**1.UNIX SYSTEM CALLS**

**Objective**

To write a programs using the following system calls of UNIX operating system:

fork, exec, getpid, exit, wait, close, stat, opendir, readdir

**Getpid**

Each process is identified by a unique *process id* (called a “pid”). The init process (which is the

supreme parent to all processes) posesses id 1. All other processes have some other (possibly

arbitrary) process id. The getpid system call returns the current process’ id as an integer.

// …

int pid = getpid();

printf(“This process’ id is %d\n“,pid);// …

**fork**

The fork system call creates a new child process. Actually, it’s more accurate to say that it *forks*

a currently running process. That is, it creates a *copy* of the current process as a new child

process, and then both processes resume execution from the fork() call. Since it creates two

processes, fork also returns two values; one to teach process. To the parent process, fork returns

the *process id of the newly created child process*. To the child process, fork returns 0. The reason

it returns 0 is precisely because this is an invalid process id. You would have no way of

differentiating between the parent and child processes if fork returned an arbitrary positive

integer to each.

*Therefore, a typical call to fork looks something like this:*

int pid;

if ( (pid = fork()) == 0 ) {

/\* child process executes inside here \*/

}

else {

/\* parent process executes inside here \*/

}

**1. Display parent id & process id**

#include<stdio.h>

int main()

{

printf("\n Parent Process ID %d",getppid());

printf("\n Child Process ID %d\n",getpid());

}

**2. Process creation using fork**

#include<stdio.h>

main()

{

printf("Before FORK \n");

fork();

printf("After FORK \n\n");

}

**3. Process with fork**

#include<stdio.h>

main()

{

int pid;

pid=fork();

if(pid>0)

{

printf("From Parent \n");

printf("Parent process id %d\n",getpid());

}

else

{

printf("From Child \n");

printf("Child process id %d\n",getpid());

}

}

**4. Making child as orphan\***

#include<stdio.h>

main()

{

int pid,pid1;

pid=fork();

if(pid>0)

{

printf("From parent process\n");

printf("Parent process %d \n",getpid());

}

else

{

sleep(1);

printf("From child process\n");

printf("child process %d \n",getpid());

}}

**6. Use of exit system call**

#include<stdio.h>

main()

{

int pid;

pid=fork();

printf("%d\n",pid);

if(pid<0)

{

perror("Child can't be executed\n");

exit(-1);

}

else

{

printf("Child created\n");

exit(0);

}

}

**7. Using fork and exec system call**

#include<stdio.h>

main()

{

int pid;

pid=fork();

printf("%d\n",pid);

if(pid==0)

{

execve("/bin/date\n",NULL,NULL);

exit(0); }

else

{

printf("Parent process %d\n",pid); }}27

**10. Create process & display pid of parent**

**& child**

#include<stdio.h>

#include<dirent.h>

main(int argc,char \*\*argv)

{

int pid,i;

for(i=0;i<atoi(argv[1]);i++)

{

pid=fork();

if(pid==0)

{

printf("child process id %d Parent process

id %d\n",getpid(),getppid());

}}}

**3.SIMULATION OF UNIX COMMANDS**

**Objective**

To simulate the following unix commands

1)ls

**Description**

**ls**

Use ls to see what files you have. Your files are kept in something called a directory.

**1. Write a program for the simulation of ls command.**

#include<stdio.h>

#include<dirent.h>

main()

{

char dirname[10];

DIR\*p;

struct dirent \*d;

printf("Enter directory name\n");

scanf("%s",dirname);

p=opendir(dirname);

if(p==NULL)

{

perror("Cannot find directory");

exit(-1);

}

while(d=readdir(p))

printf("%s\n",d->d\_name);

}

**4. CPU SCHEDULING ALGORITHMS - I**

**Objective**

To schedule the processes using FCFS(First Come First Served) and SJF(Shortest Job

First) scheduling algorithms.

**FCFS**

In this scheduling policy the processes are assigned the CPU according to the order they

arrive.

