**MEDICAPS UNIVERSITY, INDORE**



**Department of Computer Science & Engineering**

**FACULTY OF ENGINEERING**

**Practical File**

**OPERATING SYSTEM**

**[CB3CO06]**

**Submitted To:                    Submitted BY:**

Mrs. Vineeta Rathore                            Vanshika Garg

**EN20CS306057**

## EXPERIMENT 1

**AIM:** **Write the names of operating systems with their versions, release date and features.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| S.No. | Name of Operating System | Versions | Release Date | Special Features |
| 1 | LINUX | 1 | 1994 | Multiuser Capability |
|  |  | 2 | 1996 | Security |
|  |  | 3 | 2011 | Graphical user Interface |
|  |  | 4 | 2015 | Application support |
|  |  | 5 | 2019 | Open Source |
|  |  |  |  |  |
|  |  |  |  |  |
| 2 | Microsoft Windows | Xp | 2001 | Easier Navigation |
|  |  | 7 | 2009 | Can Switch between multiple workspaces at once |
|  |  | 8 | 2012 | Tablet mode designed for touch screens. |
|  |  | 10 | 2015 | Multifactor authentication technology for higher security. |
|  |  | 11 | 2021 | Automatically compress system files to reduce the storage footprint. |
|  |  |  |  |  |
|  |  |  |  |  |
| 3 | Google Anroid OS | 8.1 | 2017 | Near Field Communication (NFC) |
|  |  | 9 | 2018 | Wireless App Downloads |
|  |  | 10 | 2019 | Storage and Battery Swap |
|  |  | 11 | 2020 | Custom Home Screens |
|  |  | 12 | 2021 | Infrared Transmission |
|  |  |  |  |  |
|  |  |  |  |  |
| 4 | Mac OS | macOS 10.10 | 2014 | Software and Hardware Seamlessly Work together |
|  |  | macOS 10.12 | 2016 | Security Comes First on Mac OS X |
|  |  | macOS 11 | 2020 | Brain-Dead Easy to Use and Enjoy |
|  |  | macOS 12 | 2021 |  |
|  |  |  |  |  |
|  |  |  |  |  |
| 5 | Solaris | 9 | 2003 | Developed by Sun Microsystems |
|  |  | 10 | 2005 | Large workload processing. |
|  |  | 11 | 2011 | Managing multiple database. |
|  |  | 11.4 | 2018 |  |
|  |  |  |  |  |
|  |  |  |  |  |
| 6 | Fedora | Yarrow | 2003 |  |
|  |  | Lovelock | 2011 | Open-Source Development. |
|  |  | 21 | 2015 | Corporate Use. |
|  |  | 30 | 2019 |  |
|  |  | 32 | 2020 |  |
|  |  | 34 | 2021 |  |
|  |  | 35 | 2022 |  |
|  |  |  |  |  |
| 7 | Chrome OS | Chrome OS on Chromebook | 2021 | Web Application |
|  |  |  |  |  |
| 8 | Ubuntu | Ubuntu 4.10 | 2004 |  |
|  |  | Ubuntu 7.10 | 2007 | Open-source downloading. |
|  |  | Ubuntu 11.1 | 2011 | Running apps, browser and gaming. |
|  |  | Ubuntu 16.10 | 2016 |  |
|  |  | Ubuntu 19.10 | 2019 |  |
|  |  | Ubuntu 20.10 | 2020 |  |
|  |  |  |  |  |
| 9 | Free BSD | Free BSD 1 | 1994 |  |
|  |  | Free BSD 2 | 2006 | Networking. |
|  |  | Free BSD 7 | 2008 | Internet and Intranet Server. |
|  |  | Free BSD 11 | 2016 |  |
|  |  | Free BSD 12 | 2018 |  |
|  |  |  |  |  |
| 10 | DEBIAN | Debian 1.1 (Buzz) | 1996 |  |
|  |  | Debian 2.2 (Potato) | 2000 | Running apps. |
|  |  | Debian 3.0 (woody) | 2002 |  |
|  |  | Debian 4.0 (Etch) | 2007 |  |
|  |  | Debian 6.0 (squeeze) | 2011 |  |
|  |  | Debian 11 (Bullseye) | 2021 |  |
|  |  | Debian 12 (bookworm) | 2022 |  |

# EXPERIMENT 2

### AIM: Write a Program to get a process and its parent id.

**Theory: -** You can get the process ID of a process by calling getpid. The function getppid returns the process ID of the parent of the current process (this is also known as the *parent process ID*). Your program should include the header files unistd.h and sys/types.h to use these functions.

