# ST.ANNE'S

**COLLEGE OF ENGINEERING AND TECHNOLOGY**

ANGUCHETTYPALAYAM, PANRUTI – 607106.

**LAB MANUAL**

**Jan 21 - Jun 21 / EVEN SEMESTER**

**BRANCH:** CSE **YR/SEM:** II/IV **BATCH**: 2019 - 2023

**SUB CODE/NAME:** CS8461 – Operating Systems Laboratory

**Exp: 1a**

## IMPLEMENTATION OF FORK, EXEC, GETPID, EXIT AND WAIT SYSTEM CALL

**AIM:**

To write a C program to implement fork, exec, getpid, exit and wait system calls.

**ALGORITHM**:

**Step 1:** Using *fork* system call, create a new process.
**Step 2:** Assign the process's id in the variable pid.
**Step 3:** If pid is equal to –1, then report an error and terminate using the *exit* system call.
**Step 4:** If pid is equal to 0, it is a child process. Print the process's id using the *getpid* system call.
**Step 5:** Otherwise, it is the parent process. Print the ids of both child and parent process.
**Step 6:** Using execlp system call, the present working directory is displayed.
**Step 7:** Status of the processes is displayed using wait system call.
**Step 8:** Terminate the program using the *exit* system call.

**PROGRAM:**

```
#include<unistd.h>
#include<stdio.h>
main()
{
    int pid,status;
    pid=fork();
    if(pid==-1)
    {
        perror("Error fork()");
    }
    else if(pid==0)
    {
        printf("I am the child process\n");
        printf("Process id of the child is=%d\n", getpid());
        execlp("pwd","pwd",NULL);
    }
    else
```

```
    {
        printf("I am the parent process\n");
        printf("Process id of parent=%d\n Process id of child =%d\n",getpid(),pid);
        wait(&status);
        printf("Child Returned %d\n",status);
    }
    exit(0)
}
```

**OUTPUT:**

```
[smk@cseserver3]$ cc ex4a.c
[smk@cseserver3]$ ./a.out
PID 11924
Parent 5562
Child: created:11925 pid parent 11924
Parent after fork 11924
PID child ID 5562
```

**RESULT:**

Thus the program to implement fork, exec, getpid, exit and wait system calls has been executed successfully.

**Exp: 1b** **IMPLEMENTATION OF STAT SYSTEM CALL**

**AIM:**

To write a C program using stat system call.

**ALGORITHM:**

**Step 1:** Pass the filename where status is to be found through the command line.
**Step 2:** Use stat system call to get the inode information of the file.
**Step 3:** Print the type, size, inode number and mode of the file.

**PROGRAM:**

```
#include<stdio.h>
#include<sys/stat.h>
int main(int argc,char *argv[])
{
	struct stat s;
	stat(argv[1],&s);
	if(argc!=2)
	{
		printf("Incorrect number of arguments\n");
		exit(1);
	}
	if(s.st_mode & S_IFREG)
		printf("%s is a regular file",argv[1]);
	if(s.st_mode & S_IFDIR)
		printf("%s is a directory",argv[1]);
	printf("\n Size is %d",s.st_size);
	printf("\n Node number is %d",s.st_ino);
	printf("\n Mode is %o\n",s.st_mode);
}
```

**OUTPUT:**

```
[smk @cseserver3]$ cc exa.c
[smk @cseserver3]$ ./a.out
Sh>
Buffsh> what
smk
Buffsh>ls
a.out change.c cse.v directory.c file.c process.c read.c
```

**RESULT:**

Thus the program for stat system call has been executed successfully.

**Exp: 1c**

## IMPLEMENTATION OF OPENDIR, READDIR AND CLOSEDIR SYSTEM CALLS

**AIM:**

To write a C program using opendir, readdir and closedir system calls.

**ALGORITHM:**

**Step 1:** Open the current directory using opendir system call.
**Step 2:** If the directory is not opened report an error message.
**Step 3:** Otherwise display the contents of the directory using readdir system call.
**Step 4:** Close the opened directory using closedir system call.

**PROGRAM:**

```
#include<dirent.h>
#include<stdio.h>
main()
{
      DIR *dir;
      struct dirent *entry;
      if((dir=opendir("/"))==NULL)
            perror("Error during Opendir() error");
      else
      {
            puts("contents of the current directory");
            while((entry=readdir(dir))!=NULL)
                  printf("%s\n",entry->d_name);
            closedir(dir);
      }
}
```

**OUTPUT:**

```
[smk @cseserver3]$ cc ex3a.c
[smk @cseserver3]$ ./a.out
contents of the current directory
smk
```

**RESULT:**

Thus the program for open, read and close directory has been executed and output is verified.

**Ex : 2 READ AND WRITE OPERATION IN FILE**

**AIM:**

To write a program to perform file read and write operation.

**ALGORITHM:**

**Step 1:** Enter the filename as command lines argument.

**Step 2:** Use the access system call to find out whether the file permission are available for the mention file.

**Step 3:** if yes, then open the file as a write only file. If the file does not exist then create the file and write the given message mentioned in the write system call.

