RAGHU ENGINEERING COLLEGE

(Autonomous)

(Approved by AICTE, New Delhi, Permanently Affiliated to JNTU Kakinada,Accredited by NBA & Accredited by NAAC with A grade)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING-AI&ML

2022-23

Academic Year

B. Tech. II Year - I Semester (AR20)

STAFF LABORATORY MANUAL

for

OPERATING SYSTEMS LAB (20CS3106)

Prepared by

K L Ganapathi Reddy, Assistant Professor, CSE

Vision of the Institute

Envisioning to be a world class technical institution by synergizing quality education with ethicalvalues.

Mission of the Institute:

To enlist the services of expert faculty.

To encourage training and research in cutting-edge technologiesTo develop and strengthen strategic links with the industry

To kindle the zeal among the students and promote their quest for academic excellenceTo encourage extra-curricular activities along with good communication skills

Vision of the Department:

To generate competent professionals to become part of the industry and research organizations atthe national and international levels.

Mission of the Department:

M1: To impart high quality professional training in undergraduate level with emphasis on basic principles of Computer Science and Engineering and to foster leading edge research in the fast changing field

M2: To inculcate professional behaviour, strong ethical values values, innovative research capabilities and leadership abilities in the young minds so as to work with a commitment.

Program Educational Objectives (PEOs):

PEO No.	Program Educational Objectives Statements
PEO1	To produce graduates who have strong foundation in mathematics, science, engineering fundamentals, laboratory and work-based experiences to formulate and solve engineering problems in computer science engineering domains and shall have proficiency in implementation software tools and languages
PEO2	To progressively impart training to the students for success in various engineering positions within the core areas in computer science engineering, computational or adapting themselves to latest trends by learning themselves
PEO3	To produce graduates having the ability to pursue advanced higher studies and research. To have professional and communication skills to function as leaders and members of multi-disciplinary teams in engineering and other industries with strong work ethics, organizational skills, team work and understand the importance of being athoroughprofessional

PROGRAM OUTCOMES(POs)

Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

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.

Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate considerationfor the public health and safety, and the cultural, societal, and environmental considerations.

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.

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

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.

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.

Ethics: Apply ethical principles and commit to professional ethics and responsibilitiesand normsof the engineering practice.

Individual and team work: Function effectively as an individual, and as a member orleader indiverse teams, and in multidisciplinary settings.

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

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 leaderin a team, to manage projects and in multidisciplinary environments.

Life-long learning: Recognize the need for, and have the preparation and ability to engage inindependent and life-long learning in the broadest context of technologicalchange.

PROGRAM SPECIFIC OUTCOMES (PSOS)

PSO1:

Apply the concepts and techniques of the Computer Science & Engineering and the Mathematical foundations in the significant domains to address the complex engineering problems.

PSO2:

Employ emerging computer languages, computer networks, database management systems and platforms in developing innovative career prospects as an entrepreneur.

PSO3:

Analyze and demonstrate the knowledge of human cognition, Artificial Intelligence, Machine Learning, Deep Learning algorithms and project development skills using innovative tools and techniques to solve problems and meet future challenges

OPERATING SYSTEMS
LABAR20 - B. Tech. (CSE)
II - B. Tech., II-Semester

Course Code: 20CS4106 | **Internal Exams: 15External Exams: 35**

Course Objectives:

The course objectives of Operating Systems Lab are to discuss and make student familiar withthe

To provide an understanding of the design aspects of operating system concepts through simulation.

To know the various CPU Scheduling Algorithms and page replacement algorithms.

Introduce system call interface for process management, inter-process communication.

Course Outcomes:

By the end of the course, the student will:

Compare the performance of various CPU Scheduling Algorithms, and also implement Deadlock avoidance and Detection Algorithms.

Implement Semaphores, and also create processes and implement IPC.

Analyze the performance of the various Page Replacement Algorithms, and also implement File Organization and File Allocation Strategies.

LAB EXPERIMENTS

1. Multiprogramming-Memory management- Implementation of Fork(), Wait(), Exec() and Exit() System calls.
2. Write C programs to illustrate the IPC mechanisms using Pipes.
3. Write C programs to illustrate the following IPC mechanisms
   a) Message Passing.
   b) Shared Memory.
4. Simulate the following CPU scheduling algorithms
   a) FCFS b) SJF.
5. Simulate the following CPU scheduling algorithms
   a) Round Robin b) Priority.
6. Write a C program to implementation of Semaphores.
7. Simulate Bankers Algorithm for Dead Lock Avoidance.
8. Simulate Bankers Algorithm for Dead Lock Prevention.
9. Simulate MVT and MFT.
10. Implementation of the following Memory Allocation Methods for fixed partition
    a) First Fit b) Worst Fit c) Best Fit
11. Simulate all page replacement algorithms.
    a) FIFO b) LRU c) Optimal (LFU).
12. Simulate all File allocation strategies a) Sequenced b) Indexed c) Linked.

INDEX

Laboratory – CO mapping

Course Outcomes:

S.No	Course Outcomes:	BTL
1	Compare the performance of various CPU Scheduling Algorithms	L3
2	Implement Semaphores	L3
3	Analyze the performance of the various Page Replacement Algorithms	L4

Correlation of COs with POs & PSOs:

CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	2	2	1	1	-	-	-	-	-	-	-	1	1	**2**	**1**
CO2	1	1	1	2	-	-	-	-	-	-	-	-	1	**2**	**1**
CO3	1	1	1	1	-	-	-	-	-	-	-	1	1	**2**	**1**

WEEK-1

AIM: Implementation of Fork(),Wait(), Exec() and Exit() System calls

What is a Fork()?

In the computing field, **fork()** is the primary method of process creation on Unix-like operating systems. This function creates a new copy called the *child* out of the original process, that is called the *parent*. When the parent process closes or crashes for some reason, it also kills the child process.

The operating system is using a unique id for every process to keep track of all processes. And for that, fork() doesn't take any parameter and return an int value as following:

- Zero: if it is the child process (the process created).
- Positive value: if it is the parent process.
- Negative value: if an error occurred.

Program:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
   /* fork a process */
   fork();
   /* the child and parent will execute every line of code after the fork (each separately)*/
   printf("Hello world!\n");
   return 0;
}
```

The output will be:

```
Hello world!
Hello world!
```

Simply, we can tell that the result is 2 power of n, where n is the number of fork() system calls.

For example:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
   fork();
   fork();
   fork();
   printf("Hello world!\n");
   return 0;
}
```

The result is:

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

When a process creates a new process, then there are two possibilities for the execution exit:

- The parent continues to execute concurrently with its child.
- The parent waits until some or all of its children have terminated.

FORK():

Fork system call use for creates a new process, which is called ***child process***, which runs concurrently with process (which process called system call fork) and this process is called ***parent process***. After a new child process created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.

EXEC():

The exec family of functions replaces the current running process with a new process. It can be used to run a C programby using another C program. It comes under the header file **unistd.h.** There are many members in the exec family which are shown below with examples.

- **execvp** : Using this command, the created child process does not have to run the same program as the parentprocess does. The **exec** type system calls allow a process to run any program files, which include a binary executable or a shell script .

Syntax:

```
int execvp (const char *file, char *const argv[]);
```

file: points to the file name associated with the file being executed.

argv: is a null terminated array of character pointers.

