| EX.NO :1<br>DATE : 12.12.11 | ***SYSTEM CALLS*** |
|---|---|

***AIM :***

To study the system calls of UNIX operating system and to implement the system calls in high level programming language like c.

***SYSYEM CALLS IN UNIX:***

System calls in UNIX operating system are

| | |
|---|---|
| Fork | execl |
| getpid | exit |
| getppid | opendir |
| sleep | closedir |
| wait | readdir |
| stat | |

***fork():***

Processes initiated by us can also create children in the same manner as the swapper and the process dispatcher did .These children processes are created using the fork() function. It is by forking processes that we can exploit the multitasking capability of UNIX.

**Program 1:**

```
#include<stdio.h>
void main ()
{
     fork();
     printf ("Hello world\n");
}
```

**Output:**

```
Hello world
Hello world
```

**Program 2:**

```
#include<stdio.h>
main()
{
     printf("no fork\n");
     fork();
     printf("one fork\n");
     fork();
     printf("two fork\n");
}
```

**Output:**

```
no fork
one fork
two fork
two fork
one fork
two fork
two fork
```

**Program 3:**

```
#include<stdio.h>
main()
{
     fork();
     fork();
     fork();
     printf("hi\n")
}
```

**Output:**

```
hi
hi
hi
hi
hi
hi
hi
hi
```

***getpid():***

There is a UNIX function getpid() that enables us to get the identification number of a process. pid value will depend on the number of processes already running .It will always be unique. This pid cannot be changed although it may be re-used once the process no longer exists .UNIX itself takes care of this.

**Program 4:**

```
     main();
     {
          int pid;
          pid=getpid();
          printf("process id is %d\n",pid);
     }
```

**Output:**

Process ID is 192.

***getppid():***

A process in UNIX is not a stand alone. There is a parent-child relationship existing between processes. The ID of the parent process can be identified by the function getppid().

**Program 5:**

```
main()
{
     int pid;
     pid=fork();
     if(pid==0)
     {
      printf("Iam the child,my process id is %d\n",getpid());
   printf("the child's parent process id is %d\n",getppid());
     }
     else
     {
          printf("I am the parent , my id is %d\n",getpid());
  printf("the parent's parent process id is %d\n",getppid());
     }
}
```

**Output:**

```
I am the child, my process ID is 8012
The child's parent process ID is 8010
I am the parent, my process ID is 8010
The parent's parent process Id is 82
```

***Sleep():***

A process in UNIX can be made to remain inactive by using sleep() function.

**Program 6:**

```
     main()
     {
       sleep(50);
```

```
        printf("the handsome princess kissed me\n");
      }
```

**Output:**

<after 50 seconds>
The beautiful princess blessed me.

***Orphan and Zombie processes:***

Normally after a fork(), the time slice is given to the child process. Now suppose the parent process were to terminate before the child process, the fatherless child is known as orphan process and is adopted by process dispatcher whose pid is 1.

**Program 7**

```
main()
{
     int pid ;
     pid=fork();
     if(pid==0)
     {
          printf("I am child with pid %d\n",getpid());
          printf("the child's parent pid is %d\n",getppid());
          sleep(20);
          printf("I am child with pid  %d\n",getpid());
          printf("the child's parent pid is %d\n",getppid());
     }
     else
     {
          printf("I am the parent with pid is %d\n",getpid());
          printf("the parent's parent pid is %d\n",getppid());
     }
}
```

**Output:**

```
I am child with pid 890.
The child's parent pid is 888.
I am parent with pid 888.
The parent's parent pid is 884.
I am child with pid 890.
The child's parent pid is 1.
```

**Description :**

Parent's pid of 1 indicates that the dispatches is the parent.

Zombies are the processes that are dead but have not been removed from the process table. Assume that we had a parent process create a child. Both would now have an entry in the process table. Assume further that the child process terminated well before the parent does. Since the parent is yet in action the chills process cannot be removed from the process table. It therefore exists in the twilight zone. And that’s our Zombie.

**Program 8:**

```
main()
{
     int pid;
     pid=fork();
     if(pid==0)
     {
     printf(“I am the child with pid %d\n”,getpid());
     printf(“the child’s parent process id is
%d\n”,getppid());
     }
     else
     {
     printf(“I am the parent wit pid %d\n”,getpid());
     printf(“the parent’s parent is %d\n”,getppid());
     sleep(20);
     printf(“I am the parent wit pid%d\n”,getpid());
     printf(“the parent’s parent is %d\n”,getppid());
     }
}
```

**Output:**

```
I am the child with pid 89.
The child’s parent process id is 87.
I am the parent with pid 87.
The parent’s parent is 85
< after 20 seconds>
I am the parent with pid 87.
The parent’s parent is 85
```

## ***Wait():***

Wait() process suspends the parent process indefinitely till all the child processes gets completed. The call to th wait() function results in a number of things happening. A check is first mode to see if the parent process has any children. If it does not, a -1 is returned by wait(). If the parent process has a child that has terminated, that child’s PID is returned and it is removed from the process table. However if the parent process has a child or children that have not terminated, it is suspended till it receives a signal. The signal is received as soon as the child dies.

**Program 9:**

```
main()
{
     int pid,dip;
     pid=fork():
     if(pid==0)
     {
          printf("1st child's process id is %d\n",getpid());
          printf("1st child dead\n");
     }
     else
     {
          dip=fork();
          if(dip==0)
          {
          printf("2nd child's process id is %d\n",getpid());
          printf("2nd child dead\n");
          }
          else
          {
               sleep(15);
               printf("I am parent and I am dying\n");
          }
     }
}
```

**Output:**

```
1st child's process id is 5369.
First child dead.
2nd child's process id is 5370
Second child dead.
< after 15 seconds>
I am parent and I am dying.
```

***Execl():***

UNIX believes that small and simple is beautiful. Small programs cab be built that handle only one task. That these in turn can call others and those others will call others and those others will call others and so on till we have a synergy of small processes that come to life,execute,bring another process to life and die in this process of creation. It's like a GOTO in BASIC, where execution begins from the line specified in the GOTO and there is no returning. This new process, however has the same PID as the original process . Not only the PID but also the parent process ID, current directory, and file descriptor tables. If any are open- also remain the same.

**Program 10-A:**

```
#include<stdio.h>
main()
{
     printf("before execl id is %d\t\n",getpid());
     printf("parent id is %d\t\n",getppid());
     printf("ececl starts\n");
     sleep(5);
     execl("/home/cs2k8b/cs2k8b20/os/1h","1h",(char*)0);
     printf("wl notbe executrd");
}
```

**Program 10-B:**

```
#include<stdio.h>
main()
{
     printf("after execl id is %d\t\n",getpid());
     printf("parent id is %d \t\n",getppid());
     printf("execl ends\n");
}
```

***Output:***

```
Before execl my ID is 8012
My parent's process id is 8010
Exec starts.
After execl my ID is 8012.
Parent's process id is 8010.
Exec ends.
```

***Opendir, Closedir and Readdir:***

Opendir system call is used to open a directory and readdir is used to read the file name of the file present in the directory. The argument passed for opendir() is the name of the directory to be opened. Closedir() is the system call used to close the directory.