**Step 4:** if no, then open the file as a read only file and reads the content of the existing file using the read system call.

**PROGRAM:**

```
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
#include<fcntl.h>
int main(int argc,char *argv[])
{
        char buf[256];
        int fd,len;
        while(--argc>0)
        {
                if(access(*++argv,F_OK))
                {
                        fd=open(*argv,O_APPEND,0744);
                        write(fd,"hello world\n",512);
                }
                else
                {
                        fd=open(*argv,O_RDONLY);
                        while(len=read(fd,buf,256))
                                write(1,buf,len);
                }
                close(fd);
        }
}
```

**OUTPUT:**

```
[smk @cseserver3]$ cc ex5a.c
[smk h@cseserver3]$ ./a.out temp.txt
India
America
Australia
```

England

**RESULT:**

Thus the given file has been read and write successfully.

**Exp: 3a**

## SIMULATION OF CAT COMMAND

**AIM:**

To write a 'C' program for simulation of CAT command.

**ALGORITHM:**

**Step 1:** Include the header files essential to simulate CAT command.
**Step 2:** Define Buffer size of 1024 which can be used as the standard size for file and buffer.
**Step 3:** Create a source file with some contents in it.
**Step 4:** Open the source file in READ mode.
**Step 5:** Print the contents of the file that was opened in READ mode.
**Step 6:** Close the file and exit.

**PROGRAM**:

```
#include<stdio.h>
#include<sys/stat.h>
#include<fcntl.h>
#define bs 1024
main()
{
      FILE *f;
      char file1[bs], buf[bs];
      printf("Enter the filename:\n");
      scanf("%s", &file1);
      f=open(file1, O_RDONLY, bs);
      if(read(f, buf, bs)>0)
      {
            printf("%s", buf);
      }
      else
      {
            printf("File does not exits");
      }
      close(f);
}
```

**OUTPUT:**

```
[smk @cseserver3]$ cc ea.c
[smk @cseserver3]$ ./a.out
Enter the filename: college
File does not exits
```

**RESULT:**

Thus CAT command has been executed successfully.

**Exp: 3b**

## SIMULATION OF LS COMMAND

**AIM:**

To write a 'C' program for simulation of LS command.

**ALGORITHM:**

**Step 1:** Include the header files essential to simulate LS command.
**Step 2:** Declare a structure that will be used for accessing the directories.
**Step 3:** If the directory is Empty or Not Found then display a message "Directory not found"
**Step 4:** If the directory exists then print all the entries under the specified directory name.
**Step 5:** Close the directory and exit.

**PROGRAM:**

```
#include<stdio.h>
#include<fcntl.h>
#include<dirent.h>
void main(int argc,char *argv[],char d_name)
{
      DIR *dir;
      struct dirent *ent;
      if((dir=opendir(argv[1]))!=NULL)
      {
            while((ent=readdir(dir))!=NULL)
            {
                  printf("%s\n",ent->d_name);
            }
      }
      else
            printf("Directory does not exits");
}
```

**OUTPUT:**

```
[smk @cseserver3]$ cc prg.c
[smk @cseserver3]$ ./a.out
Directory does not exits
```

**RESULT:**

Thus LS command has been executed successfully.

**Exp: 3c**

## SIMULATION OF GREP COMMAND

**AIM:**

To write a 'C' program for simulation of GREP command.

**ALGORITHM:**

**Step 1:** Include the header files essential to simulate CAT command.
**Step 2:** Create a file with some contents in it.
**Step 3:** Give two choices
( i ) To print the similar pattern word.
( ii ) To print the entire line of the specified pattern.
**Step 4:** Define the two cases using Switch cases.
**Step 5:** Print the lines of similar pattern depending upon the choice of selection.
**Step 6:** Close the file and exit.

**PROGRAM:**

```
#include<stdio.h>
#include<fcntl.h>
#include<sys/stat.h>
#include<string.h>
#define bs 1024
void main()
{
	FILE *f1;
	char *ptr,file1[bs],str[bs],pat[bs];
	printf("Enter the file name :");
	scanf("%s",file1);
	printf("Enter the pattern to be searched :");
	scanf("%s",pat);
	f1=fopen(file1,"r");
	while(!feof(f1))
	{
		fscanf(f1,"%s",str);
		if((strstr(str,pat))!=NULL)
		{
			printf("The substring is %s\n",str);
		}
	}
}
```

**OUTPUT:**

```
[smk @cseserver3]$ cc grep.c
[smk @cseserver3]$ ./a.out
Enter the file name: college
Enter the pattern to be searched: cse
The substring is me
```

**RESULT:**

Thus GREP command has been executed successfully

**Exp: 4a**

## IMPLEMENTATION OF FCFS SCHEDULING ALGORITHM

**AIM:**

To write a 'C' program to implement FCFS Scheduling.

