**Ex.No:4 IMPLEMENT THE CPU SCHEDULING ALGORITHMS**

**Date**:

**A) FCFS B) SJFS C) Priority D) Round Robin**

AIM:

To implement FCFS, SJF, Priority and Round robin scheduling algorithms using C.

**PROBLEM DESCRIPTION:**

**FIRST COME FIRST SERVE SCHEDULING:**

With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. The FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU either by terminating or by requesting I/O. While considering the performance of the FCFS scheduling algorithm, the higher burst time process makes the lower burst time process to wait for a long time. This effect is known as convoy effect.

**SHORTEST JOB FIRST SCHEDULING:**

This algorithm associates with each process the length of the latter’s next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie. The SJF algorithm may be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue while a previous process is executing. The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes.

**PRIORITY SCHEDULING:**

A priority is associated with each process and the CPU is allocated to the process with the highest priority. Equal priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority(p) is the inverse of the next CPU burst. The larger the CPU burst, the lower the priority and vice versa.

ROUND-ROBIN SCHEDULING:

The round-robin scheduling algorithm is designed especially for time-sharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a time quantum is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as circular queue.

1.FCFS:

ALGORITHM:

1: Get the number of processes and burst time.

2: The process is executed in the order given by the user.

3: Calculate the waiting time and turn around time.

4: Display the gantt chart, avg waiting time and turn around time.

**PROGRAM:**

#include<stdio.h>

void main(int argc,char \*argv[])

{

int i,j=0,n,burst[10],wait[10],turn[10];

float w=0,t=0;

printf("Enter the no. of processes");

scanf("%d",&n);

burst[0]=0;

printf("Enter the burst time");

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

{

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

}

printf("\n\nGantt chart\n");

printf("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

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

printf("\tP%d\t|",i);

printf("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

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

{

j=j+burst[i];

wait[i+1]=j;

turn[i]=j;

printf("%d\t\t",j);

}

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

w=w+wait[i];

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

t=t+turn[i];

w=w/n;

t=t/n;

printf("\nAverage waiting time %0.2f",w);

printf("\nAverage turnaroundtime %0.2f",t);

}

**SAMPLE INPUT AND OUTPUT:**

cc fcfs.c

./a.out

Enter the no. of processes 3

Enter the burst time 3 6 8

Gantt chart

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

P1 | P2 | P3 |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

0 3 9 17

Average waiting time 4.00

Average turn around time 9.67

2.SJF:

ALGORITHM:

1: Get the number of processes and burst time.

2: Sort the process based on the burst time in ascending order.

3: Calculate the waiting time and turn around time.

4: Display the gantt chart,avg waiting time and turn around time.

**PROGRAM:**

#include<stdio.h>

void main(int argc,char \*argv[])

{

int b[10],temp,i,j,n,wait[10],burst[10],turn[10];

float w=0,t=0;

printf("Enter the no. of processes");

scanf("%d",&n);

burst[0]=0;

b[0]=0;

printf("Enter the burst time");

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

{

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

}

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

b[i]=burst[i];

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

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

if(b[i]>b[j])

{

temp=b[i];

b[i]=b[j];

b[j]=temp;

}

printf("\nGantt chart");

printf("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

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

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

if(b[i]==b[j])

printf("P%d|\t",j);

printf("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

j=0;

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

{

j=j+b[i];

wait[i+1]=j;

turn[i]=j;

printf("%d\t",j);

}

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

w=w+wait[i];

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

t=t+turn[i];

w=w/n;

t=t/n;

printf("\nAverage waiting time is %0.2f",w);

printf("\nAverage turnaroundtime is %0.2f",t);

}

**SAMPLE INPUT AND OUTPUT:**

cc sjf.c

./a.out

Enter the no. of processes 3

Enter the burst time2 1 3

Gantt chart

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

P1| P2| P3|

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

0 1 3 6

Average waiting time is 1.33

Average turn around time is 3.33

**3) PRIORITY SCHEDULING**

**ALGORITHM:**

1: Get the number of processes, priority and burst time.

2: Sort the process based on the priority in ascending order

3: Calculate the waiting time and turn around time.

4: Display the gantt chart, avg waiting time and turn around time.