**Program 11:**

```
#include<stdio.h>
#include<dirent.h>
#include<sys/types.h>
main(int argc,char *argv[])
{
     DIR *dir;
     struct dirent *rddir;
     dir=opendir(argv[1]);
     rddir=readdir(dir);
     printf("opened %s\n",argv[1]);
     printf("first entry in the directory %s is 
%s\n",argv[1],rddir->d_name);
     closedir(dir);
}
```

***Output:***

```
./a.out sample
Opened sample
First entry in the directory sample is ..
```

**Program 12:**

```
#include<stdio.h>
#include<string.h>
#include<dirent.h>
main(int argc,char *argv[])
{
     DIR *dir;
     struct dirent *rddir;
     dir=opendir(argv[1]);
     c:
     rddir=readdir(dir);
     printf("%s\n",rddir->d_name);
     if(strcmp(argv[2],rddir->d_name)==0)
     {
               printf("file found\n");
               exit(0);

     }
     if(rddir==NULL)
     {
          printf("file not found\n");
          exit(0);
     }
     else goto c;

     closedir(dir);
}
```

***Output:***

```
./a.out sample b
File found.
```

***Stat():***

Stat is a functioncall which is used to know the status of a file whether it isdeleted or modified.

Stat() call will return a value which will be less than zero,if the file is deleted. It will return the same value as before, if the file is unmodified.

**Program 13:**

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#define max 4
main(int argc,char *argv[])
{
     struct stat fs;
     int j,retval;
     long last_mtime[max+1];
     if((argc<2)||(argc>5))
     {
          printf("usage is %s file name\n",argv[0]);
          printf("max 4 files\n");
          exit(0);
     }

     for(j=1;j<argc;j++)
     {
          retval=stat(argv[j],&fs);
          if(retval<0)
          {
               perror(0);
               exit(0);
          }
          last_mtime[j]=fs.st_mtime;
     }
     sleep(20);
     for(j=1;j<argc;j++)
     {
          retval=stat(argv[j],&fs);
               if(retval<0)
               {
                    printf("file %s  ",argv[j]);
                    printf("has been deleted \n");
                    printf("%d\n",retval);
               }
               if(last_mtime[j]!=fs.st_mtime)
               {
                    printf("file %s ",argv[j]);
                    printf("has been modified \n");
                    printf("%d\n",retval);

               }

          sleep(5);
     }
}
```

***Output:***

```
./a.out a b c d
(modify a and delete b in another terminal)
File a has been modifies
0
File b has been deleted
-1
```

***Exit():***

Exit is the system call that is used to terminate the program.

**Program 14:**

```
     #include<stdio.h>
     main()
     {
          printf("enter a and b :");
          scanf("%d%d",&a,&b);
          if(b==0)
          {
               exit();
          }
          else
          {
               printf("quotient is %d",a/b);
          }
     }
```

***Output:***

```
Enter a and b: 5
0
```

The program will be terminated as b is 0.

***Result:***

Thus various system calls of UNIX operating system are studied and implemented in c.

| | |
|---|---|
| **EX.NO :2**<br>**DATE : 26.12.11** | ***I/O SYSTEM CALLS*** |

## ***AIM :***

To write programs using the I/O system calls of UNIX operating system such as open, read, write.

## ***DESCRIPTION:***

I/O system calls such as open, read and write are used to open, read and write in any files respectively. The syntax for open is

open(name, mode, accessheights)

Various modes available are

- O-WRONGLY
- O-RDONLY

**Program 1:**

```
#include<stdio.h>
#include<fcntl.h>
main()
{
     int fp,pid;
     char ch='A';
     pid=fork();
     if(pid==0)
     {
          fp=open("sarath",O_WRONLY);
          printf("\n In child: ch %c\n",ch);
          ch='B';
          write(fp,&ch,1);
          printf("\n In child: ch after writing %c\n",ch);
          close(fp);
     }
     else
     {
          wait(0);
          printf("%c",ch);
          fp=open("sarath",O_RDONLY);
          read(fp,&ch,1);
          printf("\n in parent : ch after reading 
%c\n",ch);
          close(fp);
          }
     }
```

***Output:***

In child: ch A.
In child: ch after writing B.
In parent: ch after reading B.
(O/P in file ‘sarath’)
B

**Program 2:**

```
     #include<stdio.h>
     #include<fcntl.h>
     main()
     {
          int fp,pid;
          char buff[11];
          fp=open("sar",O_RDONLY);
          pid=fork();
          if(pid==0)
          {
               printf("child begins %d\n",getpid());
               read(fp,buff,10);
               buff[10]='\0';
               printf("child reads %s\n",buff);
               printf("child exiting \n");
          }
          else
          {
               wait(0);
               read(fp,buff,10);
               buff[10]='\0';
               printf("parent reads %s\n",buff);
               printf("parent exittin \n");
          }
     }
```

***Output:***

```
Text in ‘Sar’
Hi this is sample file.
File name is sar.
./a.out
Child begins 7294
Child reads hi this is
Child exiting.
Parent reads sample fi
Parent exiting.
```

**Program 3:**

```
#include<stdio.h>
#include<fcntl.h>
main()
{
     int fp,pid;
     char buff[11];
     fp=open("raj",O_RDONLY);
     pid=fork();
     if(pid==0)
     {
          printf("file handle is %d \n",lseek(fp,01,1));
          read(fp,buff,10);
          buff[10]='\0';
          printf("file handle is %d\n",lseek(fp,01,1));
     }
     else
     {
          wait(0);
          printf("file handle is %d\n",lseek(fp,01,1));
     }
     close(fp);
}
```

***Output:***

```
File handle is 1.
File handle is 11.
File handle is 12.
```

***Result:***

Thus programs using I/O system calls of UNIX operating system are written and executed successfully.

| | |
|---|---|
| **EX.NO :3A**<br>**DATE : 8.01.12** | ***SIMULATING UNIX COMMAND - ls*** |

***AIM :***

To write c program to simulate the UNIX command IS.

***Algorithm:***

1. Start the program.
2. Open the directory using opendir() system call.
3. Read the file in the directory using readdir() system call and print the name of the file.
4. Continue 3 until NULL is read.
5. Close the directory using closedir() system call.
6. Stop the program.

**Program :**

```
#include<stdio.h>
#include<sys/types.h>
#include<dirent.h>
main(int argc,char *argv[])
{
     DIR *dir;
     struct dirent *rddir;
     dir=opendir(argv[1]);
     while((rddir=readdir(dir))!=NULL)
     {
          printf("%s\n",rddir->d_name);
     }
     closedir(dir);
}
```

***Output:***

