checks for violations such as the requested resources exceeding the process's remaining needs and the available resources.

7. **Print Function:**
   - Displays information about the processes, including their maximum needs, allocated resources, and remaining needs.
8. **Safety Check:**
   - The program checks if the system is in a safe state using the **safeseq** function.

The program makes use of Banker's Algorithm to determine whether a system is in a safe state and whether a new resource request can be granted without violating safety. The safe sequence is printed if the system is in a safe state.

Output:

1.Input
2.New Request
3.Safe State or Not
4.Print
5.Exit
Enter your choice:1

Enter the number of processes:5

Enter the number of resources:3

Enter the available resources:
For Resource Type 0:3

For Resource Type 1:3

For Resource Type 2:2

Enter Max and Allocated Resources for P 0 :
Enter the Max of Resources 0:7

Allocation of Resources 0:0

Enter the Max of Resources 1:5

Allocation of Resources 1:1

Enter the Max of Resources 2:3

Allocation of Resources 2:0

Enter Max and Allocated Resources for P 1 :
Enter the Max of Resources 0:3

Allocation of Resources 0:2

Enter the Max of Resources 1:2

Allocation of Resources 1:0

Enter the Max of Resources 2:2

Allocation of Resources 2:0

Enter Max and Allocated Resources for P 2 :
Enter the Max of Resources 0:9

Allocation of Resources 0:3

Enter the Max of Resources 1:0

Allocation of Resources 1:0

Enter the Max of Resources 2:2

Allocation of Resources 2:2

Enter Max and Allocated Resources for P 3 :
Enter the Max of Resources 0:2

Allocation of Resources 0:2

Enter the Max of Resources 1:2

Allocation of Resources 1:1

Enter the Max of Resources 2:2

Allocation of Resources 2:1

Enter Max and Allocated Resources for P 4 :
Enter the Max of Resources 0:4

Allocation of Resources 0:0

Enter the Max of Resources 1:3

Allocation of Resources 1:0

Enter the Max of Resources 2:3

Allocation of Resources 2:2

1.Input
2.New Request
3.Safe State or Not
4.Print
5.Exit
Enter your choice:3

p1p3p4p0p2
The System is in Safe State

1.Input
2.New Request
3.Safe State or Not
4.Print
5.Exit
Enter your choice:4

Number of Process:5

Number of Resources:3

Pid	Max	Allocated	Need
P0 :	7 5 3	0 1 0	7 4 3
P1 :	3 2 2	2 0 0	1 2 2
P2 :	9 0 2	3 0 2	6 0 0
P3 :	2 2 2	2 1 1	0 1 1
P4 :	4 3 3	0 0 2	4 3 1

Available:
3 3 2
1.Input
2.New Request
3.Safe State or Not
4.Print
5.Exit
Enter your choice:2

Requesting Process ID:
1

Number of Request for Resource 0:1

Number of Request for Resource 1:0

Number of Request for Resource 2:2

p1p3p4p0p2
Request committed.

1.Input
2.New Request
3.Safe State or Not
4.Print
5.Exit
Enter your choice:3

p1p3p4p0p2
The System is in Safe State

1.Input
2.New Request
3.Safe State or Not
4.Print
5.Exit
Enter your choice:4

Number of Process:5

Number of Resources:3

Pid	Max	Allocated	Need
P0 :	7 5 3	0 1 0	7 4 3
P1 :	3 2 2	3 0 2	0 2 0
P2 :	9 0 2	3 0 2	6 0 0
P3 :	2 2 2	2 1 1	0 1 1
P4 :	4 3 3	0 0 2	4 3 1

Available:
2 3 0
1.Input
2.New Request
3.Safe State or Not
4.Print
5.Exit
Enter your choice:2

Requesting Process ID:
2

Number of Request for Resource 0:3

Number of Request for Resource 1:2

Number of Request for Resource 2:2

The Process Exceeds its Max needs: Terminated

PROGRAM 6

6. Develop a C program to simulate the following contiguous memory allocation Techniques: a) Worst fit b) Best fit c) First fit

DESCRIPTION:

One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided. When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate.

Best-fit strategy chooses the block that is closest in size to the request.

First-fit chooses the first available block that is large enough.

Worst-fit chooses the largest available block.

