KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)

(Established Under Section 3 of UGC Act 1956)

Coimbatore-641 021

(For the candidates admitted from 2017 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

SUBJECT CODE: 17CSU312 SUBJECT NAME: OPERATING SYSTEM-PRACTICAL
SEMESTER : III CLASS: II B.SC CS-A L T P : 0 0 4

---

1. Write a program (using *fork()* and/or *exec()* commands) where parent and child execute:
   a) same program, same code.
   b) same program, different code.
   c) before terminating, the parent waits for the child to finish its task.
2. Write a program to report behaviour of Linux kernel including kernel version, CPU type and model. (CPU information)
3. Write a program to report behaviour of Linux kernel including information on configured memory, amount of free and used memory. (memory information)
4. Write a program to print file details including owner access permissions, file access time, where file name is given as argument.
5. Write a program to copy files using system calls.
6. Write program to implement FCFS scheduling algorithm.
7. Write program to implement Round Robin scheduling algorithm.
8. Write program to implement SJF scheduling algorithm.
9. Write program to implement non-preemptive priority based scheduling algorithm.
10. Write program to implement preemptive priority based scheduling algorithm.
11. Write program to implement SRJF scheduling algorithm.
12. Write program to calculate sum of n numbers using *thread* library.
13. Write a program to implement first-fit, best-fit and worst-fit allocation strategies.

Ex. No: 01

PARENT-CHILD PROCESS CREATION AND EXECUTION

AIM:

To write a program using (fork() and/ or exec() commands) where parent child execute :

i. Same program, same code.
ii. Same program, different code
iii. Before terminating the parent waits for the child to finish its task.

ALGORITHM:

Step 1: Start the process.

Step 2: Declare the header files.

Step 3: In main function, declare the variables as ret, pid, status of type int.

Step 4: Create a new process using fork() command.

Step 5: The execl("./","parent",NULL) function is used to execute same program same code. Step 6: The execvp("./child",NULL) function is used to execute same program different code. Step 7: If pid>0 then before terminating the parent waits for the child to finish its task using

waitpid( ) function.

Step 8: The process id is stored in ret variable

Step 9: Write a new program to execute the child process using different code. Step 10: Stop the process.

PROGRAM:

Main program :

```
#include<stdio.h>
#include<unistd.h>
main()
{
int pid,ret,status;
pid=fork();
if(pid==0)
{
        //exevp("./child",NULL);        // same program different code
        execl("./","parent",NULL);      // same program same code
}
else if(pid>0)
        {
        ret=waitpid(pid,&status,0);
        printf("\n I am in parent
        \n"); if(ret==-1)
        printf("\nError in wait ID \n");
        else if(ret==pid)
        printf("I am waiting in parent \n");
        }
else
{
        printf("Error in fork command \n");
}
}
```

Child process

```
#include<stdio.h>
void main()
{
printf("\n I am child process");
}
```

OUTPUT

i. Same program, same code.

```
I am in parent
I am waiting in parent
```

ii. Same program, different code

```
I am child process
I am in parent
I am waiting in parent
```

RESULT:

The above program has been executed successfully and output is verified.

Ex. No:02

CPU INFORMATION

AIM:

To write a program to report behaviour of linux kernel including kernel version, CPU type and model. (CPU information)

ALGORITHM:

Step 1: Start the process.

Step 2: Declare the header files.

Step 3: In main function, display the kernel version by locating the file /proc/sys/kernel/osrelease using cat command.

Step 4: CPU type and model information is displayed using system() command by locating the file /proc/cpuinfo

Step 5: Display the result.

