-

A.P.SHAHINSTITUTEOFTECHNOLOGY

Department of Computer Engineering

Program:	**T.E**
Semester:	**V**
SubjectName:	**Operating Systems(OS)**

Parshvanath Charitable Trust's

A. P. SHAH INSTITUTE OF TECHNOLOGY

(Approved by AICTE New Delhi & Govt. of Maharashtra, Affiliated to University of Mumbai)
(Religious Jain Minority)

Department of Computer Engineering

SE: SEM V **Subject: OS**

Experiment List

Sr. No.	List of Practical Experiments
01	To study basic Linux process commands.
02	To study Process related system call.
03	Implementation of Basic Process management algorithms such as FCFS, SJF and RR.
04	Implementation of Process synchronization algorithms like Producer Consumer Problem.
05	Implementation of banker's algorithm.
06	Implementation of Disk scheduling algorithms like FCFS, SSTF, SCAN and LOOK
07	Implementation of various file allocation methods such as Contiguous allocation and Indexed Allocation
08	Implementation of paging.
09	Implementing kernel compilation.

Subject In-Charge

Prof.Pravin P. Adivarekar

HOD

Prof.Sachin Malave

EXPERIMENT NO. 01

Aim: To study Process Management.

Theory :

An instance of a program is called a Process. In simple terms, any command that you give to your Linux machine starts a new process. Linux creates a process whenever a program is launched, either by you or by Linux. This process is a container of information about how that program is running and what‟s happening.

Eg. if u open a Terminal that is nothing but a one Process.

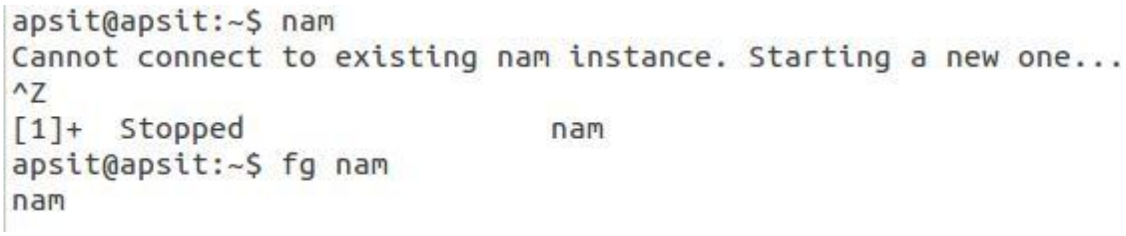
Types of Processes:

Foreground Processes: They run on the screen and need input from the user. For example : Office Programs Background Processes: They run in the background and usually do not need user input.

For example Antivirus.

Running a foreground Process

To start a foreground process you can either run it from the dash board or you can run it from the terminal.When using the Terminal, you will have to wait, until the foreground process runs.



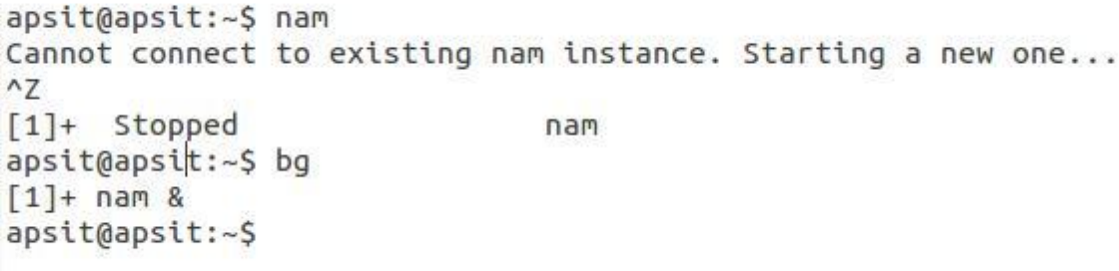
```
apsit@apsit:~$ nam
Cannot connect to existing nam instance. Starting a new one...
^Z
[1]+  Stopped                 nam
apsit@apsit:~$ fg nam
nam
```

so that user can again continue with current process.

Running a Background process

If you start a foreground program/process from the terminal, then you cannot work on the terminal, till the program is up and running. Certain, data intensive tasks take lots of processing power and may even take hours to complete. You do not want your terminal to be held up for such a long time.To avoids such a situation; you can run the program and send it to the background so that terminal remains available to you.

Example:



```
apsit@apsit:~$ nam
Cannot connect to existing nam instance. Starting a new one...
^Z
[1]+  Stopped                 nam
apsit@apsit:~$ bg
[1]+ nam &
apsit@apsit:~$
```

Brief of things to do when managing Linux processes:

See which processes are running

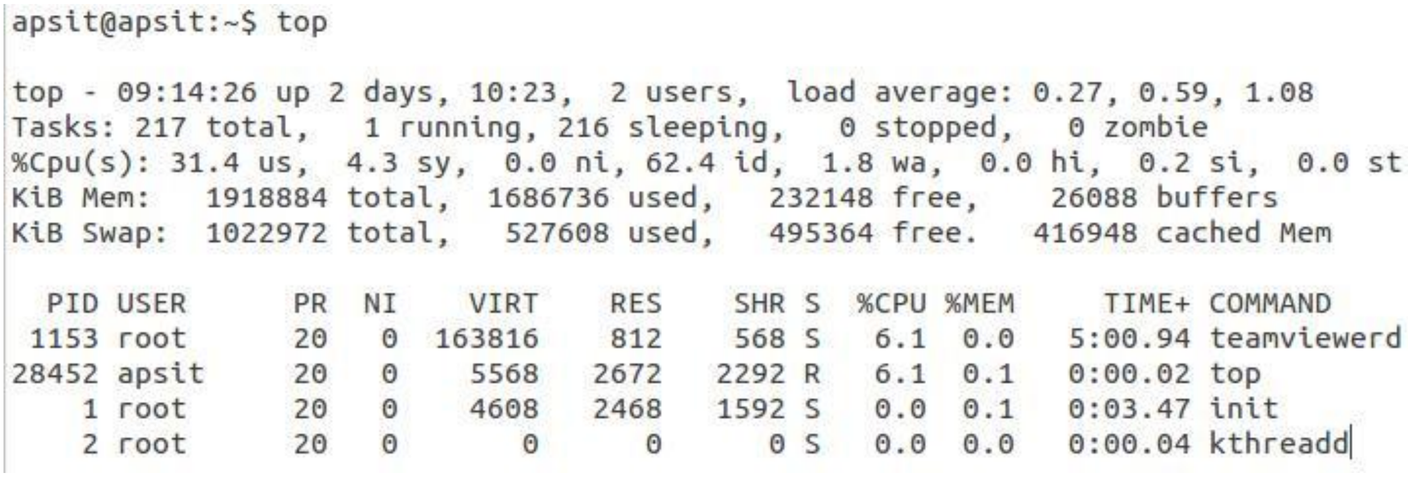
See how much of your Linux system the processes are using (especially any greedy ones) Locate a particular process to see what it‟s doing or to take action on it

Define or change the level of priority associated with that process

Terminate the process if it has outlived its usefulness or if it‟s misbehaving

Top

This utility tells the user about all the running processes on the Linux machine.
Eg: type **top** in terminal you will get :



```
apsit@apsit:~$ top

top - 09:14:26 up 2 days, 10:23,  2 users,  load average: 0.27, 0.59, 1.08
Tasks: 217 total,   1 running, 216 sleeping,   0 stopped,   0 zombie
%Cpu(s): 31.4 us,  4.3 sy,  0.0 ni, 62.4 id,  1.8 wa,  0.0 hi,  0.2 si,  0.0 st
KiB Mem:   1918884 total,  1686736 used,   232148 free,    26088 buffers
KiB Swap:  1022972 total,   527608 used,   495364 free.   416948 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 1153 root      20   0  163816    812    568 S   6.1  0.0   5:00.94 teamviewerd
28452 apsit     20   0    5568   2672   2292 R   6.1  0.1   0:00.02 top
    1 root      20   0    4608   2468   1592 S   0.0  0.1   0:03.47 init
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.04 kthreadd
```

Press '**q**' on the keyboard to move out of the process display.

The terminology follows:

PID -- Process Id (The task's unique process ID)

USER -- User Name

The effective user name of the task's owner

PR -- Priority The scheduling priority of the task.

NI -- Nice Value

The nice value of the task. A negative nice value means higher priority, whereas a positive nice value means lower priority. Zero in this field simply means priority will not be adjusted in determining a task's dispatch-ability.

VIRT -- Virtual Memory Size (KiB) The total amount of virtual memory used by the task.

RES -- Resident Memory Size (KiB) The non-swapped physical memory a task has used.

SHR -- Shared Memory Size (KiB) The amount of shared memory available to a task.

S -- Process Status

The status of the task which can be one of:

D = uninterruptible sleep

R = running

S = sleeping

T = traced or stopped

Z = zombie

%CPU -- CPU Usage The task's share of the elapsed CPU time since the last screen update,expressed as a percentage of total CPU time.

%MEM -- Memory Usage (RES) A task's currently used share of available physical memory.

TIME+ -- CPU Time, hundredths

COMMAND -- Command Name or Command Line Display the command line used to start a task or the name of the associated program.

COMMAND-LINE OPTIONS

-d

Specifies the delay between screen updates. You can change this with the **s** interactive command.

-p

Monitor only processes with given process id. This flag can be given up to twenty times. This option is neither available interactively nor can it be put into the configuration file.

-q

This causes **top** to refresh without any delay. If the caller has superuser privileges, top runs with the highest possible priority.

-S

Specifies cumulative mode, where each process is listed with the CPU time that it *as well as its dead children* has spent.

-s

Tells **top** to run in secure mode. This disables the potentially dangerous of the interactive commands.

-i

Start **top** ignoring any idle or zombie processes. See the interactive command **i** below.

-C

display total CPU states instead of individual CPUs. This option only affects SMP systems.

-c

display command line instead of the command name only. The default behavior has been changed as this seems to be more useful.

-H

Show all threads.

-n

Number of iterations. Update the display this number of times and then exit.