```
./a.out sample
A
C
..
.
B
```

***<u>Result:</u>***

Thus the c program to stimulate IS is written and executed successfully.

| | |
|---|---|
| **EX.NO :3B**<br>**DATE : 8.01.12** | ***SIMULATING UNIX COMMAND - grep*** |

***AIM :***

To write c program to simulate Grep.

***Algorithm:***

1. Start the program.
2. Open the target file.
3. Read the pattern from the user.
4. In every line, check for the presence of pattern.
5. If the pattern is presen, print the line.
6. Continue step 4 and 5 until the end of the file.
7. Stop the program.

**Program :**

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

main()
{
     char arr[80];
     FILE *fp;
     char pat[10];
     int n,i,f,line;
     fp=fopen("grep","r");
     printf("enter the pattern :");
     f=0;
     line=0;
     scanf("%s",pat);
     n=strlen(pat);
     fgets(arr,80,fp);


     while(!feof(fp))
     {

          for(i=0;i<80;i++)
          {
               if(arr[i]!='\0')
               {
                    if(f<=(n-1))
                    {
```

```
                              if(arr[i]==pat[f])
                              f++;
                         }
                    }
                    else
                         break;
                    }
                    if(f==n)
                    {
                         line++;
                         printf("line = %d\t",line);
                         printf("%s\n",arr);
                    }
                    else
                    {
                         line++;
                    }
               fseek(fp,01,80);
               fgets(arr,80,fp);
               f=0;

          }



     }
```

***Output:***

```
Enter the pattern: a
Line=2 cat
Line=6 praveen
< contents of file>
Is
Cat
His
Dog
Miss
Praveen
Sis
```

***Result:***

Thus the c program to stimulate grep is written and executed successfully.

| EX.NO :4A<br>DATE :22.1.12 | ***CPU SCHEDULING***<br>***FIRST COME, FIRST SERVE (FCFS)*** |
|---|---|

## ***AIM***

To write a C program to perform CPU scheduling by the FCFS algorithm.

## ***ALGORITHM:***

1. Start the program.
2. Declare and define the structure to create processes.
3. Input the number of process.
4. Create the process table with the specified number of process.
5. Input the process name , arrival time and burst time of the process.
6. The finishing time,turn-around time and wait time for first process are printed as it is.
7. Using the for loop calculate the finishing ,turn-around and waiting time using the previous values of the other processes.
8. Print the result in the form of a process table.
9. Stop the program.

## ***PROGRAM:***

```
#include<stdio.h>
struct process_tbl
{
      char name[5];
      int at,bt,wt,tat,ft;
};


main()
{
      int n,i,j;
      float tot_ta=0,tot_w=0;
      printf("Enter the number of processes");
      scanf("%d",&n);
      struct process_tbl tb[n];

      for(i=0;i<n;i++)
      {
          printf("Enter process name ");
          scanf("%s",&tb[i].name);
          printf("Enter arrival time and burst time ");
          scanf("%d%d",&tb[i].at,&tb[i].bt);
          tb[i].ft=tb[i].wt=tb[i].tat=0;
      }

      printf("The table is\n");
      printf("Name\tArrival time\tBurst time\n");
      for(i=0;i<n;i++)
      {
           printf("%s\t\t%d\t\t%d\n",
                          tb[i].name,tb[i].at,tb[i].bt);
       }
       struct process_tbl temp;

      for(i=1;i<n;i++)
      {
           for(j=i+1;j<n;j++)
           {
                if(tb[j].at<tb[i].at)
                {
                      temp=tb[i];
                      tb[i]=tb[j];
                      tb[j]=temp;
                }
          }
      }
      tb[0].ft=tb[0].bt;
      tb[0].wt=0;
      tb[0].tat=tb[0].ft-tb[0].at;

      for(i=1;i<n;i++)
      {
           tb[i].ft=tb[i].bt+tb[i-1].ft;
           tb[i].wt=tb[i-1].ft-tb[i].at;
           tb[i].tat=tb[i].ft-tb[i].at;
      }
      printf("The table is\n");
      printf("Name\tArrival time\tBurst time\tFinish time\tWait
time\tT A time\n");

      for(i=0;i<n;i++)
      {
            tot_w=tb[i].wt+tot_w;
            tot_ta=tb[i].tat+tot_ta;
  printf("
%s\t\t%d\t\t%d\t\t%d\t\t%d\t%d\n",tb[i].name,tb[i].at,tb[i].bt,t
b[i].ft,tb[i].wt,tb[i].tat);
      }
      printf("Average waiting time=%f\n",(tot_w/n));
      printf("Average turn around time=%f\n",(tot_ta/n));
}
```

***OUTPUT:***

```
Enter the number of processes 3
Enter process name         p1
Enter arrival time and burst time        0
5
Enter process name         p2
Enter arrival time and burst time        2
3
Enter process name         p3
Enter arrival time and burst time        4
4
The table is:
Name    Arrival time            burst time
 p1             0               5
 p2             2               3
 p3             4               4

The table is:
Name Arrival time Burst time Finish time   Wait time   TA time
 p1         0            5                 5            0              5
 p2         2             3                 8            3              6
 p3         4             4                12            4              8
```

***RESULT:***

Thus a C program to implement FCFS has been written, executed and output is verified.

| | |
|---|---|
| **EX.NO :4B**<br>**DATE :22.1.12** | ***CPU SCHEDULING***<br>***SHORTEST JOB FIRST (SJF)*** |

## ***AIM***

To write a C program to perform CPU scheduling by the SJF algorithm

## ***ALGORITHM:***

1. Start the program.
2. Declare and define the structure for creating the processes.
3. Input the number of process.
4. Create the process table with the specified number of process.
5. Input the process name , arrival time and burst time of the process.
6. Sort the table of process based on the Burst time so that process with low burst time is processed first.
7. The finishing time, turn-around time and wait time for first process are printed as it is.
8.. Using the for loop calculate the finishing ,turn-around and waiting time using the previous values of the other processes.
9. Print the result in the form of a process table.
10 .Stop the program.

## ***PROGRAM:***

```
#include<stdio.h>
struct process_tbl
{
      char name[5];
      int at,bt,wt,tat,ft;
};

main()
{
     float tot_w=0,tot_ta=0;
      int n,i,j;
      printf("Enter the number of processes");
      scanf("%d",&n);
      struct process_tbl tb[n];

      for(i=0;i<n;i++)
     {
          printf("Enter process name ");
```

```
            scanf("%s",&tb[i].name);
            printf("Enter arrival time and burst time ");
           scanf("%d%d",&tb[i].at,&tb[i].bt);
                  tb[i].ft=tb[i].wt=tb[i].tat=0;
       }

       printf("The table is\n");
      printf("Name\tArrival time\tBurst time\n");
       for(i=0;i<n;i++)
             printf("
%s\t\t%d\t\t%d\n",tb[i].name,tb[i].at,tb[i].bt);
      //sorting
      struct process_tbl temp;

       for(i=1;i<n;i++)
      {
             for(j=i+1;j<n;j++)
             {
                   if(tb[j].bt<tb[i].bt)
                   {
                        temp=tb[i];
                       tb[i]=tb[j];
                       tb[j]=temp;
                   }
                  else if(tb[j].bt==tb[i].bt)
                  {
                      if(tb[j].at<tb[i].at)
                       {
                          temp=tb[i];
                           tb[i]=tb[j];
                           tb[j]=temp;
                       }
                  }
             }
       }
       tb[0].ft=tb[0].bt;
       tb[0].wt=0;
       tb[0].tat=tb[0].ft-tb[0].at;

       for(i=1;i<n;i++)
       {
             tb[i].ft=tb[i].bt+tb[i-1].ft;
             tb[i].wt=tb[i-1].ft-tb[i].at;
             tb[i].tat=tb[i].ft-tb[i].at;
       }
       printf("The complete table is\n");
       printf("Name\tArrival time\tBurst time\tFinish time\tWait
time\tT A time\n");

       for(i=0;i<n;i++)
       {
             tot_w=tb[i].wt+tot_w;
```

```
          tot_ta=tb[i].tat+tot_ta;
  printf("
%s\t\t%d\t\t%d\t\t%d\t\t%d\t%d\n",tb[i].name,tb[i].at,tb[i].bt,t
b[i].ft,tb[i].wt,tb[i].tat);
     }
      printf("The average waititng time=%d\n",(tot_w/n));
      printf("Average turn around time=%d\n",(tot_ta/n));
}
```