### Program: -

#include<stdio.h>

#include<unistd.h>

#include<sys/types.h>

int main()

{

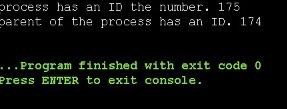
printf("process has an ID the number. %ld\n", (long) getpid());

printf("parent of the process has an ID. %ld\n",(long) getppid());

return 0;

}

### Output: -



**EXPERIMENT 3**

###### **AIM**: **Write a Program to demonstrate the fork function**

**THEORY**: - Fork system call is used for creating a new process, which is called child process, which runs concurrently with the process that makes the fork () call (parent process). After a new child process is created, both processes will execute the next instruction following the fork () system call. A child process uses the same pc (program counter), same CPU registers, same open files which use in the parent process.

#### **PROGRAM**: -

#include<stdio.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/wait.h>

int main ()

{ pid\_t fork\_pid;

printf("the main program process ID is %d\n", (int)getpid());

printf("the main program parent process ID is %d\n",

(int)getppid ());

fork\_pid = fork ();

if (fork\_pid! = 0) {

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

printf ("process ID is %d\n", (int)getpid());

printf (" parent process ID is %d\n", (int)getppid ()); printf("the child process ID is %d\n", (int)fork\_pid); sleep(10);

}

else{ wait(NULL);

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

printf("process ID is %d\n", (int)getpid());

printf(" parent process ID is %d\n", (int)getppid());

printf("logical ID of the process based on the forkfunction is

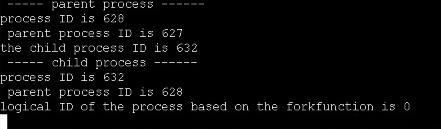
%d\n", (int)fork\_pid);

}

return 0;

}

**Output: -**



**EXPERIMENT 4**

###### **AIM:** **Write a Program to demonstrate the fork function.** **Program:**

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

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

{

printf ("--beginning of program\n"); int counter = 0;

pid\_t pid = fork (); if (pid == 0)

{

// child process int i = 0;

for (; i < 5; ++i)

{

printf ("child process: counter=%d\n", ++counter);

}

}

else if (pid > 0)

{

// parent process int j = 0;

for (; j < 5; ++j)

{

printf ("parent process: counter=%d\n", ++counter);

}

}

else {

// fork failed

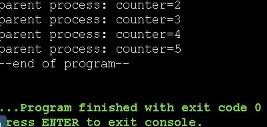
printf ("fork() failed!\n"); return 1;

}

printf ("--end of program--\n"); return 0;

}

**Output:**



## EXPERIMENT 5

##### AIM: Write a Program using fork () to duplicate a program’s process

**Program:**

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

{

pid\_t child\_pid;

printf ("the main program process ID is %d\n", (int) getpid ()); child\_pid = fork ();

if (child\_pid != 0)

{

printf ("this is the parent process, with id %d\n", (int) getpid ());

printf ("the child’s process ID is %d\n", (int) child\_pid);

}

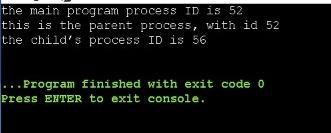
Else

printf ("this is the child process, with id %d\n", (int) getpid ());

return 0;

}

**OUTPUT: -**



# EXPERIMENT 6

##### AIM: Write a Program for First come first serve (FCFS) scheduling algorithm.

**THEORY:** - In this scheduling algorithm we allocate CPU to process that comes first in the ready queue. That is, the process that comes first in the ready queue will gets the CPU first. So, it's called first come first serve algorithm.

It is Non preemptive it means in FCFS Scheduling once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by requesting I/O.

Advantage: -

* Easy to implement
* Easy to understand
* The simplest form of a CPU scheduling algorithm

Disadvantage: -

* Average waiting time is often quite longer
* Because of its simplicity, FCFS is not very efficient
* Not an ideal technique for time-sharing systems

C program: -

In the first Code we will just use Array and in the second Code we will use structure.

Code 1: -

In this code we simply declare Some array for the arrival time, Burst time

, Waiting time, and turnaround time.

First, we take an input from the user which is number of processes in the Ready queue. Then after we take Arrival time and Burst time of all the process one by one. After that we will calculate the waiting time and Turn Around time of all the process and store it in the Array WT and TT respectively.