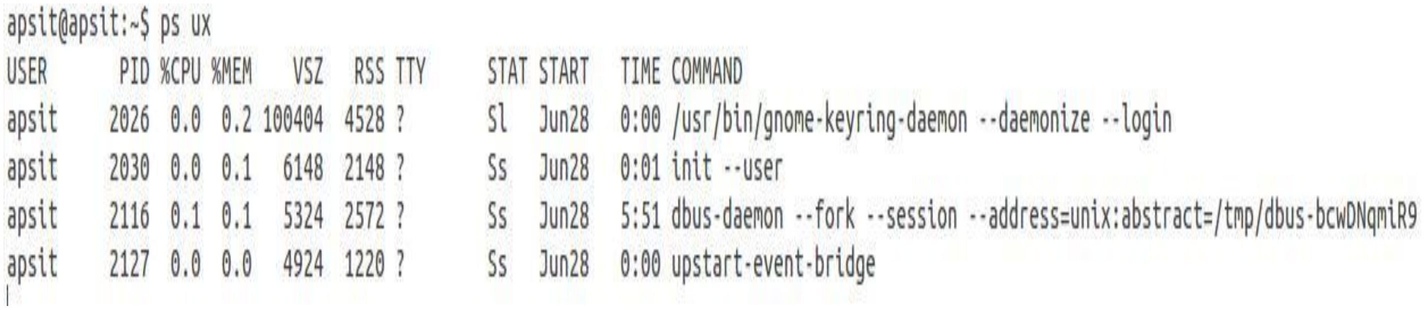
-b

Batch mode. Useful for sending output from top to other programs or to a file.

ps

This command stands for 'Process Status'. It is similar to the "Task Manager" that pop-ups in a Windows Machine when we use Cntrl+Alt+Del . This command is similar to 'top' command but the information displayed Is different. To check all the processes running under a user, use the command -

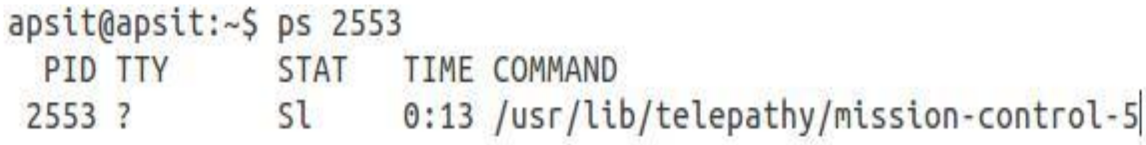
ps ux



```
apsit@apsit:~$ ps ux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
apsit     2026  0.0  0.2 100404  4528 ?        Sl   Jun28   0:00 /usr/bin/gnome-keyring-daemon --daemonize --login
apsit     2030  0.0  0.1   6148  2148 ?        Ss   Jun28   0:01 init --user
apsit     2116  0.1  0.1   5324  2572 ?        Ss   Jun28   5:51 dbus-daemon --fork --session --address=unix:abstract=/tmp/dbus-bcwDNqmiR9
apsit     2127  0.0  0.0   4924  1220 ?        Ss   Jun28   0:00 upstart-event-bridge
```

You can also check the process status of a single process , use the syntax -

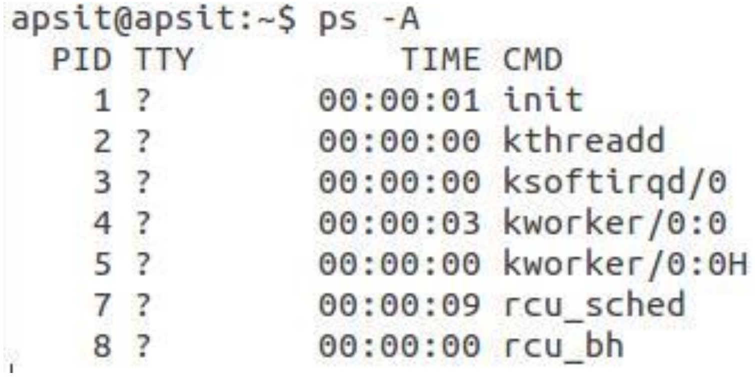
ps PID



```
apsit@apsit:~$ ps 2553
  PID TTY      STAT   TIME COMMAND
 2553 ?        Sl     0:13 /usr/lib/telepathy/mission-control-5
```

The ps command lists running processes. The following command lists all processes running on your system:

ps –A



```
apsit@apsit:~$ ps -A
  PID TTY          TIME CMD
    1 ?        00:00:01 init
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 ksoftirqd/0
    4 ?        00:00:03 kworker/0:0
    5 ?        00:00:00 kworker/0:0H
    7 ?        00:00:09 rcu_sched
    8 ?        00:00:00 rcu_bh
```

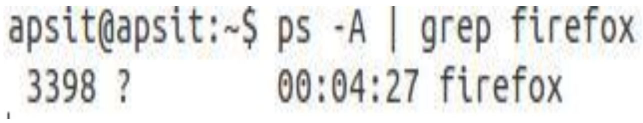
This may be too many processes to read at one time, so you can pipe the output through the less command to scroll through them at your own pace:

ps -A | less

Press q to exit when you‟re done.

You could also pipe the output through grep to search for a specific process without using any other commands. The following command would search for the Firefox process:

ps -A | grep firefox

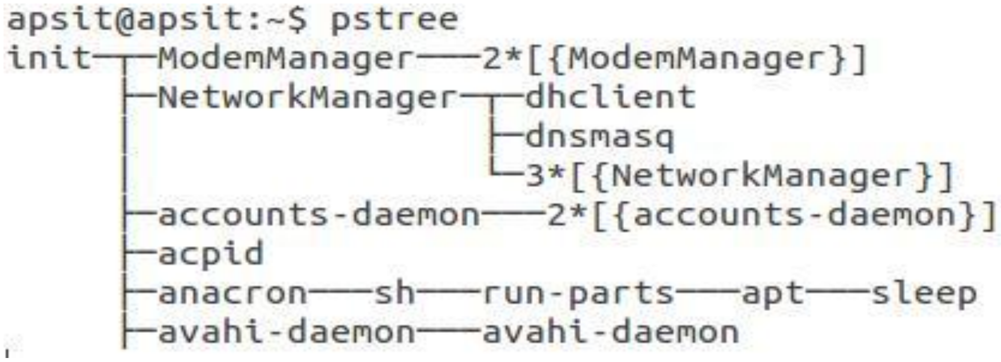


```
apsit@apsit:~$ ps -A | grep firefox
 3398 ?        00:04:27 firefox
```

pstree

A step up from the simple *ps* command, *pstree* is used to display a tree diagram of processes that also shows relationships that exist between them. Every process is generated by another process (a parent process) in Linux. if you alter something for a parent process, you affect the child processes as well. In particular, if you stop the parent, you automatically stop the children.

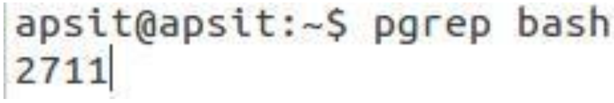
Eg.



```
apsit@apsit:~$ pstree
init─┬─ModemManager───2*[{ModemManager}]
     ├─NetworkManager─┬─dhclient
     │                ├─dnsmasq
     │                └─3*[{NetworkManager}]
     ├─accounts-daemon───2*[{accounts-daemon}]
     ├─acpid
     ├─anacron───sh───run-parts───apt───sleep
     ├─avahi-daemon───avahi-daemon
```

pgrep

A quick way of getting the PID of a process is with the pgrep command:

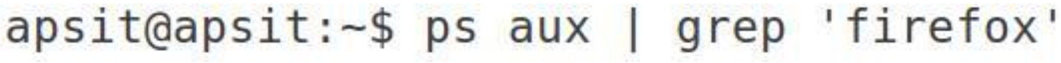


```
apsit@apsit:~$ pgrep bash
2711
```

This will simply query the process ID and return it.

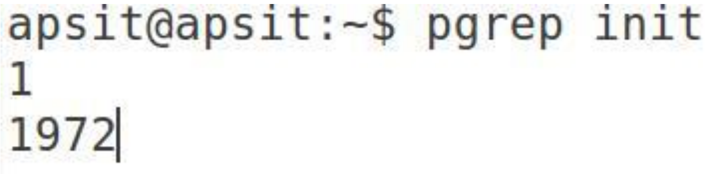
Killing a Crashed Process

It doesn't happen often, but when a program crashes, Let's say users got browser running and all of a sudden it locks up. users try and close the window but nothing happens, it has become completely unresponsive. No worries, users can easily kill Firefox and then reopen it. To start off we need to identify the process id.



```
apsit@apsit:~$ ps aux | grep 'firefox'
```

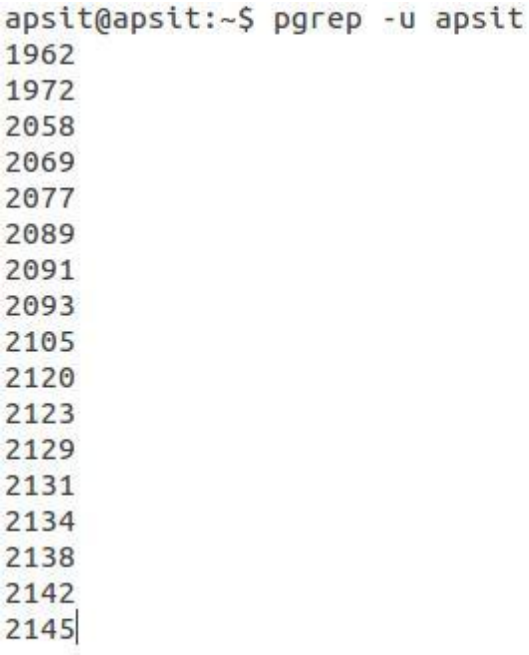
The first process spawned at boot, called *init*, is given the PID of "1".



```
apsit@apsit:~$ pgrep init
1
1972
```

users can use -u option to locate processess that are owned by specific users.

Eg:



```
apsit@apsit:~$ pgrep -u apsit
1962
1972
2058
2069
2077
2089
2091
2093
2105
2120
2123
2129
2131
2134
2138
2142
2145
```

Kill

This command **terminates a running processes** on a Linux machine. In order to use this utility you need to know the PID (process id) of the process you want to kill.

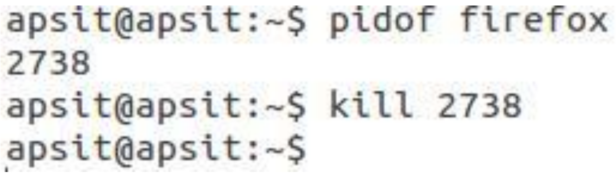
Syntax -

kill PID

To find the PID of a process simply type

pidof Processname

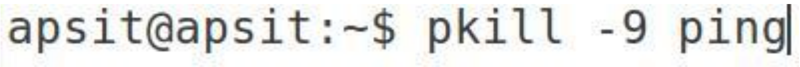
example:



```
apsit@apsit:~$ pidof firefox
2738
apsit@apsit:~$ kill 2738
apsit@apsit:~$
```

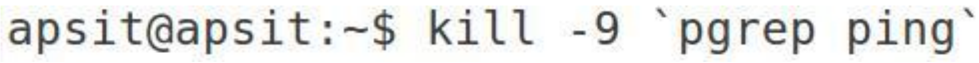
pkill

The pkill command works in almost exactly the same way as kill, but it operates on a process name instead:



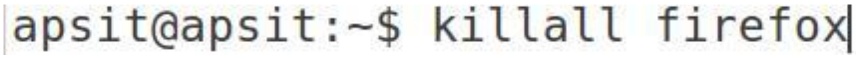
```
apsit@apsit:~$ pkill -9 ping
```

The above command is the equivalent of.:



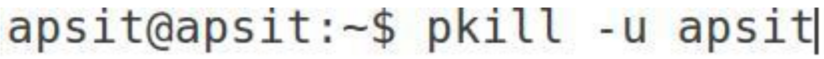
```
apsit@apsit:~$ kill -9 `pgrep ping`
```

If you would like to send a signal to every instance of a certain process, you can use the killall command:



```
apsit@apsit:~$ killall firefox
```

To kill all processes that are owned by user apsit, use the following.