**PROGRAM:**

#include<stdio.h>

struct prcs

{

int pid;

int bt;

int pr;

int tt;

int wt;

};

int main()

{

int no;

int k=0,b=0,g=0;

int i;

int w=0;

float n,m;

struct prcs p[10];

printf("enter the number of process");

scanf("%d",&no);

for(i=1;i<=no;i++)

{

printf("enter the process id");

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

printf(" enter the burst time");

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

printf("enter the priority");

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

p[i].wt=0;

p[i].tt=0;

}

for(i=1;i<=no;i++)

{

int f=1,j;

for(j=1;j<=no-i;j++)

{

int h=0,l=0,m=0;

if(p[j].pr>p[j+1].pr && i<no)

{

f=0;

l=p[j].pr;

p[j].pr=p[j+1].pr;

p[j+1].pr=l;

h=p[j].pid;

p[j].pid=p[j+1].pid;

p[j+1].pid=h;

g=p[j].bt;

p[j].bt=p[j+1].bt;

p[j+1].bt=g;

if(f)

break;

}

}

}

p[1].wt=0;

p[1].tt=p[1].bt;

for(i=2;i<=no;i++)

{

p[i].tt=p[i-1].tt+p[i].bt;

p[i].wt=p[i-1].tt;

b=p[i].wt+b;

k=p[i].tt+k;

}

k=k+p[1].tt;

m=(float)b/(float)no;

n=(float)k/(float)no;

for (i=1;i<=no;i++)

{

printf("the waiting time for p[%d]==%d\n",p[i].pid,p[i].wt);

printf("the turnarround time for p[%d]==%d\n",p[i].pid,p[i].tt);

}

printf("the average waiting time==%f\n",m);

printf("the average turnarround time==%f\n",n);

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

for (i=1;i<=no;i++)

{

printf("Process%d",p[i].pid);

}

printf("\n");

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

for(i=1;i<=no;i++)

{

printf("%d\t",p[i].wt);

}

printf("%d\n",p[no].tt); return(0);

}

**SAMPLE INPUT AND OUTPUT:**

cc pri.c

./a.out

enter the number of process3

enter the process id1

enter the burst time2

enter the priority3

enter the process id2

enter the burst time2

enter the priority1

enter the process id3

enter the burst time4

enter the priority2

the waiting time for p[2]==0

the turnarround time for p[2]==2

the waiting time for p[3]==2

the turnarround time for p[3]==6

the waiting time for p[1]==6

the turnarround time for p[1]==8

the average waiting time==2.666667

the average turnarround time==5.333333

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Process2Process3Process1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

0 2 6 8

4. RR Scheduling

ALGORITHM:

**1:** Initialize all the structure elements

**2:** Receive inputs from the user to fill process id,burst time and arrival time.

**3:** Calculate the waiting time for all the process id.

i) The waiting time for first instance of a process is calculated as:

a[i].waittime=count + a[i].arrivt

ii) The waiting time for the rest of the instances of the process is calculated as:

a) If the time quantum is greater than the remaining burst time then waiting time

is calculated as:

a[i].waittime=count + tq

b) Else if the time quantum is greater than the remaining burst time then waiting

time is calculated as:

a[i].waittime=count - remaining burst time

**4:** Calculate the average waiting time and average turnaround time

**5:** Print the results of the step 4.

**PROGRAM:**

#include<stdio.h>

void main()

{

int b[10],i,j=1,n,temp,burst[10],wait[10],turn[10],p[10],a=1,q,tat[10],t1=0;

float t=0,w=0;

printf("Enter the no of process & Q");

scanf("%d%d",&n,&q);

burst[0]=0;

b[0]=0;

tat[0]=0;

p[0]=0;

printf("Enter burst time");

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

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

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

b[i]=burst[i];

printf("\n\n\t\t Gantt chart\n");

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

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

{

if(b[i]>0)

{

a=1;

printf("P%d\t|",i);

if(b[i]>=q)

{

t1=t1+q;

p[j]=t1;

j++;

}

else if(b[i]<q)

{

t1=t1+b[i];

p[j]=t1;

j++;

}

b[i]=b[i]-q;

if(b[i]<=0)

tat[i]=t1;

}

else

a++;

if(a==n+1)

break;

if(i==n)

i=0;

}

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

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

printf("%d\t",p[i]);

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

{

t=t+tat[i];

w=w+tat[i]-burst[i];

}

w=w/n;

t=t/n;

printf("\nThe average waiting time is %0.2f",w);

printf("\nThe average turn around time is %0.2f",t);

}

**SAMPLE INPUT AND OUTPUT:**

cc rr.c

./a.out

Enter the number of processes 3

Enter the time quantum 3

Enter the Burst Time P1 3

Enter the Burst Time P2 4

Enter the Burst Time P3 2

GanttChart

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­­­­­­­­­­­­­­­­­­­­­\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

p1 p2 p3 p2

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

0 3 6 8 9

The average Waiting Time 3.666667

The average Turnaround Time 6.6666

**CONCLUSION:**

Thus FCFS, SJFS, Priority and Round Robin scheduling algorithms is implemented using C and executed.