Step 6: Stop the process.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
printf("\n The kernel version:\n");
system("cat /proc/sys/kernel/osrelease");
printf("\n The CPU space:\n");
system("cat /proc/cpuinfo");
printf("\n Amount of CPU time since system was last booted
is:\n"); system("cat /proc/uptime\n");
}
```

OUTPUT:

The kernel version is: 3.16. 0-38-generic

The CPU space:

CPU family:6

Model:23

Amount of CPU time since system was last booted

is: 9408.98 18586.20

RESULT:

The above program has been executed successfully and output is verified.

Ex. No: 03

MEMORY INFORMATION

AIM:

To write a program to report behaviour of Linux kernel including information on configured memory, amount of free and used memory. (memory information)

ALGORITHM:

Step 1: Start the process.

Step 2: Declare the header files.

Step 3: In main function, display configured memory by locating the file /proc/meminfo and displaying first record of the meminfo file using awk 'NR==1

Step 4: Amount of free memory information is displayed using system() command by locating the file /proc/meminfo and displaying second record of the meminfo file using awk 'NR==2

Step 5: Amount of used memory information is displayed calculated using the formula (a-b) [Total configured memory – free memory]

Step 6: Display the result.

Step 7: Stop the process.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
printf("\n The Configured Memory is:\n");
system("cat /proc/meminfo | awk 'NR==1{print $2}'\n");
printf("\n Amount of Free Memory:\n");
system("cat /proc/meminfo | awk 'NR==2{print $2}'\n
"); printf("\n Amount of used Memory:\n");
system("cat /proc/meminfo | awk '{if(NR==1) a=$2;if(NR==2) b=$2} END{print a-b}'\n");
}
```

OUTPUT

The Configured Memory is : 2054804

Amount of Free Memory : 1259508

Amount of used Memory : 795296

RESULT:

The above program has been executed successfully and output is verified.

Ex. No: 04

FILE DETAILS WITH ACCESS PERMISSIONS

AIM:

To write a program to print file details including owner access permissions, file access time, where file name is given as argument.

ALGORITHM:

Step 1: Start the process.

Step 2: Declare the header files.

Step 3: In main function, include int argc and char **argv to define the number of arguments and filename to be specified in the run command

Step 4: Create an object fileStat for the file struct stat.

Step 5: Display the file size using the command fileStat.st_size

Step 6: Display the file access time using the command fileStat.st_atime

Step 7: Display the file access permission Read (R), Write (W) and Execute (X) for user, group and others.

Step 8: Stop the process.

PROGRAM:

```
#include<unistd.h>
#include<stdio.h>
#include<sys/stat.h>
#include<sys/types.h>
int main(int argc,char **argv)
{
if(argc !=2)
return 1;
struct stat fileStat;
if(stat(argv[1],&fileStat)<0)
return 1;
printf("Information for %s\n",argv[1]);
printf("**************************\n");
printf("File Size: \t\t %ld bytes \n",fileStat.st_size);
printf("File Access Time: \t\t %ld \n",fileStat.st_atime);
printf("File Permission: \t");
printf( (fileStat.st_mode & S_IRUSR) ?"r":"-");
printf( (fileStat.st_mode & S_IWUSR) ?"w":"-");
printf( (fileStat.st_mode & S_IXUSR) ?"x":"-");
printf( (fileStat.st_mode & S_IRGRP) ?"r":"-");
printf( (fileStat.st_mode & S_IWGRP) ?"w":"-");
printf( (fileStat.st_mode & S_IXGRP) ?"x":"-");
printf( (fileStat.st_mode & S_IROTH) ?"r":"-");
printf( (fileStat.st_mode & S_IWOTH) ?"w":"-
"); printf( (fileStat.st_mode & S_IXOTH) ?"x":"-
"); printf("\n\n");
return 0;
}
```

OUTPUT:

Run command: ./perm program.txt

Information for program.txt
****

File Size : 7408 bytes
File Access Time : 1504175495
File Permission : rwxr-xr-x

RESULT:

The above program has been executed successfully and output is verified.

Ex. No: 05

COPY FILES USING SYSTEM CALLS

AIM:

To write a program to copy files using system calls.

ALGORITHM:

Step 1: Start the process.

Step 2: Declare the header files.

Step 3: In main function, include int argc and char *argv[] to define the number of arguments and filename to be specified in the run command.

Step 4: The source file is opened using the command fd1=open(argv[1],0)

Step 5: The destination file is created using the command fd2=creat(argv[2],0666).

Step 6: Declare the variables char buf[512], int cnt, in the filecopy method

Step 7: The source file(f1) is read and copied to the destination file (f2) using read() and write() functions.

Step 8: Display the result.