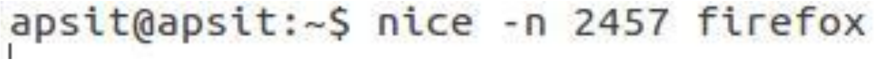


```
apsit@apsit:~$ pkill -u apsit
```

NICE

Linux can run a lot of processes at a time ,which can slow down the speed of some high priority processes and result in poor performance. To avoid this, you can tell your machine to prioritize processes as per your requirements This priority is called Niceness in Linux and it has a value between -20 to 19. The lower the Niceness index the higher would be priority given to that task. The default value of all the processes is 0. To start a process with a niceness value other than the default value use the following syntax.

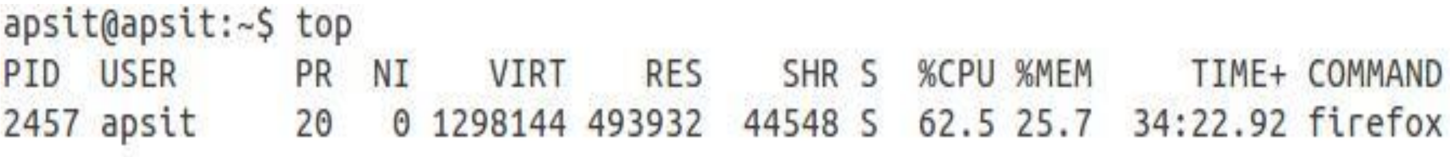
nice -n 'Nice value' process name



```
apsit@apsit:~$ nice -n 2457 firefox
```

If there is some process already running on the system then you can 'Renice' its value using syntax.

renice 'nice value' -p 'PID'



```
apsit@apsit:~$ top
PID  USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
2457 apsit     20   0 1298144 493932  44548 S  62.5 25.7  34:22.92 firefox
```

To change Niceness you can use the 'top' command to determine the PID (process id) and its Nice value.Later use the renice command to change the value. you can check it by again using top command :

Conclusion: Thus we Understood Process Management in linux.

EXPERIMENT NO: 2

Aim: To study process related system call.

Theory:

The fork() System Call

System call **fork()** is used to create processes. It takes no arguments and returns a process ID. The purpose of **fork()** is to create a ***new*** process, which becomes the *child* process of the caller. After a new child process is created, ***both*** processes will execute the next instruction following the ***fork()*** system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**:

If **fork()** returns a negative value, the creation of a child process was unsuccessful.

fork() returns a zero to the newly created child process.

fork() returns a positive value, the ***process ID*** of the child process, to the parent. The returned process ID is of type **pid_t** defined in **sys/types.h**. Normally, the process ID is an integer. Moreover, a process can use function **getpid()** to retrieve the process ID assigned to this process.

Therefore, after the system call to **fork()**, a simple test can tell which process is the child. **Please note that Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces**.

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main()

{

   // make two process which run same
   // program after this instruction
   fork();

   printf("Hello world!\n");
   return 0;
}
```

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

int main()

{

  pid_t c_pid;

  c_pid = fork(); //duplicate

  if( c_pid == 0
{
    //child: The return of fork() is zero

    printf("Child: I'm the child: %d\n", c_pid);

  }
else if (c_pid > 0)
{
    //parent: The return of fork() is the process of id of the child

    printf("Parent: I'm the parent: %d\n", c_pid);

  }


Else
{
    //error: The return of fork() is negative

    perror("fork failed");
    _exit(2); //exit failure, hard

  }

  return 0; //success
}
```

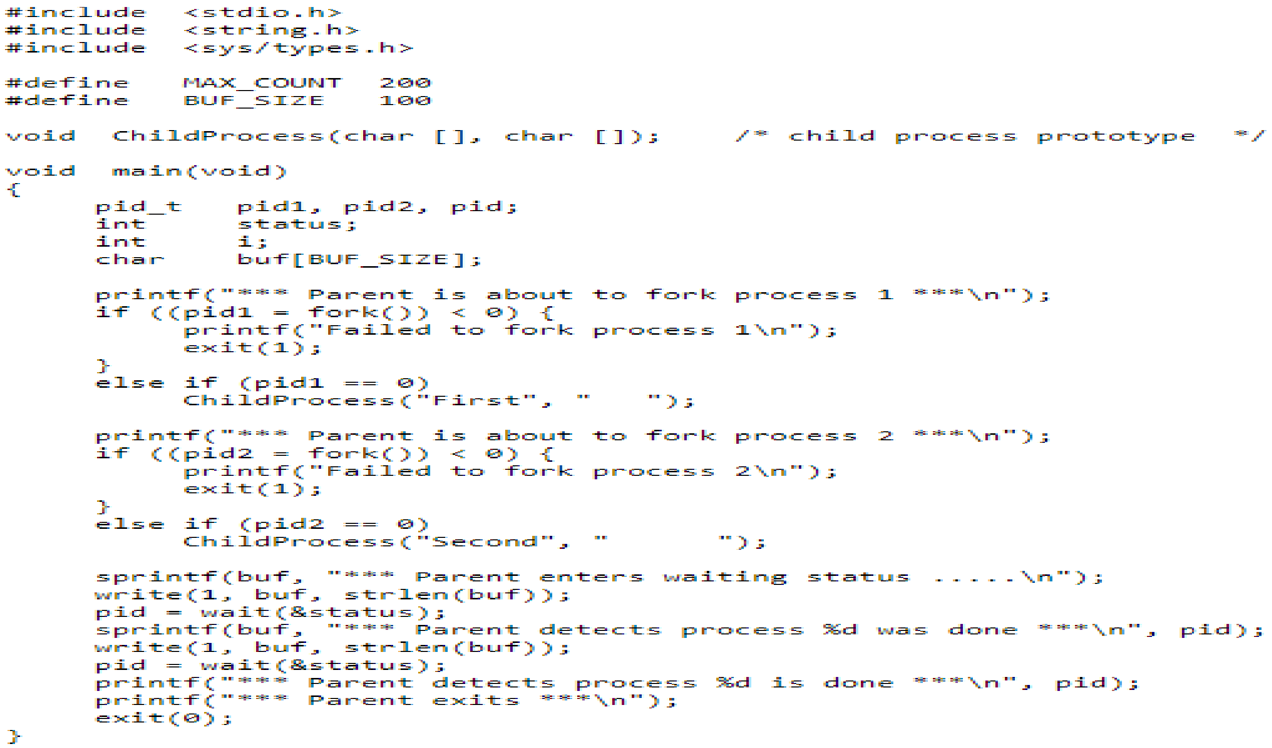
```
#include  <stdio.h>
#include  <sys/types.h>

#define   MAX_COUNT  50

void  ChildProcess(void);                /* child process prototype  */
void  ParentProcess(void);               /* parent process prototype */

void  main(void)
{
    pid_t  pid;

    pid = fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}

void  ChildProcess(void)
{
    int   i;

    for (i = 1; i <= MAX_COUNT; i++)
    printf("   This line is from child, value = %d\n", i);
    printf("   *** Child process is done ***\n");
}

void  ParentProcess(void)
{
    int   i;

    for (i = 1; i <= MAX_COUNT; i++)
        printf("This line is from parent, value = %d\n", i);
        printf("*** Parent is done ***\n");
}
```

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

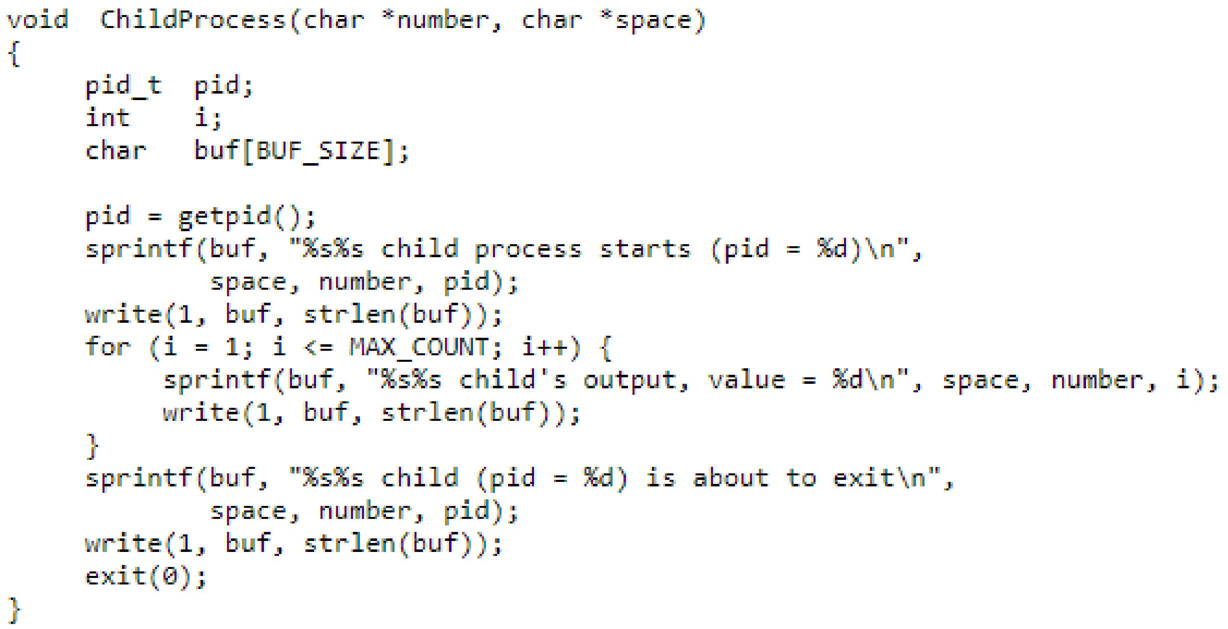
The wait() System Call

The system call **wait()** is easy. This function blocks the calling process until one of its *child* processes exits or a signal is received. For our purpose, we shall ignore signals. **wait()** takes the address of an integer variable and returns the process ID of the completed process. Some flags that indicate the completion status of the child process are passed back with the integer pointer. One of the main purposes of **wait()** is to wait for completion of child processes.
The execution of **wait()** could have two possible situations.