**ALGORITHM:**

**Step 1:** Include the header files for simulating FCFS scheme.
**Step 2:** Declare the variables to calculate the essential aspects of FCFS.
**Step 3:** Initially get the process time and burst time.
**Step 4:** Swap all the processes in such a way that they are arranged in FCFS order.
**Step 5:** Prepare the gantt chart for each process so that their waiting time, turn around time is calculated.
**Step 6:** Calculate the average waiting time and average turn around time and display the result.

**PROGRAM:**

```
#include<stdio.h>
void main()
{
    int i,j,k,n,p[10],bur[10],wat[10],tur[10],ttur=0,twat=0,t;
    float awat,atur;
    printf("\nEnter no. of process :");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("\nEnter the burst time for process %d: ",(i+1));
        scanf("%d",&bur[i]);
        p[i]=i+1;
    }
    printf("\nPROCESS \t BURST TIME \t WAITING TIME \t TURN AROUND TIME\n");
    wat[0]=0;
    for(i=0;i<n;i++)
    {
        wat[i+1]=wat[i]+bur[i];
        tur[i]=wat[i]+bur[i];
    }
    for(i=0;i<n;i++)
    {
        ttur=ttur+tur[i];
        twat=twat+wat[i];
    }
    for(i=0;i<n;i++)
    {
        printf("\n%d\t\t%d\t\t%d\t\t%d\n",p[i],bur[i],wat[i],tur[i]);
```

```
        }
        printf("\nGAANT CHART\n\n");
        for(i=0;i<n;i++)
        {
                printf("--------");
        }
        printf("\n");
        for(i=0;i<n;i++)
        {
                printf("   P%d  |",p[i]);
        }
        printf("\n");
        for(i=0;i<n;i++)
        {
                printf("--------");
        }
        printf("\n");
        for(i=0;i<=n;i++)
        {
                printf("%d\t",wat[i]);
        }
        awat=(float)twat/n;
        atur=(float)ttur/n;
        printf("\nTotal waiting time :%d\n",twat);
        printf("\nTotal turnaround time :%d\n",ttur);
        printf("\nAverage waiting time :%.2f\n",awat);
        printf("\nAverage turn around time :%.2f\n",atur);
}
```

**OUTPUT:**

```
[smk @cseserver3]$ cc prg1.c
[smk @cseserver3]$ ./a.out

Enter no. of process :3

Enter the burst time for process 1: 10

Enter the burst time for process 2: 5

Enter the burst time for process 3: 8

PROCESS        BURST TIME      WAITING TIME    TURN AROUND TIME

 1                 10                  0                  10

 2                  5                  10                 15

 3                  8                  15                 23

GAANT CHART
------------------------
  P1  |  P2  |  P3  |
------------------------
0      10     15     23

Total waiting time :25
Total turnaround time :48
Average waiting time :8.33
Average turn around time :16.00
```

**RESULT:**

Thus the program for FCFS scheduling has been executed successfully.

**Exp: 4 IMPLEMENTATION OF SJF SCHEDULING ALGORITHM**

**AIM:**

To write a 'C' program to implement SJF Scheduling.

**ALGORITHM:**

**Step 1:** Include the header files for simulating SJF scheme.
**Step 2:** Declare the variables to calculate the essential aspects of SJF.
**Step 3:** Initially get the process time and burst time.
**Step 4:** Swap all the processes in such a way that they are arranged in shortest job first (SJF) order.
**Step 5:** Prepare the gantt chart for each process so that their waiting time, turn around time is calculated.
**Step 6:** Calculate the average waiting time and average turn around time and display the result.

**PROGRAM:**

```
#include<stdio.h>
void main()
{
    int i,j,k,n,p[10],bur[10],wat[10],tur[10],ttur=0,twat=0,t;
    float awat,atur;
    printf("\nEnter no. of process :");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("\nEnter the burst time for process %d: ",(i+1));
        scanf("%d",&bur[i]);
        p[i]=i+1;
    }
    for(i=0;i<n;i++)
    {
        for(j=i+1;j<n;j++)
        {
            if(bur[i]>bur[j])
            {
                t=bur[i];
                bur[i]=bur[j];
                bur[j]=t;

                t=p[i];
                p[i]=p[j];
                p[j]=t;
            }
        }
    }
```

```
	printf("\nPROCESS\tBURST TIME \t WAITING TIME \t TURN AROUND TIME\n");
	wat[0]=0;
	for(i=0;i<n;i++)
	{
		wat[i+1]=wat[i]+bur[i];
		tur[i]=wat[i]+bur[i];
	}
	for(i=0;i<n;i++)
	{
		ttur=ttur+tur[i];
		twat=twat+wat[i];
	}
	for(i=0;i<n;i++)
	{
		printf("\n%d\t\t%d\t\t%d\t\t%d\n",p[i],bur[i],wat[i],tur[i]);
	}
	printf("\nGAANT CHART\n\n");
	for(i=0;i<n;i++)
	{
		printf("--------");
	}
	printf("\n");
	for(i=0;i<n;i++)
	{
		printf("   P%d  |",p[i]);
	}
	printf("\n");
	for(i=0;i<n;i++)
	{
		printf("--------");
	}
	printf("\n");
	for(i=0;i<=n;i++)
	{
		printf("%d\t",wat[i]);
	}
	awat=(float)twat/n;
	atur=(float)ttur/n;
	printf("\nTotal waiting time :%d\n",twat);
	printf("\nTotal turnaround time :%d\n",ttur);
	printf("\nAverage waiting time :%.2f\n",awat);
	printf("\nAverage turn around time :%.2f\n",atur);
}
```