***OUTPUT:***

```
Enter the number of processes 3
Enter process name        p1
Enter arrival time and burst time       0
5
Enter process name        p2
Enter arrival time and burst time       2
4
Enter process name        p3
Enter arrival time and burst time       4
2


The table is:
Name   Arrival time          burst time
 p1          0           5
 p2          2           4
 p3          4           2


The table is:
Name Arrival time Burst time Finish time   Wait time   TA time
 p1        0          5                   5              0          5
 p3        4           2             7            1            3
 p2        2           4             11           5            9
```

***RESULT:***

Thus a C program to implement the SJF of the specified number of process is executed and output is verified.

| | |
|---|---|
| **EX.NO :4C**<br>**DATE :29.1.12** | ***CPU SCHEDULING***<br>***PRIORITY SCHEDULING*** |

## ***AIM***

To write a C program to perform CPU scheduling by the priority algorithm

## ***ALGORITHM:***

1. Start the program.
2. Declare and define the structure for creating the processes.
3. Input the number of process.
4. Create the process table with the specified number of process.
5. Input the process name , arrival time, priority and burst time of the process.
6. Sort the table of process based on the Burst time so that process with low burst time is processed first. Then sort using priority.
7. The finishing time, turn-around time and wait time for first process are printed as it is.
8.. Using the for loop calculate the finishing ,turn-around and waiting time using the previous values of the other processes.
9. Print the result in the form of a process table.
10 .Stop the program.

## ***PROGRAM:***

```
#include<stdio.h>
struct process_tbl
{
      char name[5];
      int at,bt,wt,tat,ft,priority;
};

main()
{
     float tot_w=0,tot_ta=0;
      int n,i,j;
      printf("Enter the number of processes");
      scanf("%d",&n);
      struct process_tbl tb[n];

      for(i=0;i<n;i++)
     {
           printf("Enter process name ");
           scanf("%s",&tb[i].name);
           printf("Enter arrival time ,priority and burst time
");
          scanf("%d%d",&tb[i].at&tb[i].priority,&tb[i].bt);
                 tb[i].ft=tb[i].wt=tb[i].tat=0;
      }

      printf("The table is\n");
     printf("Name\tArrival time\tBurst time\n");
      for(i=0;i<n;i++)
            printf("
%s\t\t%d\t\t%d\n",tb[i].name,tb[i].at,tb[i].bt);
     //sorting
     struct process_tbl temp;

      for(i=1;i<n;i++)
     {
            for(j=i+1;j<n;j++)
            {
                  if(tb[j].bt<tb[i].bt)
                  {
                       temp=tb[i];
                      tb[i]=tb[j];
                      tb[j]=temp;
                  }
                 else if(tb[j].bt==tb[i].bt)
                 {
                     if(tb[j].at<tb[i].at)
                      {
                         temp=tb[i];
                          tb[i]=tb[j];
                          tb[j]=temp;
                      }
                 }
               if(tb[j].priority<tb[i].priority)
               {
                   temp=tb[i];
                   tb[i]=tb[j];
                   tb[j]=temp;
               }
            }
      }
      tb[0].ft=tb[0].bt;
      tb[0].wt=0;
      tb[0].tat=tb[0].ft-tb[0].at;

      for(i=1;i<n;i++)
      {
            tb[i].ft=tb[i].bt+tb[i-1].ft;
            tb[i].wt=tb[i-1].ft-tb[i].at;
            tb[i].tat=tb[i].ft-tb[i].at;
      }
      printf("The complete table is\n");
      printf("Name\tArrival time\tBurst time\tFinish time\tWait
time\tT A time\n");

      for(i=0;i<n;i++)
      {
            tot_w=tb[i].wt+tot_w;
           tot_ta=tb[i].tat+tot_ta;
  printf("
%s\t\t%d\t\t%d\t\t%d\t\t%d\t%d\n",tb[i].name,tb[i].at,tb[i].bt,t
b[i].ft,tb[i].wt,tb[i].tat);
     }
      printf("The average waititng time=%d\n",(tot_w/n));
      printf("Average turn around time=%d\n",(tot_ta/n));
}
```

***OUTPUT:***

```
Enter the number of processes 4
Enter process name          p1
Enter arrival time ,priority and burst time     0
3
7
Enter process name          p2
Enter arrival time, priority and burst time     3
2
5
Enter process name          p3
Enter arrival time, priority and burst time     6
1
4
Enter process name          p4
Enter arrival time, priority and burst time     9
4
2



The table is:
Name   Arrival time          burst time priority
 p1          0          5          3
 p2          3          2          2
 p3          6          4          1
p4           9          7          4
```

```
The table is:
Name          Arrival time          Burst time  Finish time   Wait time            TA
time
 p1          0             5             3            7             0        7
 p2          3             2             2            16            8        13
 p3          6             4             1            11            1        5
 p4          9             7             4            18            7        9
```

***RESULT:***

Thus a C program to implement the priority scheduling of the specified number of process is executed and output is verified.

| EX.NO :5<br>DATE :29.1.12 | ***CPU SCHEDULING***<br>***ROUND ROBIN SCHEDULING*** |
|---|---|

***AIM***

To write a c program for the implementation of cpu scheduling using round robin algorithm.

***ALGORITHM:***

1. Start the program
2. Get the number of process and corresponding arrival time and burst time.
3. Get the time quantum
4. For each process, allot the process time of given time quantum until it becomes zero and if it is less than given time quantum, take that burst time itself and make it to zero.
5. Compute the finish time of each process
6. Using finish time, compute turn around time form which waiting time can be calculated.
7. Print the resultant table along with average waiting and turn around time.
8. Stop the program