1. If there are at least one child processes running when the call to **wait()** is made, the caller will be blocked until one of its child processes exits. At that moment, the caller resumes its execution.
2. If there is no child process running when the call to **wait()** is made, then this **wait()** has no effect at all. That is, it is as if no **wait()** is there



```
#include   <stdio.h>
#include   <string.h>
#include   <sys/types.h>

#define    MAX_COUNT   200
#define    BUF_SIZE    100

void   ChildProcess(char [], char []);     /* child process prototype   */

void   main(void)
{
       pid_t    pid1, pid2, pid;
       int      status;
       int      i;
       char     buf[BUF_SIZE];

       printf("*** Parent is about to fork process 1 ***\n");
       if ((pid1 = fork()) < 0) {
            printf("Failed to fork process 1\n");
            exit(1);
       }
       else if (pid1 == 0)
            ChildProcess("First", "    ");

       printf("*** Parent is about to fork process 2 ***\n");
       if ((pid2 = fork()) < 0) {
            printf("Failed to fork process 2\n");
            exit(1);
       }
       else if (pid2 == 0)
            ChildProcess("Second", "        ");

       sprintf(buf, "*** Parent enters waiting status .....\n");
       write(1, buf, strlen(buf));
       pid = wait(&status);
       sprintf(buf, "*** Parent detects process %d was done ***\n", pid);
       write(1, buf, strlen(buf));
       pid = wait(&status);
       printf("*** Parent detects process %d is done ***\n", pid);
       printf("*** Parent exits ***\n");
       exit(0);
 }
```



```
void  ChildProcess(char *number, char *space)
{
     pid_t  pid;
     int    i;
     char   buf[BUF_SIZE];

     pid = getpid();
     sprintf(buf, "%s%s child process starts (pid = %d)\n",
             space, number, pid);
     write(1, buf, strlen(buf));
     for (i = 1; i <= MAX_COUNT; i++) {
          sprintf(buf, "%s%s child's output, value = %d\n", space, number, i);
          write(1, buf, strlen(buf));
     }
     sprintf(buf, "%s%s child (pid = %d) is about to exit\n",
             space, number, pid);
     write(1, buf, strlen(buf));
     exit(0);
}
```

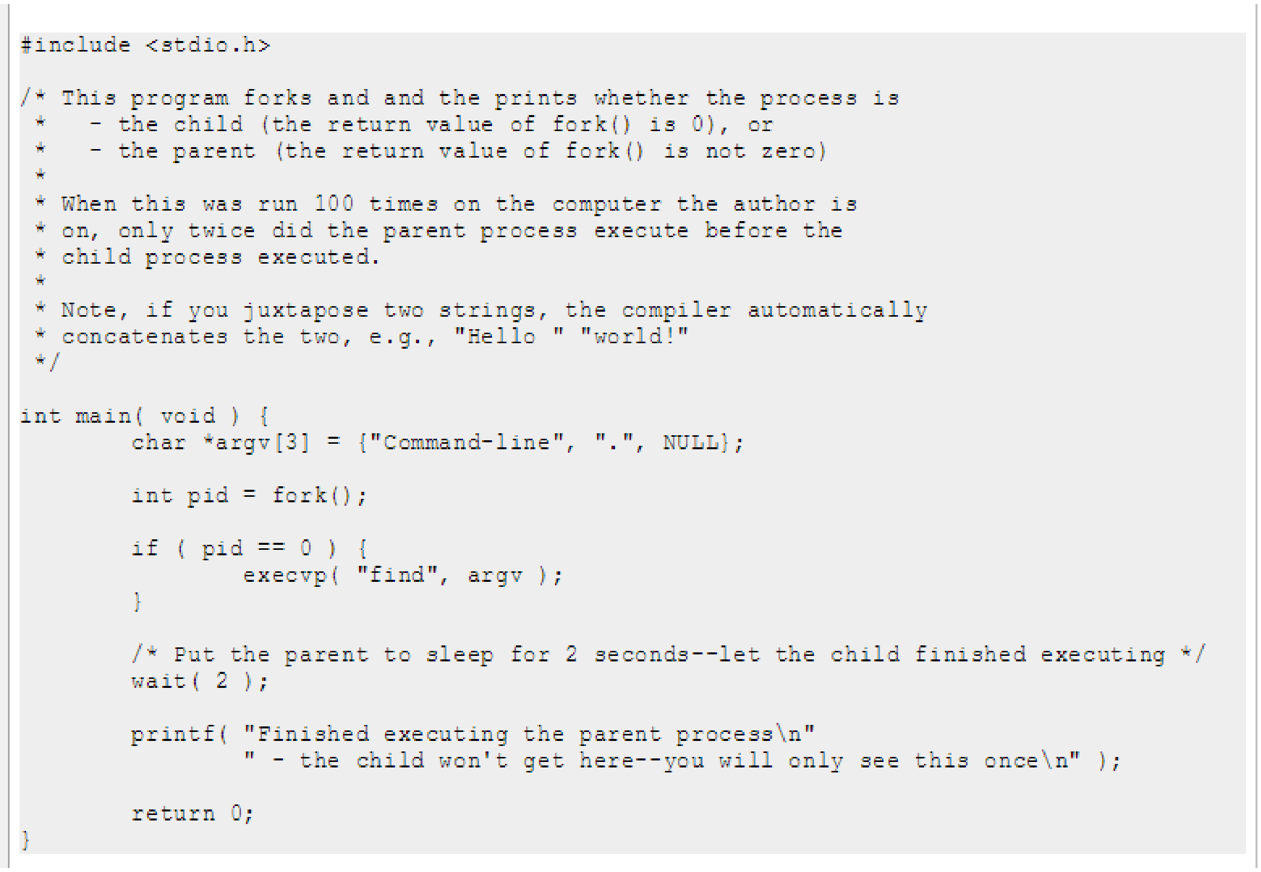
```
------------------------------------------------------------------------------------------------------------------------------------
-
  #include <sys/types.h>
  #include <sys/wait.h>
void main()
{
 int status;
pid_t  pid;
pid = fork();
if(pid == -1)
 printf("\nERROR child not created");
 else if (pid == 0) /* child process */
{
printf("\n I'm the child!");
exit(0);
}
else /* parent process */
{
wait(&status);
printf("\n I'm the parent!");
printf("\n Child returned: %d \n", status);
}
}

------------------------------------------------------------------------------------------------------------------------
```

The exec() System Call

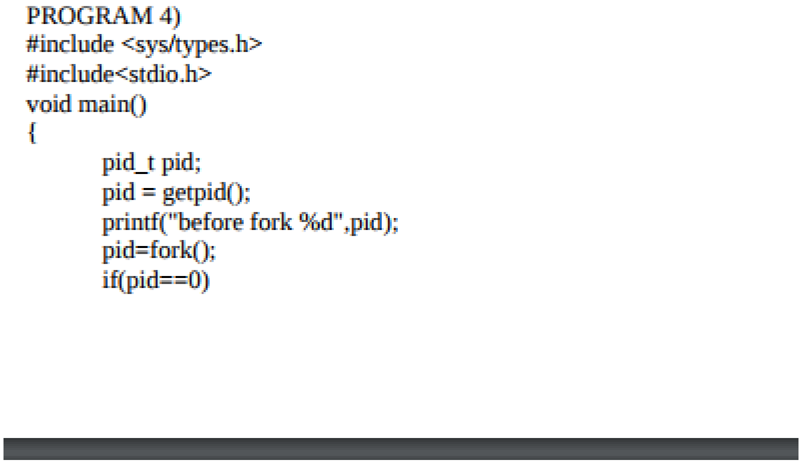
The created child process does not have to run the same program as the parent process does. The **exec** type system calls allow a process to run any program files, which include a binary executable or a shell script.

In computing, **exec** is a functionality of an operating system that runs an executable file in the context of an already existing process, replacing the previous executable. This act is also referred to as an **overlay**. It is especially important in Unix-like systems, although other operating systems implement it as well. Since a new process is not created, the original process identifier (PID) does not change, but the machine code, data, heap, and stack of the process are replaced by those of the new program.

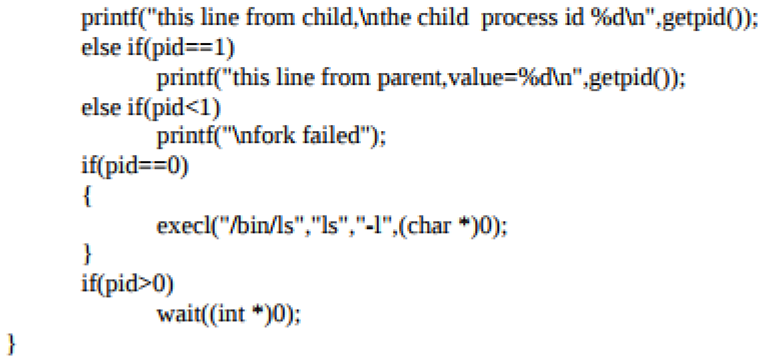
The *exec* call is supported in many programming languages, including compilable languages and some scripting languages. In OS command interpreters, the exec built-in commandreplaces the shell process with the specified program.



```
#include <stdio.h>

/* This program forks and and the prints whether the process is
 *   - the child (the return value of fork() is 0), or
 *   - the parent (the return value of fork() is not zero)
 *
 * When this was run 100 times on the computer the author is
 * on, only twice did the parent process execute before the
 * child process executed.
 *
 * Note, if you juxtapose two strings, the compiler automatically
 * concatenates the two, e.g., "Hello " "world!"
 */

int main( void ) {
        char *argv[3] = {"Command-line", ".", NULL};

        int pid = fork();

        if ( pid == 0 ) {
                execvp( "find", argv );
        }

        /* Put the parent to sleep for 2 seconds--let the child finished executing */
        wait( 2 );

        printf( "Finished executing the parent process\n"
                " - the child won't get here--you will only see this once\n" );

        return 0;
}
```



```
PROGRAM 4)
#include <sys/types.h>
#include<stdio.h>
void main()
{
        pid_t pid;
        pid = getpid();
        printf("before fork %d",pid);
        pid=fork();
        if(pid==0)
```



```
        printf("this line from child,\nthe child  process id %d\n",getpid());
        else if(pid==1)
                printf("this line from parent,value=%d\n",getpid());
        else if(pid<1)
                printf("\nfork failed");
        if(pid==0)
        {
                execl("/bin/ls","ls","-l",(char *)0);
        }
        if(pid>0)
                wait((int *)0);
}
```

Conclusion: Thus we have studied various system call related to process management

Parshvanath Charitable Trust's

A. P. SHAH INSTITUTE OF TECHNOLOGY

(Approved by AICTE New Delhi & Govt. of Maharashtra, Affiliated to University of Mumbai)
(Religious Jain Minority)

Department of Computer Engineering

SE: SEM V **Subject: OS**

Experiment No. 3

Aim: Implementation of Basic Process management algorithms such as FCFS, SJF and RR.