- **execv** : This is very similar to execvp() function in terms of syntax as well. The syntax of **execv()** is as shownbelow:

Syntax:

```
int execv(const char *path, char *const argv[]);
```

path: should point to the path of the file being executed.

argv[]: is a null terminated array of character pointers.

execlp and execl : These two also serve the same purpose but the syntax of of them are a bit different which isas shown below:**Syntax:**

- **execvpe and execle** : These two also serve the same purpose but the syntax of them are a bit different from all the above members of exec family. The synatxes of both of them are shown below :

```
int execlp(const char *file, const char *arg,.../* (char  *) NULL */);


int execl(const char *path, const char *arg,.../* (char  *) NULL */);
```

Syntax:

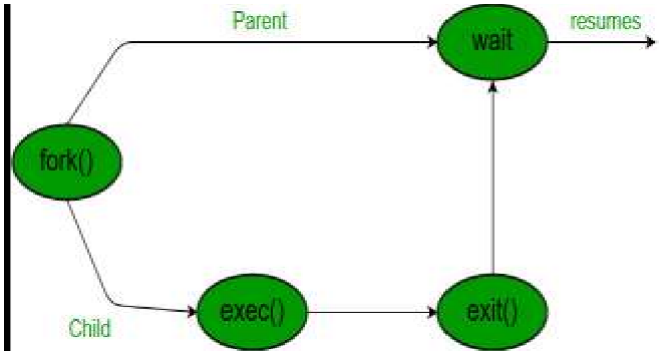
WAIT() :

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent ***continues*** its execution after wait system call instruction.
Child process may terminate due to any of these:

- It calls exit();
- It returns (an int) from main
- It receives a signal (from the OS or another process) whose default action is to terminate.

EXIT():

It terminates the calling process without executing the rest code which is after the exit() function.



Program:

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

```
#include <stdlib.h>
int main()
{
int pid;   //process id
pid = fork();     //create another process
if( pid< 0 )
{  //fail
printf("\nFork failed\n");
exit (-1);
}
else if( pid == 0 )
{  //child
execlp("/bin/ls","ls","-l",NULL);  //execute ls
}
else
{  //parent
wait (NULL);   //wait for child
printf("\nchild complete\n");
exit (0);

}
}
```

Output:

```
pandu@pandu-desktop:~/Desktop/os lab$ cc week1.c
pandu@pandu-desktop:~/Desktop/os lab$ ./a.out
total 2044
-rwxrwxr-x 1 pandu pandu    8472 Oct 20 20:27  a.out
-rw-r--r-- 1 pandu pandu   78788 Oct 20 20:27  exe1.docx
-rw-r--r-- 1 pandu pandu   14995 Sep 28 20:49  exe2.docx
-rw-r--r-- 1 pandu pandu   70437 Oct 20 20:23  exe3.docx
```

WEEK-2

AIM: To Ilustrate the IPC mechanisms using Pipes

Pipes are used for communication between processes. The named pipes are fifos. They enable two way communication unlike ordinary pipes. But they are half duplex, i.e. communication can take place only in one direction at a time.

Program1:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main(int argc,char *argv[])
{
int fd[2],pid,k;
k=pipe(fd);
if(k==-1)
{
perror("pipe");
exit(1);
}
pid=fork();
if(pid==0)
{
close(fd[0]);
dup2(fd[1],1);
close(fd[1]);
execlp(argv[1],argv[1],NULL);
perror("execl");
}
else
{
close(fd[1]);
dup2(fd[0],0);
close(fd[0]);
execlp(argv[2],argv[2],NULL);
perror("execl");
}
}
```

Output:

```
pandu@pandu-desktop:~/Desktop/os lab$ cc week2.c
pandu@pandu-desktop:~/Desktop/os lab$ ./a.out ls sort
a.out
exe1.docx
exe2.docx

exe3.docx
week1.c
week2a.c
week2.c
```

week3msgqueueread.c
week3msgqueuewrite.c
week3sharedmemory.c

Program2:

Algorithm
Step 1 − Create a pipe.
Step 2 − Send a message to the pipe.
Step 3 − Retrieve the message from the pipe and write it to the standard output.
Step 4 − Send another message to the pipe.
Step 5 − Retrieve the message from the pipe and write it to the standard output.

```
#include<stdio.h>
#include<unistd.h>
int main() {
int pipefds[2];
int returnstatus;
char writemessages[2][20]={"Hi", "Hello"};
char readmessage[20];
returnstatus = pipe(pipefds);
if (returnstatus == -1) {
printf("Unable to create pipe\n");
return 1;
}
printf("Writing to pipe - Message 1 is %s\n",writemessages[0]);
write(pipefds[1], writemessages[0], sizeof(writemessages[0]));
read(pipefds[0], readmessage, sizeof(readmessage));
printf("Reading from pipe – Message 1 is %s\n", readmessage);
printf("Writing to pipe - Message 2 is %s\n",writemessages[1]);
write(pipefds[1], writemessages[1], sizeof(writemessages[0]));
read(pipefds[0], readmessage, sizeof(readmessage));
printf("Reading from pipe – Message 2 is %s\n", readmessage);
return 0;
}
```

output:

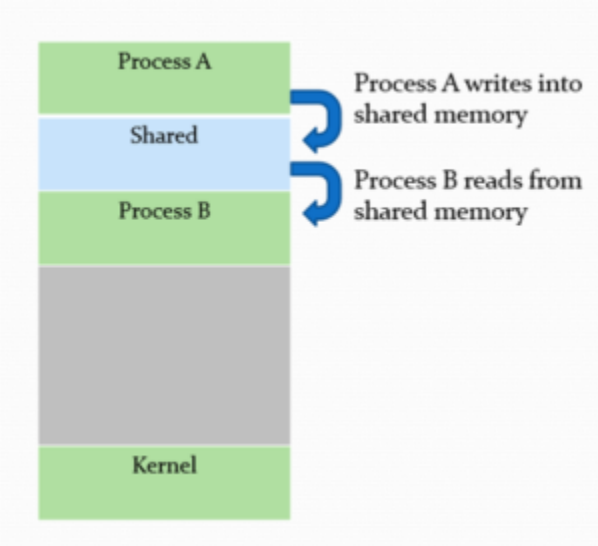
```
pandu@pandu-desktop:~/Desktop/os lab$ cc week2a.c
pandu@pandu-desktop:~/Desktop/os lab$ ./a.out
```

Writing to pipe - Message 1 is Hi
Reading from pipe – Message 1 is Hi
Writing to pipe - Message 2 is Hello
Reading from pipe – Message 2 is Hello

WEEK-3

AIM: To Ilustrate the following IPC mechanisms a) Message Passing. b) Shared Memory.

Shared Memory is the fastest inter-process communication (IPC) method. The operating system maps a memory segment in the address space of several processes so that those processes can read and write in thatmemorysegment.
Two functions:shmget() and shmat() are used for IPC using shared memory. shmget() function is used to create the shared memory segment while shmat() function is used to attach the shared segment with the address space of the process.



IPC through shared memory
Algorithm:

- Server reads from the input file.
- The server writes this data in a message using either a pipe, fifo or message queue.
- The client reads the data from the IPC channel,again requiring the data to be copied from kernel's IPC buffer to the client's buffer. Finally the data is copied from the client's buffer.
- A total of four copies of data are required (2 read and 2 write). So, shared memory provides a way by letting two or more processes share a memory segment. With
- Shared Memory the data is only copied twice – from input file into shared memory and from shared memory to the output file.

SYSTEM CALLS USED ARE:

ftok(): is use to generate a unique key.
shmget(): int shmget(key_t,size_tsize,intshmflg); upon successful completion, shmget() returns an identifier for the shared memory segment.
shmat(): Before you can use a shared memory segment, you have to attach yourself to it using shmat().
void *shmat(int shmid ,void *shmaddr ,int shmflg);
shmid is shared memory id. shmaddr specifies specific address to use but we should set it to zero and OS will automatically choose the address.

Program1:

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