Step 9: Stop the process.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<fcntl.h>
void filecopy(int f1,int f2); int
main(int argc,char *argv[])
{
if(argc!=3)
{
        printf("File not
        found"); exit(1);
}
int fd1=open(argv[1],0);
if(fd1==-1)
{
        printf("Error in file
        opening"); exit(1);
}
int fd2=creat(argv[2],0666);
if(fd2==-1)
{
        printf("Error while creating file
        "); exit(1);
}
filecopy(fd1,fd2);
close(fd1);
close(fd2);
```

```
printf("\n File
Copied\n"); return 0;
}
void filecopy(int f1,int f2)
{
        char buf[512];
         int cnt;
while(cnt=read(f1,buf,sizeof(buf)))
{
        write(f2,buf,cnt);
}
}
```

OUTPUT:

Compile : gcc pgm5.c -o pgm5

Run Command: ./pgm5 file1.txt file2.txt

File Copied

RESULT:

The above program has been executed successfully and output is verified.

Ex.No: 06

FCFS SCHEDULING ALGORITHM

AIM:

To write a program to implement FCFS scheduling algorithm.

ALGORITHM:

Step 1: Start the process.

Step 2: Declare the header files.

Step 3: In main function, declare the variables as n, bt[20], wt[20], tat[20], avwt=0, avtat=0,i,j of type int.

Step 4: Get the input for the Number of process and process Burst time from the user.

Step 5: The process that requests the CPU first is allocated the CPU first using for loop.

Step 6: Calculate the waiting time and turn around time for each process using for loop.

Step 7: Display the Process number, Burst Time, Waiting time, and Turnaround time for each process.

Step 8: Calculate the average waiting time and average turn around time using avwt /=i; avtat /=i;

Step 9: Display the result.

Step 10: Stop the process.

PROGRAM:

```
#include<stdio.h>
int main()
{
int n,bt[20],wt[20],tat[20],avwt=0,avtat=0,i,j;
printf("\n Enter total number of Processes(maximum 20):");
scanf("%d",&n);
printf("\n Enter Process Burst
Time\n"); for(i=0;i<n;i++)
{
        printf("P[%d]:",i+1);
        scanf("%d",&bt[i]);
}
wt[0]=0;
for(i=0;i<n;i++)
{
        wt[i]=0;
        for(j=0;j<i;j++)
        wt[i]+=bt[j];
}
printf("\n Process\tBurst time\tWaiting time\tTurnaround time");
for(i=0;i<n;i++)
{
        tat[i]=bt[i]+wt[i];
        avwt+=wt[i];
        avtat+=tat[i];
        printf("\n P[%d]\t\t%d\t\t%d\t\t%d",i+1,bt[i],wt[i],tat[i]);
```

```
}
avwt/=i;
avtat/=i;
printf("\n Average Waiting Time:%d",avwt);
printf("\n Average Turnaround Time:%d",avtat);
return 0;
}
```

OUTPUT:

Enter total number of processes(maximum 20) : 3

Enter Process Burst Time

P[1] : 24

P[2] : 3

P[3] : 3

Process	Burst time	Waiting time	Turnaround time
P[1]	24	0	24
P[2]	3	24	27
P[3]	3	27	30

Average Waiting Time : 17

Average Turnaround Time : 27

RESULT:

The above program has been executed successfully and output is verified.

Ex.No:07

ROUND ROBIN SCHEDULING ALGORITHM

AIM:

To write a program to implement Round Robin Scheduling Algorithm.

ALGORITHM:

Step 1: Start the process.

Step 2: Declare the header files.

Step 3: In main function, declare the variables count, j, n, time, remain, flag=0, time_quantum of type int.

Step 4: Get the input values for the Number of process, process Burst time and Arrival time from the user.

Step 5: Get the input values for the time quantum from the user.

Step 6: For the given time quantum each process takes its turns until the process burst time is completed using for and if statements.

Step 7: Calculate the waiting time and turn around time for each process using for loop. Step 8 : Display the Process number, Waiting time, and Turnaround time for each process.

Step 9 : Calculate the average waiting time and average turn around time using wait_time*1.0/n and turnaround_time*1.0/n.

Step 10: Display the result.

Step 11: Stop the process.

PROGRAM:

```
#include<stdio.h>
int main()
{
int count,j,n,time,remain ,flag=0,time_quantum;
int wait_time=0,turnaround_time=0,at[10],bt[10],rt[10];
printf("Enter total process:\t");
scanf("%d",&n);
remain=n;
for(count=0;count<n;count++)
{
        printf("Enter Arrival time and Burst time for process number
        %d:",count+1); scanf("%d",&at[count]);
        scanf("%d",&bt[count]);
        rt[count]=bt[count];
}
printf("Enter time Quantum:\t");
scanf("%d",&time_quantum);
printf("\n\n Process\tTurnaround Time\tWaiting
Time\n\n"); for(time=0,count=0;remain!=0;)
{
        if(rt[count]<=time_quantum && rt[count]>0)
                {
                time+=rt[count];
                rt[count]=0;
                flag=1;
                }
```

```
        else if(rt[count]>0)
        {
        rt[count]-=time_quantum;
        time+=time_quantum;
        }
if(rt[count]==0 && flag==1)
{
        remain--; printf("P[%d]\t|\t%d\t|\t%d\n",count+1,time-at[count],time-at[count]-
        bt[count]); wait_time+=time-at[count]-bt[count];

        turnaround_time+=time-
        at[count]; flag=0;
}
if(count==n-
1) count=0;
else if(at[count+1]<=time)
count++;
else
count=0;
}
printf("\n Average Waiting Time =%f\n",wait_time*1.0/n);
printf("\n Average Turnaround Time
=%f\n",turnaround_time*1.0/n); return 0;
}
```

OUTPUT:

Enter total process : 3

Enter Arrival time and Burst time for process number 1 :

0

3

Enter Arrival time and Burst time for process number 2 :

0

4

Enter Arrival time and Burst time for process number 3 :

0

3

Enter time Quantum : 1

Process	Turnaround Time	Waiting Time
P[1]	7	4
P[3]	9	6
P[2]	10	6

Average Waiting time : 5.333333

Average Turnaround Time : 8.666667

RESULT:

The above program has been executed successfully and output is verified.

Ex.No: 08

SJF SCHEDULING ALGORITHM

AIM:

To write a program to implement SJF scheduling algorithm.

ALGORITHM:

Step 1: Start the process.

Step 2: Declare the header files.

Step 3: In main function, declare the variables bt[20], p[20], wt[20], tat[20], i, j, n, total=0, pos, temp of type integer..

Step 4: Get the input for the Number of process and process Burst time from the user.

Step 5: In shortest job first scheduling algorithm, the processor selects the waiting process with the smallest execution time to execute next.

Step 6: The processes are sorted in ascending order using temp variable.

Step 7: Calculate the waiting time and turn around time for each process using for loop.

Step 8: Display the Process number, Burst Time, Waiting time, and Turnaround time for each process.

Step 9: Calculate the average waiting time and average turn around time using avwt /=i; avtat /=i;

Step 10: Display the result.

Step 11: Stop the process.

PROGRAM:

```
#include<stdio.h>
void main()
{
int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;
float avg_wt,avg_tat;
printf("\n Enter number of
process:"); scanf("%d",&n);
printf("\n Enter Burst time:\n");
for(i=0;i<n;i++)
{
        printf("P%d:",i+1);
        scanf("%d",&bt[i]);
        p[i]=i+1;
}
   //sorting burst time in ascending order using selection sort

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

```
temp=p[i];
p[i]=p[pos];
p[pos]=temp;
}
wt[0]=0; //waiting time for first process will be zero
   //calculate waiting time
for(i=1;i<n;i++)
{
        wt[i]=0;
        for(j=0;j<i;j++)
        wt[i]+=bt[j];
        total+=wt[i];
}
avg_wt=(float)total/n; //average waiting time total=0;

printf("\n \t Process \t Burst time \t Waiting time \t Turnaround time");
for(i=0;i<n;i++)
{
        tat[i]=bt[i]+wt[i];      //calculate turnaround time
        total+=tat[i];
        printf("\nP %d \t\t %d\t\t %d \t\t%d\t\t",p[i],bt[i],wt[i],tat[i]);
}
avg_tat=(float)total/n; //average turnaround time
printf("\n Average Waiting Time=%f",avg_wt);
printf("\n Average Turnaround Time=%f\n",avg_tat);
printf("\n");
}
```

OUTPUT:

Enter number of process : 3

Enter burst time :

P1: 6

P2: 8

P3: 7

Process	Burst time	Waiting time	Turnaround time
P1	6	0	6
P3	7	6	13
P2	8	13	21

Average Waiting time : 6.333333

Average Turnaround time : 13.333333

RESULT:

The above program has been executed successfully and output is verified.