Theory:

1) First Come First Served (FCFS) Process Management Algorithm:

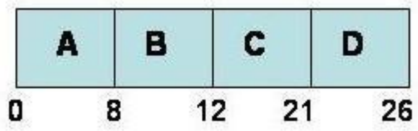
- It is the simplest algorithm and NON-PREEMPTIVE.
- The process that requests the CPU first is allocated the CPU first.
- The implementation is easily managed by queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue
- The average waiting time, however, is long. It is not minimal and may vary substantially if the process CPU burst time varies greatly.
- This algorithm is particularly troublesome for time-sharing systems.

It is the simplest scheduling algorithm, FIFO simply queues processes in the order that they arrive in the ready queue. When evaluating a scheduler‟s performance, we use a **Gantt Chart**, which is a horizontal timeline indicating which processes are run and at what times they run starting from when all tasks are submitted and ending when all tasks have been completed.

To illustrate it, suppose the scheduler is given 4 tasks, A, B, C and D. Each task requires a certain number of time units to complete.

Task	Time units
A	8
B	4
C	9
D	5

The FCFS scheduler‟s Gantt chart for these tasks would be:



The tasks are inserted into the queue in order A, B, C and D. as shown above. Task A takes 8 time units to complete, B takes 4 units to complete (therefore, B completes at time 12), etc. Task D ends at time 26, which is the time it took to run and complete all processes.

Metric	FCFS
Turnaround time	(8+12+21+25+6cs)/4 = 16.5
Waiting time	(0+8+12+21+6cs)/4 = 10.25

The average waiting time of FCFS is usually quite long.

Algorithm :

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Set the waiting of the first process as „0‟ and its burst time as its turn around time

Step 5: for each process in the Ready Q calculate

(a) Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)

(b) Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

Step 6: Calculate

(a) Average waiting time = Total waiting Time / Number of process

(b) Average Turnaround time = Total Turnaround Time / Number of process

Step 7: Stop the process

Program:

```
#include<stdio.h>

void main()

{

 int i,n,sum,wt,tat,twt,ttat;

 int t[10];

 float awt,atat;

 clrscr();

 printf("Enter number of processors:\n");

 scanf("%d",&n);

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

 {

   printf("\n Enter the Burst Time of the process %d",i+1);

   scanf("\n %d",&t[i]);

 }

 printf("\n\n FIRST COME FIRST SERVE SCHEDULING ALGORITHM \n");

 printf("\n Process ID \t Waiting Time \t Turn Around Time \n");

 printf("1 \t\t 0 \t\t %d \n",t[0]);

 sum=0;

 twt=0;

 ttat=t[0];

 for(i=1;i<n;i++)

 {
  sum+=t[i-1];
  wt=sum;
  tat=sum+t[i];
  twt=twt+wt;
  ttat=ttat+tat;
  printf("\n %d \t\t %d \t\t %d",i+1,wt,tat);
  printf("\n\n");
 }
  awt=(float)twt/n;
  atat=(float)ttat/n;
  printf("\n Average Waiting Time %4.2f",awt);
  printf("\n Average Turnaround Time %4.2f",atat);
  getch();
}
```

OUTPUT:

```
Enter number of processors:
3
 Enter the Burst Time of the process 1:   2
 Enter the Burst Time of the process 2:    5
 Enter the Burst Time of the process 3:    4


FIRST COME FIRST SERVE SCHEDULING ALGORITHM
Process ID      Waiting Time    Turn Around Time
```

1	0	2
2	2	7
3	7	11

Average Waiting Time 3.00

Average Turnaround Time 6.67

2. Shortest Job First (SJF) Process Management Algorithm:

- The SJF is a special case of priority scheduling.
- In priority scheduling algorithm, a priority is associated with each process, and the CPU is allocated to the process with the highest priority.

Two Schemes:

Non-preemptive (SRTF- Shortest Remaining Time First)- Once CPU given to the process it cannot be preempted until completes its CPU burst**.**

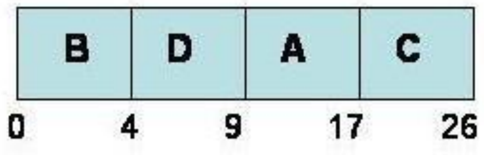
Preemptive- If a new process arrives with CPU burst length less than remaining time of current executing process, preempt.

This scheme is known as the **Shortest-Remaining-Time-First(SRTF)**.

SJF is optimal- Gives minimum average waiting time for a given set of processes To illustrate it, suppose the scheduler is given 4 tasks, A, B, C and D. Each task requires a certain number of time units to complete.

Task	Time units
A	8
B	4
C	9
D	5

The SJF Gantt Chart would be:



Metric	SJF
Turnaround time	(4+9+CS+17+2CS+26+3CS)/4 = 14
Waiting Time	(0+4+CS+9+2CS+17+3CS)/4 = 7.5

From the metrics, we see that SJF achieves better performance than FCFS. However, unlike FCFS, there is the potential for **starvation** in SJF. **Starvation** occurs when a large process never gets run to run because shorter jobs keep entering the queue.

In addition, SJF needs to know how long a process is going to run (i.e. it needs to predict the future). This runtime estimation feature may be hard to implement, and thus SJF is not a widely used scheduling scheme.

Algorithm:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Start the Ready Q according the shortest Burst time by sorting according to lowest to highest burst time.

Step 5: Set the waiting time of the first process as „0‟ and its turnaround time as its burst time.

Step 6: For each process in the ready queue, calculate

(c) Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)

(d) Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

Step 6: Calculate

(c) Average waiting time = Total waiting Time / Number of process

(d) Average Turnaround time = Total Turnaround Time / Number of process

Step 7: Stop the process

Program:

```
#include<stdio.h>
void main()
{
 int i,j,k,n,sum,wt[10],tt[10],twt,ttat;
 int t[10],p[10];
 float awt,atat;
 clrscr();
 printf("Enter number of process\n");
 scanf("%d",&n);
 for(i=0;i<n;i++)
 {
    printf("\n Enter the Burst Time of Process %d",i);
    scanf("\n %d",&t[i]);
 }
 for(i=0;i<n;i++)
   p[i]=i;
   for(i=0;i<n;i++)
   {
     for(k=i+1;k<n;k++)
     {
       if(t[i]>t[k])
       {
              int temp;
```

```
            temp=t[i];

            t[i]=t[k];

            t[k]=temp;

temp=p[i];

            p[i]=p[k];

            p[k]=temp;

    }

  }

  printf("\n\n SHORTEST JOB FIRST SCHEDULING ALGORITHM");

  printf("\n PROCESS ID \t BURST TIME \t WAITING TIME \t TURNAROUND TIME \n\n");

  wt[0]=0;

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

  {

    sum=0;

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

    {

          wt[i]=sum+t[k];

          sum=wt[i];

    }

  }

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

  {

    tt[i]=t[i]+wt[i];

  }

  for(i=0;i<n;i++)
```

```
    {       printf("%5d \t\t5%d \t\t %5d \t\t %5d\n\n",p[i],t[i],wt[i],tt[i]);
    }
    twt=0;
    ttat=t[0];
    for(i=1;i<n;i++)
    {
            twt=twt+wt[i];
            ttat=ttat+tt[i];
    }
    awt=(float)twt/n;
    atat=(float)ttat/n;
    printf("\n AVERAGE WAITING TIME %4.2f",awt);
    printf("\n AVERAGE TURN AROUND TIME %4.2f",atat);
    getch();
  }
}
```

OUTPUT:

```
Enter number of process  3

Enter the Burst Time of Process 04


Enter the Burst Time of Process 13

Enter the Burst Time of Process 25


 SHORTEST JOB FIRST SCHEDULING ALGORITHM

 PROCESS ID      BURST TIME      WAITING TIME    TURNAROUND TIME
```

1	3	0	3
0	4	3	7
2	5	7	12

AVERAGE WAITING TIME 3.33

AVERAGE TURN AROUND TIME 7.33

3. Round Robin (RR) Process Management Algorithm:

- It is designed specially for time-sharing systems.
- It is similar to FCFS, but preemption is added to switch between processes. A time quantum is defined.
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum. If a processes CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue.
- The scheduler assigns a fixed time unit per process, and cycles through them.
- RR scheduling involves extensive overhead, especially with a small time unit.
- Balanced throughput between FCFS and SJF, shorter jobs are completed faster than in FCFS and longer processes are completed faster than in SJF.
- Fastest average response time, waiting time is dependent on number of processes, and not average process length.
- Because of high waiting times, deadlines are rarely met in a pure RR system.

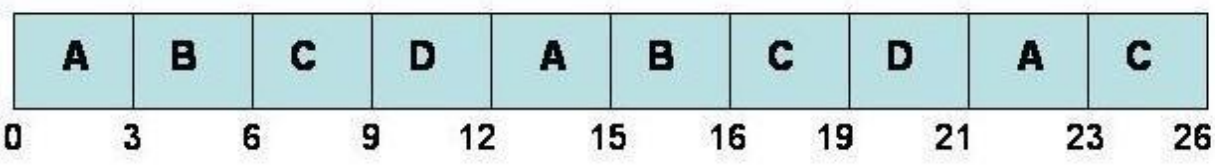
Starvation can never occur, since no priority is given. Order of time unit allocation is based upon process arrival time, similar to FCFS.

To illustrate it, suppose the scheduler is given 4 tasks, A, B, C and D. Each task requires a certain number of time units to complete.

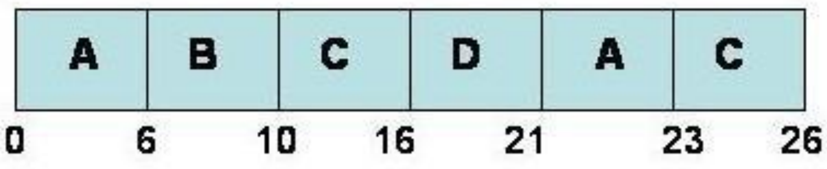
Task	Time units
A	8
B	4
C	9
D	5

RR assigns a **time quantum** (i.e. time slot) to each process waiting to be run. For the jobs A, B, C and D, RR‟s Gantt chart would be:

If time quantum = 3



If time quantum = 6



Metric	**RR**
Turnaround time	(23+16+26+21)/4 = 21.5
Waiting time	(15+12+17+16)/4 = 15

Algorithm:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue and time quantum (or) time slice

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Calculate the no. of time slices for each process where No. of time slice for process(n) = burst time process(n)/time slice

Step 5: If the burst time is less than the time slice then the no. of time slices =1.