***PROGRAM:***

```
#include<stdio.h>
struct  process
{
     char name[5];
     int at,bt,wt,tat,ft;
};
main()
     {
          int i,j,k,m,n,tq,b[10],rr[10][10],count[10];
          int max=0;
          float tot_w=0, tot_ta=0;
          printf("enter the number of processes:");
          scanf("%d",&n);
          struct process p[n];
          for(i=1;i<=n;i++)
          {
               printf("enter the name,at,bt:");
               scanf("%s%d%d",p[i].name,&p[i].at,&p[i].bt);
          }
          for(i=1;i<=n;i++)
          {
               b[i]=a[i].bt;
               if(max<b[i])
               max=b[i];
               wt[i]=0;
          }
     printf("enter the time quantum:");
     scanf("%d",&tq);
     m=max/tq+1     //to find the dimension of the rr array
     for(i=1;i<=n;i++)   //initialize rr array
     {
          for(j=1;j<=m;j++)
               {
                    rr[i][j]=0;
               }
     }
               //placing value in rr array
               i=1;
     while(i<=n)
     {
          j=1;
          while(b[i]>0)
               {
                    if(b[i]>=tq)
                         {
                              b[i]=b[i]-tq;
                              rr[i][j]=tq;
                              j++;
                         }
                    else
                         {
                              rr[i][j]=b[i];
                              b[i]=0;
                              j++;
                         }
               }
          count[i]=j-1;
          i++;
     }
     //calculating finish time
     for(k=1;k<=n;k++)
          {
               for(i=1;i<=count[k];i++)
                    {
                         for(j=1;j<=n;j++)
                              {
                                   p[k].ft=p[k].ft+rr[j][i];
                                   if(j==k && rr[j][i]<tq)
                                        goto next;
                              }
                    }
                         next:
          }
     for(i=1;i<=n;i++)
          {
```

```
                p[i].tat=p[i].ft-p[i].at;
                p[i].wt=p[i].tat-p[i].bt;
           }
      printf("resultant table is:");
      printf("NAME\tAT\tBT\tFT\tWT\tTAT\n");
      for(i=1;i<=n;i++)
           {
                printf("%s%d%d%d%d%d",p[i].name,p[i].at,p[i].bt,p
           [i].ft,p[i].wt,p[i].tat);
                tot_w=tot_w+p[i].wt;
                tot_ta=tot_ta+p[i].tat;
           }
      printf("avg wt is %f", (tot_w/n));
      printf("avg tat is %f",(tot_ta/n));
}
```

***OUTPUT:***

```
enter the number of processes: 4
enter the name, at, bt:
p1    0    7
p2    3    5
p3    6    4
p4    9    2
resultant table is
NAME      AT   BT   FT   WT   TAT
p1        0    7    18   11   18
p2        3    5    16   8    13
p3        6    4    17   7    11
p4        9    2    11   0    2

avg wt is 6.5
avg tat is  11
```

***RESULT:***

Thus the c program for the implementation of cpu scheduling using round robin algorithm has been executed successfully.

| | |
|---|---|
| **EX.NO :6** | ***MEMORY MANAGEMENT SCHEME II*** |
| **DATE :05.02.12** | ***MEMORY ALLOCATION STRATEGIES*** |

## ***AIM:***

To write a C program to perform memory allocation strategies.

## ***ALGORITHM:***

1. Start the program.
2. Declare and define the structure for process and the partitions.
3. Read the number of process value n and allocate memory for them.
4. Read the process name and size required for the process and also input the memory
   partition size.
5. Perform First fit,Best fit and Worst fit based on the choice entered.
6. First fit: Allocate the first partition that suits for the process by checking all the
   partition size.
7. Best fit: Allocate partition for the process by checking the partition and process size
   that is accurate and exact by sorting the partition and perform first fit.
8. Worst fit: Allocate maximum partition size for the process by sorting the partition in
   decreasing order and perform first fit.
9. Print the result with the partition allotted for each process.
10. Stop the program.

## ***PROGRAM:***

```
#include<stdio.h>
#include<string.h>
int n,m;
struct process
{    char name[5];
     int na,f,size;
};
struct part
{    char name[5];
     int f,psize;
};
```

```
     void firstfit(struct process p[],struct part pa[],int i,int
j)
     {         if(p[i].f==0 && pa[j].f==0)
               {  if(p[i].size<=pa[j].psize)
                    {    p[i].na=pa[j].psize;
                         p[i].f=1;
                         pa[j].f=1;
                    }
                 else
                    firstfit(p,pa,i,j+1);
               }

     }
     void tsort(struct part pa[])
     {    int i,j;struct part t;
          for(i=0;i<m;i++)
           for(j=i+1;j<m;j++)
             if(pa[i].psize>pa[j].psize)
             { t=pa[i];
               pa[i]=pa[j];
               pa[j]=t;
             }
     }
     void sort(struct part pa[])
     {    int i,j;struct part t;
          for(i=0;i<m;i++)
           for(j=i+1;j<m;j++)
            if(pa[i].psize<pa[j].psize)
            {  t=pa[i];
               pa[i]=pa[j];
               pa[j]=t;
            }
     }
     main()
     {    int s,i,j;
          printf("\nEnter no.of process");scanf("%d",&n);
          printf("\nEnter no. of partition");scanf("%d",&m);
          struct process p[n];struct part pa[m];
          for(i=0;i<n;i++)
          {    printf("\nEnter the process name and size");
               scanf("%s%d",&p[i].name,&p[i].size);
               p[i].f=0;
          }
          for(i=0;i<m;i++)
          {    printf("\nEnter the partition size");
               scanf("%d",&pa[i].psize);
               pa[i].f=0;
          }
          printf("\n1.First fit\t2.Best Fit\t3.Worst fit\nEnter
choice:");
          scanf("%d",&s);
           switch(s)
           {
                 case 1:printf("\nFirst fit\n");
                for(i=0;i<n;i++)
                 for(j=0;j<m;j++)
                    firstfit(p,pa,i,j);
                 break;
            case 2:printf("\nBest fit\n");
                tsort(pa);
                for(i=0;i<n;i++)
                  for(j=0;j<m;j++)
                    firstfit(p,pa,i,j);
                break;
            case 3:printf("\nWorst fit\n");
                sort(pa);
                for(i=0;i<n;i++)
                 for(j=0;j<m;j++)
                    firstfit(p,pa,i,j);
                 break;
           default: printf("\nInvalid choice");exit(0);
          }
         for(i=0;i<n;i++)
          if(p[i].f==1)
            printf("\n%s
Memory:%d\tPartition:%d",p[i].name,p[i].size,p[i].na);
           else
          printf("\n%s-Partition not allocated",p[i].name);
     }
```

***OUTPUT:***

```
Enter no. of process:4
Enter no. of partition:5
Enter the process name and size:p1
215
Enter the process name and size:p2
100
Enter the process name and size:p3
300
Enter the process name and size:p4
415
Enter the partition size:100
Enter the partition size:500
Enter the partition size:600
Enter the partition size:400
Enter the partition size:300

1.First fit     2.Best fit     3.Worst fit   Enter choice:2
```

```
Best fit:
p1      Memory:215          Partition:300
p2      Memory:100          Partition:100
p3      Memory:300          Partition:400
p4      Memory:415          Partition:500
```

***<u>RESULT:</u>***

Thus, a C program to perform memory allocation strategies is executed and verified.

| | |
|---|---|
| **EX.NO :7**<br>**DATE :12.02.12** | ***PRODUCER – CONSUMER PROBLEM USING SEMAPHORE*** |

## ***AIM***

To write a c program to implement the producer-consumer problem using semaphores.