Ex.No:5 IMPLEMENTION OF PRODUCER-CONSUMER PROBLEM USING

Date: SEMAPHORES

AIM:

To write a C program to implement the Producer & consumer Problem (Semaphore).

**PROBLEM DESCRIPTION:**

The producer-consumer problem (also known as the bounded-buffer problem) is a classical example of a multi-[process](http://en.wikipedia.org/wiki/Process_%28computing%29) [synchronization](http://en.wikipedia.org/wiki/Synchronization_%28computer_science%29) problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size [buffer](http://en.wikipedia.org/wiki/Buffer_%28computer_science%29). The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time the consumer is consuming the data (i.e. removing it from the buffer) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer. The solution for the producer is to go to sleep if the buffer is full. The next time the consumer removes an item from the buffer, it wakes up the producer who starts to fill the buffer again. In the same way the consumer goes to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be reached by means of [inter-process communication](http://en.wikipedia.org/wiki/Inter-process_communication), typically using [semaphores](http://en.wikipedia.org/wiki/Semaphores). An inadequate solution could result in a [deadlock](http://en.wikipedia.org/wiki/Deadlock) where both processes are waiting to be awakened.

**ALGORITHM:**

**1:** The Semaphore mutex, full & empty are initialized.

**2:** In the case of producer process

a)Produce an item in to temporary variable.

b)If there is empty space in the buffer check the mutex value for enter into the critical section.

c)If the mutex value is 0, allow the producer to add value in the temporary variable to the buffer.

**3:** In the case of consumer process

a)It should wait if the buffer is empty

b)If there is any item in the buffer check for mutex value, if the mutex==0,remove item from buffer

c)Signal the mutex value and reduce the empty value by 1.Consume the item.

**4**: Print the result

**PROGRAM:**

#include<stdio.h>

char buf[20],p[20],cos[20];

int mutex,i,k,c,sz,n;

mutex=0;

void prosig()

{

mutex=mutex+1;

}

void consig()

{

mutex=mutex-1;

}

int buffer(int mutex)

{

if(mutex==0)

return 1;

else

return 0;

}

void producer(int sz)

{

int c;

c=buffer(mutex);

if(c==1)

{

printf("\nProducer can produce the item and give $ for exit\n");

i=0;

while(i<sz&&(p[i]=getchar())!='$')

{

buf[i]=p[i];

i++;

}

k=i;

prosig();

printf("\nProduction done successfully\n");

}

else if(k<sz)

{

printf("Producer can also produce items");

while((p[k]=getchar())!='$')

{

buf[k]=p[k];

k++;

}

prosig();

printf("\nProduction done successfully\n");

}

else if(k>=sz)

{

printf("\nBuffer is full,can't produce\n");

}

}

void consumer()

{

int c1;

c1=buffer(mutex);

if(c1==0)

{

printf("\nConsumer can consume item\n");

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

cos[i]=buf[i];

printf("\nConsumed item is:\n");

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

printf("\n%c",cos[i]);

consig();

printf("\nSuccessfully done\n");

}

else

{

printf("\nBuffer is empty,can't consume\n");

}

}

int main()

{

int op,sz;

printf("Enter the buffer size");

scanf("%d",&sz);

do

{

printf("\n1.Producer\t2.Consumer\t3.Exit\n");

printf("\nEnter your choice\n");

scanf("%d",&op);

switch(op)

{

case 1:

producer(sz);

break;

case 2:

consumer();

break;

case 3:

exit(0);

}

}

while(op<=2);

return 0;

}

**SAMPLE INPUT AND OUTPUT:**

cc pcp.c

./a.out

Enter the buffer size5

1.Producer 2.Consumer 3.Exit

Enter your choice

1

Producer can produce the item and give $ for exit

ho$

Production done successfully

1.Producer 2.Consumer 3.Exit

Enter your choice

1

Producer can also produce items ney$

Production done successfully

1.Producer 2.Consumer 3.Exit

Enter your choice

1

Buffer is full,can't produce

1.Producer 2.Consumer 3.Exit

Enter your choice

2

Consumer can consume item

Consumed item is:

h

o

n

e

y

Successfully done

1.Producer 2.Consumer 3.Exit

Enter your choice

3

**CONCLUSION:**

The Producer & consumer Problem using Semaphore is implemented using C and executed.