Ex.No: 09

NON-PREEMPTIVE PRIORITY SCHEDULING ALGORITHM

AIM:

To write a program to implement non-preemptive priority scheduling algorithm.

ALGORITHM:

Step 1: Start the process.

Step 2: Declare the header files.

Step 3: In main function, declare the necessary variables of type integer and float.

Step 4: Get the input for the Number of process and process Burst time from the user.

Step 5: Shortest job first scheduling algorithm acts as a nonpreemptive scheduling algorithm.

Step 6: The processes are sorted in ascending order using temp variable.

Step 7: Calculate the waiting time and turnaround time for each process using for loop.

Step 8: Display the Process number, Burst Time, Waiting time, and Turnaround time for each process.

Step 9: Calculate the average waiting time and average turnaround time.

Step 10: Display the result.

Step 11: Stop the process.

PROGRAM:

```
#include<stdio.h>
int main()
{
int i,n,p[10]={1,2,3,4,5,6,7,8,9,10},min,k=1,btime=0;
int bt[10],temp,j,at[10],wt[10],tt[10],ta=0,sum=0;
float wavg=0,tavg=0,tsum=0,wsum=0;
printf("\n ------- SHORTEST JOB FIRST (NP) --------");
printf("\n Enter the no.of process:\n");
scanf("%d",&n);
for(i=0;i<n;i++)
{
        printf("\n Enter the burst time of P%d process : ",i+1);
        scanf("%d",&bt[i]);
        printf("\n Enter the arrival time of P%d process: ",i+1);
        scanf("%d",&at[i]);
}
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
if(at[i]<at[j])
{
        temp=p[j];
        p[j]=p[i];
        p[i]=temp;
        temp=at[j];
        at[j]=at[i];
```

```
            at[i]=temp;
            temp=bt[j];
            bt[j]=bt[i];
            bt[i]=temp;
    }
    }
    }
    for(j=0;j<n;j++)
    {
            btime=btime+bt[j];
            min=bt[k];
    for(i=k;i<n;i++)
    {
    if(btime>=at[i] && bt[i]<min)
    {
            temp=p[k];
            p[k]=p[i];
            p[i]=temp;
            temp=at[k];
            at[k]=at[i];
            at[i]=temp;
            temp=bt[k];
            bt[k]=bt[i];
            bt[i]=temp;
    }
    }
            k++;
    }
    wt[0]=0;
```

```
for(i=1;i<n;i++)
{
        sum=sum+bt[i-1];
        wt[i]=sum-at[i];
        wsum=wsum+wt[i];
}
        wavg=(wsum/n);
for(i=0;i<n;i++)
{
        printf("\n **************");
        printf("\n Result: \n");
        printf("\n Process \t Burst \t Arrival \t Waiting \t Turnaround ");
for(i=0;i<n;i++)
{
        printf("\n P %d \t %d \t %d \t %d \t %d ",p[i],bt[i],at[i],wt[i],tt[i]);}
        printf("\n Average waiting time: %f",wavg);
        printf("\n Average turnaround time:
        %f",tavg); return 0;
}
}
```

OUTPUT:

------- SHORTEST JOB FIRST (NP) --------

Enter the no.of process:2

Enter the burst time of P1 process:10

Enter the arrival time of P1 process:2

Enter the burst time of P2 process:5

Enter the arrival time of P2 process:1

RESULT:
******

Process	Burst	Arrival	Waiting	Turn-around
P[2]	5	1	0	4
P[1]	10	2	3	13

Average waiting time:1.500000

Average turn around time:8.500000

RESULT:

The above program has been executed successfully and output is verified.

Ex.No: 10

PREEMPTIVE PRIORITY SCHEDULING ALGORITHM

AIM:

To write a program to implement preemptive priority scheduling algorithm.

ALGORITHM:

Step 1: Start the process.

Step 2: Declare the header files.

Step 3: In main function, declare the necessary variables of type integer and float.

Step 4: Get the input for the Number of process and process Burst time from the user.

