EXPERIMENT: 0 2

OBJECTIVE

Simulate the following CPU scheduling algorithms to find turnaround time and waiting time

a) FCFS b) SJF c) Round Robin d) Priority

DESCRIPTION

Assume all the processes arrive at the same time.

a) FCFS CPU SCHEDULING ALGORITHM

AIM: Write a C Program to implement FCFS CPU Scheduling Method

For FCFS scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. The scheduling is performed on the basis of arrival time of the processes irrespective of their other parameters. Each process will be executed according to its arrival time. Calculate the waiting time and turnaround time of each of the processes accordingly.

Algorithm for FCFS

Step 1: Create the number of process.

Step 2: Get the ID and Service time for each process.

Step 3: Initially, Waiting time of first process is zero and Total time for the first process is the starting time of that process.

Step 4: Calculate the Total time and Processing time for the remaining processes.

Step 5: Waiting time of one process is the Total time of the previous process.

Step 6: Total time of process is calculated by adding Waiting time and Service time.

Step 7: Total waiting time is calculated by adding the waiting time for lack process.

Step 8: Total turn around time is calculated by adding all total time of each process.

Step 9: Calculate Average waiting time by dividing the total waiting time by total number of process.

Step 10: Calculate Average turn around time by dividing the total time by the number of process.

Step 11: Display the result.

SOURCE CODE:

```
#include<stdio.h>
void main()
{
int bt[20], wt[20], tat[20], i, n;
float wtavg, tatavg;
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\nEnter Burst Time for Process %d -- ", i);
scanf("%d", &bt[i]);
}
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0];
for(i=1;i<n;i++)
{
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);
 printf("\nAverage Waiting Time -- %f", wtavg/n);
 printf("\nAverage Turnaround Time -- %f", tatavg/n);
}
```

INPUT

```
Enter the number of processes -- 3
Enter Burst Time for Process 0 -- 24
Enter Burst Time for Process 1 -- 3
Enter Burst Time for Process 2 --  3
```

OUTPUT

PROCESS TIME	BURST TIME	WAITING TIME	TURNAROUND TIME
P0	24	0	24
P1	3	24	27
P2	3	27	30

Average Waiting Time-- 17.000000

Average Turnaround Time --27.000000

b) SJF CPU SCHEDULING ALGORITHM

AIM: Write a C Program to implement SJF CPU Scheduling Method

For SJF scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. Arrange all the jobs in order with respect to their burst times. There may be two jobs in queue with the same execution time, and then FCFS approach is to be performed. Each process will be executed according to the length of its burst time. Then calculate the waiting time and turnaround time of each of the processes accordingly.

Algorithm for SJF

Step 1: Declare the array size.

Step 2: Get the number of elements to be inserted.

Step 3: Select the process which have shortest burst will execute first.

Step 4: If two process have same burst length then FCFS scheduling algorithm used.

Step 5: Make the average waiting the length of next process.

Step 6: Start with the first process from it's selection as above and let other process to be in queue.

Step 7: Calculate the total number of burst time.

Step 8: Display the values.

SOURCE CODE:

```
#include<stdio.h>
void main()
{
 int p[20], bt[20], wt[20], tat[20], i, k, n, temp;
```

```
 float wtavg, tatavg;
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
 for(i=0;i<n;i++)
 {
p[i]=i;
printf("Enter Burst Time for Process %d -- ", i);
scanf("%d", &bt[i]);
 }
for(i=0;i<n;i++)
 for(k=i+1;k<n;k++)
 if(bt[i]>bt[k])
{
 temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;
 temp=p[i];
 p[i]=p[k];
 p[k]=temp;
 }
 wt[0] = wtavg = 0;
 tat[0] = tatavg = bt[0];
 for(i=1;i<n;i++)
 {
wt[i] = wt[i-1] +bt[i-1];
 tat[i] = tat[i-1] +bt[i];
 wtavg = wtavg + wt[i];
 tatavg = tatavg + tat[i];
}
 printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);
 printf("\nAverage Waiting Time -- %f", wtavg/n);
 printf("\nAverage Turnaround Time -- %f", tatavg/n);
 }
```

INPUT

Enter the number of processes --4

Enter Burst Time for Process 0 --6

Enter Burst Time for Process 1 --8

Enter Burst Time for Process 2 --7

Enter Burst Time for Process 3 --3

OUTPUT

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
P0	6	3	9
P1	8	16	24
P2	7	9	16
P3	3	0	3

Average Waiting Time --7.000000

Average Turnaround Time --13.000000

c) ROUND ROBIN CPU SCHEDULING ALGORITHM

AIM: Write a C Program to implement Round Robin CPU Scheduling Method

For round robin scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the size of the time slice. Time slices are assigned to each process in equal portions and in circular order, handling all processes execution. This allows every process to get an equal chance. Calculate the waiting time and turnaround time of each of the processes accordingly.

