**Fcfs algorithm**: **Problem Description:**

Write an FCFS Scheduling Program in C to determine the average waiting time and average turnaround time has given n processes and their burst times.

**FCFS Scheduling Algorithm:**

The CPU scheduling algorithm First Come, First Served (**FCFS**), also known as First In, First Out (**FIFO**), allocates the CPU to the processes in the order they are queued in the ready queue.

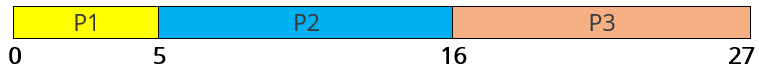
**FCFS** uses non-preemptive scheduling, which means that once a CPU has been assigned to a process, it stays assigned to that process until it is either not terminated or may be interrupted by an I/O interrupt.

**Problem Solution**

1. Enter all the processes and their burst time.  
2. Find waiting time, **WT** of all the processes.  
3. For the 1st process, **WT = 0**.  
4. For all the next processes i, **WT[i] = BT[i-1] + WT[i-1]**.  
5. Calculate Turnaround **time = WT + BT** for all the processes.  
6. Calculate **average waiting time** = total waiting time/no. of processes.  
7. Calculate **average turnaround time** = total turnaround time/no. of processes.

**Example:**

|  |  |  |
| --- | --- | --- |
| **Process** | **Arrival Time** | **Burst Time** |
| P1 | 0 | 5 |
| P2 | 0 | 11 |
| P3 | 0 | 11 |

**Gantt Chart:**  


#include<stdio.h>

void main()

{

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

int t[10];

float awt,atat;

clrscr();

printf("enter no.of processor:\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");

printf("\n process id\twaiting time\tTurn aroundtime");

printf("\n1\t\t0\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("\nAverage waiting time%4.2f",awt);

printf("\nAvereage Turn around time:%4.2f",atat);

getch();

}

**SJF algorithm:**

Write an SJF scheduling program in C to determine the average waiting time and average turnaround time given n processes and their burst times.

**SJF Scheduling Algorithm in C:**

The CPU scheduling algorithm Shortest Job First (**SJF**), allocates the CPU to the processes according to the process with smallest execution time.

**SJF** uses both preemptive and non-preemptive scheduling. The preemptive version of SJF is called **SRTF** (Shortest Remaining Time First). Here we will discuss about SJF i.e., the non-preemptive scheduling.

**Advantages of SJF:**

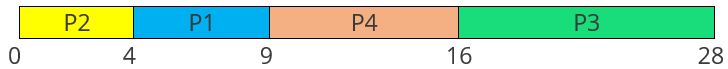
* It has the minimum waiting time among all the scheduling algorithms.
* A process having larger burst time may get into starvation but the problem can be solved using concept of Ageing.
* It is a greedy algorithm and provides optimal scheduling.

**Problem Solution**

1. Enter number of processes.  
2. Enter the **burst time** of all the processes.  
3. Sort all the processes according to their **burst time**.  
4. Find waiting time, **WT** of all the processes.  
5. For the smallest process, **WT = 0**.  
6. For all the next processes **i**, find waiting time by adding burst time of all the previously completed process.  
7. Calculate **Turnaround time = WT + BT** for all the processes.  
8. Calculate **average waiting time = total waiting time / no. of processes**.  
9. Calculate **average turnaround time= total turnaround time / no. of processes**.

**SJF Example:**

|  |  |  |
| --- | --- | --- |
| **Process** | **Arrival Time** | **Burst Time** |
| P1 | 0 | 5 |
| P2 | 0 | 4 |
| P3 | 0 | 12 |
| P4 | 0 | 7 |

**Gantt Chart:**  


|  |
| --- |
| #include<stdio.h>   int main()  {      int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;      float avg\_wt,avg\_tat;      printf("Enter number of process:");      scanf("%d",&n);        printf("nEnter Burst Time:n");      for(i=0;i<n;i++)      {          printf("p%d:",i+1);          scanf("%d",&bt[i]);          p[i]=i+1;      }       //sorting of burst times      for(i=0;i<n;i++)      {          pos=i;          for(j=i+1;j<n;j++)          {              if(bt[j]<bt[pos])                  pos=j;          }            temp=bt[i];          bt[i]=bt[pos];          bt[pos]=temp;            temp=p[i];          p[i]=p[pos];          p[pos]=temp;      }        wt[0]=0;          for(i=1;i<n;i++)      {          wt[i]=0;          for(j=0;j<i;j++)              wt[i]+=bt[j];            total+=wt[i];      }        avg\_wt=(float)total/n;      total=0;        printf("nProcesst    Burst Time    tWaiting TimetTurnaround Time");  for(i=0;i<n;i++)  {          tat[i]=bt[i]+wt[i];          total+=tat[i];          printf("np%dtt  %dtt    %dttt%d",p[i],bt[i],wt[i],tat[i]);      }        avg\_tat=(float)total/n;      printf("nnAverage Waiting Time=%f",avg\_wt);      printf("nAverage Turnaround Time=%fn",avg\_tat);  } |

**Priority algorithm:**

**Priority Scheduling** is a CPU scheduling algorithm in which the CPU performs the task having higher priority at first. If two processes have the same priority then scheduling is done on **FCFS** basis (first come first serve). Priority Scheduling is of two types : **Preemptive** and **Non-Preemptive**.

**Preemptive:** In this case, resources can be voluntarily snatched.

**Non-Preemptive:** In this type, if a process is once started, it will execute completely i.e resources cannot be snatched.

**Following are the basic terminologies:**

**Waiting Time:** Time for which the process has to wait in the ready queue.

**Turn Around Time:** Total time taken by the process for execution (waiting time + burst time).

**Problem Description:**

Write a C Program to implement priority scheduling.