#include<stdio.h>

int main ()

{

int AT [10], BT [10], WT [10], TT [10],n;

int burst=0, cmpl\_T;

float Avg\_WT,Avg\_TT,Total=0;

printf("Enter number of the process\n");

scanf("%d",&n);

printf("Enter Arrival time and Burst time of the process\n");

printf("AT\tBT\n");

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

{

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

}

// Logic for calculating Waiting time

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

{

if(i==0)

WT[i]=AT[i];

else

WT[i]=burst-AT[i];

burst+=BT[i];

Total+=WT[i];

}

Avg\_WT=Total/n;

// Logic for calculating Turnaround time

cmpl\_T=0;

Total=0;

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

{

cmpl\_T+=BT[i];

TT[i]=cmpl\_T-AT[i];

Total+=TT[i];

}

Avg\_TT=Total/n;

// printing of outputs

printf("Process ,Waiting\_time ,TurnA\_time\n");

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

{

printf("%d\t\t%d\t\t%d\n",i+1,WT[i],TT[i]);

}

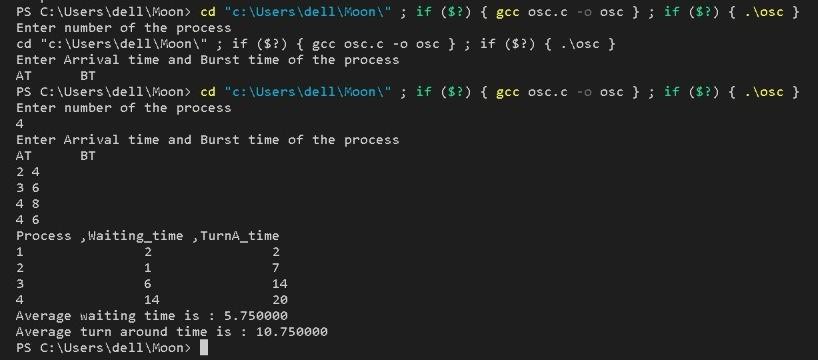
printf("Average waiting time is : %f\n",Avg\_WT);

printf("Average turn around time is : %f\n",Avg\_TT);

return 0;

}

**Output: -**



Code 2: -

In This Code first we create the structure for the process. Then after we declare an array of structure type.

cmpl\_T stands for completion time of the process which we are calculating in the program.

#include<stdio.h> struct process

{

int AT, WT, TT, BT;

};

struct process a[10]; int main()

{

int n;

int burst=0, cmpl\_T;

float Avg\_WT,Avg\_TT,Total=0;

printf("Enter number of process\n");

scanf("%d",&n);

printf("Enter Arrival time and Burst time of the process\n");

printf("At BT\n");

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

{

scanf("%d%d",&a[i].AT,&a[i].BT);

}

// Logic for calculating Waiting time

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

{

if(i==0

else

a[i]. WT=a[i].AT;

a[i]. WT=burst-a[i].AT;

burst+=a[i].BT;

Total+=a[i]. WT;

}

Avg\_WT=Total/n;

// Logic for calculating Turnaround time

cmpl\_T=0;

Total=0;

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

{

cmpl\_T+=a[i].BT;

a[i].TT=cmpl\_T-a[i].AT;

Total+=a[i].TT;

}

Avg\_TT=Total/n;

// printing of outputs

printf("Process , Waiting\_time , TurnA\_time\n");

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

{

printf("%d\t\t%d\t\t%d\n”, i+1,a[i].WT,a[i].TT);

}

printf("Average waiting time is : %f\n",Avg\_WT);

printf("Average turnaround time is : %f\n",Avg\_TT);

return 0;

}

**Output: -**

Enter number of process

3

Enter Arrival time and Burst time of the process At BT

0 2

1 4

2 3

Process ,

1

2

3

Waiting\_time , TurnA\_time

0

1

4

2

5

7

Average waiting time is : 1.666667

Average turn around time is : 4.666667

**EXPERIMENT 7**

##### AIM: Write a Program for First come first serve (FCFS) scheduling algorithm. (Assuming arrival time of process are same i.e., zero)

C program: -

// first come first serve #include<bits/stdc++.h> using namespace std;

int main(){ int n;

cout<<"Enter no. of processes: "; cin>>n;

cout<<"Enter burst time for n process: "; int bt[n];

for(int i=0;i<n;i++) cin>>bt[i];

cout<<"NOTE: We assume the arrival time of process are same i.e, zero"<<endl;

int wait[n+1],turnarnd[n+1]; wait[0]=0; turnarnd[0]=bt[0];

double avgwait=0.0, avgturnarndT=0.0; for(int i=1;i<n;i++)