```
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666|IPC_CREAT); //creates shared memory segment with key
2345, having size 1024 bytes. IPC_CREAT is used to create the shared segment if it does not exist.
0666 are the permisions on the shared segment
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0); //process attached to shared memory segment
printf("Process attached at %p\n",shared_memory); //this prints the address where the segment is
attached with this process
printf("Enter some data to write to shared memory\n");
read(0,buff,100); //get some input from user
strcpy(shared_memory,buff); //data written to shared memory
printf("You wrote : %s\n",(char *)shared_memory);
}
```

Output

```
pandu@pandu-desktop:~/Desktop/os lab$ cc week3sharedmemory.c
pandu@pandu-desktop:~/Desktop/os lab$ ./a.out
Key of shared memory is 32803
Process attached at 0x7f22bda7e000
Enter some data to write to shared memory
hi gana
You wrote : hi gana
```

IPC using Message Queues(MESSAGE PASSING)

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by msgget().

New messages are added to the end of a queue by msgsnd(). Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd() when the message is added to a queue. Messages are fetched from a queue by msgrcv(). We don't have to fetch the messages in a first-in,
first-out order. Instead, we can fetch messages based on their type field.

All processes can exchange information through access to a common system message queue. The sending process places a message (via some(OS) message-passing module) onto a queue which can be read by another process. Each message is given an identification or type so that processes can select the appropriate message. Process must share a common key in order to gain access to the queue in the first place.

System calls used for message queues:

ftok(): is use to generate a unique key.

msgget(): either returns the message queue identifier for a newly created message queue or returns the identifiers for a queue which exists with the same key value.
msgsnd(): Data is placed on to a message queue by calling msgsnd().
msgrcv(): messages are retrieved from a queue.
msgctl(): It performs various operations on a queue. Generally it is use to destroy message queue.

// C Program for Message Queue (Writer Process)

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX 10
// structure for message queue
struct mesg_buffer {long mesg_type;
char mesg_text[100];
} message;
int main()
{
key_t key;
int msgid;
// ftok to generate unique key
key = ftok("progfile", 65);
// msgget creates a message queue
// and returns identifier
msgid = msgget(key, 0666 | IPC_CREAT);
message.mesg_type = 1;
printf("Write Data : ");
fgets(message.mesg_text,MAX,stdin);
// msgsnd to send message
msgsnd(msgid, &message, sizeof(message), 0);
// display the message
printf("Data send is : %s \n", message.mesg_text);
return 0;
}
```

// C Program for Message Queue (Reader Process)

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
// structure for message queue
struct mesg_buffer {
long mesg_type;
char mesg_text[100];
} message;
int main()
{

key_t key;
int msgid;
// ftok to generate unique key
```

```
key = ftok("progfile", 65);
// msgget creates a message queue
// and returns identifier
msgid = msgget(key, 0666 | IPC_CREAT);
// msgrcv to receive message
msgrcv(msgid, &message, sizeof(message), 1, 0);// display the message
printf("Data Received is : %s \n",
message.mesg_text);
// to destroy the message queue
msgctl(msgid, IPC_RMID, NULL);
return 0;
}
```

OUTPUT:

```
pandu@pandu-desktop:~/Desktop/os lab$ cc write.c
pandu@pandu-desktop:~/Desktop/os lab$ ./a.out
Write Data : hi
Data send is : hi

pandu@pandu-desktop:~/Desktop/os lab$ cc read.c
pandu@pandu-desktop:~/Desktop/os lab$ ./a.out
Data Received is : hi
```

WEEK4

AIM: To simulate the following CPU scheduling algorithms a) FCFS b) SJF.

DESCRIPTION:

- Jobs are executed on first come, first serve basis.
- It is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.

Program:

```
#include<stdio.h>
main()
{
int n,a[10],b[10],t[10],w[10],g[10],i,m;
float att=0,awt=0;
for(i=0;i<10;i++)
{
a[i]=0; b[i]=0; w[i]=0; g[i]=0;
}
printf("enter the number of process");
scanf("%d",&n);
printf("enter the burst times");
for(i=0;i<n;i++)
scanf("%d",&b[i]);
printf("\nenter the arrival times");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
g[0]=0;
for(i=0;i<10;i++)
g[i+1]=g[i]+b[i];
for(i=0;i<n;i++){
w[i]=g[i]-a[i];
t[i]=g[i+1]-a[i];
awt=awt+w[i];
att=att+t[i];
}
awt =awt/n;
att=att/n;
printf("\n\tprocess\twaiting time\tturn arround time\n");
for(i=0;i<n;i++)
{
printf("\tp%d\t\t%d\t\t%d\n",i,w[i],t[i]);
}
printf("the average waiting time is %f\n",awt);
printf("the average turn around time is %f\n",att);
```

}

OUTPUT:

enter the number of process 4
enter the burst times
4
9
8
3
enter the arrival times
0
2
4
3

process	waiting time	turn arround time
p0	0	4
p1	2	11
p2	9	17
p3	18	21

the average waiting time is 7.250000

the average turn around time is 13.250000

SJF(SHORTEST JOB FIRST SCHEDULING ALGORITHM

DESCRIPTION:

- This is also known as **shortest job first**, or SJF
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known.
- The processer should know in advance how much time process will take.

Program:

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

```
int main()
{
int n,j,temp,temp1,temp2,pr[10],b[10],t[10],w[10],p[10],i;
float att=0,awt=0;
for(i=0;i<10;i++)
{
b[i]=0;w[i]=0;
}
printf("enter the number of process");
scanf("%d",&n);
printf("enter the burst times");
for(i=0;i<n;i++)
{
scanf("%d",&b[i]);
p[i]=i;
}
for(i=0;i<n;i++)
{
for(j=i;j<n;j++)
{
if(b[i]>b[j])
{
temp=b[i];
temp1=p[i];
b[i]=b[j];
p[i]=p[j];
b[j]=temp;
p[j]=temp1;
}
}
}
w[0]=0;
for(i=0;i<n;i++)
w[i+1]=w[i]+b[i];
for(i=0;i<n;i++)
{
t[i]=w[i]+b[i];
awt=awt+w[i];
att=att+t[i];}
awt=awt/n;
att=att/n;
printf("\n\t process \t waiting time \t turn around time \n");
for(i=0;i<n;i++)
printf("\t p[%d] \t %d \t\t %d \n",p[i],w[i],t[i]);
printf("the average waitingtimeis %f\n",awt);
printf("the average turn around time is %f\n",att);
```