**SJF**

In this scheduling the process with shortest burst will be selected first. The processes are

sorted in ascending order according to the CPU burst time.

**FIRST COME FIRST SERVED (FCFS)**

**Program:**

#include<stdio.h>

struct process

{

int burst,wait;

}p[20]={0,0};

int main()

{

int n,i,totalwait=0,totalturn=0;

printf("\nEnter The No Of Process :");

scanf("%d",&n);

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

{

printf("Enter The Burst Time (in ms) For Process #%2d :",i+1);

scanf("%d",&p[i].burst);

}

printf("\nProcess \t Waiting Time TurnAround Time ");

printf("\n \t (in ms) (in ms)");

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

{

printf("\nProcess # %-12d%-15d%-15d",i+1,p[i].wait,p[i].wait+p[i].burst);

p[i+1].wait=p[i].wait+p[i].burst;

totalwait=totalwait+p[i].wait;

totalturn=totalturn+p[i].wait+p[i].burst;

}

printf("\n\nAVERAGE\n--------- ");

printf("\nWaiting Time : %f ms",totalwait/(float)n);

printf("\nTurnAround Time : %f ms\n\n",totalturn/(float)n);

return 0;

}

**Output:**

Enter The No Of Process :3

Enter The Burst Time (in ms) For Process # 1 :10

Enter The Burst Time (in ms) For Process # 2 :30

Enter The Burst Time (in ms) For Process # 3 :20

Process Waiting Time TurnAround Time

(in ms) (in ms)

Process # 1 0 10

Process # 2 10 40

Process # 3 40 60

AVERAGE

---------

Waiting Time : 16.666667 ms

TurnAround Time : 36.666667 ms31

**SHORTEST JOB FIRST(SJF)**

**Program:**

#include<stdio.h>

struct process{

int burst,wait,no;

}p[20]={0,0};

int main(){

int n,i,j,totalwait=0,totalturn=0;

printf("\nEnter The No Of Process :");

scanf("%d",&n);

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

printf("Enter The Burst Time (in ms) For Process #%2d :",i+1);

scanf("%d",&p[i].burst);

p[i].no=i+1;

}

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

for(j=0;j<n-i-1;j++)

if(p[j].burst>p[j+1].burst){

p[j].burst^=p[j+1].burst^=p[j].burst^=p[j+1].burst;

p[j].no^=p[j+1].no^=p[j].no^=p[j+1].no;

}

printf("\nProcess \t Waiting Time TurnAround Time ");

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

printf("\nProcess # %-12d%-15d%-15d",p[i].no,p[i].wait,p[i].wait+p[i].burst);

p[i+1].wait=p[i].wait+p[i].burst;

totalwait=totalwait+p[i].wait;

totalturn=totalturn+p[i].wait+p[i].burst;

}

printf("\n\nAverage\n---------");

printf("\nWaiting Time : %f ms",totalwait/(float)n);

printf("\nTurnAround Time : %f ms\n\n",totalturn/(float)n);

return 0;

}

**Output:**

Enter The No Of Process :3

Enter The Burst Time (in ms) For Process # 1 :20

Enter The Burst Time (in ms) For Process # 2 :30

Enter The Burst Time (in ms) For Process # 3 :10

Process Waiting Time TurnAround Time

Process # 3 0 10

Process # 1 10 30

Process # 2 30 60

Average

---------

Waiting Time : 13.333333 ms

TurnAround Time : 33.333333 ms

**5. CPU SCHEDULING ALGORITHMS - II**

**Objective**

To schedule the processes using Priority and Round Robin scheduling algorithms.

**Description**

**Priority**

In this scheduling policy the processes are given certain priorities usually specified as a

number. They are sorted according to the priorities and the process with highest priority is

scheduled first.

**Round Robin**

In this algorithm, a time quantum is fixed for the process to get executed in the CPU.

After that time quantum, the process is pre-empted and CPU is scheduled to another process.

This will continue until all processes in the system complete their turn.