{

wait[i]=bt[i-1] +wait[i-1];

avgwait+=wait[i]; turnarnd[i]=bt[i]+wait[i]; avgturnarndT+=turnarnd[i];

}

//avgturnarndT=avgturnarndT/n; avgwait=avgwait/n;

cout<<"process Burst\_time turnaround\_time waiting\_time "<<endl; for(int i =0 ;i <n;i++)

{

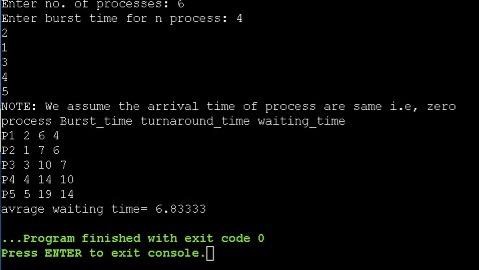
cout<<"P"<<i<<" "<<bt[i]<<" "<< turnarnd[i]<<" "<<wait[i]<<endl;

}

cout<<"avrage waiting time= "<<avgwait; return 0;

}

**OUTPUT: -**



## EXPERIMENT 8

#### **AIM:** **Write a Program for Shortest Job First (SJF) scheduling algorithm. (Non-Preemptive)**

**THEORY:** - As it is clear by the name of this scheduling algorithm the job which have the less burst time will get the CPU first .it is the best method to minimize the waiting time .it is of two type

1. preemptive
2. non preemptive

**Characteristics: -**

* + Siff scheduling can be either preemptive or non-preemptive.
  + IN SJF CPU is assigned to the process that has the smallest next CPU Burst time.
  + If the next CPU Burst of two process is the same then FCFS scheduling is used to break the tie.
  + This process gives the minimum average waiting time for a given processes.

**Code: -**

this code is for Non preemptive shortest job first algorithm

In this code first we are creating the structure for the process in which we are declaring the id, waiting time, Arrival time, Burst time and turnaround time. then after an array of the structure type.

**Logic: -**

1. After taking the input first we sort the input on the basis of Arrival time (i.e the process which have less Arrival time will come first).

Here one more thing we will do. we will check that the arrival time of all the processes is different or not. (Example: - if all the processes comes at the same time then we don't need to sort the array on the basis of arrival time) . for checking this we will use check\_ar.

1. After doing this if processes are arrived at the different time, then we can easily calculate the waiting and turnaround time for the first process. and for other process first we will find the minimum burst time of the process which are arrived at cmp\_time (completion time of the previous process).

**Code: -**

#include<stdio.h>

struct process

{

int id,AT,BT,WT,TAT;

};

struct process a[10];

// declaration of the ready queue

int queue [100];

int front=-1;

int rear=-1;

// function for insert the element

// into queue

void insert (int n)

{

if(front==-1)

front=0;

rear=rear+1;

queue[rear]=n;

}

// function for delete the

// element from queue int delete ()

{

int n; n=queue[front];

front=front+1;

return n;

}

int main()

{

int n,TQ,p,TIME=0;

int temp [10], exist [10]={0};

float total\_wt=0, total\_tat=0,Avg\_WT,Avg\_TAT;

printf("Enter the number of the process\n");

scanf("%d",&n);

printf("Enter the arrival time and burst time of the process\n"); printf("AT BT\n");

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

{

scanf("%d%d",&a[i].AT,&a[i].BT); a[i].id=i;

temp[i]=a[i].BT;

}

printf("Enter the time quantum\n"); scanf("%d",&TQ);

// logic for round robin scheduling

// insert first process

// into ready queue insert (0); exist[0]=1;

// until ready queue is empty

while(front<=rear)

{

p=delete (); if(a[p].BT>=TQ)

{

}

else

{

}

a[p].BT=a[p].BT-TQ; TIME=TIME+TQ;

TIME=TIME+a[p].BT; a[p].BT=0;

//if process is not exist

// in the ready queue even a single

// time then insert it if it arrives

// at time 'TIME' for(int i=0;i<n;i++)

{

if(exist[i]==0 && a[i].AT<=TIME)

{

insert(i); exist[i]=1;

}

}

// if process is completed

if(a[p].BT==0)