Algorithm for RR

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 round 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

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

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

Step 8: Stop the process

SOURCE CODE:

```
#include<stdio.h>
void main()
{
int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;
float awt=0,att=0,temp=0;
printf("Enter the no of processes -- ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("\nEnter Burst Time for process %d -- ", i+1);
scanf("%d",&bu[i]);
ct[i]=bu[i];
}
printf("\nEnter the size of time slice -- ");
scanf("%d",&t);
max=bu[0];
for(i=1;i<n;i++)
if(max<bu[i])
max=bu[i];
for(j=0;j<(max/t)+1;j++)
for(i=0;i<n;i++)
if(bu[i]!=0)
if(bu[i]<=t)
{
tat[i]=temp+bu[i];
temp=temp+bu[i];
bu[i]=0;
```

```
}
else
{
bu[i]=bu[i]-t;
temp=temp+t;
}
for(i=0;i<n;i++)
{
wa[i]=tat[i]-ct[i];
att+=tat[i];
awt+=wa[i];
}
printf("\nThe Average Turnaround time is -- %f",att/n);
printf("\nThe Average Waiting time is -- %f ",awt/n);
printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]);
}
```

INPUT:

Enter the no of processes – 3

Enter Burst Time for process 1 – 24

Enter Burst Time for process 2 –3

Enter Burst Time for process 3 –3

Enter the size of time slice – 3

OUTPUT:

The Average Turnaround time is – 15.666667

The Average Waiting time is -- 5.666667

PROCESS TIME	BURST TIME	WAITING TIME	TURN AROUND
1	24	6	30
2	3	4	7
3	3	7	10

d)PRIORITY CPU SCHEDULING ALGORITHM

AIM: Write a C Program to implement Priority CPU Scheduling Method

For priority scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the priorities. Arrange all the jobs in order with respect to their priorities. There may be two jobs in queue with the same priority, and then FCFS approach is to be performed. Each process will be executed according to its priority. Calculate the waiting time and turnaround time of each of the processes accordingly.

Algorithm for Priority

STEP 1: Declare the array size.

STEP 2: Get the number of elements to be inserted.

STEP 3: Get the priority for each process and value

STEP 4: Start with the higher priority process from it's initial position let other process to be queue.

STEP 5: Calculate the total number of burst time.

STEP 6: Display the values

SOURCE CODE:

```
#include<stdio.h>
void main( )
{
int et[20],at[10],n,i,j,temp,p[10],st[10],ft[10],wt[10],ta[10]; int totwt=0,totta=0;
float awt,ata;
char pn[10][10],t[10]; clrscr();
printf("Enter the number of process:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Enter process name,arrivaltime,execution time & priority:");
scanf("%s%d%d%d",pn[i],&at[i],&et[i],&p[i]);
}
for(i=0;i<n;i++)
for(j=0;j<n;j++)
```

```
{
if(p[i]<p[j])
{
temp=p[i];
p[i]=p[j];
p[j]=temp;
 temp=at[i];
at[i]=at[j];
at[j]=temp;
temp=et[i];
et[i]=et[j];
et[j]=temp;
strcpy(t,pn[i]);
strcpy(pn[i],pn[j]);
strcpy(pn[j],t);
}
}
for(i=0;i<n;i++)
{
if(i==0)
{
st[i]=at[i];wt[i]=st[i]-at[i];ft[i]=st[i]+et[i];ta[i]=ft[i]-at[i];
}
else
{
st[i]=ft[i-1];wt[i]=st[i]-at[i];ft[i]=st[i]+et[i];ta[i]=ft[i]-at[i];
}
totwt+=wt[i];
totta+=ta[i];
}
awt=(float)totwt/n;
ata=(float)totta/n;
printf("\nPname\tarrivaltime\texecutiontime\tpriority\twaitingtime\ttatime");
for(i=0;i<n;i++)
printf("\n%s\t%5d\t\t%5d\t\t%5d\t\t%5d\t\t%5d",pn[i],at[i],et[i],p[i],wt[i],ta[i]);
printf("\nAverage waiting time is:%f",awt);
printf("\nAverage  turnaroundtime  is:%f",ata);
```

}

INPUT

Enter the number of processes: 3

Enter the Process Name, Arrival Time, execution time & priority: p1 2 3 1

Enter the Process Name, Arrival Time, execution time & priority: p2 4 5 2

Enter the Process Name, Arrival Time, execution time & priority: p3 5 6 3

OUTPUT

PROCESS	ARRIVAL TIME	EXECUTION TIME	PRIORITY	TA TIME
P1	2	3	1	3
P2	4	5	2	16
P3	5	6	3	11

Average Waiting Time: 2.0000

Average Turn Around Time: 6.6667

EXPERIMENT: 03

PRODUCER-CONSUMER PROBLEM

Aim: Write a C program to simulate producer-consumer problem using semaphores.