PROGRAM

a) Worst fit

```
#include<stdio.h>
#define max 25
void main()
{
        int frag[max],b[max],f[max],i,j,nb,nf,temp,highest=0;
        static int bf[max],ff[max];
        printf("\n\tMemory Management Scheme - Worst Fit");
        printf("\nEnter the number of blocks:");
        scanf("%d",&nb);
        printf("Enter the number of files:");
        scanf("%d",&nf);
        printf("\nEnter the size of the blocks:-\n");
        for(i=1;i<=nb;i++)
```

```
        {
                printf("Block %d:",i);
                scanf("%d",&b[i]);
        }
        printf("Enter the size of the files :-\n");
        for(i=1;i<=nf;i++)
        {
                printf("File %d:",i);
                scanf("%d",&f[i]);
        }
        for(i=1;i<=nf;i++)
        {
                for(j=1;j<=nb;j++)
                {
                        if(bf[j]!=1) //if bf[j] is not allocated
                        {
                                temp=b[j]-f[i];
                                if(temp>=0)
                                if(highest<temp)
                                {
                                        ff[i]=j;
                                        highest=temp;
                                }
                        }
                }
                frag[i]=highest;
                bf[ff[i]]=1;
                highest=0;
        }
        printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement");
        for(i=1;i<=nf;i++)
        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
        //getch();
}
```

Explanation:

1. **Header and Definition:**
   - **#include<stdio.h>**: Includes the standard input/output functions.
   - **#define max 25**: Defines a macro **max** with a value of 25.
2. **Main Function:**
   - Declares arrays and variables for storing block sizes, file sizes, fragmentations, allocation details, and temporary values.
   - Initializes two static arrays, **bf** (block flags) and **ff** (file flags), to keep track of allocated blocks and files.
   - Reads the number of blocks (**nb**) and the number of files (**nf**) from the user.
   - Reads the sizes of blocks and files from the user.
   - Uses nested loops to find the worst fit for each file in the available blocks.
   - Calculates and stores the fragmentation for each file.
   - Prints the allocation details and fragmentations for each file.
3. **Explanation:**
   - The program first reads the number of blocks and files, along with their sizes, from the user.
   - It then iterates through each file and searches for the worst fit block (the largest available block) for that file.
   - After finding the worst fit block, it marks the block as allocated and calculates the fragmentation for that file.
   - The program prints the allocation details, including file number, file size, allocated block number, block size, and fragmentation.

OUTPUT

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

Enter the size of the blocks:-
Block 1:100
Block 2:500
Block 3:200
Block 4:300
Block 5:600
Enter the size of the files :-
File 1:212
```

File 2:417
File 3:112
File 4:426

File_no:	File_size :	Block_no:	Block_size:	Fragement
1	212	5	600	388
2	417	2	500	83
3	112	4	300	188
4	426	0	0	0

b) Best fit

```
#include<stdio.h>
#define max 25
void main()
{
	int frag[max],b[max],f[max],i,j,nb,nf,temp,lowest=10000;
	static int bf[max],ff[max];
	printf("\nEnter the number of blocks:");
	scanf("%d",&nb);
	printf("Enter the number of files:");
	scanf("%d",&nf);
	printf("\nEnter the size of the blocks:-\n");
	for(i=1;i<=nb;i++)
	{
		printf("Block %d:",i);
		scanf("%d",&b[i]);
	}
	printf("Enter the size of the files :-\n");
	for(i=1;i<=nf;i++)
	{
		printf("File %d:",i);
		scanf("%d",&f[i]);
	}
		for(i=1;i<=nf;i++)
		{
			for(j=1;j<=nb;j++)
			{
```

```
                    if(bf[j]!=1)
                    {
                        temp=b[j]-f[i];
                        if(temp>=0)
                        if(lowest>temp)
                        {
                            ff[i]=j;
                            lowest=temp;
                        }
                    }
                }
                frag[i]=lowest;
                bf[ff[i]]=1;
                lowest=10000;
            }
            printf("\nFile No\tFile Size \tBlock No\tBlock Size\tFragment");
            for(i=1;i<=nf && ff[i]!=0;i++)
            printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);