Step 6: Consider the ready queue is a circular Q, calculate

(a) Waiting time for process(n) = waiting time of process(n-1)+ burst time of process(n-1) + the time difference in getting the CPU from process(n-1)

(b) Turn around time for process(n) = waiting time of process(n) + burst time of process(n)+ the time difference in getting CPU from process(n).

Step 7: Calculate

(e) Average waiting time = Total waiting Time / Number of process

(f) Average Turnaround time = Total Turnaround Time / Number of process

Step 8: Stop the process

Program:

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

void main()
{
 int ts,pid[10],need[10],wt[10],tat[10],i,j,n,n1;
 int bt[10],flag[10],ttat=0,twt=0;
 float awt,atat;
 clrscr();
 printf("\t\t ROUND ROBIN SCHEDULING \n");
 printf("Enter the number of Processors \n");
 scanf("%d",&n);
 n1=n;
 printf("\n Enter the Timeslice \n");
 scanf("%d",&ts);
 for(i=1;i<=n;i++)
 {
   printf("\n Enter the process ID %d",i);
   scanf("%d",&pid[i]);
   printf("\n Enter the Burst Time for the process");
   scanf("%d",&bt[i]);
   need[i]=bt[i];
 }
 for(i=1;i<=n;i++)
 {
  flag[i]=1;
  wt[i]=0;
 }
 while(n!=0)
 {
   for(i=1;i<=n;i++)
   {
    if(need[i]>=ts)
     {
       for(j=1;j<=n;j++)
       {
                if((i!=j)&&(flag[i]==1)&&(need[j]!=0))
                wt[j]+=ts;
       }
       need[i]-=ts;
       if(need[i]==0)
       {
               flag[i]=0;
               n--;
       }
     }
     else
     {
       for(j=1;j<=n;j++)
       {
                if((i!=j)&&(flag[i]==1)&&(need[j]!=0))
                wt[j]+=need[i];
       }
       need[i]=0;
       n--;
       flag[i]=0;
   }
}
}
for(i=1;i<=n1;i++)
{
  tat[i]=wt[i]+bt[i];
  twt=twt+wt[i];
  ttat=ttat+tat[i];
}
awt=(float)twt/n1;
atat=(float)ttat/n1;

printf("\n\n ROUND ROBIN SCHEDULING ALGORITHM \n\n");
printf("\n\n Process \t Process ID \t BurstTime \t Waiting Time \t TurnaroundTime \n ");
for(i=1;i<=n1;i++)
{
  printf("\n %5d \t %5d \t\t %5d \t\t %5d \t\t %5d \n", i,pid[i],bt[i],wt[i],tat[i]);
}

printf("\n The average Waiting Time=4.2f",awt);
printf("\n The average Turn around Time=4.2f",atat);
```

```
getch();
}
```

OUTPUT:

```
ROUND ROBIN SCHEDULING
Enter the number of Processors
4

 Enter the Timeslice
5

 Enter the process ID 1  5

 Enter the Burst Time for the process 10

 Enter the process ID 2  6

 Enter the Burst Time for the process 15

 Enter the process ID 3  7

 Enter the Burst Time for the process 20

 Enter the process ID 4  8

 Enter the Burst Time for the process 25


 ROUND ROBIN SCHEDULING ALGORITHM

 Process      Process ID    BurstTime     Waiting Time    TurnaroundTime


    1            5              10            15              25

    2            6              15             25             40

    3            7              20            25              45

    4             8             25            20               45

 The average Waiting Time=4.2f
 The average Turn around Time=4.2f
```

Conclusion: Hence, we have successfully implemented basic Process management algorithms such as FCFS, SJF and RR.

Experiment no:4

Aim: Implementation of Process synchronization algorithms like Producer Consumer Problem .

Theory:

1. PRODUCER CONSUMER PROBLEM:

Algorithm:

```
Step 1: start
Step 2: display the menu and read
Step 3: If choice=1 then do the following steps
        a) Get the process to be produced.
        b) Check whether the process already exists.
           If yes display the message
           Else
           Produce the process and display the process list.
Step 4: If choice=2 then do the following steps
        a) Get the process to be consumed
        b) Check whether the process is already produced
           If yes consume the process
           Else
           Display the waiting list
Step 5: stop.
```

Program:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>
struct prod
{
int s;
char wait[20][20];
};
static struct prod se={0};
char produce[20][20],consume[20];
int flag,i,j,z=1;
void main()
{
int ch;
void producer();
void consumer();
clrscr();
do
{
printf("\n\t\t MENU");
printf("\n\t\t ****");
printf("\n 1.producer");
printf("\n 2.consumer");
printf("\n 3.exit");
printf("\n enter your choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:
producer();
break;
case 2:
consumer();
break;
case  3:
exit(0);
break;
}
}
while(ch!=3);
}
void producer()
{
flag=0;
printf("\n enter the producer process name:");
scanf("%s",&produce[++se.s]);
for(i=0;i<se.e;i++)
{
if(strcmp(produce[i],produce[se.e])==0)
{
printf("\n process already exist");
}
```

```
getch();
flag=1;
se.e--;
break;
}
}
for(i=0;i<se.e;i++)
{
if(strcmp(se.wait[i],produce[se.e])==0)
{
j=1;
printf("\n process %s now consumed",produce[se.e]);
se.e--;
flag=2;
break;
}
}
if(flag==1)
return;
else if(flag==2)
{
for(i=j;i<z;i++)
strcpy(se.wait[i],se.wait[i+1]);
z--;
}
else if(flag==0)
{
printf("list of produced process\n");
```

```
for(i=1;i<se.e;i++)
printf("%s\n",produce[i]);
}
}
void consumer()
{
flag=0;
printf("\n enter the consumer process name:);
scanf("%s",&consume);
for(i=1;i<se.e;i++)
{
if(strcmp(produce[i],consume)==0)
{
printf("\n process %s now consumed",produce[i]);
j=1;
flag=1;
break;
}
}
for(i=0;i<z;i++)
{
if(strcmp(produce[i],consume)==0)
{
printf("\n process already exists");
flag=2;
break;
}}
if(flag==1)
```

```
{
for(i=1;i<se.e;i++)
strcpy(produce[i],produce[i+1]);
se.e--;
}
else if(flag==0)
{
strcpy(se.wait[++z],consume);
z++;
printf("list of waiting process\n");
for(i=1;i<z;i++)
printf("%s\n",se.wait[i]);
}}
```

OUTPUT:

```
MENU
1. Producer
2. consumer
3. exit
enter your choice:1
enter the producer process name:p1
list of producer process:p1

MENU
1. producer
2. consumer
3. exit
enter your choice:1
enter the producer process name:p2
list of producer process:p1
p2
```

```
MENU
1. Producer
2. consumer
3. exit
enter your choice:1
enter the producer process name:p1
process already exists

MENU
1. Producer
2. consumer
3. exit
enter your choice:1
enter the producer process name:p3
list of producer process:p1
p2
p3

MENU
1. Producer
2. consumer
3. exit
enter your choice:2
enter the producer process name:p1
process p1 is consumed

MENU
1. Producer
2. consumer
3. exit
enter your choice:2
enter the producer process name:p4
list of waiting process name:p4

MENU
1. producer
2. consumer
3. exit
```

```
enter your choice:3
```

Conclusion: Hence, we have successfully implemented Process synchronization algorithms like Producer Consumer Problem .

Parshvanath Charitable Trust's

A. P. SHAH INSTITUTE OF TECHNOLOGY

(Approved by AICTE New Delhi & Govt. of Maharashtra, Affiliated to University of Mumbai)
(Religious Jain Minority)

Department of Computer Engineering

SE: SEM V **Subject: OS**

EXPERIMENT NO. 05

AIM: Simulate Banker's Algorithm for Deadlock Avoidance to find whether the system is in safe state or not.

Theory:

DEAD LOCK AVOIDANCE

To implement deadlock avoidance & Prevention by using Banker's Algorithm.

Banker's Algorithm:

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

Data structures

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

Safety Algorithm

1. Work and Finish be the vector of length m and n respectively, Work=Available and Finish[i] =False.

2. Find an i such that both

- Finish[i] =False
- Need<=Work

If no such I exists go to step 4.

3. work=work+Allocation, Finish[i] =True;

4. if Finish[1]=True for all I, then the system is in safe state.

Resource request algorithm

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

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

Available=Available-Request I;
Allocation I =Allocation+Request I;
Need i=Need i-Request I;

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

ALGORITHM:

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

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
struct da {
int max[10],al[10],need[10],before[10],after[10];
}p[10];
void main() {
int i,j,k,l,r,n,tot[10],av[10],cn=0,cz=0,temp=0,c=0;
clrscr();
printf("\n Enter the no of processes:");
scanf("%d",&n);
printf("\n Enter the no of resources:");
scanf("%d",&r);
for(i=0;i<n;i++) {
printf("process %d \n",i+1);
for(j=0;j<r;j++) {
printf("maximum value for resource %d:",j+1);
scanf("%d",&p[i].max[j]);
}
for(j=0;j<r;j++) {
printf("allocated from resource %d:",j+1);
scanf("%d",&p[i].al[j]);
p[i].need[j]=p[i].max[j]-p[i].al[j];
}
}
for(i=0;i<r;i++) {
printf("Enter total value of resource %d:",i+1);
scanf("%d",&tot[i]);
}
for(i=0;i<r;i++) {
for(j=0;j<n;j++)
temp=temp+p[j].al[i];
av[i]=tot[i]-temp;
temp=0;
}
printf("\n\t max allocated needed total avail");
for(i=0;i<n;i++) {
```

```
printf("\n P%d \t",i+1);
for(j=0;j<r;j++)

printf("%d",p[i].max[j]);
printf("\t");
for(j=0;j<r;j++)
printf("%d",p[i].al[j]);
printf("\t");
for(j=0;j<r;j++)
printf("%d",p[i].need[j]);
printf("\t");
for(j=0;j<r;j++)
{
if(i==0)
printf("%d",tot[j]);
}
printf(" ");
for(j=0;j<r;j++) {
if(i==0)
printf("%d",av[j]);
}
}
printf("\n\n\t AVAIL BEFORE \t AVAIL AFTER");
for(l=0;l<n;l++)
{
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{
if(p[i].need[j]>av[j])
cn++;
if(p[i].max[j]==0)
cz++;
}
if(cn==0 && cz!=r)
{
for(j=0;j<r;j++)
{
p[i].before[j]=av[j]-p[i].need[j];
p[i].after[j]=p[i].before[j]+p[i].max[j];
av[j]=p[i].after[j];
p[i].max[j]=0;
}
printf("\n p%d \t",i+1);
for(j=0;j<r;j++)
printf("%d",p[i].before[j]);
printf("\t");
for(j=0;j<r;j++)
printf("%d",p[i].after[j]);
cn=0;
cz=0;
c++;
break;
}
else {
cn=0;cz=0;