**OUTPUT:**

```
[smk@cseserver3]$ cc prg3.c
[smk@cseserver3]$ ./a.out

Enter no. of process :3

Enter the burst time for process 1: 10

Enter the burst time for process 2: 5

Enter the burst time for process 3: 8

PROCESS       BURST TIME      WAITING TIME    TURN AROUND TIME

2                 5                 0                 5

3                 8                 5                 13

1                 10                13                23

GAANT CHART

------------------------
  P2  |  P3  |  P1  |
------------------------
0     5     13    23

Total waiting time :18
Total turnaround time:41
Average waiting time :6.00
Average turn around time:13.67
```

**RESULT:**

Thus the program for SJF scheduling has been executed successfully.

**Exp: 5a**

## IMPLEMENTATION OF PRIORITY SCHEDULING ALGORITHM

**AIM:**

To write a 'C' program to implement PRIORITY Scheduling.

**ALGORITHM:**

**Step 1:** Include the header files for simulating priority scheme.
**Step 2:** Declare the variables to calculate the essential aspects of priority scheme.
**Step 3:** Initially get the process time, arrival time and burst time along with their priorities.
**Step 4:** Execute the process by referring to their priority values.
**Step 5:** Prepare the gantt chart for each process so that their waiting time, turn around time is calculated.
**Step 6:** Calculate the average waiting time and average turn around time and display the result.

**PROGRAM:**

```
#include<stdio.h>
void main()
{
    int i,j,k,n,p[10],pri[10],bur[10],wat[10], tur[10],ttur=0,twat=0,t;
    float awat,atur;
    printf("\nEnter no. of process :");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("\nEnter the burst time for process %d: ",(i+1));
        scanf("%d",&bur[i]);
        p[i]=i+1;
    }
    for(i=0;i<n;i++)
    {
        printf("\nEnter the priority for process %d: ",(i+1));
        scanf("%d",&pri[i]);
    }
    for(i=0;i<n;i++)
    {
        for(j=i+1;j<n;j++)
        {
            if(pri[i]>pri[j])
            {
                t=bur[i];
                bur[i]=bur[j];
                bur[j]=t;
```

```
                    t=p[i];
                p[i]=p[j];
                    p[j]=t;
                t=pri[i];
                pri[i]=pri[j];
                pri[j]=t;
            }
        }
    }
    printf("\nPROCESS\tBURST TIME \t PRIORITY \t WAITING TIME \tTURNAROUND
TIME\n");
        wat[0]=0;
        for(i=0;i<n;i++)
        {
            wat[i+1]=wat[i]+bur[i];
            tur[i]=wat[i]+bur[i];
        }
        for(i=0;i<n;i++)
        {
            ttur=ttur+tur[i];
            twat=twat+wat[i];
        }
        for(i=0;i<n;i++)
        {
            printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d\n",p[i],bur[i],pri[i],wat[i],tur[i]);
        }
        printf("\nGAANT CHART\n\n");
        for(i=0;i<n;i++)
        {
            printf("--------");
        }
        printf("\n");
        for(i=0;i<n;i++)
        {
            printf("   P%d  |",p[i]);
        }
        printf("\n");
        for(i=0;i<n;i++)
        {
            printf("--------");
        }
        printf("\n");
        for(i=0;i<=n;i++)
        {
            printf("%d\t",wat[i]);
        }
        awat=(float)twat/n;
        atur=(float)ttur/n;
        printf("\nTotal waiting time :%d\n",twat);
```

```
        printf("\nTotal turnaround time :%d\n",ttur);
        printf("\nAverage waiting time :%.2f\n",awat);
        printf("\nAverage turn around time :%.2f\n",atur);
}
```

**OUTPUT:**

```
[smk@cseserver3]$ cc prg6.c
[smk@cseserver3]$ ./a.out

Enter no. of process :3

Enter the burst time for process 1: 14

Enter the burst time for process 2: 26

Enter the burst time for process 3: 34

Enter the priority for process 1: 3

Enter the priority for process 2: 1

Enter the priority for process 3: 2

PROCESS      BURST TIME      PRIORITY      WAITING TIME   TURNAROUND TIME

 2                26                1              0                26

 3                34                2              26               60

 1                14                3              60               74

GAANT CHART

------------------------
  P2  |  P3  |  P1  |
------------------------
0      26     60     74

Total waiting time :86

Total turnaround time :160

Average waiting time :28.67

Average turn around time :53.33
```

**RESULT:**

Thus the program for priority scheduling is successfully implemented.