}
```

Explanation:

1. **Header and Definition:**
   - **#include<stdio.h>**: Includes the standard input/output functions.
   - **#define max 25**: Defines a macro **max** with a value of 25.
2. **Main Function:**
   - Declares arrays and variables for storing block sizes, file sizes, fragmentations, allocation details, and temporary values.
   - Initializes two static arrays, **bf** (block flags) and **ff** (file flags), to keep track of allocated blocks and files.
   - Reads the number of blocks (**nb**) and the number of files (**nf**) from the user.
   - Reads the sizes of blocks and files from the user.
   - Uses nested loops to find the best fit for each file in the available blocks.

- Calculates and stores the fragmentation for each file.
- Prints the allocation details and fragmentations for each file.

3. **Working:**

- The program first reads the number of blocks and files, along with their sizes, from the user.
- It then iterates through each file and searches for the best fit block (the smallest available block) for that file.
- After finding the best fit block, it marks the block as allocated and calculates the fragmentation for that file.
- The program prints the allocation details, including file number, file size, allocated block number, block size, and fragmentation.

OUTPUT

```
Enter the number of blocks:5
Enter the number of files:4

Enter the size of the blocks:-
Block 1:100
Block 2:500
Block 3:200
Block 4:300
Block 5:600
Enter the size of the files :-
File 1:212
File 2:417
File 3:112
File 4:426
```

File No	File Size	Block No	Block Size	Fragment
1	212	4	300	88
2	417	2	500	83
3	112	3	200	88
4	426	5	600	174

c) First Fit

```
#include<stdio.h>

void main()

{

        int bsize[10], psize[10], bno, pno, flags[10], allocation[10], i, j;

        for(i = 0; i < 10; i++)

        {
```

```
            flags[i] = 0;
            allocation[i] = -1;
    }
    printf("Enter no. of blocks: ");
    scanf("%d", &bno);
    printf("Enter no. of processes: ");
    scanf("%d", &pno);

    printf("Enter size of each block:\n");
    for(i = 0; i < bno; i++)
    {
            printf("Block %d:",i);
            scanf("%d", &bsize[i]);
    }
    printf("\nEnter size of each process:\n");
    for(i = 0; i < pno; i++)
    {
            printf("Process %d:",i);
            scanf("%d", &psize[i]);
    }
    for(j = 0; j < bno; j++)        //allocation as per first fit
            for(i = 0; i < pno; i++)
                    if((flags[i] == 0) && (bsize[j] >= psize[i]))
                    {
                            allocation[i] = j;
                            flags[i] = 1;
                            break;
                    }
    //display allocation details
    printf("\nFile No\t\tFile size\tBlock no.\tBlock size\tFragment");
    for(i = 0; i < pno; i++)
    {
            printf("\n%d\t\t%d\t\t", i+1, psize[i]);
            if(flags[i] == 1)
```

```
                printf("%d\t\t%d\t\t%d",allocation[i]+1,bsize[allocation[i]],
                (bsize[allocation[i]]-psize[i]));
            else
                printf("0\t\t0\t\t0");
        }
}
```

Explanation:

1. **Main Function:**
    - Declares arrays and variables for storing block sizes, process sizes, flags, and allocation details.
    - Initializes flags to keep track of allocated processes and the allocation array to store block numbers for each process.
    - Reads the number of blocks (**bno**) and the number of processes (**pno**) from the user.
    - Reads the sizes of blocks and processes from the user.
    - Uses nested loops to perform the first-fit allocation. For each process, it checks the available blocks and allocates the first block that is large enough.
    - Prints the allocation details for each process, including file number, file size, allocated block number, block size, and fragmentation.
2. **Working:**
    - The program iterates through each process and, for each process, iterates through each block to find the first available block that can accommodate the process.
    - It updates the flags and allocation arrays to mark the process as allocated and store the block number for that allocation.
    - The program then prints the allocation details for each process, including the file number, file size, allocated block number, block size, and fragmentation.

OUTPUT