## ***Algorithm:***

1. Start the program
2. Declare global variables representing buffer and semaphore for mutex,full and empty.
3. Define the functions wait and signal which respectively decrement and increment the variable passed to them.
4. Define producer function which places item in buffer(increment buffer variable) after invoking wait on mutex and empty. After its task is done it invokes signal on mutex and full.
5. Similarly consumcer function removes item from buffer(decrement buffer variable) after invoking wait on mutex and full. After its task is done it invokes signal on mutex and empty.
6. Using switch case based on user input call either producer or consumer function.
7. Track buffer contents and display appropriate message when buffer is full or empty.
8. Stop the program

## ***Program:***

```
#include<stdio.h>
int buf=0,i=0,j=0,n;
int mutex=0,empty=0,full=0,ch;
void wait(int s)
{
     if(s<=0)
     s--;
}
void signal(int s)
{
     s++;
}
void producer(int mutex,int empty,int full)
{
     wait(mutex);
     wait(empty);
     i++;
     buf++;
     printf("Producer:%d\n",i);
     signal(full);
     signal(mutex);
}
void consumer(int mutex,int empty,int full)
{
     wait(mutex);
     wait(full);
     j++;
     buf--;
     printf("Consumer:%d\n",j);
     signal(empty);
     signal(mutex);
}
main()
{
     int ch;
     printf("Enter the buffer size\n");
     scanf("%d",&n);
     do
          {
               printf("Enter choice\n");
               printf("1.Producer\n2.Consumer\n3.Exit\n");
               scanf("%d",&ch);
               switch(ch)
               {
                    case 1:if(buf<n)
                              producer(mutex,empty,full);
                           else
                              printf("Buffer is full\n");
                              break;
                    case 2:if(buf>0)
                              consumer(mutex,empty,full);
                           else
                              printf("Buffer is empty\n");
                              break;
                    case 3:break;
                    default:printf("Invlid choice\n");
               }
          }while(ch!=3);
}
```

***Output:***

```
Enter buffer size 5
1. Producer
2.Consumer
3.Exit
Enter choice
1
```

```
1
1. Producer
2.Consumer
3.Exit
Enter choice
1
2
1. Producer
2.Consumer
3.Exit
Enter choice
1
3
1. Producer
2.Consumer
3.Exit
Enter choice
2
2
1. Producer
2.Consumer
3.Exit
Enter choice
2
1
1. Producer
2.Consumer
3.Exit
Enter choice
2
Buffer is empty
1. Producer
2.Consumer
3.Exit
Enter choice
1
1
1. Producer
2.Consumer
3.Exit
Enter choice
1
```

```
2
1. Producer
2.Consumer
3.Exit
Enter choice
3
```

***<u>Result:</u>***

Thus the c program to implement the producer-consumer problem using semaphores was executed successfully and the output was verified.

| EX.NO :8<br>DATE :19.02.12 | ***INTER PROCESS COMMUNICATION USING PIPES*** |
|---|---|

## ***AIM:***

To implement inter-process communication using pipes.

## ***DESCRIPTION:***

A pipe is a one way communication channel. Input from one end becomes output at the other end. By using pipes we can have characters being passed from one task to another. In a program, a pipe is created using the system call pipe().It is passed as an array with two elements as parameter. The pipe ( ) function will return in these two elements two file descriptors.

**Program 1:**

In this program, we pass the array p to pipe (),i.e. the starting address of this array. And the values of the two file descriptors which are returned in this array are then printed. The first file descriptor is where data is read from (input end), the second file is for writing (output end).These file descriptors are common or public to all processes.

**Source Code:**

```
 #include<stdio.h>
main()
{
 int p[2];
 pipe(p);
 printf("%d\n%d",p[0],p[1]);
}
```

**Output:**

```
2
3
```

**Program 2:**

There is a limit to how many pipes we can have at one time. The following program tells how many pipes can be open at one time.

**Source Code:**

```
#include<stdio.h>
main()
{
 int p[2],retval,i=0;
 while(1)
 {
  retval=pipe(p);
  if(retval==-1)
  {
   printf("Limit:%d",i);
   break;
  }
  i++;
  printf("%d\n%d\n",p[0],p[1]);
}
}
```

**Output:**

```
Limit:1
Limit:2
…
…
…
Limit:511
```

In the above program, it will print the value of variable I which tells us how many pipes can be open at one time.

**Program 3:**

In this program, we fork a child process. In both the parent and child processes we print the value of the elements in the array p. Since a fork ( ) creates an identical process with the lines of code and variables matching, the pipe ( ) function returns two copies of the file descriptors. But the pipe itself is not forked. It remains one. Both the child and parent processes will print the same file descriptor values, thereby proving that the pipe is shared over processes.

**Source Code:**

```
 #include<stdio.h>
main()
{
 int p[2],pid;
 pipe(p);
 pid=fork();
 if(pid==0)
     printf("Child:%d\n%d\n",p[0],p[1]);
 else
     printf("Parent:%d\n%d\n",p[0],p[1]);
}
```

**Output:**

```
Child:2
3
 Parent:2
 3
```

**Program 4:**

This program explains how pipe in and pipe out takes place. We define three pointers to characters msg1,msg2,msg3.A call to pipe() with p as parameter results in two file descriptors being returned to the elements 0 and 1 of the array p. This program demonstrates that any write to the write end will immediately be reflected at the read end.

**Source Code:**

```
#include<stdio.h>
#define SIZE 25
main()
{
 char *msg1="SVCE,CSE,SEM4,BATCH1";
 char *msg2="SVCE,CSE,SEM4,BATCH2";
 char *msg3="SVCE,CSE,SEM4,BATCH3";
 char buffer[SIZE];
 int p[2],i;
 pipe(p);
 write(p[1],msg1,SIZE);
 write(p[1],msg2,SIZE);
 write(p[1],msg3,SIZE);
 for(i=0;i<3;i++)
 {
  read(p[0],buffer,SIZE);
  printf("%s\n",buffer);
 }
}
```

**Output:**

```
SVCE,CSE,SEM4,BATCH1
SVCE,CSE,SEM4,BATCH2
SVCE,CSE,SEM4,BATCH3
```

**Program 5:**

In this program we have two calls to the write();one in the parent process where it really belongs, and one in the child process .Both write the message to the write end of the pipe. Only the child process prints the contents of the read end.

**Source Code:**

```
#include<stdio.h>
#define SIZE 25
main()
{
 char* msg="OS";
 char buffer[SIZE];
 int p[2],i,pid;
 pipe(p);
 pid=fork();
 if(pid==0)
 {
  write(p[1],msg,SIZE);
  for(i=0;i<=1;i++)
  {
   read(p[0],buffer,SIZE);
   printf("%s\n",buffer);
  }
 }
 else
     write(p[1],msg,SIZE);
 exit(0);
}
```

**Output:**

```
OS
OS
```

**Program 6:**

Pipes are meant for one way processes. To ensure that only a one way process takes place we need to take certain precautions. In the next program we are closing the read end related to the parent process and the write end that is related to the child process. The advantage of doing is that two file descriptors which would have been otherwise unused are now given back to the system.