```
return 1;
}
```

OUTPUT:

```
Enter number of process: 4
Enter Burst Time:
P1:4
P2:8
P3:3
P4:7
```

Average Waiting Time=6.000000
Average Turnaround Time=11.500000

Process	Burst Time	Waiting Time	Turnaround Time
P3	3	0	3
P1	4	3	7
P4	7	7	14
P2	8	14	22

WEEK5

AIM: To simulate the following CPU scheduling algorithms a) Round Robin b) Priority

DESCRIPTION:

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms inbatch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first served basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Program:

/* C Program to implement Priority CPU Scheduling Algorithm */

```
#include<stdio.h>
#define max 10
int main()
{
int i,j,n,bt[max],p[max],wt[max],tat[max],pr[max],total=0,pos,temp;
float avg_wt,avg_tat;
printf("Enter Total Number of Process:");
scanf("%d",&n);
printf("\nEnter Burst Time and Priority For ");
for(i=0;i<n;i++)
{
printf("\nEnter Process %d: ",i+1);
scanf("%d",&bt[i]);
scanf("%d",&pr[i]);
p[i]=i+1;
}
for(i=0;i<n;i++)
{
pos=i;
for(j=i+1;j<n;j++)
{
if(pr[j]<pr[pos])
pos=j;
}
temp=pr[i];
pr[i]=pr[pos];
pr[pos]=temp;
temp=bt[i];
bt[i]=bt[pos];
bt[pos]=temp;
temp=p[i];
p[i]=p[pos];
p[pos]=temp;

}
wt[0]=0;
for(i=1;i<n;i++)
{
wt[i]=0;
for(j=0;j<i;j++)
wt[i]+=bt[j];
```

```
total+=wt[i];
}
avg_wt=total/n;
total=0;
printf("\n\nProcess\t\tBT\t\tWT\t\tTAT");
for(i=0;i<n;i++)
{
tat[i]=bt[i]+wt[i];
total+=tat[i];
printf("\n P%d\t\t%d\t\t%d\t\t%d",p[i],bt[i],wt[i],tat[i]);
}
avg_tat=total/n;
printf("\n\nAverage Waiting Time = %.2f",avg_wt);
printf("\nAvg Turn Around Time = %.2f\n",avg_tat);
return 0;
}
```

OUTPUT:

```
Enter Total Number of Process: 4Enter Burst Time and Priority:
P[1]
Burst Time: 6Priority:3
P[2]
Burst Time: 2Priority:2
P[3]
Burst Time: 14Priority:1
P[4]
Burst Time: 6Priority:4
```

Process	Burst Time	Waiting Time	Turnaround Time
P[3]	14	0	14
P[2]	2	14	16
P[1]	6	16	22
P[4]	6	22	28

```
Average Waiting Time=13
Average Turnaround Time=20
```

DESCRIPTION:

Step 1: Organize all processes according to their arrival time in the ready queue. The queue structure of the ready queue is based on the FIFO structure to execute all CPU processes.

Step 2: Now, we push the first process from the ready queue to execute its task for a fixed time, allocated by each process that arrives in the queue.

Step 3: If the process cannot complete their task within defined time interval or slots because it is stopped by another process that pushes from the ready queue to execute their task due to arrival time of the next process is reached. Therefore, CPU saved the previous state of the process, which helps to resume from the point where it is interrupted. (If the burst time of the process is left, push the process end of the ready queue).

Program:

/* C Program to implement Round robin CPU Scheduling Algorithm */

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

void main()
{
   // initlialize the variable name
   int i, NOP, sum=0,count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];
   float avg_wt, avg_tat;
   printf(" Total number of process in the system: ");
   scanf("%d", &NOP);
   y = NOP; // Assign the number of process to variable y

// Use for loop to enter the details of the process like Arrival time and the Burst Time
for(i=0; i<NOP; i++)
{
printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1);
printf(" Arrival time is: \t");  // Accept arrival time
scanf("%d", &at[i]);
printf(" \nBurst time is: \t"); // Accept the Burst time
scanf("%d", &bt[i]);
temp[i] = bt[i]; // store the burst time in temp array
}
// Accept the Time qunat
printf("Enter the Time Quantum for the process: \t");
scanf("%d", &quant);
// Display the process No, burst time, Turn Around Time and the waiting time
printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");
for(sum=0, i = 0; y!=0; )
{
if(temp[i] <= quant && temp[i] > 0) // define the conditions
{
    sum = sum + temp[i];
   temp[i] = 0;
  count=1;
  }
 else if(temp[i] > 0)
```

```
 {
  temp[i] = temp[i] - quant;
sum = sum + quant;
}
 if(temp[i]==0 && count==1)
  {
    y--; //decrement the process no.
printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);
  wt = wt+sum-at[i]-bt[i];
  tat = tat+sum-at[i];
    count =0;
  }
  if(i==NOP-1)
   {
   i=0;
   }
   else if(at[i+1]<=sum)
   {
      i++;
   }
   else
   {
      i=0;
   }
}
// represents the average waiting time and Turn Around time
avg_wt = wt * 1.0/NOP;
avg_tat = tat * 1.0/NOP;
printf("\n Average Turn Around Time: \t%f", avg_wt);
printf("\n Average Waiting Time: \t%f", avg_tat);
getch();
}
```

OUTPUT:

```
Enter Total Process: 4

Enter Arrival Time and Burst Time for Process
Process Number  1 : 09

Enter Arrival Time and Burst Time for Process
Process Number  2 : 15

Enter Arrival Time and Burst Time for Process
Process Number 3 : 23

Enter Arrival Time and Burst Time for Process
Process Number  4 : 34
```

Enter Time Quantum: 5

Process	Turnaround Time	Waiting Time
P[2]	9	4
P[3]	11	8
P[4]	14	10
P[1]	21	12

Average Waiting Time= 8.500000
Avg Turnaround Time = 13.70000

WEEK-6

AIM: program to implementation of Semaphores.

DESCRIPTION:

The producer consumer problem is a synchronization problem. We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section.

To solve this problem, we need two counting semaphores – Full and Empty. "Full" keeps track of number of items in the buffer at any given time and "Empty" keeps track of number of unoccupied slots.

Semaphore : A semaphore S is an integer variable that can be accessed only through two standard operations :

- wait() - The wait() operation reduces the value of semaphore by 1
- signal() - The signal() operation increases its value by 1.

Mutual Exclusion: One or more than one resource are non-sharable (Only one process can use at a time). A **mutex** provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by producer, the consumer needs to wait, and vice versa.