**Sample Input:**

Enter the number of processes:3

Process 1

Enter the CPU burst time: 5

Process 2

Enter the CPU burst time: 10

Process 3

Enter the CPU burst time:4

**Sample Output:**

Process Name ArrivalTime BurstTime Wait time start End

The order in which the

processes are executed:

Waiting time for every

Proess Total waiting time is:

Average waiting time

for given FCFS :

Average turnaround time:

**PRIORITY SCHEDULING**

**Program:**

#include<stdio.h>

struct process{

int burst,wait,no,priority;

}p[20]={0,0};

int main(){

int n,i,j,totalwait=0,totalturn=0;

printf("\nEnter The No Of Process :");

scanf("%d",&n);

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

printf("Enter The Burst Time (in ms) For

Process #%2d :",i+1);

scanf("%d",&p[i].burst);

printf("Enter The Priority For Process

#%2d :",i+1);

scanf("%d",&p[i].priority);

p[i].no=i+1;

}

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

for(j=0;j<n-i-1;j++)

if(p[j].priority>p[j+1].priority){

p[j].burst^=p[j+1].burst^=p[j].burst^=p[j+1]

.burst;

p[j].no^=p[j+1].no^=p[j].no^=p[j+1].no;

//Simple way to swap 2 var’s

p[j].priority^=p[j+1].priority^=p[j].priority^

=p[j+1].priority;

//printf("j");

}

printf("\nProcess \t Starting Ending

Waiting TurnAround ");

printf("\n \t Time Time Time

Time ");

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

printf("\nProcess # %-11d%-10d%-10d%-

10d%10d",p[i].no,p[i].wait,p[i].wait+p[i].burst,p[i].wait,p[i].wait+p[i].burst);

p[i+1].wait=p[i].wait+p[i].burst;

totalwait=totalwait+p[i].wait;

totalturn=totalturn+p[i].wait+p[i].burst;

}

printf("\n\nAverage\n---------");

printf("\nWaiting Time : %f

ms",totalwait/(float)n);

printf("\nTurnAround Time : %f

ms\n\n",totalturn/(float)n);

return 0;

}

**Output:**

Enter The No Of Process :3

Enter The Burst Time (in ms) For Process #

1 :30

Enter The Priority For Process # 1 :2

Enter The Burst Time (in ms) For Process #

2 :20

Enter The Priority For Process # 2 :1

Enter The Burst Time (in ms) For Process #

3 :40

Enter The Priority For Process # 3 :3

Process Starting Ending Waiting

TurnAround

Time Time Time Time

Process # 2 0 20 0 20

Process # 1 20 50 20 50

Process # 3 50 90 50 90

Average

---------

Waiting Time : 23.333333 ms

TurnAround Time : 53.333333 ms33

**ROUND ROBIN SCHEDULING**

**Program:**

#include<stdio.h>

struct process{

int burst,wait,comp,f;

}p[20]={0,0};

int main(){

int

n,i,j,totalwait=0,totalturn=0,quantum,flag=1,

time=0;

printf("\nEnter The No Of Process :");

scanf("%d",&n);

printf("\nEnter The Quantum time (in ms) :");

scanf("%d",&quantum);

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

printf("Enter The Burst Time (in ms) For Process #%2d :",i+1);

scanf("%d",&p[i].burst);

p[i].f=1;

}

printf("\nOrder Of Execution \n");

printf("\nProcess Starting Ending Remaining");

printf("\n Time Time Time");

while(flag==1){

flag=0;

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

if(p[i].f==1){

flag=1;

j=quantum;

if((p[i].burst-p[i].comp)>quantum){

p[i].comp+=quantum;

}

else{

p[i].wait=time-p[i].comp;

j=p[i].burst-p[i].comp;

p[i].comp=p[i].burst;

p[i].f=0;

}

printf("\nprocess # %-3d %-10d %-10d %-10d",i+1,time,time+j,p[i].burstp[i].comp);

time+=j;

}}}

printf("\n\n------------------");

printf("\nProcess \t Waiting Time

TurnAround Time ");