}
}
}
if(c==n)
printf("\n the above sequence is a safe sequence");
else
printf("\n deadlock occured");
getch();
}
```

Conclusion : Thus we have studied bankers algorithm.

EXPERIMENT NO. 06

Aim: Implementation of Disk scheduling algorithms like FCFS, SSTF, SCAN and LOOK.

Theory:

Concepts Used:

1. First Come First Served Scheduling: First Request will be processed at first.

2. Shortest Seek Time First Scheduling: The SSTF algorithm selects the request with the minimum seek time from the current head position. Since seek time increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest to the current head position.

3. Scan Scheduling: In the SCAN algorithm, the disk arm starts at one end of the disk and move towards the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk.

4. Look Scheduling: The arm goes only as far as the final request in each direction. Then, it reverses the direction immediately, without going all the way to the end of the disk.

1) FIRST COME FIRST SERVED (FCFS) SCHEDULING:

Program:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
int i,sum=0,n,st;
int a[20],b[20],dd[20];
clrscr();
do
{
printf("\nEnter the block number between 0 and 200: ");
scanf("%d",&st);
}
while((st>=200)||(st<0));
printf("\nOur disk head is on the %d block",st);
a[0]=st;
printf("\nEnter the no. of request: ");
scanf("%d",&n);
printf("\nEnter request: ");
for(i=1;i<=n;i++)
{
printf("\nEnter %d request: ",i);
scanf("%d",&a[i]);
do
{
if((a[i]>200)||(a[i]<0))
{
printf("\nBlock number must be between 0 and 200!");
}}while((a[i]>200)||(a[i]<0));
}
for(i=0;i<=n;i++)
dd[i]=a[i];
printf("\n\t\tFIRST COME FIRST SERVE: ");
printf("\nDISK QUEUE:");
for(i=0;i<=n;i++)
printf("\t%d",a[i]);
printf("\n\nACCESS ORDER:");
for(i=0;i<=n;i++)
{
printf("\t%d",dd[i]);
if(i!=n)
sum+=abs(dd[i]-dd[i+1]);
}
printf("\n\nTotal no. of head movements: %d",sum);
getch();
}
```

OUTPUT:

```
Enter the block number between 0 and 200: 53
Our disk head is on the 53 block
Enter the no. of request: 8
Enter request:
Enter 1 request: 98
Enter 2 request: 183
Enter 3 request: 37
Enter 4 request: 122
Enter 5 request: 14
Enter 6 request: 124
Enter 7 request: 65
Enter 8 request: 67

FIRST COME FIRST SERVED:
DISK QUEUE: 53 98 183 37 122 14 124 65 67
ACCESS ORDER: 53 98 183 37 122 14 124 65 67
Total no. of head movements: 640
```

2. SHORTEST SEEK TIME FIRST (SSTF) SCHEDULING:

Program:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
int i,j,z,sum=0,c=0,n,n1,st,min;
int a[20],b[20],dd[20];
clrscr();
do
{
printf("\nEnter the block number between 0 and 200: ");
scanf("%d",&st);
}while((st>=200)||(st<0));
printf("\nOur disk head is on the %d block",st);
a[0]=st;
printf("\nEnter the no. of request: ");
scanf("%d",&n);
printf("\nEnter request: ");
for(i=1;i<=n;i++)
{
printf("\nEnter %d request: ",i);
scanf("%d",&a[i]);
do
{
if((a[i]>200)||(a[i]<0))
{
printf("\nBlock number must be between 0 and 200!");
}}while((a[i]>200)||(a[i]<0));
}
for(i=0;i<=n;i++)
dd[i]=a[i];
n1=n;
b[0]=dd[0];
st=dd[0];
while(n1>0)
{
j=1;
min=abs(dd[0]-dd[1]);
for(i=2;i<n1+1;i++)
{
if(abs(st-dd[i])<=min)
{
min=abs(st-dd[i]);
j=i;
}}
c++;
b[c]=dd[j];
st=dd[j];
dd[0]=dd[j];
--n1;
for(z=j;z<n1+1;z++)
dd[z]=dd[z+1];
dd[z]='\0';
}
printf("\n\t\tSHORTEST SEEK TIME FIRST: ");
printf("\nDISK QUEUE:");
for(i=0;i<=n;i++)
printf("\t%d",a[i]);
printf("\n\nACCESS ORDER:");
```

```
for(i=0;i<=c;i++)
{
printf("\t%d",b[i]);
if(i!=c)
sum+=abs(b[i]-b[i+1]);
}
printf("\n\nTotal no. of head movements: %d",sum);
getch();
}
```

OUTPUT:

```
Enter the block number between 0 and 200: 53
Our disk head is on the 53 block
Enter the no. of request: 8
Enter request:
Enter 1 request: 98
Enter 2 request: 183
Enter 3 request: 37
Enter 4 request: 122
Enter 5 request: 14
Enter 6 request: 124
Enter 7 request: 65
Enter 8 request: 67

SHORTEST SEEK TIME FIRST:
DISK QUEUE: 53 98 183 37 122 14 124 65 67
ACCESS ORDER: 53 65 67 37 14 98 122 124 183
Total no. of head movements: 236
```

3. SCAN SCHEDULING:

Program:

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

```
int i,j,sum=0,c=0,n,st,temp,t;
int a[20],b[20],dd[20];
clrscr();
do
{
printf("\nEnter the block number between 0 and 200: ");
scanf("%d",&st);
}while((st>=200)||(st<0));
printf("\nOur disk head is on the %d block",st);
a[0]=st;
printf("\nEnter the no. of request: ");
scanf("%d",&n);
printf("\nEnter request: ");
for(i=1;i<=n;i++)
{
printf("\nEnter %d request: ",i);
scanf("%d",&a[i]);
do
{
if((a[i]>200)||(a[i]<0))
{
printf("\nBlock number must be between 0 and 200!");
}
}while((a[i]>200)||(a[i]<0));
}
for(i=0;i<=n;i++)
dd[i]=a[i];
for(i=0;i<=n;i++)
for(j=i+1;j<=n;j++)
if(dd[i]>dd[j])
{
temp=dd[i];
dd[i]=dd[j];
dd[j]=temp;
}
for(i=0;i<=n;i++)
{
if(st==dd[i])
{
t=i+1;
b[c]=st;
for(j=i-1;j>=0;j--)
b[++c]=dd[j];
b[++c]=0;
for(j=t;j<=n;j++)
b[++c]=dd[j];
}
}
printf("\n\t\tSCAN TECHNIQUE: ");
printf("\nDISK QUEUE:");
for(i=0;i<=n;i++)
printf("\t%d",a[i]);
printf("\n\nACCESS ORDER:");
for(i=0;i<=c;i++)
{
printf("\t%d",b[i]);
if(i!=c)
sum+=abs(b[i]-b[i+1]);
}
printf("\n\nTotal no. of head movements: %d",sum);
getch();
}
```

OUTPUT:

```
Enter the block number between 0 and 200: 53
Our disk head is on the 53 block
Enter the no. of request: 8
Enter request:
Enter 1 request: 98
Enter 2 request: 183
Enter 3 request: 37
Enter 4 request: 122
Enter 5 request: 14
Enter 6 request: 124
Enter 7 request: 65
Enter 8 request: 67

SCAN TECHNIQUE:
DISK QUEUE: 53 98 183 37 122 14 124 65 67
ACCESS ORDER: 53 37 14 0 65 67 98 122 124 183
Total no. of head movements: 236
```

4. LOOK SCHEDULING:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
int i,j,sum=0,c=0,n,st,temp,t,s;
int a[20],b[20],dd[20];
clrscr();
do
{
printf("\nEnter the block number between 0 and 200: ");
scanf("%d",&st);
}while((st>=200)||(st<0));
printf("\nOur disk head is on the %d block",st);
a[0]=st;
printf("\nEnter the no. of request: ");
scanf("%d",&n);
printf("\nEnter request: ");
for(i=1;i<=n;i++)
{
printf("\nEnter %d request: ",i);
scanf("%d",&a[i]);
do{
if((a[i]>200)||(a[i]<0))
{
printf("\nBlock number must be between 0 and 200!");
}}while((a[i]>200)||(a[i]<0));
}
for(i=0;i<=n;i++)
dd[i]=a[i];
s=a[0];
for(i=0;i<=n;i++)
for(j=i+1;j<=n;j++)
if(dd[i]>dd[j])
{
temp=dd[i];
dd[i]=dd[j];
dd[j]=temp;
}
for(i=0;i<=n;i++)
{
if(s==dd[i])
{
b[c]=st;
for(j=i-1;j>=0;j--)
b[++c]=dd[j];
b[++c]=200;
for(j=n;j>i;j--)
b[++c]=dd[j];
}}
printf("\n\t\tLOOK TECHNIQUE: ");
printf("\nDISK QUEUE:");
for(i=0;i<=n;i++)
printf("\t%d",a[i]);
printf("\n\nACCESS ORDER:");
for(i=0;i<=c;i++)
{
printf("\t%d",b[i]);
if(i!=c)
sum+=abs(b[i]-b[i+1]);
}
printf("\n\nTotal no. of head movements: %d",sum);
getch();
}
```

OUTPUT:

```
Enter the block number between 0 and 200: 53
Our disk head is on the 53 block
Enter the no. of request: 8
Enter request:
Enter 1 request: 98
Enter 2 request: 183
Enter 3 request: 37
Enter 4 request: 122
Enter 5 request: 14
Enter 6 request: 124
Enter 7 request: 65
Enter 8 request: 67

LOOK TECHNIQUE:
DISK QUEUE: 53 98 183 37 122 14 124 65 67
ACCESS ORDER: 53 37 14 200 183 124 122 98 67 65
Total no. of head movements: 360
```

Conclusion: Hence, we have successfully implemented Disk scheduling algorithms like FCFS, SSTF, SCAN and LOOK.

Parshvanath Charitable Trust's

A. P. SHAH INSTITUTE OF TECHNOLOGY

(Approved by AICTE New Delhi & Govt. of Maharashtra, Affiliated to University of Mumbai)
(Religious Jain Minority)

Department of Computer Engineering

SE: SEM V **Subject: OS**

EXPERIMENT NO. 07

Aim: Implementation of various file allocation methods such as Contiguous allocation and Indexed Allocation

Theory:

File Allocation Methods:

One main problem in file management is how to allocate space for files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are:

1) Contiguous Allocation

2) Indexed Allocation.