**Exp: 5b IMPLEMENTATION OF ROUND ROBIN SCHEDULING ALGORITHM**

**AIM:**

To write a 'C' program to implement ROUND ROBIN Scheduling.

**ALGORITHM:**

**Step 1:** Include the header files for simulating ROUND ROBIN scheduling scheme.
**Step 2:** Declare the variables to calculate the essential aspects of ROUND ROBIN.
**Step 3:** Initially get the time slice along with process time and burst time.
**Step 4:** Start from the 1st process and execute it for the given time slice and move to 2nd process and so on till all process gets executed for each of the time slice allotted.
**Step 5:** Prepare the gantt chart for each process getting executed within the time slice in a round robin fashion.
**Step 6:** Calculate the waiting time and turn around time along with their averages display the result.

**PROGRAM:**

```
#include<stdio.h>
void main()
{
    int i,x=-1,k[10],m=0,n,t,s=0;
    int a[50],temp,b[50],p[10],bur[10],bur1[10];
    int wat[10],tur[10],ttur=0,twat=0,j=0;
    float awat,atur;
    printf("Enter no. of process :");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Enter the burst time for process %d: ",(i+1));
        scanf("%d",&bur[i]);
        bur1[i]=bur[i];
    }
    printf("\nEnter the slicing time :");
    scanf("%d",&t);
    for(i=0;i<n;i++)
    {
        b[i]=bur[i]/t;
        if((bur[i]%t)!=0)
            b[i]=b[i]+1;
        m=b[i]+m;
    }
    printf("\nGAANT CHART\n\n");
    for(i=0;i<m;i++)
    {
        printf("--------");
    }
```

```
printf("\n");
a[0]=0;
while(j<m)
{
	if(x==n-1)
		x=0;
	else
		x++;
	if(bur[x]>=t)
	{
		bur[x]=bur[x]-t;
		a[j+1]=a[j]+t;
		if(b[x]==1)
		{
			p[s]=x;
			k[s]=a[j+1];
			s++;
		}
		j++;
		b[x]=b[x]-1;
		printf("  P%d   |",x+1);
	}
	else if(bur[x]!=0)
	{
		a[j+1]=a[j]+bur[x];
		bur[x]=0;
		if(b[x]==1)
		{
			p[s]=x;
			k[s]=a[j+1];
			s++;
		}
		j++;
		b[x]=b[x]-1;
		printf("  P%d  |",x+1);
	}
}
printf("\n");
for(i=0;i<m;i++)
	printf("--------");
printf("\n");
for(j=0;j<=m;j++)
	printf("%d\t",a[j]);
for(i=0;i<n;i++)
{
	for(j=i+1;j<n;j++)
	{
		if(p[i]>p[j])
		{
```

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

                    temp=k[i];
                    k[i]=k[j];
                    k[j]=temp;
                }
            }
        }
        for(i=0;i<n;i++)
        {
            wat[i]=k[i]-bur1[i];
            tur[i]=k[i];
        }
        printf("\nPROCESS\tBURST TIME \t WAITING TIME \t TURN AROUND TIME\t");
        for(i=0;i<n;i++)
        {           printf("\n%d\t\t%d\t\t%d\t\t%d\t\n",p[i]+1,bur1[i],wat[i],
            tur[i]);       }
        for(i=0;i<n;i++)
        {
            ttur=ttur+tur[i];
            twat=twat+wat[i];
        }
        awat=(float)twat/n;
        atur=(float)ttur/n;
        printf("\nTotal waiting time:%d",twat);
        printf("\nTotal turnaround time:%d",ttur);
        printf("\nAverage waiting time :%.2f",awat);
        printf("\nAverage turn around time :%.2f",atur);
}
```

**OUTPUT:**

```
[smk@cseserver3]$ cc prg7.c
[smk@cseserver3]$ ./a.out
Enter no. of process :3
Enter the burst time for process 1: 4
Enter the burst time for process 2: 7
Enter the burst time for process 3: 1
Enter the slicing time :2

GAANT CHART
---------------------------------------------------
 P1  | P2  | P3 | P1  | P2  | P2  | P2 |
---------------------------------------------------
0     2     4     5     7     9     11    12
PROCESS   BURST TIME     WAITING TIME   TURN AROUND TIME
1                 4                    3                    7
```

| | | | |
|---|---|---|---|
| 2 | 7 | 5 | 12 |
| 3 | 1 | 4 | 5 |

Total waiting time:12
Total turnaround time:24
Average waiting time :4.00
Average turn around time :8.00

**RESULT:**

Thus program for round robin scheduling is performed.

**Exp: 6 DEVELOPING APPLICATION USING INTERPROCESS COMMUNICATION**

**AIM:**

To write a program to develop application using inter process communication.

**ALGORITHM:**

**Step1**: Start the process.
**Step2**: Enter the limit for Fibonacci series in the parent process.
**Step3**: Display the series in child process.
**Step4**: Stop the process.

**PROGRAM:**