**Source Code:**

```
#include<stdio.h>
#define SIZE 25
main()
{
 char* msg="OS";
 char buffer[SIZE];
 int p[2],i,pid;
 pipe(p);
 pid=fork();
 if(pid>0)
 {
  close(p[1]);
  write(p[1],msg,SIZE);
 }
 else
 {
  close(p[0]);
  read(p[0],buffer,SIZE);
  printf("%s\n",buffer);
 }
 exit(0);
}
```

**Output:**

```
OS
```

**Program 7:**

In this program, we make a call to pipe ( ) function, which returns two file descriptors. A fork ( ) create a child process. Now each process has a copy of the two file descriptors. The child process, here terminates immediately as it is given the time slice first and by default both the read and write end of the child process are closed. Now, we only have parent process. The parent process can do both read and write operation since we haven't closed one of these ends. But since we have specified the process can do only a read and also we have not mentioned its write end explicitly, the read end keeps waiting and the process hangs.

**Source Code:**

```
#include<stdio.h>
main()
{
 int p[2],pid;
 int retval;
 char *buff="hello";
 char inbuff[5];
 pipe(p);
 pid=fork();
 if(pid==0)
     printf("Child Exiting\n");
 else
 {
  close(p[1]);
  retval=read(p[0],inbuff,5);
  printf("Value returned %d\n",retval);
 }
 }
```

**Output:**

```
Child exiting
```

```
Value returned:5
```

**Program 8:**

In this program we first close the write end related to the child process and then put this process to sleep. Thus the time slice is handed to the parent processes. In the parent process, we do not close off the write end. As a result, the read ( ) is blocked. After 4 seconds, the child process executes and immediately terminates. So the time slice is handed back to the parent process, where the read ( ) is waiting expectantly.

**Source Code:**

```
#include<stdio.h>
main()
{
 int p[2],pid;
 int retval;
 char *buff="hello";
 char inbuff[5];
 pipe(p);
 pid=fork();
 if(pid==0)
 {
  close(p[1]);
  sleep(4);
  printf("Child Exiting\n");
 }
 else
 {
  retval=read(p[0],inbuff,5);
  printf("Value returned %d\n",retval);
 }

}
```

**Output:**

```
….
Child exiting
```

**Program 9:**

In this program, closing the write end in both the child and parent processes, however results in the parent terminating if the child has been put to sleep ( ).A close of the write end is first end is performed in the child process, after which the time slice is being handed to the parent process. Here, too, we close the write end. Since both ends are closed and the time slice is with the parent process, it terminates immediately. After the 5 seconds, it is the child process's turn to terminate.

**Source Code:**

```
#include<stdio.h>
main()
{
 int p[2],pid;
 int retval;
 char *buff="hello";
 char inbuff[5];
 pipe(p);
 pid=fork();
 if(pid==0)
 {
  close(p[1]);
  sleep(4);
  printf("Child Exiting\n");
 }
 else
 {
  close(p[1]);
  retval=read(p[0],inbuff,5);
  printf("Value returned %d\n",retval);
 }

}
```

**Output:**

```
Value returned:5
…
Child exiting
```

**Program 10:**

In the next program, we first put the child process to sleep. Thus the time slice is handed to the parent process. In it we close the read end .but the write ( ) will not return an error as the read end of the child process is not yet open. The parent process terminates .And in this time the child process too terminates.

**Source Code:**

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

```
main()
{
 int p[2],pid;
 int retval;
 char buff[10];
 pipe(p);
 pid=fork();
 if(pid==0)
 {
  printf("Child Exiting\n");
 }
 else
 {
  close(p[0]);
  retval=write(p[1],buff,1);
  printf("Value returned %d\n",retval);
  printf("Parent Exiting\n");
 }
}
```

**Output:**

```
Child Exiting
Parent exiting
```

**Program 11:**

In the next program, we call signal ( ) function to handle the SIGPIPE signal. To it we assign a function. After the child process terminates, the time slice is handled to the parent process. In it the read end being closed. The write ( ) consequently returns a -1, since it does not sense any read ends open for it.

**Source Code:**

```
#include<signal.h>
void abc();
main()
{
int pid,p[2];
int retval;
char buff[10];
pipe(p);
signal(SIGPIPE,abc);
pid=fork();
if(pid==0)
{
printf("Child exiting\n");
}
else
{
close(p[0]);
retval=write(p[1],buff,1);
printf("Value returned :%d\n",retval);
printf("Parent exiting\n");
}
}
void abc()
{
printf("parent failed");
}
```

**Output:**

```
Child Exiting
Parent failed
```

**Program 12:**

In the next program we try an lseek ( ) on pipes to see what happens. The entire message is read by the read ( ) function. This is because the lseek ( ) does not work on pipes. Pipes, as we know, channelize data in a unidirectional flow. Because of this unidirectional movement, an lseek ( ) does not work on them.

**Source Code:**

```
#include<stdio.h>
#define SIZE 25
main()
{
 char* msg="OS";
 char buffer[SIZE];
 int p[2],i,pid;
 pipe(p);
 pid=fork();
 if(pid>0)
 {
  write(p[1],msg,SIZE);
 }
 else
 {
     printf("%d\n",lseek(p[0],0L,2));
  read(p[0],buffer,SIZE);
  printf("%s\n",buffer);
 }
 exit(0);
}
```

**Output:**

```
OS
```

***RESULT:***

Thus, the inter-process communication was implemented using pipes.

| | |
|---|---|
| **EX.NO :9A**<br>**DATE :05.03.12** | ***MEMORY MANAGEMENT SCHEME-I***<br>***PAGING*** |

## ***AIM:***

To write a c program to find the physical memory using the paging method.

## ***ALGORITHM:***

1. Start the program.
2. Declare and define the structure for the page table.
3. Input the number of pages and size of the page.
4. Create the page table using linked list and input the page no and corresponding frame number.
5. Print the contents in table format.
6. Enter the logical address.
7. Find the page no, frame no, and the offset using the corresponding formulas.
8. Calculate the physical address using the formula pa = (fno * sp) + off.
9. Print the physical address.
10. Stop the program.

## ***PROGRAM:***

```
#include<stdlib.h>
#include<stdio.h>
struct pgtbl
{
     int pgno;
     int frno;
     struct pgtbl *next;
};
int np,sp;
struct pgtbl *tbl;
void tblinput()
{
     struct pgtbl *temp;
     struct pgtbl *p;
     int i;
     printf("Enter number of pages");
     scanf("%d",&np);
     printf("Enter size of page");
     scanf("%d",&sp);
     tbl=(struct pgtbl*)malloc(sizeof(struct pgtbl));
     tbl->pgno=-3;
     tbl->frno=-9;
     tbl->next=NULL;
```

```
          p=tbl;
          for(i=0;i<np;i++)
          {
          printf("Enter page no and corresponding frame no ");
          temp=(struct pgtbl*)malloc(sizeof(struct pgtbl));
          scanf("%d",&temp->pgno);
          scanf("%d",&temp->frno);
          temp->next=NULL;
          p->next=temp;
          p=temp;
          }
     }
     void prnttbl()
     {
          struct pgtbl *temp;
          temp=tbl->next;
          printf("Page Table\n");
          while(temp)
          {
               printf("%d\t%d\n",temp->pgno,temp->frno);
               temp=temp->next;
          }
     }
     int search(int pno)
     {
          struct pgtbl *temp;
          temp=tbl->next;
          int fno=0;
          while(temp&&temp->pgno!=pno)
               temp=temp->next;
          if(temp)
               fno=temp->frno;
          return fno;
     }

     main()
     {
          int lgadr,pno,fno,off,pa;
          tblinput();
          prnttbl();
          printf("Enter the logical address ");
          scanf("%d",&lgadr);
          pno=(int)lgadr/sp;
          fno=search(pno);
          off=lgadr%sp;
          pa=(fno*sp)+off;
               printf("Logical Address:%d\nCorresponding
          Physical address:%d\n",lgadr,pa);
     }
```