Each method has its advantages and disadvantages. Accordingly, some systems support all three (e.g. Data General's RDOS). More commonly, a system will use one particular method for all files.

1) Contiguous File Allocation:

- Each file occupies a set of contiguous block on the disk
- Allocation using first fit/best fit.
- A need for compaction
- Only starting block and length of file in blocks are needed to work with the file
- Allows random access
- Problems with files that grow

EXPECTED OUTPUT AND ITS FORM:

Allocates blocks of memory for files using file allocation techniques.

Advantages:

- Effective memory utilization
- Each file occupies set of contiguous blocks on the disk. Random access

Disadvantages:

- Time consuming
- Wastes lot of memory space

Algorithm:

Step 1: Start the process

Step 2: Declare the necessary variables

Step 3: Get the number of files

Step 4: Get the total no. of blocks that fit in to the file

Step 5: Display the file name, start address and size of the file.

Step 6: Stop the program

Program:

```
#include<stdio.h>
 void main()
{
int i,j,n,block[20],start: printf("Enter the no. of file:\n");
scanf("%d",&n);
printf("Enter the number of blocks needed for each file:\n"); for(i=0,i<n;i++)
scanf("%d",&block[i]);
start=0;
printf("\t\tFile name\tStart\tSize of file\t\t\n");
printf("\n\t\tFile1\t\t%d\t\t\t%d\n",start,block[0]); for(i=2;i<=n;i++)
{
Start=start+block[i-2];
Printf("\t\tFile%d\t\t%d\t\tD\n",i,start,block[i-1]);
}
}
```

OUTPUT:

```
Enter the number of file:4

Enter the number of blocks needed for each file: 3

5

6

1


Filename   start     size of file
```

File 1	0	3
File 2	3	5
File 3	8	6
File 4	14	1

2) Indexed File Allocation:

Algorithm:

Step 1: Start.

Step 2: Let n be the size of the buffer

Step 3: check if there are any producer

Step 4: if yes check whether the buffer is full

Step 5: If no the producer item is stored in the buffer

Step 6: If the buffer is full the producer has to wait

Step 7: Check there is any consumer. If yes check whether the buffer is empty

Step 8: If no the consumer consumes them from the buffer

Step 9: If the buffer is empty, the consumer has to wait.

Step 10: Repeat checking for the producer and consumer till required

Step 11: Terminate the process.

Program:

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

OUTPUT:

```
Enter no. of files:2

Enter starting block and size of file1: 2   5

Enter blocks occupied by file1:10

enter blocks of file1:3

2   5   4   6  7   2   6  4  7

Enter starting block and size of file2: 3   4

Enter blocks occupied by file2:5

enter blocks of file2: 2  3   4  5   6

File     index  length

 1          2       10

 2          3        5


Enter file name: venkat

file name is:12803

Index is:0

Block occupied are:
```

Conclusion: Hence, we have successfully implemented various file allocation methods such as Allocation and Indexed Allocation

EXPERIMENT NO. 8

Aim: Implementation of paging .

Theory:

1) PAGING MEMORY MANAGEMENT POLICY:

- Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous.
- The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages.
- Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d). The page number is used as an index into a page table.
- The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.
- The size of a page is typically a power of 2.

Algorithm:

Step 1: Read all the necessary input from the keyboard.

Step 2: Pages - Logical memory is broken into fixed - sized blocks.

Step 3: Frames – Physical memory is broken into fixed – sized blocks.

Step 4: Calculate the physical address using the following

Physical address = (Frame number * Frame size) + offset

Step 5: Display the physical address.

Step 6: Stop the process.

Program:

```
#include <stdio.h>
#include <conio.h>
struct pstruct
{
        int fno;
        int pbit;
}ptable[10];
int pmsize,lmsize,psize,frame,page,ftable[20],frameno;
void info()
{
        printf("\n\nMEMORY MANAGEMENT USING PAGING\n\n");
        printf("\n\nEnter the Size of Physical memory: ");
        scanf("%d",&pmsize);
        printf("\n\nEnter the size of Logical memory: ");
        scanf("%d",&lmsize);
        printf("\n\nEnter the partition size: ");
        scanf("%d",&psize);
        frame = (int) pmsize/psize;
        page = (int) lmsize/psize;
        printf("\nThe physical memory is divided into %d no.of frames\n",frame);
        printf("\nThe Logical memory is divided into %d no.of pages",page);
}
void assign()
{
        int i;
        for (i=0;i<page;i++)
        {
                ptable[i].fno = -1;
                ptable[i].pbit= -1;
                }
                for(i=0; i<frame;i++)
                    ftable[i] = 32555;
                for (i=0;i<page;i++)
                {
                printf("\n\nEnter the Frame number where page %d must be placed: ",i);
                    scanf("%d",&frameno);
                    ftable[frameno] = i;
                    if(ptable[i].pbit == -1)
                    {
                            ptable[i].fno = frameno;
                            ptable[i].pbit = 1;
                    }
                }
                getch();
    //          clrscr();
                printf("\n\nPAGE TABLE\n\n");
                printf("PageAddress FrameNo. PresenceBit\n\n");
                for (i=0;i<page;i++)
                    printf("%d\t\t%d\t\t%d\n",i,ptable[i].fno,ptable[i].pbit);
                printf("\n\n\n\tFRAME TABLE\n\n");
                printf("FrameAddress    PageNo\n\n");
                for(i=0;i<frame;i++)
                    printf("%d\t\t%d\n",i,ftable[i]);
}
```

```
void cphyaddr()
{
                int laddr,paddr,disp,phyaddr,baddr;
                getch();
    //          clrscr();
                printf("\n\n\n\tProcess to create the Physical Address\n\n");
                printf("\nEnter the Base Address: ");
                scanf("%d",&baddr);
                printf("\nEnter theLogical Address: ");
                scanf("%d",&laddr);
                paddr = laddr / psize;
                disp = laddr % psize;
                 if(ptable[paddr].pbit == 1 )
                    phyaddr = baddr + (ptable[paddr].fno*psize) + disp;
                printf("\nThe Physical Address where the instruction present: %d",phyaddr);
}
void main()
{
                clrscr();
                info();
                assign();
                cphyaddr();
                getch();
}
```

OUTPUT:

```
MEMORY MANAGEMENT USING PAGING
Enter the Size of Physical memory: 16
Enter the size of Logical memory: 8
Enter the partition size: 2
```

```
The physical memory is divided into 8 no.of frames
The Logical memory is divided into 4 no.of pages
Enter the Frame number where page 0 must be placed: 5
Enter the Frame number where page 1 must be placed: 6
Enter the Frame number where page 2 must be placed: 7
Enter the Frame number where page 3 must be placed: 2

PAGE TABLE
PageAddress     FrameNo.        PresenceBit
     0              5               1
     1              6               1
     2              7               1
     3              2               1

FRAME TABLE
  FrameAddress        PageNo
       0              32555
       1              32555
       2                3
       3              32555
       4              32555
       5                0
       6                1
       7                2
Process to create the Physical Address
Enter the Base Address: 1000
Enter theLogical Address: 3
The Physical Address where the instruction present: 1013
```

Conclusion: Hence, we have successfully implemented/simulated paging memory management policies.

Experiment no:09

Aim:- To study compilation of linux kernel.

1. Download the latest kernel from kernel.org

The kernel comes as a 20 to 30 MB tar.gz or tar.bz2 file. It will decompress to about 200 MB and during the later compilation you will need additional space.

Example:

```
wget http://www.kernel.org/pub/linux/kernel/v2.4/linux-2.4.19.tar.gz
tar zxvf linux-2.4.19.tar.gz
cd linux-2.4.19
```

2. Configure the kernel options

This is where you select all the features you want to compile into the kernel (e.g. SCSI support, sound support, networking, etc.)

```
make menuconfig
```

* There are different ways to configure what you want compiled into the kernel; if you have an existing configuration from an older kernel, copy the old .config file to the top level of your source and use **make oldconfig** instead of menuconfig. This oldconfig process will carry over your previous settings, and prompt you if there are new features not covered by your earlier .config file. **This is the best way to 'upgrade' your kernel, especially among relatively close version numbers.** Another possibility is **make xconfig** for a graphical version of menuconfig, if you are running X.

3. Make dependencies

After saving your configuration above (it is stored in the ".config" file) you have to build the dependencies for your chosen configuration. This takes about 5 minutes on a 500 MHz system.

```
make dep
```

4. Make the kernel

You can now compile the actual kernel. This can take about 15 minutes to complete on a 500 MHz system.

```
make bzImage
```

The resulting kernel file is "arch/i386/boot/bzImage"

5. Make the modules

Modules are parts of the kernel that are loaded on the fly, as they are needed. They are stored in individual files (e.g. ext3.o). The more modules you have, the longer this will take to compile:

```
make modules
```

6. Install the modules

This will copy all the modules to a new directory, "/lib/modules/a.b.c" where a.b.c is the kernel version

```
make modules_install
```

*** In case you want to re-compile...**

If you want to re-configure the kernel from scratch and re-compile it, you must also issue a couple "make" commands that clean intermediate files. **Note that "make mrproper" deletes your .config file**. The complete process is:

```
make mrproper
make menuconfig
make dep
make clean
make bzImage
make modules
make modules_install
```

*** Installing and booting the new kernel**

For the remainder of this discussion, I will assume that you have LILO installed on your boot sector. Throughout this process, always have a working bootable recovery floppy disk, and **make backups of any files you modify or replace**. A good trick is to name all new files with -a.b.c (kernel version suffix) instead of overwriting files with the same name, although this is not shown in the example that follows.

On most Linux systems, the kernels are stored in the /boot directory. Copy your new kernel to that location and give it a unique name.

Example:

```
cp arch/i386/boot/bzImage /boot/vmlinuz-2.4.19
```

There is also a file called "System.map" that must be copied to the same boot directory.

```
cp System.map /boot
```

Now you are ready to tell LILO about your new kernel. Edit "/etc/lilo.conf" as per your specific needs. Typically, your new entry in the .conf file will look like this:

```
image = /boot/vmlinuz-2.4.19
  label = "Linux 2.4.19"
```

Make sure the image points to your new kernel. It is recommended you keep your previous kernel in the file; this way, if the new kernel fails to boot you can still select the old kernel from the lilo prompt.

Tell lilo to read the changes and modify your boot sector:

```
lilo -v
```

Read the output carefully to make sure the kernel files have been found and the changes have been made. You can now reboot.

Summary of important files created during kernel build:
.config (kernel configuration options, for future reference)
arch/i386/boot/bzImage (actual kernel, copy to /boot/vmlinuz-a.b.c)
System.map (map file, copy to /boot/System.map)
/lib/modules/a.b.c (kernel modules)

Conclusion: successfully compiled linux kernel.