```
#include<stdio.h>
#include<unistd.h>
#include<sys/ipc.h>
#include<sys/io.h>
#include<sys/types.h>
main()
{
        int pid,pfd[2],n,a,b,c;
         if(pipe (pfd)== -1)
        {
                printf("\n error in the pipe connection\n");
                exit(1);
        }
        pid=fork();
        if(pid>0)
        {
                printf("\n parent process");
                printf("\n Fibonacci series\n");
                printf("\n  enter the limit for the series:");
                scanf("%d", &n);
                close(pfd[0]);
                write(pfd[1], &n, sizeof(n));
                close(pfd[1]);
                exit(0);
         }
         else
        {
                close(pfd[1]);
                read(pfd[0], &n , sizeof(n));
                printf("child process");
                a=0;
                b=1;
                close(pfd[0]);
```

```
                printf("\n fibonacci series");
                printf("\n\n %d \n %d",a,b);
               while(n>2)
               {
                        c=a+b;
                         printf("\n %d",c);
                         a=b;
                         b=c;
                         n--;
                   }
          }
}
```

**OUTPUT:**

```
[smk@networkserver ~]$ cc fib.c
[smk@networkserver ~]$ ./a.out
Parent process
Fibonacci series
Enter the limit for the series: 6
Child process
[smk@networkserver ~]$ Fibonacci series
0
1
1
2
3
5
```

**RESULT:**

Thus the program to implement interprocess communication is executed.

**Exp .7 PRODUCER CONSUMER PROBLEM USING SEMAPHORE**

**AIM:**

To implement producer consumer problem using semaphore.

**ALGORITHM:**

1. Union a variable "mysemun" as integer type to implement semaphore.
2. A buffer "sembuf" variable for producer consumer is written
3. In producer, buffer "sembuf" full condition is checked and if it is full then it is displayed.
4. In consumer, buffer "sembuf" empty condition is checked and a message displayed.
5. Producer produces an element sequentially from main().and a message displayed.
6. Consumer consumes an element sequentially from main().and a message displayed.
7. The producer consumer produces messages are displayed till the user defines in the program.

**PROGRAM:**

```
#include<stdio.h>
#include<fcntl.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
#include<sys/shm.h>
#define num 10
#define se 10
#define sf 0
int main()
{
        int rc,pid,semid;
        int shmid,status,i;
        char elem;
        union semun
        {
                int val;
        }mysemun;
        struct sembuf waitempty={se,-1,SEM_UNDO};
        struct sembuf signalempty={se,1,IPC_NOWAIT};
        struct sembuf waitfull={sf,-1,SEM_UNDO};
        struct sembuf signalfull={sf,1,IPC_NOWAIT};
        struct shmid_ds myshmid_ds;
        void *shmptr;
        semid=semget(IPC_PRIVATE,2,0666 | IPC_CREAT);
        mysemun.val=num;
        semctl(semid,se,SETVAL,mysemun);
        mysemun.val=0;
        semctl(semid,sf,SETVAL,mysemun);
        shmid=shmget(IPC_PRIVATE,num,0666 | IPC_CREAT);
```

```
        pid=fork();
        if(pid==0)
        {
                shmptr=shmat(shmid,0,SHM_R);
                for(i=0;i<10;i++)
                {
                        semop(semid,&waitfull,1);
                        elem=*((char *)shmptr+(i%num));
                        printf("consumed element %c\n",elem);
                        semop(semid,&signalempty,1);
                        sleep(1);
                }exit(0);
        }
        else
        {
                shmptr=shmat(shmid,0,SHM_W);
                for(i=0;i<10;i++)
                {
                        semop(semid,&waitempty,1);
                        elem='a'+i;
                        printf("produced element %c\n",elem);
                        *((char *)shmptr+(i%num))=elem;
                        semop(semid,&signalfull,1);
                        sleep(2);
                }
        }
        wait(&status);
        shmctl(shmid,IPC_RMID,&myshmid_ds);
        semctl(semid,se,IPC_RMID,mysemun);
        exit(0);
}
```

**OUTPUT:**

```
[smk@cseserver3]$ cc ex8.c
[smk@cseserver3]$./a.out
produced element a
consumed element a
produced element b
consumed element b
produced element c
consumed element c
produced element d
consumed element d
produced element e
consumed element e
produced element f
consumed element f
produced element g
consumed element g
produced element h
```

consumed element h
produced element i
consumed element i
produced element j
consumed element j

**RESULT:**

Thus the producer consumer problem program using semaphore is written and executed successfully.

**Exp: 8 MEMORY MANAGEMENT SCHEME-I FIRST FIT**

**AIM:**

To write a C program to simulate first fit memory allocation.

**ALGORITHM:**

**Step 1:** Start.

**Step 2:** Maintain an available list of free nodes and each contain the starting and ending address of memory block.

**Step 3:** Maintain a list containing information about the process occupying the memory such as process id, starting and ending address of memory block.

**Step 4:** Insertion

i. Get the process id and amount of memory needed.
ii. Search the available for the first block that is big enough to hold the process.
iii. Make the entry of process id and starting and ending address of process in the occupied list.
iv. Make the updation in the available list.

**Step 5:** Deletion

i. Get the process id of the process to be deleted.
ii. Search the occupied list. If the process found, delete it else display error message.
iii. Create a new node at the appropriate position in the available list.

**PROGRAM:**