{

}

else

{

}

}

a[p].TAT=TIME-a[p].AT;

a[p].WT=a[p].TAT-temp[p]; total\_tat=total\_tat+a[p].TAT; total\_wt=total\_wt+a[p].WT;

insert(p);

Avg\_TAT=total\_tat/n; Avg\_WT=total\_wt/n;

// printing of the answer

printf("ID WT TAT\n"); for(int i=0;i<n;i++)

{

printf("%d %d %d\n",a[i].id,a[i].WT,a[i].TAT);

}

printf("Average waiting time of the processes is : %f\n",Avg\_WT);

printf("Average turn around time of the processes is :

%f\n",Avg\_TAT); return 0;

}

**Output: -**

Enter the number of process

4

Enter the Arrival time and Burst time of the process AT BT

process are

Avg waiting time is:- 2.000000

Avg turn around time is:- 5.000000

|  |  |
| --- | --- |
| 0 | 2 |
| 2 | 3 |
| 2 | 4 |
| 3 | 3 |
| The | |

|  |  |  |
| --- | --- | --- |
| ID | WT | TAT |
| 1 | 0 | 2 |
| 2 | 0 | 3 |
| 4 | 2 | 5 |
| 3 | 6 | 10 |

**EXPERIMENT 9**

#### **AIM:** **Write a Program for Shortest Job First (SJF) scheduling algorithm. {Preemptive}**

**Theory: -** this code is for Preemptive Shortest job first algorithm

In this code first we are creating the structure for the process in which we are declaring the waiting time, Arrival time , Burst time ,Priority and turnaround time .then after an array of the structure type.

**logic: -**.

* 1. First, we copy the burst time of the process in a new

array temp [] because in the further calculation we will be going to decrease the Burst time of the process but we will have to need the real burst time of the process in the calculation of the waiting time.

* 1. we initialize the burst time of a process with the maximum (you can take any maximum value). and we will use 9th process because we assumed that there will not be more than 10 process but you can use any number.
  2. In this code we are going to use a loop which executed until all the processes are completed. for checking how many processes are completed we use **count. Initially** its value is 0 (i.e. no processes are completed yet).
  3. In each cycle we will find the process which have shortest burst time and arrive at time t and burst time of the process is not equal to zero.
  4. After doing this we will decrease the burst time of the process by 1 in each cycle of the time.
  5. And if the process will be complete (Burst time =0) then we will increase the value of the count by 1 (i.e one process is completed)
  6. For calculating the waiting time we will use a formula (WT= time- arrival-Burst time)
  7. For calculating turnaround time we simply use TAT= completion time - arrival time.

**Code: -**

#include<stdio.h> struct process

{

int id,AT,BT,WT,TAT;

};

struct process a[10];

// declaration of the ready queue

int queue [100];

int front=-1; int rear=-1;

// function for insert the element

// into queue

void insert (int n)

{

if(front==-1) front=0; rear=rear+1; queue[rear]=n;

}

// function for delete the

// element from queue

int delete ()

{

int n; n=queue[front]; front=front+1; return n;

}

int main ()

{

int n,TQ,p,TIME=0;

int temp [10], exist [10] ={0};

float total\_wt=0, total\_tat=0,Avg\_WT,Avg\_TAT; printf("Enter the number of the process\n"); scanf("%d",&n);

printf("Enter the arrival time and burst time of the process\n");

printf("AT BT\n"); for(int i=0;i<n;i++)

{

scanf("%d%d",&a[i].AT,&a[i].BT); a[i].id=i;

temp[i]=a[i].BT;

}

printf("Enter the time quantum\n"); scanf("%d",&TQ);

// logic for round robin scheduling

// insert first process

// into ready queue

insert (0);

exist [0]=1;

// until ready queue is empty while(front<=rear)

{

p=delete(); if(a[p].BT>=TQ)

{

a[p].BT=a[p].BT-TQ; TIME=TIME+TQ;

}

else

{

TIME=TIME+a[p].BT; a[p].BT=0;

}

//if process is not exist

// in the ready queue even a single

// time then insert it if it arrive

// at time 'TIME'

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

{

if(exist[i]==0 && a[i].AT<=TIME)

{

insert(i); exist[i]=1;

}

}

// if process is completed if(a[p].BT==0)

{

a[p].TAT=TIME-a[p].AT;

a[p].WT=a[p].TAT-temp[p]; total\_tat=total\_tat+a[p].TAT; total\_wt=total\_wt+a[p].WT;