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

printf("\nProcess # %-12d%-15d%-

15d",i+1,p[i].wait,p[i].wait+p[i].burst);

totalwait=totalwait+p[i].wait;

totalturn=totalturn+p[i].wait+p[i].burst;

}

printf("\n\nAverage\n------------------");

printf("\nWaiting Time : %f

ms",totalwait/(float)n);

printf("\nTurnAround Time : %f

ms\n\n",totalturn/(float)n);

return 0;

}

**Output:**

Enter The No Of Process :3

Enter The Quantum time (in ms) :5

Enter The Burst Time (in ms) For Process #

1 :25

Enter The Burst Time (in ms) For Process #

2 :30

Enter The Burst Time (in ms) For Process #

3 :54

Order Of Execution

Process Starting Ending Remaining

Time Time Time

process # 1 0 5 20

process # 2 5 10 25

process # 3 10 15 49

process # 1 15 20 15

process # 2 20 25 20

process # 3 25 30 44

process # 1 30 35 10

process # 2 35 40 15

process # 3 40 45 39

process # 1 45 50 5

process # 2 50 55 10

process # 3 55 60 34

process # 1 60 65 0

process # 2 65 70 5

process # 3 70 75 29

process # 2 75 80 0

process # 3 80 85 24

process # 3 85 90 19

process # 3 90 95 14

process # 3 95 100 9

process # 3 100 105 4

process # 3 105 109 0

Process Waiting Time TurnAround Time

Process # 1 40 65

Process # 2 50 80

Process # 3 55 109

Average

Waiting Time : 48.333333 ms

TurnAround Time : 84.666667 ms

**8. MEMORY MANAGEMENT SCHEMES - I**

**Objective**

To implement first fit, best fit and worst fit storage allocation algorithms for memory

management.

**First-Fit**

Allocate the first hole that is big enough. Searching can start either at the beginning of the

set of holes or where the previous first-fit search ended. We can stop searching as soon as we

find a free hole that is large enough.

**Best-Fit**

Allocate the smallest hole that is big enough. We must search the entire list unless the list

is kept ordered by size. The strategy produces the smallest leftover hole.

**Worst fit**

Allocate the biggest hole.

**FIRST FIT**

**Program:**

#include<stdio.h>

struct process{

int size;

char name[20];

int id;

}p[20]={0,0};

struct block{

int size;

int id;

}b[20]={0,0};

int main(){

int

nb,np,i,j,totalwait=0,totalturn=0,quantum=4,

flag=1,time=0;

printf("\nEnter The No Of Blocks :");

scanf("%d",&nb);

for(i=0;i<nb;i++){

printf("Enter The Size of Block $ %-3d

:",i+1);

scanf("%d",&b[i].size);

}

printf("\nEnter The No Of Processes

:");

scanf("%d",&np);

for(i=0;i<np;i++){

printf("Enter The Name of process # %-3d

:",i+1);

scanf("%s",p[i].name);

printf("Enter The Size of process # %-3d

:",i+1);

scanf("%d",&p[i].size);

}

for(i=0;i<np;i++){

for(j=0;j<nb;j++){

if(b[j].id==0&&p[i].size<=b[j].size){

b[j].id=i+1;

p[i].id=j+1;

flag=1;

break;

}}

}

printf("Block \n\n-----------");

printf("\nBlock ID Block\_Size

Process\_Name Process\_Size");

for(i=0;i<nb;i++){

if(b[i].id)

printf(" \nBlock #%-7d%-10d%-10s%-10d ",i+1,b[i].size,p[b[i].id-1].name, p[b[i].id- 1].size);

else

printf(" \nBlock #%-7d%-10dEmpty

Empty ",i+1,b[i].size);

}

printf("\n\nProcess \n-----------");

printf("\nProcess\_Name Process\_Size

Block ID Block\_Size");

for(i=0;i<np;i++){

if(p[i].id)

printf(" \nProcess $ %-10s%-10d%-10d%-

10d ",p[i].name,p[i].size,p[i].id, b[p[i].id-

1].size);

else

printf(" \nProcess $ %-10s%-10dWaiting

Waiting ",p[i].name,p[i].size);