```
#include<stdio.h>
struct partition
{
	int size;
	int job,job_size;
};
main()
{
	int i,j,n,k=-1,s[10],temp,job[10];
	struct partition pr[10];
	for(i=0;i<5;i++)
	{
		printf("\nEnter the partition size of process %d",i);
		printf(":\t");
		scanf("%d",&n);
		pr[i].size=n;
		pr[i].job=0;
		pr[i].job_size=0;
		printf("\n");
	}
	for(i=0;i<4;i++)
	{
		printf("\nEnter the job:");
```

```
            scanf("%d",&job[i]);
            printf("\nenter the job size :");
            scanf("%d",&s[i]);
            printf("\n");
        }
        for(i=0;i<4;i++)
        {
            k=-1;
            for(j=0;j<5;j++)
            {
                if((pr[j].size>=s[i]))
                {
                temp=(pr[j].size-s[i]);
                k=j;
                break;
                }
            }
            if(k!=-1)
            {
                pr[k].job=job[i];
                pr[k].job_size=s;
                pr[k].size=temp;
                printf("\npartition%d fits the job:%d",k,job[i]);
                printf("\n");
            }
            else
                printf("\nThe job %d has not been allocated",job[i]);
        }
        printf("\npartition after allocation:\n");
        for(i=0;i<5;i++)
        {
            printf("%d",pr[i].size);
            printf("\n");
        }
}
```

**OUTPUT :**

```
[smk@cseserver3]$ cc ex10b.c
[smk@cseserver3]$./a.out
Enter the partition size of process 0:  100
Enter the partition size of process 1:  500
Enter the partition size of process 2:  200
Enter the partition size of process 3:  300
Enter the partition size of process 4:  600
Enter the job:1
enter the job size :212
Enter the job:2
enter the job size :417
Enter the job:3
enter the job size :112
Enter the job:4
enter the job size :426
partition1 fits the job:1
partition4 fits the job:2
partition1 fits the job:3
The job 4 has not been allocated
partition after allocation:
100
176
200
300
183
```

**RESULT:**

Thus the program is written in C to implement memory allocation strategy in the first fit algorithm basis.

**Exp: 9**

## MEMORY MANAGEMENT SCHEME – II BEST FIT

**AIM:**

To write a C program to simulate best fit memory allocation.

**ALGORITHM:**

**Step 1:** Start.

**Step 2:** Maintain an available list of free nodes and each contain the starting and ending address of memory block.

**Step 3:** Maintain a list containing information about the process occupying the memory such as process id, starting and ending address of memory block.

**Step 4:** Insertion

i. Get the process id and amount of memory needed.
ii. Search the available list for the smallest block that is big enough to hold the process.
iii. Make the entry of process in memory in the occupied list.
iv. Make the updation in the available list.

**Step 5:** Deletion

i. Get the process id of the process to be deleted.
ii. Search the occupied list. If the process found, delete it else display error message.
iii. Update the available list by inserting a free node in to it.

**PROGRAM:**

```
#include<stdio.h>
struct partition
{
        int size;
        int job,job_size,pnum;
};
main()
{
        int i,j,n,k=-1,s[10],temp,job[10];
        struct partition pr[10];
        for(i=0;i<5;i++)
        {
                printf("\nEnter the partition size of process %d",i);
                printf(":\t");
                scanf("%d",&n);
                pr[i].size=n;
                pr[i].job=0;
                pr[i].job_size=0;
                pr[i].pnum=i;
                printf("\n");
        }
        for(i=0;i<4;i++)
        {
```

```
                printf("\nEnter the job:");
                scanf("%d",&job[i]);
                printf("\nenter the job size :");
                scanf("%d",&s[i]);
        }

        for(i=0;i<4;i++)
        {
                k=-1;
                sort(5,pr);

                for(j=0;j<5;j++)
                {
                        if((pr[j].size>=s[i]))
                        {
                                temp=(pr[j].size-s[i]);
                                k=j;
                                break;
                        }
                }

                if(k!=-1)
                {
                        pr[k].job=job[i];
                        pr[k].job_size=s[i];
                        pr[k].size=temp;
                        printf("\npartition%d fits the job:%d",pr[k].pnum,job[i]);
                        printf("\n");
                }
                else
                        printf("\nThe job %d has not been allocated",job[i]);
        }
        printf("\npartition after allocation:\n");
        for(i=0;i<5;i++)
        {
                printf("%d",pr[i].size);
                printf("\n");
        }
}
sort(int num,struct partition pr[])
{
        int i,j,tem;
        for(i=0;i<num;i++)
        {
                for(j=i+1;j<num;j++)
                {
                        if(pr[j].size<pr[i].size)
                        {
                                tem=pr[i].size;
```