DESCRIPTION

Producer-Consumer problem is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process. One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, there must be available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Algorithm:

Step 1: Start

Step 2: Define the maximum buffer size.

Step 3:Enter the number of producers and consumers.

Step 4: The producer produces the job and put it in the buffer.

Step 5:The consumer takes the job from the buffer.

Step 6:If the buffer is full the producer goes to sleep.

Step 7:If the buffer is empty then consumer goes to sleep.

Step 8:Stop

SOURCE CODE:

```
#include<stdio.h>
void main()
{
int buffer[10], bufsize, in, out, produce, consume, choice=0;
in = 0;
out = 0;
bufsize = 10;
while(choice !=3)
```

```
{
printf("\n1. Produce \t 2. Consume \t3. Exit");
printf("\nEnter your choice: ");
scanf("%d", &choice);
        switch(choice)

{
case 1: if((in+1)%bufsize==out)
printf("\nBuffer is Full");
else
{
printf("\nEnter the value: ");
scanf("%d", &produce);
buffer[in] = produce;
in = (in+1)%bufsize;
} break;
case 2: if(in == out)
printf("\nBuffer is Empty");
else
{
consume = buffer[out];
printf("\nThe consumed value is %d", consume);
out = (out+1)%bufsize;
}
break;
}
}
}
```

***OUTPUT*:**

```
1. Produce  2. Consume 3. Exit
Enter your choice: 2
Buffer is Empty
1. Produce  2. Consume 3. Exit
Enter your choice: 1
Enter the value: 100
1. Produce  2. Consume 3. Exit
Enter your choice: 2
```

```
The consumed value is 100
1. Produce  2. Consume 3. Exit
Enter your choice: 3
```

EXPERIMENT: 01

Objective:

Develop a C program to implement the process system calls (fork (), exec (), wait (), create process, terminate process)

AIM:

Write a C Program to implement the process system calls.

ALGORITHM:

STEP 1: Start the program.

STEP 2: Declare the variables pid,pid1,pid2.

STEP 3: Call fork() system call to create process.

STEP 4: If pid==-1, exit.

STEP 5: Ifpid!=-1 , get the process id using getpid().

STEP 6: Print the process id.

STEP 7:Stop the program

Source Code:

```
#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/wait.h>

#include<unistd.h>

int main()
{

pid_t child_pid;

//Fork a child process

child_pid=fork();

if(child_pid== -1)

{
```

```
Perror("Fork failed");
exit(EXIT_FAILURE);
}
if(child_pid==0)
{
//This is the child process
printf("Child process: PID =%d\n",getpid());
//Execute a command in the child process using exec
execlp("/bin/ls","ls","-1",(char*)NULL);
//If exec fails
perror("Exec failed");
exit(Exit_FAILURE);
}
else
{
//This is the parent process
//Wait for the child process to complete

wait(NULL);
printf("parent process: Child process terminated\n");
}
return ():
}
```

EXPERIMENT-04

Objective:

Develop a C program which demonstrates inter process communication between a reader process and a writer process. Use mkfifo , open, read, write and close APIs in your program.

AIM:

Write a C Program to implement the inter Process communication between a reader process and a writer process.

ALGORITHM:

STEP 1: Start the Program.

STEP 2: Create a named pipe using mkfifo() function.

STEP 3: In the child process (reader), open the named pipe for reading using open() function.

STEP 4: In the parent process (writer), open the named pipe for writing using open() function.

STEP 5: Write data to the named pipe using write() function in the parent process.

STEP 6: Read data from the named pipe using read() function in the child process.

STEP 7: Close the named pipe using close() function in both processes.

STEP 8: Clean up by removing the named pipe using unlink() function.

STEP 9: Stop the Program.

SOURCE CODE:

mkfifo.c

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

```
int main()
{
int res;
res = mkfifo("fifo1",0777);
printf("pipe created\n");
}
```

sender.c

```
#include<unistd.h>
#include<stdio.h>
#include<fcntl.h>
int main()
{
int res,n;
res=open("fifo1",O_WRONLY);
write(res,"Message",7);
printf("Sender Process %d sent the data\n",getpid());
 }
```

receiver.c

```
#include<unistd.h>
#include<stdio.h>
#include<fcntl.h>
int main()
{
int res,n;
char buffer[100];
```

```
res=open("fifo1",O_RDONLY);

n=read(res,buffer,100);

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

printf("Data received by receiver %d is: %s\n",getpid(), buffer);

}
```

EXPERIMENT-05

DEADLOCK AVOIDANCE

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock. Deadlock avoidance is one of the techniques for handling deadlocks. This approach requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process. Banker's algorithm is a deadlock avoidance algorithm that is applicable to a system with multiple instances of each resource type.