Initialization of semaphores

mutex = 1
Full = 0 // Initially, all slots are empty. Thus full slots are 0
Empty = n // All slots are empty initially

PROGRAM
/*Write a C program to implement semaphores producer and consumer problem*/

```
#include<stdio.h>
#include<stdlib.h>
int mutex=1,full=0,empty=3,x=0;
int main()
{
  int n;
  void producer();
  void consumer();
  int wait(int);
  int signal(int);
  printf("\n 1.Producer  \n 2.Consumer \n 3.Exit");
  while(1)
  {
```

```
    printf("\n Enter 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;
    }
  }
}
int wait(int s)
{
  return (--s);
}
int signal(int s)
{
  return(++s);
}
void producer()
{
  mutex=wait(mutex);
  full=signal(full);
  empty=wait(empty);
  x++;
  printf("\n Producer produces the item %d",x);
  mutex=signal(mutex);
}
void consumer()
{
  mutex=wait(mutex);
  full=wait(full);
  empty=signal(empty);
  printf("\n Consumer consumes item %d",x);
  x--;
```

```
  mutex=signal(mutex);
}
```

OUTPUT:

```
[examuser35@localhost Jebastin]$ cc pc.c
 1.Producer
 2.Consumer
 3.Exit
 Enter your choice:1
 Producer produces the item 1
 Enter your choice:1
 Producer produces the item 2
 Enter your choice:1
 Producer produces the item 3
 Enter your choice:1
 Buffer is full
 Enter your choice:2
 Consumer consumes item 3
 Enter your choice:2
 Consumer consumes item 2
 Enter your choice:2
 Consumer consumes item 1
 Enter your choice:2
 Buffer is empty
 Enter your choice:3
```

WEEK7&8

AIM: Simulate Bankers Algorithm for Dead Lock Avoidance & Dead Lock Prevention

DESCRIPTION:

Banker's Algorithm:

When a new process enters a system, it must declare the maximum number of instances of each resource type it needed. This number may exceed the total number of resources in the system. When the user request a set of resources, the system must determine whether the allocation of each resources will leave the system in safe state. If it will the resources are allocation; otherwise the process must wait until some other process release the resources.

Data structures

- n-Number of process, m-number of resource types.
- Available: Available[j]=k, k – instance of resource type Rj is available.
- Max: If max[i, j]=k, Pi may request at most k instances resource Rj.
- Allocation: If Allocation [i, j]=k, Pi allocated to k instances of resource Rj
- Need: If Need[I, j]=k, Pi may need k more instances of resource type Rj,
- Need[I, j]=Max[I, j]-Allocation[I, j];

Safety Algorithm

1. Work and Finish be the vector of length m and n respectively, Work=Available and Finish[i] =False.
2. Find an i such that both
    - Finish[i] =False
    - Need<=Work

If no such I exists go to step 4.

3. work=work+Allocation, Finish[i] =True;
4. if Finish[1]=True for all I, then the system is in safe state.

Resource request algorithm

Let Request i be request vector for the process Pi, If request i=[j]=k, then process Pi wants k instances of resource type Rj.

1. if Request<=Need I go to step 2. Otherwise raise an error condition.
2. if Request<=Available go to step 3. Otherwise Pi must since the resources are available.
3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows;

Available=Available-Request I;

Allocation I =Allocation+Request I;

Need i=Need i-Request I;

If the resulting resource allocation state is safe, the transaction is completed and process Pi is allocated its resources. However if the state is unsafe, the Pi must wait for Request i and the old resource-allocation state is restored.

ALGORITHM:

1. Start the program.
2. Get the values of resources and processes.

3. Get the avail value.
4. After allocation find the need value.
5. Check whether its possible to allocate.
6. If it is possible then the system is in safe state.
7. Else system is not in safety state.
8. If the new request comes then check that the system is in safety.
9. or not if we allow the request.
10. stop the program.

PROGRAM

/* BANKER'S ALGORITHM */

```
#include<stdio.h>
#include<conio.h>
struct da
{
int max[10],a1[10],need[10],before[10],after[10];
}p[10];
void main()
{
int i,j,k,l,r,n,tot[10],av[10],cn=0,cz=0,temp=0,c=0;

printf("\n ENTER THE NO. OF PROCESSES:");
scanf("%d",&n);
printf("\n ENTER THE NO. OF RESOURCES:");
scanf("%d",&r);
for(i=0;i<n;i++)
{
printf("PROCESS %d \n",i+1);
for(j=0;j<r;j++)
{
printf("MAXIMUM VALUE FOR RESOURCE %d:",j+1);
scanf("%d",&p[i].max[j]);
}
for(j=0;j<r;j++)
{
printf("ALLOCATED FROM RESOURCE %d:",j+1);
scanf("%d",&p[i].a1[j]);
p[i].need[j]=p[i].max[j]-p[i].a1[j];
}
}
for(i=0;i<r;i++)
```

```
{
printf("ENTER TOTAL VALUE OF RESOURCE %d:",i+1);
scanf("%d",&tot[i]);
}
for(i=0;i<r;i++)
{
for(j=0;j<n;j++)
temp=temp+p[j].a1[i];
av[i]=tot[i]-temp;
temp=0;
}
printf("\n\t RESOURCES ALLOCATED NEEDED TOTAL AVAIL");
for(i=0;i<n;i++)
{
printf("\n P%d \t",i+1);
for(j=0;j<r;j++)
printf("%d",p[i].max[j]);
printf("\t");
for(j=0;j<r;j++)
printf("%d",p[i].a1[j]);
printf("\t");
for(j=0;j<r;j++)
printf("%d",p[i].need[j]);
printf("\t");
for(j=0;j<r;j++)
{
if(i==0)
printf("%d",tot[j]);
}
printf(" ");
for(j=0;j<r;j++)
{
if(i==0)
printf("%d",av[j]);
}
}
printf("\n\n\t AVAIL BEFORE \t AVAIL AFTER ");
for(l=0;l<n;l++)
{
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{
if(p[i].need[j] >av[j])
cn++;
```

```
if(p[i].max[j]==0)
cz++;
}
if(cn==0 && cz!=r)
{
for(j=0;j<r;j++)
{
p[i].before[j]=av[j]-p[i].need[j];
p[i].after[j]=p[i].before[j]+p[i].max[j];
av[j]=p[i].after[j];
p[i].max[j]=0;
}
printf("\n P %d \t",i+1);
for(j=0;j<r;j++)
printf("%d",p[i].before[j]);
printf("\t");
for(j=0;j<r;j++)
printf("%d",p[i].after[j]);
cn=0;
cz=0;
c++;
break;
}
else
{
cn=0;cz=0;
}
}
}
if(c==n)
printf("\n THE ABOVE SEQUENCE IS A SAFE SEQUENCE");
else
printf("\n DEADLOCK OCCURED");
getch();
}
```

Output: testcase 1: safe state

```
//Test Case 1:
Enter The No. Of Processes:4
Enter The No. Of Resources:3
Process 1
Maximum Value For Resource 1:3
Maximum Value For Resource 2:2
Maximum Value For Resource 3:2
Allocated From Resource 1:1
Allocated From Resource 2:0
```

Allocated From Resource 3:0
Process 2
Maximum Value For Resource 1:6
Maximum Value For Resource 2:1
Maximum Value For Resource 3:3
Allocated From Resource 1:5
Allocated From Resource 2:1
Allocated From Resource 3:1
Process 3
Maximum Value For Resource 1:3
Maximum Value For Resource 2:1
Maximum Value For Resource 3:4
Allocated From Resource 1:2
Allocated From Resource 2:1
Allocated From Resource 3:1
Process 4
Maximum Value For Resource 1:4
Maximum Value For Resource 2:2
Maximum Value For Resource 3:2
Allocated From Resource 1:0
Allocated From Resource 2:0
Allocated From Resource 3:2
Enter Total Value Of Resource 1:9
Enter Total Value Of Resource 2:3
Enter Total Value Of Resource 3:6