```
                                        pr[i].size=pr[j].size;
                                        pr[j].size=tem;
                                        tem=pr[i].pnum;
                                        pr[i].pnum=pr[j].pnum;
                                        pr[j].pnum=tem;
                                }
                        }
                }
```

**OUTPUT:**

```
[smk@cseserver3]$ cc ex10a.c
[smk@cseserver3]$./a.out
Enter the partition size of process 0:  100
Enter the partition size of process 1:  500
Enter the partition size of process 2:  200
Enter the partition size of process 3:  300
Enter the partition size of process 4:  600
Enter the job:1
enter the job size :212
Enter the job:2
enter the job size :417
Enter the job:3
enter the job size :112
Enter the job:4
enter the job size :426
partition3 fits the job:1
partition1 fits the job:2
partition2 fits the job:3
partition4 fits the job:4
partition after allocation:
83
88
88
100
174
```

**RESULT:**

Thus the memory management scheme is implement using best fit algorithm.

**Exp: 10 IMPLEMENTATION OF FILE ALLOCATION TECHNIQUES-CONTIGUOUS**

**AIM:**

To write a C program to implement linked or contiguous file allocation techniques.

**ALGORITHM:**

**Step 1:** Start
**Step 2:** Get the file ID and number of blocks needed by it.
**Step 3:** Get the number of blocks needed by it.
**Step 4:** Allocate the file in contiguous manner.
**Step 5:** Print the value.
**Step 6:** Stop.

**PROGRAM:**

```
#include<stdio.h>
#include<stdlib.h>
int a[20],num=0;
int fid[10],length[10],start[10];
void filedescriptor();
void filedescriptor()
{
        int i;
        printf("\n file id \t starting address \t length \n");
        for(i=0;i<num;i++)
              printf("%d\t\t\t %d\t\t%d\n",fid[i],start[i],length[i]);
}
void display()
{
         int i;
         for(i=0;i<20;i++)
               printf("%2d",i);
        printf("\n");
        for(i=0;i<20;i++)
              printf("%2d", a[i]);
}
int main()
{
         int i,n,k,temp,st,l,id,flag=0,cho;
        for(i=0;i<20;i++)
              a[i]=0;
        printf("\n memory before allocation:\n");
        display();
        while(1)
        {
              printf("\n enter the file id:");
```

```
            scanf("%d",&id);
            printf("\n enter the number of blocks the file occupies:");
            scanf("%d", &l);


           fid[num]=id;
          length[num]=l;
         printf("\n enter the starting address:");
          l1:scanf("%d", &st);
         flag=0;
        if((st+1)>20)
        {
              printf("\n sorry the given memory goes out of space: Enter new starting address ");
              goto l1;
        }
        for(i=st;i<(st+l);i++)
        {
               if(a[i]!=0)
               {
                        flag=1;
                        break;
               }
        }
        if(flag==0)
        {
               start[num]=st;
              for(i=st;i<(st+l);i++)
                        a[i]=id;
        }
        else
        {
              printf("\n sorry the given blocks are already occupied: Enter new starting address");
              goto l1;
        }
        flag=0;
        num++;
       filedescriptor();
       printf("\n memory after allocation \n");
       display();
       printf("\n want to continue? \n1.yes \n2.no");
       scanf("%d",&cho);
       if(cho==2)
             exit(0);
      }
      return 0;
}
```

**OUTPUT:**

```
[smk@networkserver~]$ cc contiguous.c
[smk@networkserver~]$ ./a.out
Memory before allocation:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
0 0 0 0 0 0 0 0 0 0  0   0   0   0   0   0   0   0   0   0
Enter file id: 1
Enter the number of blocks the file occupies: 5
Enter the starting address: 7
FILE ID                 starting address                length
1                         7                             5
Memory after allocation
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
0 0 0 0 0 0 0 1 1 1  1   1   0   0    0  0   0   0   0    0
Want to continue?
1.yes
2.no 1
Enter the file id:2
Enter the number of blocks the file occupies:3
Enter the starting address: 12
FILE ID                 starting address                length
1                         7                             5
2                         12                            3
Memory after allocation
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
0 0 0 0 0 0 0 1 1 1  1   1   2   2   2   0   0   0  0   0
Want to continue?
1.yes
2.no 1
Enter the file id:3
Enter the number of blocks the file occupies:4
Enter the starting address:20
Sorry the given memory location goes out of space. Please give some new starting address: 0
File id                 starting address            length
1                         7                         5
2                          12                       3
3                          0                        4
Memory after allocation
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
3 3 3 3 0 0 0 1 1 1  1   1     2  2   2   0   0  0   0   0
Want to continue?
1.yes
2.no 2
```

**RESULT:**

Thus program to implement file allocation technique is implemented.