Objective:

Develop a C program to Simulate Bankers Algorithm for Deadlock Avoidance.

Aim:

To write a C program to implement bankers algorithm for dead lock avoidance.

Algorithm:

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

SOURCE CODE:

```
#include<stdio.h>
struct file
{
int all[10];
```

```
int max[10];
int need[10];
int flag;
};
 void main()
{
struct file f[10];
int fl;
 int i, j, k, p, b, n, r, g, cnt=0, id, newr;
 int avail[10],seq[10];
printf("Enter number of processes -- ");
scanf("%d",&n);
printf("Enter number of resources -- ");
scanf("%d",&r);
for(i=0;i<n;i++)
{
 printf("Enter details for P%d",i);
 printf("\n Enter allocation\t -- \t");
for(j=0;j <r;j++)
scanf("%d",&f[i].max[j]);
f[i].flag=0;
}
 printf("\nEnter Available Resources\t -- \t");
for(i=0;i i<r;i++)
scanf("%d",&avail[i]);
printf("\n Enter New Request Details -- ");
printf("\n Enter pid \t -- \t");
scanf("%d",&id);
printf("Enter Request for Resources \t -- \t");
for(i=0;i<r;j++)
{
scanf("%d",&newr);
f[id].all[i] += newr;
```

```
avail[i]=avail[i] - newr;
}
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{
f[i].need[j]=f[i].max[j]-f[i].all[j];
if(f[i].need[j]<0)
f[i].need[j]=0;
}
}
cnt=0;
fl=0;
while(cnt!=n)
{
 g=0;
for(j=0;j<n;j++)
{
 if(f[j].flag==0)
 {
 b=0;
 for(p=0;p<r;p++)
{
 if(avail[p]>=f[j].need[p])
b=b+1;
 else
b=b-1;
}
 if(b==r)
{
 printf("\nP%d is visited",j);
seq[fl++]=j;
f[j].flag=1;
```

```
for(k=0;k <r;k++)
avail[k]=avail[k]+f[j].all[k];
cnt=cnt+1;
 printf("(");
for(k=0;k<r;k++)
printf("%3d",avail[k]);
 printf(")");
 g=1;
}
 }
}
if(g==0)
{
printf("\n REQUEST NOT GRANTED -- DEADLOCK OCCURRED");
printf("\n SYSTEM IS IN UNSAFE STATE");
goto y;
}
}
printf("\n SYSTEM IS IN SAFE STATE");
printf("\n The Safe Sequence is -- (");
for(i=0;i<fl;i++)
printf("P%d ",seq[i]);
printf(")");
y: printf("\nProcess\t\tAllocation\t\tMax\t\t\tNeed\n");
for(i=0;i<n;i++)
{
 printf("P%d\t",i);
for(j=0;j<r;j++)
printf("%6d",f[i].all[j]);
for(j=0;j<r;j++)
printf("%6d",f[i].max[j]);
for(j=0;j<r;j++)
printf("%6d",f[i].need[j]);
```

```
printf("\n");
}
}
```

INPUT:

```
Enter number of processes -- 5
Enter number of resources -- 3
Enter details for P0
Enter allocation – 0 1 0
Enter Max -- 7 5 3

Enter details for P1
Enter allocation -- 2 0 0
Enter Max ---- 3 2 2

Enter details for P2
Enter allocation -- 3 0 2
Enter Max -- 9 0 2

Enter details for P3
Enter allocation -- 2 1 1
Enter Max -- 2 2 2

Enter details for P4
Enter allocation -- 0 0 2
Enter Max -- 4 3 3

Enter Available Resources -- 3 3 2
Enter New Request Details –
Enter pid-- 1
Enter Request for Resources – 1 0 2
```

OUTPUT:

P1 is visited (5 3 2)

P3 is visited (7 4 3)

P4 is visited (7 4 5)

P0 is visited (7 5 5)

P2 is visited (10 5 7)

SYSTEM IS IN SAFE STATE

The Safe Sequence is -- (P1 P3 P4 P0 P2)

Process	Allocation			Max			Need		
P0	0	1	0	7	5	3	7	4	3
P1	3	0	2	3	2	2	0	2	0
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	0	0	2	4	3	3	4	3	1