RESOURCES	**ALLOCATED**	**NEEDED**	**TOTAL**	**AVAIL**
P1	322	100	222	936 112
P2	613	511	102	
P3	314	211	103	
P4	422	002	420	

	AVAIL BEFORE	**AVAIL AFTER**
P 2	010	623
P 1	401	723
P 3	620	934
P 4	514	936

THE ABOVE SEQUENCE IS A SAFE SEQUENCE

//TEST CASE:2 unsafe state(deadlock occurrence)

Enter The No. Of Processes:4
Enter The No. Of Resources:3
Process 1
Maximum Value For Resource 1:3
Maximum Value For Resource 2:2
Maximum Value For Resource 3:2
Allocated From Resource 1:1
Allocated From Resource 2:0
Allocated From Resource 3:1

Process 2
Maximum Value For Resource 1:6
Maximum Value For Resource 2:1
Maximum Value For Resource 3:3
Allocated From Resource 1:5
Allocated From Resource 2:1
Allocated From Resource 3:1
Process 3
Maximum Value For Resource 1:3
Maximum Value For Resource 2:1
Maximum Value For Resource 3:4
Allocated From Resource 1:2
Allocated From Resource 2:1
Allocated From Resource 3:2
Process 4
Maximum Value For Resource 1:4
Maximum Value For Resource 2:2
Maximum Value For Resource 3:2
Allocated From Resource 1:0
Allocated From Resource 2:0
Allocated From Resource 3:2
Enter Total Value Of Resource 1:9
Enter Total Value Of Resource 2:3
Enter Total Value Of Resource 3:6

RESOURCES	**ALLOCATED**	**NEEDED**	**TOTAL**	**AVAIL**
P1 322	101	221	936	110
P2 613	511	102		
P3 314	212	102		
P4 422	002	420		

AVAIL BEFORE **AVAIL AFTER**
DEADLOCK OCCURED

WEEK-9

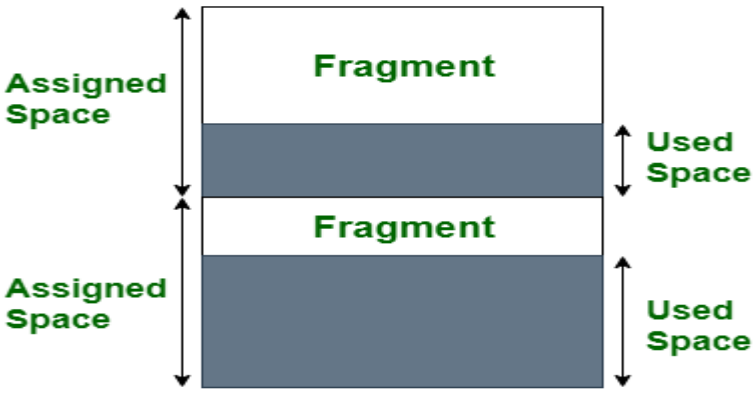
AIM: Simulate MVT and MFT.

DESCRIPTION:

MFT (Multiprogramming with a Fixed number of Tasks) :
It is one of the old memory management techniques in which the memory is partitioned into fixed size partitions and each job is assigned to a partition. The memory assigned to a partition does not change.mft suffers from internal fragmentation

Internal Fragmentation:
Internal fragmentation happens when the memory is split into fixed size blocks. Whenever a process is requested for the memory, the fixed size block is allotted to the process. In the case where the memory allotted to the process is somewhat larger than the memory requested, then the difference between allotted and requested memory is called internal fragmentation.

Internal fragmentation happens when the process size is smaller than the memory.
Internal fragmentation occurs when memory is divided into fixed-sized partitions.



Internal Fragmentation

The above diagram clearly shows the internal fragmentation because the difference between memory allocated and required space or memory is called Internal fragmentation.

Algorithm:
Step1: start the process.
Step2: Declare variables.
Step3: Enter total memory size.
Step4: Allocate memory for os.
Step5: allocate total memory to the pages.
Step6: Display the wastage of memory.
Step7: Stop the process.
/* MFT */

Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int ms,i,ps[20],n,size,p[20],s,intr=0;

printf("Enter size of memory:");
scanf("%d",&ms);
printf("Enter memory for OS:");
scanf("%d",&s);
ms-=s;
printf("Enter no.of partitions to be divided:");
scanf("%d",&n);
size=ms/n;
for(i=0;i<n;i++)
{
printf("Enter process and process size");
scanf("%d%d",&p[i],&ps[i]);
if(ps[i]<=size)
{
intr=intr+size-ps[i];
printf("process%d is allocated\n",p[i]);
}
else
printf("process%d is blocked",p[i]);
}
printf("internal fragmentation is %d",intr);
getch();
}
```

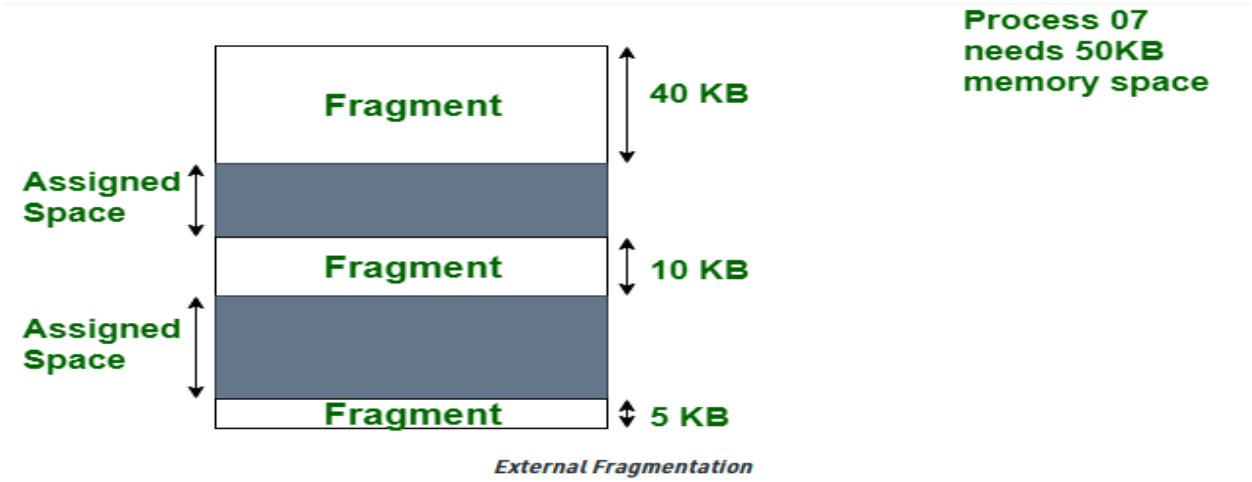
OUTPUT:

```
Enter size of memory:50
Enter memory for OS:10
Enter no.of partitions to be divided:4
Enter process and process size1 10
process1 is allocated
Enter process and process size2 9
process2 is allocated
Enter process and process size3 9
process3 is allocated
Enter process and process size4 8
process4 is allocated
internal fragmentation is 4
```

MVT (Multiprogramming with a Variable number of Tasks):
It is the memory management technique in which each job gets just the amount of memory it needs. That is, the partitioning of memory is dynamic and changes as jobs enter and leave the system. MVT is a more ``efficient'' user of resources. MVT suffers with external fragmentation.

External Fragmentation:
External fragmentation happens when there's a sufficient quantity of area within the memory to satisfy the memory request of a method. However, the process's memory request cannot be fulfilled because the memory offered is in a non-contiguous manner. Whether you apply a first-fit or best-fit memory allocation strategy it'll cause external fragmentation.

External fragmentation happens when the process size is larger than the memory
External fragmentation occurs when memory is divided into variable size partitions based on the size of processes.
External fragmentation happens when the method or process is removed.



External Fragmentation

In the above diagram, we can see that, there is enough space (55 KB) to run a process-07 (required 50 KB) but the memory (fragment) is not contiguous. Here, we use compaction, paging, or segmentation to use the free space to run a process.

Step1: start the process.
Step2: Declare variables.
Step3: Enter total memory size.
Step4: Allocate memory for os.
Step5: allocate total memory to the pages.
Step6: Display the wastage of memory.
Step7: Stop the process.
/* MVT */

Program:

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

```
int i,m,n,tot,s[20];

printf("Enter total memory size:");
scanf("%d",&tot);
printf("Enter no. of processes:");
scanf("%d",&n);
printf("Enter memory for OS:");
scanf("%d",&m);
for(i=0;i<n;i++)
{
printf("Enter size of process%d:",i+1);
scanf("%d",&s[i]);
}
tot=tot-m;
for(i=0;i<n;i++)
{
if(tot>=s[i])
{
printf("Allocate page %d\n",i+1);
tot=tot-s[i];
}
else
printf("process p%d is blocked\n",i+1);
}
printf("External Fragmentation is=%d",tot);
getch();
}
```

OUTPUT:

```
Enter total memory size:50
Enter no. of processes:4
Enter memory for OS:10
Enter size of process1:10
Enter size of process2:10
Enter size of process3:9
Enter size of process4:9
process allocated is 1
process allocated is 2
process allocated is 3
process allocated is 4
External Fragmentation is=2
```

WEEK-10

AIM: Implementation of the following Memory Allocation Methods for fixed partition

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

DESCRIPTION:

a) Program for First Fit

First Fit:

In the first fit approach is to allocate the first free partition or hole large enough which can accommodate the process. It finishes after finding the first suitable free partition.

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp;
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)
{
ff[i]=j;
break;
}
}
}
frag[i]=temp;
bf[ff[i]]=1;
}
printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragment");
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();
}
```

Output:

```
Enter the number of blocks:4
Enter the number of files:3

Enter the size of the blocks:-
Block 1:5
Block 2:8
Block 3:4
Block 4:10
Enter the size of the files:-
File 1:1
File 2:4
File 3:7
```

File_no:	File_size :	Block_no:	Block_size:	Fragment
1	1	1	5	4
2	4	2	8	4
3	7	4	10	3

b) Program Code for Best Fit

Best Fit:

The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire list of free partitions and considers the smallest hole that is adequate. It then tries to find a hole which is close to actual process size needed

```
#include<stdio.h>
#include<conio.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]);
getch();
}
```

Output:

Enter the number of blocks:4
Enter the number of files:3

Enter the size of the blocks:-
Block 1:5
Block 2:8
Block 3:4
Block 4:10
Enter the size of the files:-
File 1:1
File 2:4
File 3:7

File_no:	File_size :	Block_no:	Block_size:	Fragment
1	1	3	4	3
2	4	1	5	1
3	7	2	8	1

c) Program Code for Worst Fit

Worst fit:
In worst fit approach is to locate largest available free portion so that the portion left will be big enough to be useful. It is the reverse of best fit.

```
#include<stdio.h>
#include<conio.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("\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 \tFragment");
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();
}
```

Output:

```
Enter the number of blocks:4
Enter the number of files:3
Enter the size of the blocks:-
Block 1:5
Block 2:8
Block 3:4
Block 4:10
Enter the size of the files:-
File 1:1
File 2:4
File 3:7
```

File_no:	File_size :	Block_no:	Block_size:	Fragment
1	1	4	10	9
2	4	2	8	4
3	7	0	0	0

WEEK-11

AIM: Simulate all page replacement algorithms.
a) FIFO b) LRU c) Optimal (LFU).

DESCRIPTION:

FIFO (First In First Out)

ALGORITHM:

FIFO:

Step 1: Create a queue to hold all pages in memory

Step 2: When the page is required replace the page at the head of the queue

Step 3: Now the new page is inserted at the tail of the queue

```
/* FIFO Page Replacement Algorithm */
#include<stdio.h>
#include<conio.h>
int i,j,nof,nor,flag=0,ref[50],frm[50],pf=0,victim=-1;
void main()
{
clrscr();
printf("\n \t\t\t FIFI PAGE REPLACEMENT ALGORITHM");
printf("\n Enter no.of frames....");
scanf("%d",&nof);
printf("Enter number of reference string..\n");
scanf("%d",&nor);
printf("\n Enter the reference string..");
for(i=0;i<nor;i++)
scanf("%d",&ref[i]);
printf("\nThe given reference string:");
for(i=0;i<nor;i++)
printf("%4d",ref[i]);
for(i=1;i<=nof;i++)
frm[i]=-1;
printf("\n");
for(i=0;i<nor;i++)
{
flag=0;
printf("\n\t Reference np%d->\t",ref[i]);
for(j=0;j<nof;j++)
{
```

```
if(frm[j]==ref[i])
{
flag=1;
break;
}}
if(flag==0)
{
pf++;
victim++;
victim=victim%nof;
frm[victim]=ref[i];
for(j=0;j<nof;j++)
printf("%4d",frm[j]);
}
}
printf("\n\n\t\t No.of pages faults...%d",pf);
getch();
}
```

OUTPUT:

```
FIFO PAGE REPLACEMENT ALGORITHM
Enter no.of frames....4
Enter number of reference string..
6
Enter the reference string..
5 6 4 1 2 3
The given reference string:
..... 5 6 4 1 2 3
Reference np5-> 5 -1 -1 -1
Reference np6-> 5 6 -1 -1
Reference np4-> 5 6 4 -1
Reference np1-> 5 6 4 1
Reference np2-> 2 6 4 1
Reference np3-> 2 3 4 1
No.of pages faults...6
```

LRU PAGE REPLACEMENT ALGORITHM

AIM: To implement page replacement algorithm

LRU (Least Recently Used)

LRU (Lease Recently Used)

Here we select the page that has not been used for the longest period of time.