}

printf("\n");

}

**Output:**

Enter The No Of Blocks :3

Enter The Size of Block $ 1 :30

Enter The Size of Block $ 2 :20

Enter The Size of Block $ 3 :10

Enter The No Of Processes :3

Enter The Name of process # 1 :P

Enter The Size of process # 1 :10

Enter The Name of process # 2 :Q

Enter The Size of process # 2 :30

Enter The Name of process # 3 :R

Enter The Size of process # 3 :20

Block

-----------

Block ID Block\_Size Process\_Name

Process\_Size

Block #1 30 P 10

Block #2 20 R 20

Block #3 10 Empty Empty

Process

-----------

Process\_Name Process\_Size Block ID

Block\_Size

Process $ P 10 1 30

Process $ Q 30 Waiting Waiting

Process $ R 20 2 2036

**BEST FIT**

**Program:**

#include<stdio.h>

#include<conio.h>

struct process{

int size;

int id;

}p[20]={0,0};

struct block{

int no;

int size;

int id;

}b[20]={0,0};

int main(){

int

nb,np,i,j,totalwait=0,totalturn=0,quantum=4,

flag=1,time=0;

clrscr();

printf("\nEnter The No Of Blocks :");

scanf("%d",&nb);

for(i=0;i<nb;i++){

printf("Enter The Size of Block $ %-3d :",i+1);

scanf("%d",&b[i].size);

b[i].no=i+1;

}

printf("\nEnter The No Of Processes :");

scanf("%d",&np);

for(i=0;i<np;i++){

printf("Enter The Size of process # %-3d :",i+1);

scanf("%d",&p[i].size);

}

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

for(j=0;j<nb-i-1;j++)

if(b[j].size>b[j+1].size){

b[j].size^=b[j+1].size^=b[j].size^=b[j+1].size;

b[j].no^=b[j+1].no^=b[j].no^=b[j+1].no;

}

for(i=0;i<np;i++){

for(j=0;j<nb;j++){

if(b[j].id==0&&p[i].size<=b[j].size){

b[j].id=i+1;

p[i].id=b[j].no;

flag=1;

break;

}}}

printf("Block \n\n-----------");

printf("\nBlock\_ID Block\_Size

Process\_ID Process\_Size");

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

//for(j=0;j<nb;j++)

{

//if(j+1==b[j].no){

if(b[i].id)

printf(" \nBlock #%-7d%-10d%-10d%-10d ",b[i].no,b[i].size,b[i].id, p[b[i].id-1].size);

else

printf(" \nBlock #%-7d%-10dEmpty Empty ",b[i].no,b[i].size);

}

//}

printf("\n\nProcess \n-----------");

printf("\nProcess\_ID Process\_Size

Block\_ID Block\_Size");

for(i=0;i<np;i++){

if(p[i].id)

printf(" \nProcess $ %-3d%-14d%-10d%- 10d ",i+1,p[i].size,p[i].id, b[p[i].id-1].size);

else

printf(" \nProcess $ %-3d%-14dWaiting Waiting ",i+1,p[i].size);

}

printf("\n");

getch(); }

**Output:**

Enter The No Of Blocks :3

Enter The Size of Block $ 1 :30

Enter The Size of Block $ 2 :20

Enter The Size of Block $ 3 :10

Enter The No Of Processes :3

Enter The Size of process # 1 :10

Enter The Size of process # 2 :20

Enter The Size of process # 3 :30

Block

-----------

Block\_ID Block\_Size Process\_ID

Process\_Size

Block #3 10 1 10

Block #2 20 2 20

Block #1 30 3 30

Process

-----------

Process\_ID Process\_Size Block\_ID

Block\_Size

Process $ 1 10 3 30

Process $ 2 20 2 20

Process $ 3 30 1 10