Step 5: In preemptive scheduling algorithm, the process with the highest priority is first executed.

Step 6: Calculate the waiting time and turnaround time for each process using for loop.

Step 7 : Display the Process number, Burst Time, Waiting time, and Turnaround time for each process.

Step 8: Calculate Average waiting time and turnaround time using wait_time/limit and turnaround_time/limit

Step 9: Display the result.

Step 10: Stop the process.

PROGRAM:

```
#include<stdio.h>
int main()
{
int i,j,time,sum_wait=0,sum_turnaround=0,smallest,n;
int at[10],bt[10],pt[10],rt[10],remain;//rt=remaining
time printf("\n Enter no of process:");
scanf("%d",&n);
remain=n;
for(i=0;i<n;i++)
{
printf("Enter arrival time,Burst time and priority for process p%d:",i+1);
scanf("%d",&at[i]);
scanf("%d",&bt[i]);
scanf("%d",&pt[i]);
rt[i]=bt[i];
}
pt[9]=11;
printf("\n\n Process\t|\tTurnaround Time\t|waiting Time\n ");
for(time=0;remain!=0;time++)
{
smallest=9;
for(i=0;i<n;i++)
{
if(at[i]<=time && pt[i]<pt[smallest] && rt[i]>0)
{
smallest=i;
}
```

```
}
rt[smallest]--;
if(rt[smallest]==0)
{
remain--;
printf("p[%d]\t|\t %d\t|\t%d\n",smallest+1,time+1-at[smallest],time+1-at[smallest]-
bt[smallest]);
sum_wait+=time+1-at[smallest];
sum_turnaround+=time+1-at[smallest]-bt[smallest];
}
}
printf("\n avg waitig time=%f\n",sum_wait*1.0/n);
printf("Avg turnaroud
time=%f",sum_turnaround*1.0/n); return 0;
}
```

OUTPUT:

Enter no of process : 3

Enter arrival time, Burst time and priority for process p1 :

0 10

3

Enter arrival time,Burst time and priority for process p2 : 0

20 2

Enter arrival time,Burst time and priority for process p3 : 0

30 1

Process	Turnaround Time	waiting Time
p[3]	30	0
p[2]	50	30
p[1]	60	50

Avg waiting time=46.666667

Avg turnaround time=26.666667

RESULT:

The above program has been executed successfully and output is verified.

Ex. No: 11

SRJF SCHEDULING ALGORITHM

AIM:

To write a program to implement SJRF scheduling algorithm.

ALGORITHM:

Step 1: Start the process.

Step 2: Declare the header files.

Step 3: In main function, declare the necessary variables.

Step 4: Get the input for the Number of process and process Burst time from the user.

Step 5: In this scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute.

Step 6: The remaining time of the processes is compared with the next process using if statement.

Step 7: Calculate the waiting time and turnaround time for each process using for loop.

Step 8 : Display the Process number, Waiting time, and Turnaround time for each process.

Step 9: Calculate the average waiting time and average turnaround time using sum_wait*1.0/n and sum_turnaround*1.0/n

Step 10: Display the result.

Step 11: Stop the process.

PROGRAM:

```
#include<stdio.h>
int main()
{
int at[10],bt[10],rt[10],endTime,i,smallest;
int remain=0,n,time,sum_wait=0,sum_turnaround=0;
printf("\n Enter no.of process: ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
        printf("\n Enter arrival time for process P %d: ",i+1);
        scanf("%d",&at[i]);
        printf("\n Enter burst time for process P %d: ",i+1);
        scanf("%d",&bt[i]);
        rt[i]=bt[i] ;
}
printf("\n Process \t Turnaround Time| Waiting time
\n\n"); rt[9]=9999;
for(time=0;remain!=n;time++)
{
        smallest=9;
for(i=-0;i<n;i++)
{
if(at[i]<=time && rt[i]<rt[smallest] && rt[i]>0)
{
        smallest=i;
}
}
```

```
        rt[smallest]--
; if(rt[smallest]==0)
{
remain++;endTime=time+1;
printf("\n P[%d] \t %d \t %d ",smallest+1,endTime-at[smallest],endTime-
bt[smallest]-at[smallest]);
sum_wait+=endTime-bt[smallest]-at[smallest];
sum_turnaround+=endTime-at[smallest];}
}
printf("\n Avg waiting time = %f \n",sum_wait*1.0/n); printf("\n
Avg turnaround time = %f \n",sum_turnaround*1.0/n); return 0;

}
```