ALGORITHM:

Step 1: Create a queue to hold all pages in memory

Step 2: When the page is required replace the page at the head of the queue

Step 3: Now the new page is inserted at the tail of the queue

Step 4: Create a stack

Step 5: When the page fault occurs replace page present at the bottom of the stack

```
/* LRU Page Replacement Algorithm */
#include<stdio.h>
#include<conio.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()
{
clrscr();
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);
getch();
}
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;
}
```

OUTPUT:

```
LRU PAGE REPLACEMENT ALGORITHM
Enter no.of Frames....3
Enter no.of reference string..6
Enter reference string..
6 5 4 2 3 1
LRU PAGE REPLACEMENT ALGORITHM
The given reference string:
…….. 6 5 4 2 3 1
Reference NO 6-> 6 -1 -1
Reference NO 5-> 6 5 -1
Reference NO 4-> 6 5 4
Reference NO 2-> 2 5 4
Reference NO 3-> 2 3 4
Reference NO 1-> 2 3 1
No.of page faults...6
```

OPTIMAL(LFU) PAGE REPLACEMENT ALGORITHM

AIM: To implement page replacement algorithms

Optimal (The page which is not used for longest time)

ALGORITHM:

Optimal algorithm

Here we select the page that will not be used for the longest period of time.

OPTIMAL:

Step 1: Create a array

Step 2: When the page fault occurs replace page that will not be used for the longest period of time

```
/*OPTIMAL(LFU) page replacement algorithm*/
#include<stdio.h>
#include<conio.h>
int i,j,nof,nor,flag=0,ref[50],frm[50],pf=0,victim=-1;
int recent[10],optcal[50],count=0;
int optvictim();
void main()
{
clrscr();
printf("\n OPTIMAL PAGE REPLACEMENT ALGORITHN");
printf("\n.....");
printf("\nEnter the no.of frames");
scanf("%d",&nof);
printf("Enter the no.of reference string");
scanf("%d",&nor);
printf("Enter the reference string");
for(i=0;i<nor;i++)
scanf("%d",&ref[i]);
clrscr();
printf("\n OPTIMAL PAGE REPLACEMENT ALGORITHM");
printf("\n.....");
printf("\nThe given string");
printf("\n.....\n");
for(i=0;i<nor;i++)
printf("%4d",ref[i]);
for(i=0;i<nof;i++)
{
frm[i]=-1;
optcal[i]=0;
}
for(i=0;i<10;i++)
recent[i]=0;
printf("\n");
for(i=0;i<nor;i++)
{
flag=0;
```

```
printf("\n\tref 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=optvictim(i);
pf++;
frm[victim]=ref[i];
for(j=0;j<nof;j++)
printf("%4d",frm[j]);
}
}
printf("\n Number of page faults: %d",pf);
getch();
}
int optvictim(int index)
{
int i,j,temp,notfound;
for(i=0;i<nof;i++)
{
notfound=1;
for(j=index;j<nor;j++)
if(frm[i]==ref[j])
{
notfound=0;
optcal[i]=j;
break;
}
if(notfound==1)
return i;
}
temp=optcal[0];
for(i=1;i<nof;i++)
if(temp<optcal[i])
temp=optcal[i];
```

```
for(i=0;i<nof;i++)
if(frm[temp]==frm[i])
return i;
return 0;
}
```

OUTPUT:

```
OPTIMAL PAGE REPLACEMENT ALGORITHM
Enter no.of Frames....3
Enter no.of reference string..6
Enter reference string..
6 5 4 2 3 1
OPTIMAL PAGE REPLACEMENT ALGORITHM
The given reference string:
……. 6 5 4 2 3 1
Reference NO 6-> 6 -1 -1
Reference NO 5-> 6 5 -1
Reference NO 4-> 6 5 4
Reference NO 2-> 2 5 4
Reference NO 3-> 2 3 4
Reference NO 1-> 2 3 1
No.of page faults...6
```

WEEK-12

AIM: Simulate all File allocation strategies a) Sequenced b) Indexed c) Linked

DESCRIPTION:

a) SEQUENCEDDESCRIPTION:

Each file occupies a contiguous set of blocks on the disk. For example, if a file requires n blocks and is given a block b as the starting location, then the blocks assigned to the file will be: *b, b+1, b+2,……b+n-1.* This meansthat given the starting block address and the length of the file (in terms of blocks required), we can determine the blocks occupied by the file.

The directory entry for a file with contiguous allocation contains

- Address of starting block
- Length of the allocated portion.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
int main()
{
int n,i,j,b[20],sb[20],t[20],x,c[20][20];
printf("Enter no.of files:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Enter no. of blocks occupied by file%d",i+1);
scanf("%d",&b[i]);
printf("Enter the starting block of file%d",i+1);
scanf("%d",&sb[i]);
t[i]=sb[i];
for(j=0;j<b[i];j++)
c[i][j]=sb[i]++;
}
printf("Filename\tStart block\tlength\n");
for(i=0;i<n;i++)
printf("%d\t %d \t%d\n",i+1,t[i],b[i]);
printf("Enter file name:");
scanf("%d",&x);
printf("File name is:%d",x);
printf("length is:%d",b[x-1]);
printf("blocks occupied:");
for(i=0;i<b[x-1];i++)
printf("%4d",c[x-1][i]);
getch();
}
```

OUTPUT:

```
Enter no.of files: 2
Enter no. of blocks occupied by file1 4
Enter the starting block of file1 2
Enter no. of blocks occupied by file2 10
Enter the starting block of file2 5
Filename Start block length
1 2 4
2 5 10
Enter file name: rajesh
File name is:12803 length is:0blocks occupied
```

INDEXED FILE ALLOCATION

A special block known as the **Index block** contains the pointers to all the blocks occupied by a file. Each file has its own index block. The ith entry in the index block contains the disk address of the ith file block. The directoryentry contains the address of the index block

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
int main()
{
int n,m[20],i,j,sb[20],s[20],b[20][20],x;
printf("Enter no. of files:");
scanf("%d",&n);
for(i=0;i<n;i++)
{ printf("Enter starting block and size of file%d:",i+1);
scanf("%d%d",&sb[i],&s[i]);
printf("Enter blocks occupied by file%d:",i+1);
scanf("%d",&m[i]);
printf("enter blocks of file%d:",i+1);
for(j=0;j<m[i];j++)
scanf("%d",&b[i][j]);
} printf("\nFile\t index\tlength\n");
for(i=0;i<n;i++)
{
printf("%d\t%d\t%d\n",i+1,sb[i],m[i]);
}printf("\nEnter file name:");
scanf("%d",&x);
printf("file name is:%d\n",x);
i=x-1;
printf("Index is:%d",sb[i]);
printf("Block occupied are:");
for(j=0;j<m[i];j++)
printf("%3d",b[i][j]);
getch();
}
```

OUTPUT:

```
Enter no. of files:2
Enter starting block and size of file1: 2 5
Enter blocks occupied by file1:10
enter blocks of file1:3
2 5 4 6 7 2 6 4 7
Enter starting block and size of file2: 3 4
Enter blocks occupied by file2:5
```

enter blocks of file2: 2 3 4 5 6 File index length
1 2 10
2 3 5
Enter file name: venkat
file name is:12803
Index is:0Block occupied are:

LINKED FILE ALLOCATION:

Each file is a linked list of disk blocks which **need not be** contiguous. The disk blocks can be scattered anywhere on the disk. The directory entry contains a pointer to the starting and the ending file block. Each block contains a pointer to the next block occupied by the file.

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();
}
```

OUTPUT:

```
Enter how many blocks already
allocated: 3Enter blocks already
allocated:1 3 5
Enter index starting block and
length: 24
2_____> 1

3  Block is already
allocated4__> 1
5  Block is already
allocated6__> 1
7_____> 1

Do you want to enter more file(Yes - 1/No - 0) 0
```