}

else

{

insert(p);

}

}

Avg\_TAT=total\_tat/n;

Avg\_WT=total\_wt/n;

// printing of the answer

printf("ID WT TAT\n");

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

{

printf("%d %d %d\n",a[i].id,a[i].WT,a[i].TAT);

}

printf("Average waiting time of the processes is :

%f\n",Avg\_WT);

printf("Average turn around time of the processes is :

%f\n",Avg\_TAT);

return 0;

}

**Output: -**

mber of the process

rival time and burst time of the process

Avg waiting time of the process is 1.666667

Avg turn around time of the process 5.666667

|  |  |  |
| --- | --- | --- |
| Enter the nu 3  Enter the ar  AT WT | | |
| 0 | 7 |  |
| 2 | 3 |  |
| 7 | 2 |  |
| Id | WT | TAT |
| 1 | 5 | 12 |
| 2 | 0 | 3 |
| 3 | 0 | 2 |

## EXPERIMENT 10

**AIM:** **Write a Program for round robin algorithm.**

### Theory: -

* In this technique ready queue is treated as circular queue.
* In this technique each process is provided a fix time execute which is called time quantum (or time slice).
* CPU goes around the ready queue allocating the CPU to each process for a time interval up to 1 time quantum.
* It is only preemptive.
* This algorithm gives minimum average response time for a given set of process.
* Widely used scheduling method in traditional OS.
* It is designed especially for time sharing system or multi-tasking system.

**Code: -**

#include<stdio.h> struct process

{

int id,AT,BT,WT,TAT;

};

struct process a[10];

// declaration of the ready queue

int queue [100];

int front=-1; int rear=-1;

// function for insert the element

// into queue

void insert (int n)

{

if(front==-1) front=0; rear=rear+1; queue[rear]=n;

}

// function for delete the

// element from queue

int delete()

{

int n; n=queue[front]; front=front+1; return n;

}

int main()

{

int n,TQ,p,TIME=0;

int temp[10],exist[10]={0};

float total\_wt=0,total\_tat=0,Avg\_WT,Avg\_TAT;

printf("Enter the number of the process\n"); scanf("%d",&n);

printf("Enter the arrival time and burst time of the process\n");

printf("AT BT\n"); for(int i=0;i<n;i++)

{

scanf("%d%d",&a[i].AT,&a[i].BT); a[i].id=i;

temp[i]=a[i].BT;

}

printf("Enter the time quantum\n"); scanf("%d",&TQ);

// logic for round robin scheduling

// insert first process

// into ready queue insert(0); exist[0]=1;

// until ready queue is empty while(front<=rear)

{

p=delete(); if(a[p].BT>=TQ)

{

a[p].BT=a[p].BT-TQ; TIME=TIME+TQ;

}

else

{

TIME=TIME+a[p].BT; a[p].BT=0;

}

//if process is not existed

// in the ready queue even a single

// time then insert it if it arrives

// at time 'TIME'

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

{

if(exist[i]==0 && a[i].AT<=TIME)

{

insert(i); exist[i]=1;

}

}

// if process is completed if(a[p].BT==0)

{

a[p].TAT=TIME-a[p].AT;

a[p].WT=a[p].TAT-temp[p];

total\_tat=total\_tat+a[p].TAT; total\_wt=total\_wt+a[p].WT;

}

else

{

insert(p);

}

}

Avg\_TAT=total\_tat/n; Avg\_WT=total\_wt/n;

// printing of the answer

printf("ID WT TAT\n"); for(int i=0;i<n;i++)

{

printf("%d %d %d\n",a[i].id,a[i].WT,a[i].TAT);

}

printf("Average waiting time of the processes is :

%f\n",Avg\_WT);

printf("Average turnaround time of the processes is :

%f\n",Avg\_TAT); return 0;

}

**Output: -**

number of the process

arrival time and burst time of the process

time quantum

Average waiting time of the processes is : 6.333333

Average turn around time of the processes is : 12.333333

|  |  |  |
| --- | --- | --- |
| Enter the 3  Enter the AT BT  0 5  2 7  4 6  Enter the 3 | | |
| ID | WT | TAT |
| 0 | 3 | 8 |
| 1 | 9 | 16 |
| 2 | 7 | 13 |

## EXPERIMENT 11

#### **AIM: Write a Program for Priority Scheduling Algorithm (Non- Preemptive)**

**Theory: -** As it is clear with the name that this scheduling is based on the priority of the processes. The process which has the higher priority will get the CPU first. whereas jobs with equal priorities are carried out on a round-robin or FCFS basis.