OUTPUT:

Enter total process: 2

Enter arrival time and burst time for process P1: 3

Enter arrival time and burst time for process P2: 5

Enter arrival time and burst time for process P3: 7

Enter arrival time and burst time for process P4: 9

Process	Turnaround Time	Waiting Time
P[1]	5	0
P[2]	10	1

Average waiting time= 0.50000

Average turnaround time= 3.750000

RESULT:

The above program has been executed successfully and output is verified.

Ex.No: 12

SUM OF N NUMBERS USING THREAD LIBRARY

AIM:

To write a program to calculate sum of n numbers using thread library.

ALGORITHM:

Step 1: Start the process.

Step 2: Declare the header files.

Step 3: In main function, include int argc and char *argv[] to define number of arguments and integer value to be specified in the run command.

Step 4: Declare the pthread library objects thread ID (tid), attr.

Step 5: The if(atoi(argv[1]<0)) statement is used specify the input value should be a positive integer.

Step 6: The pthread_create()function starts a new thread in the calling Process, and pthread_join() function waits for the thread specified by thread to terminate.

Step 7: In the run function the atoi(param) function is used to return the converted integer value. The param parameter is used to set the upper limit.

Step 8: Calculate sum of n numbers using sum+=i.

Step 9: Display the result.

Step 10: Stop the process.

PROGRAM:

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<pthread.h>
int sum;
void *run(void *param);
int main(int argc,char *argv[])
{
        pthread_t tid;
        pthread_attr_t attr;
if(argc!=2)
{
        printf("ERROR !");
        return 1;
}
if(atoi(argv[1]<0))
{
        printf("\n No.should be
        +ve"); return 1;
}
        pthread_attr_init(&attr);
        pthread_create(&tid,&attr,run,argv[1]);
        pthread_join(tid,NULL);
        printf("\n SUM= %d \n",sum);
}
```

```
void *run(void *param)
{
        int i,upper; sum=0;

        upper=atoi(param);

if(upper>0)
{
        for(i=1;i<=upper;++i)
        sum+=i;
}
pthread_exit(0);
}
```

OUTPUT:

Compile command : gcc **-pthread** pthread.c -o pthread

Run command: ./pthread 5

SUM= 15

RESULT:

The above program has been executed successfully and output is verified.

Ex.No: 13

FIRST-FIT, BEST-FIT AND WORST-FIT ALLOCATION STRATEGIES

AIM:

To write a program to implement first-fit, best-fit and worst-fit allocation strategies.

ALGORITHM:

Step 1: Start the process.

Step 2: Declare the header files.

Step 3: In main function, declare the necessary variables.

Step 4: Get the input for the Number of block and number of process a from the user.

Step 5: Get the input for the size of the block and size of the process from the user.

Step 6: In best fit implementation the algorithm first selects the smallest block which can adequately fulfill the memory request by the respective process.

Step 7: In first fit implementation the algorithm first selects the first block which can adequately fulfill the memory request by the respective process.

Step 8: In worst fit implementation the algorithm first selects the largest block which can adequately fulfill the memory request by the respective process.

Step 9: Display the result.

Step 10: Stop the process.

PROGRAM:

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("\n Enter size of each block:");
for(i=0;i<bno;i++)
scanf("%d",&bsize[i]);
printf("\n Enter no.of processes");
scanf("%d ",&pno);
printf("\n Enter size of each process:");
for(i=0;i<pno;i++)
scanf("%d",&psize[i]);
for(i=0;i<pno;i++)
for(j=0;j<bno;j++)
if(flags[j]==0 && bsize[j]>=psize[i])
{
allocation[j]=i;
flags[j]=1;
break;
```

```
}
printf("\n block no.\t size\t process no.\t\t size");
 for(i=0;i<bno;i++)
{
printf("\n %d\t\t%d\t\t",i+1,bsize[i]);
if(flags[i]==1)
printf("%d \t\t%d",allocation[i]+1,psize[allocation[i]]);
else
printf("Not allocation");
}
}
```

OUTPUT:

Enter no. of blocks:2

Enter size of each block: 10

5

Enter no. of processes: 2

Enter size of each process: 7

3

block no.	block size	process no.	process size
1	10	1	7
2	5	2	3

BEST FIT:

```
#include<stdio.h>
void main()
{
int fragment[20], b[20], p[20], i, j, nb, np, temp, lowest=9999;
static int barray[20],parray[20];
printf("\n\t\tMEMORY MANAGEMENT SCHEME - BEST
FIT"); printf("\n Enter the number of block:");
scanf("%d",&nb);
printf("Enter the number of processes:");
scanf ("%d",&np);
printf("\n Enter the size of blocks:\n");
for(i=1;i<=nb;i++)
{
printf("Block number %d:",i);
scanf("%d",&b[i]);
}
printf("\n Enter the size of processes:\n");
for(i=1;i<=np;i++)
{
printf("\n Process number %d:",i);
scanf("%d",&p[i]);
}
for(i=1;i<=np;i++)
{
for(j=1;j<=nb;j++)
{
```

```
if(barray[j]!=1)
{
temp=b[j]-p[i];
if(temp>=0)
if(lowest>temp)
{
parray[i]=j;
lowest=temp;
}
}
}
fragment[i]=lowest;
barray  [parray[i]]=1;
lowest=10000;
}
printf("\nProcess_no\tProcess_size\tBlock_no\tBlock_size\tFragment");
for(i=1;i<=np && parray[i]!=0;i++)
printf("\n %d\t\t %d\t\t %d\t\t %d\t\t %d",i,p[i],parray[i],b[parray[i]],fragment[i]);
}
```

OUTPUT

MEMORY MANAGEMENT SCHEME - BEST FIT

Enter the number of block : 4
Enter the number of processes : 3

Enter the size of blocks
Block number 1 : 10
Block number 2 : 5
Block number 3 : 10
Block number 4 : 4

Enter the size of processes
Process number 1 : 4
Process number 2 : 5
Process number 3 : 3

Process_no	Process_size	Block_no	Block_size	Fragment
1	4	4	4	0
2	5	2	5	0
3	3	1	10	7

WORST FIT :

```
#include<stdio.h>

int main()
{
    int fragments[10], blocks[10], files[10];
    int m, n, number_of_blocks, number_of_files, temp, top =
    0; static int block_arr[10], file_arr[10];
    printf("\nEnter the Total Number of Blocks:\t");
    scanf("%d",&number_of_blocks); printf("Enter
    the Total Number of Files:\t");
    scanf("%d",&number_of_files);
    printf("\nEnter the Size of the Blocks:\n");
    for(m = 0; m < number_of_blocks; m++)
    {
        printf("Block No.[%d]:\t", m + 1);
        scanf("%d", &blocks[m]);
    }
    printf("Enter the Size of the Files:\n");
    for(m = 0; m < number_of_files; m++)
    {
        printf("File No.[%d]:\t", m +
        1); scanf("%d", &files[m]);
    }
    for(m = 0; m < number_of_files; m++)
    {
        for(n = 0; n < number_of_blocks; n++)
        {
```

```
            if(block_arr[n] != 1)
            {
                temp = blocks[n] - files[m];
                if(temp >= 0)
                {
                    if(top < temp)
                    {
                        file_arr[m] = n;
                        top = temp;
                    }
                }
            fragments[m] = top;
            block_arr[file_arr[m]] = 1;
            top = 0;
        }
    }
    printf("\nFile Number\tFile Size\tBlock Number\tBlock Size\tFragment");
    for(m = 0; m < number_of_files; m++)
    {
        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", m, files[m], file_arr[m],
blocks[file_arr[m]], fragments[m]);
    }
    printf("\n");
    return 0;
}
```

OUTPUT:

```
Enter the total number of blocks: 3
Enter the total number of files: 3
Enter the size of blocks:
Block No.[1]: 7
Block No.[2]: 8
Block No.[3]: 9
Enter the size of files:

File No.[1]: 4
File No.[2]: 5
File No.[3]: 6
```

File Number	File Size	Block Number	Block Size	Fragment
0	4	3	9	5
1	5	0	8	3
2	6	0	7	1

RESULT:

The above program has been executed successfully and the output is verified.