***OUTPUT:***

```
Enter the number of pages:3
Enter the size of page:10
Enter pageno. And frame no.:1
3
Enter pageno. And frame no.:2
2
Enter pageno. And frame no.:3
1
Page table:
Pageno     fno
1          3
2          2
3          1
Enter logical address:20
Logical Address:20
Physical Address:20
```

***RESULT:***

Thus, the c program to perform paging operation is executed successfully.

|  |  |
|---|---|
| **EX.NO :9B** <br> **DATE :12.03.12** | ***MEMORY MANAGEMENT SCHEME-I*** <br> ***SEGMENTATION*** |

## ***AIM:***

To write a c program to perform segmentation.

## ***ALGORITHM:***

1. Start the program.
2. Declare and define the structure for the segment table.
3. Input the number of segments.
4. Create the page table by entering the segment no, base and limit.
5. Enter the segment number and offset for logical address.
6. Check whether the physical address is less than base + limit.
7. If yes, then print as valid address.
8. If no, then print invalid address.
9. Stop the program.

## ***PROGRAM:***

```
#include<stdlib.h>
#include<stdio.h>
struct sgtbl
{
     int sgno;
     int base;
     int limit;
     struct sgtbl *next;
};
int ns;
struct sgtbl *tbl;
void tblinput()
{
     struct sgtbl *temp;
     struct sgtbl *p;
     int i;
     printf("Enter number of segments");
     scanf("%d",&ns);
     tbl=(struct sgtbl*)malloc(sizeof(struct sgtbl));
     tbl->sgno=-3;
     tbl->base=-9;
     tbl->limit=-99;
     tbl->next=NULL;
     p=tbl;
```

```
	for(i=0;i<ns;i++)
	{
		printf("Enter segment no, base and limit ");
		temp=(struct sgtbl*)malloc(sizeof(struct sgtbl));
		scanf("%d",&temp->sgno);
		scanf("%d",&temp->base);
		scanf("%d",&temp->limit);
		temp->next=NULL;
		p->next=temp;
		p=temp;
	}
}
void prnttbl()
{
	struct sgtbl *temp;
	temp=tbl->next;
	printf("Segment Table\n");
	while(temp)
	{
printf("%d\t%d\t%d\n",temp->sgno,temp->base,temp->limit);
		temp=temp->next;
	}
}

main()
{
	int lgadr,pno,fno,off,pa,f;
	f=0;
	segtbl *temp;
	tblinput();
	prnttbl();
	printf("Enter the segment no and offset");
	scanf("%d",&segno);
	scanf("%d",&off);
	temp=tbl->next;
	for(i=0;i<ns && temp->sgno==segno;i++)
		if((temp->base+off)<(temp->base+temp->limit)
		printf("Valid Address:%d",(temp->base+off));f=0;
		else
			temp=temp->next;
		f=1;
	if(f=1)
		printf("\nInvalid address");
}
```

***OUTPUT:***

```
Enter number of segments:2
Enter segno,base and limit:1
5000
200
```

```
Enter segno,base and limit:2
5500
100
Segment table:
Segmno.   Base   limit
   1         5000    200
   2         5500    100
Enter segment no and offset:
1
100

Valid Address:5100
```

***RESULT:***

Thus, the c program to perform segmentation is executed successfully.

| | |
|---|---|
| **EX.NO :10** | ***FILE ALLOCATION TECHNIQUE*** |
| **DATE :19.03.12** | ***CONTIGUOUS FILE ALLOCATION*** |

## *AIM:*

To write a c program to allocate memory for file using sequential contiguous memory allocation and direct access contiguous memory allocation.

## *ALGORITHM:*

1. Start the program
2. Read the number of blocks in memory
3. Read the choice (sequential or direct access) from user.
4. For direct access get the file name, limit and start block
5. Ensure that the blocks to be allocated for one file does not interfere with another file
6. For sequential access, get name and limit and allocate block starting from zero based on each file's limit.
7. In both choices the number of blocks required by the files should not exceed total number of blocks available in the system.
8. Based on choice do appropriate memory allocation.
9. Print the file names along with their allocated starting blocks
10. Stop the program.

## *PROGRAM:*

```
#include<stdio.h>
struct filedet
{
     char name[10];
     int srtblk;
     int limit;
     struct filedet* next;
}*h;
int n,flag=0;
createtbl()
{
     int i=0;
     struct filedet *p,*temp;
     int nof;
     printf("Enter number of files");
     scanf("%d",&nof);
     h=(struct filedet*)malloc(sizeof(struct filedet));
     h->limit=0;
     h->srtblk=-9;
```

```
      if(flag==0)
      {
      for(i=0;i<nof;i++)
      {
       temp=(struct filedet*)malloc(sizeof(struct filedet));
      printf("Eter file name and limit");
      scanf("%s",&temp->name);
      scanf("%d",&temp->limit);
      temp->next=NULL;
      p->next=temp;
      p=temp;
      }
      }
      else
      {
      for(i=0;i<nof;i++)
      {
      temp=(struct filedet*)malloc(sizeof(struct filedet));
       printf("Eter file name,limit aand start block");
      scanf("%s",&temp->name);
      scanf("%d",&temp->limit);
      scanf("%d",&temp->srtblk);
      temp->next=NULL;
      p->next=temp;
      p=temp;
      }
      }
}
void allot()
{
      if(flag==1)
      {
            int x=0;
            struct filedet *temp;
            temp=h->next;
            while(temp)
            {
                  temp->srtblk=x;
                  x=x+temp->limit;
                  temp=temp->next;
            }
      }
}
void print()
{
      struct filedet *temp;
      temp=h->next;
      printf("Name\tStart Block\tLimit\n");
      while(temp)
      {
       printf("temp->name\t\ttemp->srtblk\ttemp->limit\n");
      temp=temp->next;
```

```
          }
}
void main()
{
          printf("Enter number of blocks");
          scanf("%d",&n);
          printf("0.Sequential Access\n1.Direct Access");
          printf("\nEnter your Choice ");
          scnaf("%d",&flag");
          createtbl();
          allot();
          print();
}
```

***OUTPUT:***

```
Enter number of blocks 15
0. Sequential Access
1. Direct Access
Enter choice 0
Enter number of files 3
Enter name and limit
A
5
B
2
C
4
Name Start Block    Limit
A         0         5
B         5         2
C         7         4
Enter number of blocks 15
0. Sequential Access
1. Direct Access
Enter choice 1
Enter number of files 3
Enter name, limit and start block
A
5
1
B
2
13
C
```

```
4
7
Name  Start Block     Limit
A             1             5
B             13             2
C             7             4
```

***RESULT:***

Thus the c program to allocate memory for file using sequential contiguous memory allocation and direct access contiguous memory allocation was executed successfully and output was verified.