This scheduling is of two types: -

1. Non preemptive
2. Preemptive

**Characteristics: -**

* 1. It schedules the process based on the priority of the processes.
  2. Lower the number higher the priority.
  3. If the two or more processes have the same priority then we schedules on the basis of FCFS.
  4. Major problem with priority scheduling is problem of starvation.
  5. Solution of the problem of the starvation is aging ,where aging is a technique of gradually increasing the priority of the processes that wait in the system from long time.

**Drawbacks: -**

1. Major problem with priority scheduling is problem of starvation.
2. Solution of the problem of the starvation is aging ,where aging is a technique of gradually increasing the priority of the processes that wait in the system from long time.

**Code: -**

1. After taking the input first we sort the input on the basis of Arrival time (i.e., the process which have less Arrival time will come first).
2. Here one more thing we will do. we will check that the arrival time of all the processes is different or not. (Example: - if all the processes comes at the same time then we don't need to sort the array on the basis of arrival time) . for checking this we will use check\_ar.
3. After doing this if processes are arrived at the different time, then we can easily calculate the waiting and turnaround time for the first process. and for other process first we will find the highest priority (less priority number) of the process which are arrived

at cmp\_time (completion time of the previous process)

#include<stdio.h> struct process

{

int id,WT,AT,BT,TAT,PR;

};

struct process a [10];

// function for swapping void swap(int \*b,int \*c)

{

int tem; tem=\*c;

\*c=\*b;

\*b=tem;

}

//Driver function int main()

{

int n,check\_ar=0; int Cmp\_time=0;

float Total\_WT=0, Total\_TAT=0, Avg\_WT,Avg\_TAT;

printf("Enter the number of process \n");

scanf("%d",&n);

printf("Enter the Arrival time , Burst time and priority of the process\n");

printf("AT BT PR\n"); for(int i=0;i<n;i++)

{

scanf("%d%d%d",&a[i].AT,&a[i].BT,&a[i].PR);

a[i].id=i+1;

// here we are checking that arrival time

// of the process are same or different

if(i==0)

check\_ar=a[i].AT;

if(check\_ar!=a[i].AT ) check\_ar=1;

}

// if process are arrived at the different time

// then sort the process on the basis of AT

if(check\_ar!=0)

{

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

{

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

{

if(a[j].AT>a[j+1].AT)

{

swap(&a[j].id,&a[j+1].id);

swap(&a[j].AT,&a[j+1].AT);

swap(&a[j].BT,&a[j+1].BT);

swap(&a[j].PR,&a[j+1].PR);

}

}

}

} // logic of Priority scheduling ( non-preemptive) algo

// if all the process are arrived at different time if(check\_ar!=0)

{

a[0].WT=a[0].AT; a[0].TAT=a[0].BT-a[0].AT;

// cmp\_time for completion time

Cmp\_time=a[0].TAT; Total\_WT=Total\_WT+a[0].WT; Total\_TAT=Total\_TAT+a[0].TAT; for(int i=1;i<n;i++)

{

int min=a[i].PR; for(int j=i+1;j<n;j++)

{

if(min>a[j].PR && a[j].AT<=Cmp\_time)

{

min=a[j].PR; swap(&a[i].id,&a[j].id);

swap(&a[i].AT,&a[j].AT);

swap(&a[i].BT,&a[j].BT);

swap(&a[i].PR,&a[j].PR);

}

}

a[i].WT=Cmp\_time-a[i].AT; Total\_WT=Total\_WT+a[i].WT;

// completion time of the process Cmp\_time=Cmp\_time+a[i].BT;

// Turn Around Time of the process

// compl-Arival a[i].TAT=Cmp\_time-a[i].AT; Total\_TAT=Total\_TAT+a[i].TAT;

}

}

// if all the process are arrived at same time else

{

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

{

int min=a[i].PR; for(int j=i+1;j<n;j++)

{

if(min>a[j].PR && a[j].AT<=Cmp\_time)

{

min=a[j].PR; swap(&a[i].id,&a[j].id);

swap(&a[i].AT,&a[j].AT);

swap(&a[i].BT,&a[j].BT);

swap(&a[i].PR,&a[j].PR);

}

}

a[i].WT=Cmp\_time-a[i].AT;