```
Enter no. of blocks: 5
Enter no. of processes: 4
Enter size of each block:
Block 0:100
Block 1:500
Block 2:200
Block 3:300
Block 4:600
```

Enter size of each process:
Process 0:212
Process 1:417
Process 2:112
Process 3:426

File No	File size	Block no.	Block size	Fragment
1	212	2	500	288
2	417	5	600	183
3	112	3	200	88
4	426	0	0	0

PROGRAM 7

7. Develop a C program to simulate page replacement algorithms: a) FIFO b) LRU

DESCRIPTION:

Page Replacement Algorithms: Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. If the recent past is used as an approximation of the near future, then the page that has not been used for the longest period of time can be replaced. This approach is the Least Recently Used (LRU) algorithm. LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. Least frequently used (LFU) page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count.

PROGRAM

a. FIFO

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

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

1. **Main Function:**
   - Declares variables for the number of pages (**n**), page array (**a**), frame array (**frame**), the number of frames (**no**), and auxiliary variables (**i**, **j**, **k**, **avail**, **count**).
   - Reads the number of pages and the page numbers from the user.
   - Initializes the frame array with -1 to represent empty frames.
   - Initializes the index **j** to track the next frame to be replaced.
   - Prints the reference string and the page frames after each page access.
   - Counts and prints the number of page faults.
2. **Explanation:**
   - The program simulates the FIFO page replacement algorithm, where the oldest page in the frames is replaced when a page fault occurs.
   - It uses a circular buffer (**frame**) to represent the frames, and the variable **j** keeps track of the next frame to be replaced.

- The program iterates through the page array and checks if the page is already present in the frames.
- If the page is not present, it increments the page fault count, replaces the oldest page in the frames, and updates the frame array.
- The program prints the reference string and the current state of the page frames after each page access.

Output:

```
 ENTER THE NUMBER OF PAGES:
8

 ENTER THE PAGE NUMBER :
2
4
5
4
6
7
1
5

 ENTER THE NUMBER OF FRAMES :3
        ref string       page frames
2               2       -1      -1
4               2       4       -1
5               2       4       5
4
6               6       4       5
7               6       7       5
1               6       7       1
5               5       7       1
Page Fault Is 7
Process returned 0 (0x0)   execution time : 34.878 s
Press any key to continue.
```

b. LRU

```
#include<stdio.h>
int i,j,nof,nor,flag=0,ref[50],frm[50],pf=0,victim=-1;
int recent[10],lrucal[50],count=0;
int lruvictim();
void main()
{
        printf("\n\t\t\t LRU PAGE REPLACEMENT ALGORITHM");
        printf("\n Enter no.of Frames....");
        scanf("%d",&nof);
        printf(" Enter no.of reference string..");
        scanf("%d",&nor);
        printf("\n Enter reference string..");
```

```
for(i=0;i<nor;i++)
scanf("%d",&ref[i]);
printf("\n\n\t\t LRU PAGE REPLACEMENT ALGORITHM ");
printf("\n\t The given reference string:");
printf("\n……..");
for(i=0;i<nor;i++)
printf("%4d",ref[i]);
for(i=1;i<=nof;i++)
{
        frm[i]=-1;
        lrucal[i]=0;
}
for(i=0;i<10;i++)
recent[i]=0;
printf("\n");
for(i=0;i<nor;i++)
{
        flag=0;
        printf("\n\t Reference NO %d->\t",ref[i]);
        for(j=0;j<nof;j++)
        {
                if(frm[j]==ref[i])
                {
                        flag=1;
                        break;
                }
        }
        if(flag==0)
        {
                count++;
                if(count<=nof)
                victim++;
                else
                victim=lruvictim();
                pf++;
```

```
                        frm[victim]=ref[i];
                        for(j=0;j<nof;j++)
                        printf("%4d",frm[j]);
                }
                recent[ref[i]]=i;
        }
        printf("\n\n\t No.of page faults...%d",pf);
}
int lruvictim()
{
        int i,j,temp1,temp2;
        for(i=0;i<nof;i++)
        {
                temp1=frm[i];
                lrucal[i]=recent[temp1];
        }
        temp2=lrucal[0];
        for(j=1;j<nof;j++)
        {
                if(temp2>lrucal[j])
                temp2=lrucal[j];
        }
        for(i=0;i<nof;i++)
        if(ref[temp2]==frm[i])
        return i;
        return 0;
}
```

1. **Main Function:**
    - Declares variables for the number of frames (**nof**), number of references (**nor**), an array to store the reference string (**ref**), and arrays to represent the frames (**frm**) and the LRU calculation array (**lrucal**).
    - Initializes arrays and variables.
    - Reads the number of frames and the reference string from the user.
    - Initializes the frames and LRU calculation arrays.

- Calls the **lruvictim** function to find the victim for replacement based on the LRU algorithm.
- Prints the reference string and performs page replacement using the LRU algorithm.
- Prints the number of page faults.

2. **lruvictim Function:**

- Takes no arguments and returns the index of the victim frame for replacement.
- Calculates the LRU values for each frame based on the recent array.
- Finds the frame with the minimum LRU value and returns its index.

3. **Explanation:**

- The program simulates the LRU page replacement algorithm by maintaining an array (**recent**) to store the recent usage of each page.
- The **lruvictim** function calculates the LRU values for each frame and returns the index of the frame with the minimum LRU value.
- The main loop iterates through the reference string and checks if the page is present in the frames.
- If the page is not present, it selects a victim frame for replacement using the LRU algorithm and increments the page fault count.
- The frames are updated, and the reference string and frame status are printed after each page replacement.

OUTPUT LRU:

```
                          LRU PAGE REPLACEMENT ALGORITHM
 Enter no.of Frames....4
 Enter no.of reference string..20

 Enter reference string..7
0
1
2
0
3
0
4
2
3
0
3
2
1
2
0
1
7
0
1
```

```
                LRU PAGE REPLACEMENT ALGORITHM
        The given reference string:
àààààààààààà..   7   0   1   2   0   3   0   4   2   3   0   3   2   1   2   0   1   7   0   1
         Reference NO 7->          7  -1  -1  -1
         Reference NO 0->          7   0  -1  -1
         Reference NO 1->          7   0   1  -1
         Reference NO 2->          7   0   1   2
         Reference NO 0->
         Reference NO 3->          3   0   1   2
         Reference NO 0->
         Reference NO 4->          3   0   4   2
         Reference NO 2->
         Reference NO 3->
         Reference NO 0->
         Reference NO 3->
         Reference NO 2->
         Reference NO 1->          3   0   1   2
         Reference NO 2->
         Reference NO 0->
         Reference NO 1->
         Reference NO 7->          7   0   1   2
         Reference NO 0->
         Reference NO 1->

        No.of page faults...8
Process returned 25 (0x19)   execution time : 24.982 s
Press any key to continue.
```

PROGRAM 8

8. Simulate following File Organization Techniques a) Single level directory b) Two level directory

DESCRIPTION:

a) SINGLE LEVEL DIRECTORY:

The directory structure is the organization of files into a hierarchy of folders. In a single-level directory system, all the files are placed in one directory. There is a root directory which has all files. It has a simple architecture and there are no sub directories. Advantage of single level directory system is that it is easy to find a file in the directory.

PROGRAM

```
#include<stdlib.h>
#include<string.h>
#include<stdio.h>
struct
{
    char dname[10],fname[10][10];
    int fcnt;
}dir;
void main()
{
    int i,ch;
    char f[30];
    dir.fcnt = 0;
    printf("\nEnter name of directory -- ");
    scanf("%s", dir.dname);
    while(1)
    {
        printf("\n\n1. Create File\t2. Delete File\t3. Search File \n 4. Display Files\t5.
        Exit\nEnter your choice -- ");
        scanf("%d",&ch);
        switch(ch)
        {
```

```
case 1: printf("\nEnter the name of the file -- ");
scanf("%s",dir.fname[dir.fcnt]);
dir.fcnt++;
break;
case 2: printf("\nEnter the name of the file -- ");
scanf("%s",f);
for(i=0;i<dir.fcnt;i++)
{
        if(strcmp(f, dir.fname[i])==0)
        {
                printf("File %s is deleted ",f);
                strcpy(dir.fname[i],dir.fname[dir.fcnt-1]); break; } }
                if(i==dir.fcnt) printf("File %s not found",f);
                else
                dir.fcnt--;
                break;
                case 3: printf("\nEnter the name of the file -- ");
                scanf("%s",f);
                for(i=0;i<dir.fcnt;i++)
                {
                        if(strcmp(f, dir.fname[i])==0)
                        {
                                printf("File %s is found ", f);
                                break;
                        }
                }
                if(i==dir.fcnt)
                printf("File %s not found",f);
                break;
                case 4: if(dir.fcnt==0)
                printf("\nDirectory Empty");
                else
                {
                        printf("\nThe Files are -- ");
```