// completion time of the process Cmp\_time=Cmp\_time+a[i].BT;

// Turn Around Time of the process

// compl-Arrival a[i].TAT=Cmp\_time-a[i].AT;

Total\_WT=Total\_WT+a[i].WT; Total\_TAT=Total\_TAT+a[i].TAT;

}

}

Avg\_WT=Total\_WT/n; Avg\_TAT=Total\_TAT/n;

// Printing of the results printf("The process are\n"); printf("ID WT TAT\n"); for(int i=0;i<n;i++)

{

printf("%d\t%d\t%d\n",a[i].id,a[i].WT,a[i].TAT);

}

printf("Avg waiting time is: %f\n",Avg\_WT); printf("Avg turn around time is: %f",Avg\_TAT); return 0;

}

**Output: -**

Enter the number of process 4

Enter the Arrival time , Burst time and priority of the process AT BT PR

0 5 3

1 2 4

2 2 1

3 6 2

The process are ID WT TAT

1 0 5

3 3 5

4 4 10

2 12 14

Avg waiting time is: 4.750000 Avg turn around time is: 8.500000

## EXPERIMENT 12

#### **AIM: Write a Program for Priority Scheduling Algorithm. (Preemptive)**

**Logic: -**

* 1. First, we copy the burst time of the process in a new

array temp[] because in the further calculation we will be going to decrease the Burst time of the process but we will have to need the real burst time of the process in the calculation of the waiting time .(If you confused then don't worry you will be able understand after going through code)

* 1. we initialize the priority of a process with the maximum (you can take any maximum value). and we will use 9th process because we assumed that there will not be more than 10 process but you can use any number.
  2. In this code we are going to use a loop which executed until all the processes are completed. for checking how many processes are completed we use **count. Initially** its value is 0 (i.e no processes are completed yet).
  3. In each cycle we will find the process which have highest priority (lowest priority number like 1 have high priority than 2) and arrived at time t and burst time of the process is not equal to zero.
  4. After doing this we will decrease the burst time of the process by 1 in each cycle of the time.
  5. And if the process will be complete (Burst time =0) then we will increase the value of the count by 1 (i.e one process is completed)
  6. For calculating the waiting time, we will use a formula (WT= time- arrival-Burst time)
  7. 8.For calculating turnaround time we simply use TAT= completion time - arrival time.

**Code: -**

#include<stdio.h> struct process

{

int WT,AT,BT,TAT,PT;

};

struct process a[10];

int main()

{

int n,temp[10],t,count=0,short\_p;

float total\_WT=0,total\_TAT=0,Avg\_WT,Avg\_TAT;

printf("Enter the number of the process\n"); scanf("%d",&n);

printf("Enter the arrival time , burst time and priority of the process\n");

printf("AT BT PT\n");

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

{

scanf("%d%d%d",&a[i].AT,&a[i].BT,&a[i].PT);

// copying the burst time in

// a temp array fot futher use temp[i]=a[i].BT;

}

// we initialize the burst time

// of a process with maximum

a[9].PT=10000;

for(t=0;count!=n;t++)

{

short\_p=9;

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

{

if(a[short\_p].PT>a[i].PT && a[i].AT<=t && a[i].BT>0)

{

short\_p=i;

}

}

a[short\_p].BT=a[short\_p].BT-1;

// if any process is completed if(a[short\_p].BT==0)

{

// one process is completed

// so count increases by 1

count++;

a[short\_p].WT=t+1-a[short\_p].AT-temp[short\_p]; a[short\_p].TAT=t+1-a[short\_p].AT;

// total calculation

total\_WT=total\_WT+a[short\_p].WT; total\_TAT=total\_TAT+a[short\_p].TAT;

}

}

Avg\_WT=total\_WT/n; Avg\_TAT=total\_TAT/n;

// printing of the answer

printf("ID WT TAT\n"); for(int i=0;i<n;i++)

{

printf("%d %d\t%d\n",i+1,a[i].WT,a[i].TAT);

}

printf("Avg waiting time of the process is %f\n",Avg\_WT); printf("Avg turnaround time of the process is %f\n",Avg\_TAT);

return 0;

}

**Output: -**

Enter the number of the process 3

Enter the arrival time, burst time and priority of the process AT BT PT

0 3 3

1 5 1

2 2 2

ID WT TAT 1 7 10

2 0 5

3 4 6

Avg waiting time of the process is 3.666667 Avg turnaround time of the process is 7.000000