```
                    for(i=0;i<dir.fcnt;i++)
                    printf("\t%s",dir.fname[i]);
                }
                break;
                default: exit(0);
            }
        }
    }
```

Explanation:

1. **Directory Structure:**
   - The program defines a structure named **dir** with fields:
     - **dname**: Represents the directory name.
     - **fname**: Represents an array of file names.
     - **fcnt**: Represents the count of files in the directory.
2. **Main Function:**
   - Declares variables for user input (**ch** for choice, **f** for file name).
   - Initializes the file count (**fcnt**) to 0.
   - Asks the user to enter the name of the directory.
   - Uses a **while** loop for the main menu until the user chooses to exit.
   - The main menu includes options for creating, deleting, searching, displaying files, and exiting.
3. **Menu Options:**
   - **Create File (case 1):**
     - Asks the user to enter the name of the file and adds it to the directory.
   - **Delete File (case 2):**
     - Asks the user to enter the name of the file to be deleted.
     - Searches for the file in the directory and deletes it if found.
   - **Search File (case 3):**
     - Asks the user to enter the name of the file to be searched.
     - Searches for the file in the directory and prints a message indicating whether it is found or not.
   - **Display Files (case 4):**

- Displays the names of files in the directory.

- **Exit (case 5):**
  - Exits the program.

OUTPUT:

```
Enter name of directory -- aa


1. Create File  2. Delete File  3. Search File
 4. Display Files       5. Exit
Enter your choice -- 1

Enter the name of the file -- ds


1. Create File  2. Delete File  3. Search File
 4. Display Files       5. Exit
Enter your choice -- 3

Enter the name of the file -- ds
File ds is found

1. Create File  2. Delete File  3. Search File
 4. Display Files       5. Exit
Enter your choice -- 4

The Files are --        ds

1. Create File  2. Delete File  3. Search File
 4. Display Files       5. Exit
Enter your choice -- 2

Enter the name of the file -- ds
File ds is deleted
```

b) TWO LEVEL DIRECTORY

In the two-level directory system, each user has own user file directory (UFD). The system maintains a master block that has one entry for each user. This master block contains the addresses of the directory of the users. When a user job starts or a user logs in, the system's master file directory (MFD) is searched. When a user refers to a particular file, only his own UFD is searched.

PROGRAM:

```
#include<stdio.h>

struct st
{
    char dname[10];
    char sdname[10][10];
    char fname[10][10][10];
    int ds,sds[10];
}dir[10];
void main()
{
    int i,j,k,n;
    char name[10];
    printf("enter number of users:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
            printf("enter user directory %d names:",i+1);
            scanf("%s",&dir[i].dname);
            printf("enter size of directories:");
            scanf("%d",&dir[i].ds);
            for(j=0;j<dir[i].ds;j++)
            {
                    printf("enter subdirectory name and size:");
                    scanf("%s",&dir[i].sdname[j]);
                    scanf("%d",&dir[i].sds[j]);
                    for(k=0;k<dir[i].sds[j];k++)
                    {
                            printf("enter file name:");
                            scanf("%s",&dir[i].fname[j][k]);
                    }
            }
```

```
    }
    printf("\ndirname\t\tsize\tsubdirname\tsize\tfiles");
    printf("\n*************************************************\n");
    for(i=0;i<n;i++)
    {
        printf("%s\t\t%d",dir[i].dname,dir[i].ds);
        for(j=0;j<dir[i].ds;j++)
        {
            printf("\t%s\t\t%d\t",dir[i].sdname[j],dir[i].sds[j]);
            for(k=0;k<dir[i].sds[j];k++)
            printf("%s\t",dir[i].fname[j][k]);
            printf("\n\t\t");
        }
            printf("\n");
    }

}
```

Explanation:

1. **Directory Structure:**
   - The program defines a structure named **st** to represent the directory structure.
   - Fields of the **st** structure:
     - **dname**: Directory name.
     - **sdname**: Array to store subdirectory names.
     - **fname**: Two-dimensional array to store file names.
     - **ds**: Size of the directory (number of subdirectories).
     - **sds**: Array to store the sizes of subdirectories.
2. **Main Function:**
   - Declares variables for user input (**i**, **j**, **k**, **n**, **name**).
   - Declares an array of structures (**dir**) to store information for multiple users.
   - Asks the user to enter the number of users (**n**).

- Uses nested loops to collect information about directories, subdirectories, and files for each user.

3. **User Input Loop:**
   - For each user, the program prompts for the user's directory name, the size of directories, subdirectory names, and the number of files in each subdirectory.
4. **Displaying Directory Structure:**
   - After collecting user input, the program prints the directory structure in tabular form.
   - Displays information such as **dirname**, **size**, **subdirname**, **size**, and **files**.

OUTPUT:

```
enter number of users:2
enter user directory 1 names:DS
enter size of directories:2
enter subdirectory name and size:sem1
2
enter file name:f1
enter file name:f2
enter subdirectory name and size:sem3
3
enter file name:v1
enter file name:v2
enter file name:v3
enter user directory 2 names:IOT
enter size of directories:3
enter subdirectory name and size:AIML
2
enter file name:a1
enter file name:a2
enter subdirectory name and size:VLSI
1
enter file name:B1
enter subdirectory name and size:ISE
2
enter file name:K1
enter file name:K2

dirname         size    subdirname      size    files
**************************************************
DS              2       sem1            2       f1      f2
                        sem3            3       v1      v2      v3

IOT             3       AIML            2       a1      a2
                        VLSI            1       B1
                        ISE             2       K1      K2


Process returned 2 (0x2)   execution time : 77.500 s
Press any key to continue.
```

PROGRAM 9

9. Develop a C program to simulate the Linked file allocation strategies

DESCRIPTION:

In the chained method file allocation table contains a field which points to starting block of memory. From it for each bloc a pointer is kept to next successive block. Hence, there is no external fragmentation

PROGRAM:

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

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

Explanation:

1. **Variables:**
   - **f[50]**: Array to represent the status of blocks (0 for unallocated, 1 for allocated).
   - **p**: Number of blocks already allocated.
   - **i, st, len, j, c, k, a**: Loop control and temporary variables.
2. **Allocation Loop:**
   - The program prompts the user to enter the number of blocks already allocated (**p**) and the blocks that are already allocated.
   - It initializes the **f** array based on the already allocated blocks.
3. **File Allocation (First Fit):**

- The program uses a loop labeled as **x** to repeatedly ask the user for the starting block (**st**) and length (**len**) of the file.
- For each file, it checks if the starting block is unallocated. If yes, it allocates consecutive blocks starting from the specified block.
- If any block in the range is already allocated, it increments the length **k** to find the next available block.
- It prints the allocated blocks for each file.

4. **User Interaction:**
   - After allocating blocks for a file, the program asks the user if they want to enter more files.
   - If the user enters 1 (Yes), it goes back to the label **x** and repeats the process.
   - If the user enters 0 (No), the program exits.

OUTPUT

```
Enter how many blocks already allocated: 6
Enter blocks already allocated: 5
8
2
4
9
12
Enter index starting block and length: 3
8
3-------->1
4 Block is already allocated
5 Block is already allocated
6-------->1
7-------->1
8 Block is already allocated
9 Block is already allocated
10-------->1
11-------->1
12 Block is already allocated
13-------->1
14-------->1
15-------->1
Do you want to enter more file(Yes - 1/No - 0)1
Enter index starting block and length: 6
4
6 starting block is already allocated
Do you want to enter more file(Yes - 1/No - 0) 0
```

PROGRAM 10

10. Develop a C program to simulate SCAN disk scheduling algorithm.

DESCRIPTION:

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

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;
   printf("Enter the number of Requests\n");
   scanf("%d",&n);
   printf("Enter the Requests sequence\n");
   for(i=0;i<n;i++)
   scanf("%d",&RQ[i]);
   printf("Enter initial head position\n");
   scanf("%d",&initial);
   printf("Enter total disk size\n");
   scanf("%d",&size);
   printf("Enter the head movement direction for high 1 and for low 0\n");
   scanf("%d",&move);

   // logic for Scan disk scheduling

      /*logic for sort the request array */
   for(i=0;i<n;i++)
   {
```

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

    }
}

int index;
for(i=0;i<n;i++)
{
   if(initial<RQ[i])
   {
      index=i;
      break;
   }
}

// if movement is towards high value
if(move==1)
{
   for(i=index;i<n;i++)
   {
      TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
      initial=RQ[i];
   }
   //  last movement for max size
   TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);
   initial = size-1;
```

```
        for(i=index-1;i>=0;i--)
        {
            TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
            initial=RQ[i];

        }
    }
    // if movement is towards low value
    else
    {
        for(i=index-1;i>=0;i--)
        {
            TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
            initial=RQ[i];
        }
        //  last movement for min size
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);
        initial =0;
        for(i=index;i<n;i++)
        {
            TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
            initial=RQ[i];

        }
    }

    printf("Total head movement is %d",TotalHeadMoment);
    return 0;
}
```

OUTPUT

```
Enter the number of Requests
7
Enter the Requests sequence
82
170
43
140
24
16
190
Enter initial head position
50
Enter total disk size
200
Enter the head movement direction for high 1 and for low 0
1
Total head movement is 332
Process returned 0 (0x0)   execution time : 29.118 s
Press any key to continue.
```