**Example:**  
Following is the example of non preemptive scheduling with arrival time zero.

|  |  |  |
| --- | --- | --- |
| **Process** | **Burst Time** | **Priority** |
|  |  |  |
| P1 | 5 | 1 |
| P2 | 7 | 6 |
| P3 | 2 | 4 |
| P4 | 3 | 5 |

Since scheduling is non preemptive, which means that the process will be fully executed once its execution is started. So processes will be executed in the same order of priority.

**Order:** P2, P4, P3, P1

P2 will be executed from 0 to 7.  
P4 will be executed from 7 to 10.  
P3 will be executed from 10 to 12.  
P1 will be executed from 12 to 17.

|  |  |  |  |
| --- | --- | --- | --- |
| **Process Id** | **Burst Time** | **Wait Time** | **Turn Around Time** |
|  |  |  |  |
| P2 | 7 | 0 | 7 |
| P4 | 3 | 7 | 10 |
| P3 | 2 | 10 | 12 |
| P1 | 5 | 12 | 17 |

**Problem Solution:**

**Priority Scheduling Algorithm:**

**Take**[**C Programming Tests**](https://test.sanfoundry.com/c-programming-tests/)**Now!**

Step 1: Start the Program.  
Step 2: Input the number of processes.  
Step 3: Input the burst time and priority for each process.  
Step 4: Sort the element on the basis of priority.  
Step 5: Print order of execution of their process with their time stamp (wait time and turnaround time).  
Step 6: End the Program.

#

#include<stdio.h>

// structure representing a structure

struct priority\_scheduling

{

// name of the process

char process\_name;

// time required for execution

int burst\_time;

// waiting time of a process

int waiting\_time;

// total time of execution

int turn\_around\_time;

// priority of the process

int priority;

};

int main()

{

// total number of processes

int number\_of\_process;

// total waiting and turnaround time

int total = 0;

// temporary structure for swapping

struct priority\_scheduling temp\_process;

// ASCII numbers are used to represent the name of the process

int ASCII\_number = 65;

// swapping position

int position;

// average waiting time of the process

float average\_waiting\_time;

// average turnaround time of the process

float average\_turnaround\_time;

printf("Enter the total number of Processes: ");

// get the total number of the process as input

scanf("%d", & number\_of\_process);

// initializing the structure array

struct priority\_scheduling process[number\_of\_process];

printf("\nPlease Enter the Burst Time and Priority of each process:\n");

// get burst time and priority of all process

for (int i = 0; i < number\_of\_process; i++)

{

// assign names consecutively using ASCII number

process[i].process\_name = (char) ASCII\_number;

printf("\nEnter the details of the process %c \n", process[i].process\_name);

printf("Enter the burst time: ");

scanf("%d", & process[i].burst\_time);

printf("Enter the no processes priority: ");

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

// increment the ASCII number to get the next alphabet

ASCII\_number++;

}

// swap process according to high priority

for (int i = 0; i < number\_of\_process; i++)

{

position = i;

for (int j = i + 1; j < number\_of\_process; j++)

{

// check if priority is higher for swapping

if (process[j].priority > process[position].priority)

position = j;

}

// swapping of lower priority process with the higher priority process

temp\_process = process[i];

process[i] = process[position];

process[position] = temp\_process;

}

// First process will not have to wait and hence has a waiting time of 0

process[0].waiting\_time = 0;

for (int i = 1; i < number\_of\_process; i++) {

process[i].waiting\_time = 0;

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

// calculate waiting time

process[i].waiting\_time += process[j].burst\_time;

}

// calculate total waiting time

total += process[i].waiting\_time;

}

// calculate average waiting time

average\_waiting\_time = (float) total / (float) number\_of\_process;

// assigning total as 0 for next calculations

total = 0;

printf("\n\nProcess\_name \t Burst Time \t Waiting Time \t Turnaround Time\n");

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

for (int i = 0; i < number\_of\_process; i++)

{

// calculating the turnaround time of the processes

process[i].turn\_around\_time = process[i].burst\_time + process[i].waiting\_time;

// calculating the total turnaround time.

total += process[i].turn\_around\_time;

// printing all the values

printf("\t %c \t\t %d \t\t %d \t\t %d", process[i].process\_name, process[i].burst\_time, process[i].waiting\_time, process[i].turn\_around\_time);

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

}

// calculating the average turn\_around time

average\_turnaround\_time = (float) total / (float) number\_of\_process;

// average waiting time

printf("\n\n Average Waiting Time : %f", average\_waiting\_time);

// average turnaround time

printf("\n Average Turnaround Time: %f\n", average\_turnaround\_time);

return 0;

}

**ROUND ROBIN SCHEDULING ALGORITHM**

**Problem Description:**

Write a C Program that implements the Round Robin Scheduling algorithm and determines the average waiting time and turnaround time.

**What is Round Robin Scheduling in C?**

**Round Robin Scheduling** is a CPU scheduling algorithm in which each process is executed for a fixed time slot. Since the resources are snatched after the time slot, round robin is preemptive.

**Preemptive:** In this type, resources can be voluntarily snatched.

**Non-Preemptive:** In this type, if a process is once started, it will execute completely i.e resources cannot be snatched.

Following are the basic terminologies:

**Turnaround Time:** Difference between completion time and arrival time.

**Turnaround Time = Completion Time – Arrival Time**

**Waiting Time:** Time Difference between tur**Problem Solution**

**Round Robin Scheduling Algorithm:**

Step 1: Start the Program.  
Step 2: Input the number of processes.  
Step 3: Input the burst time and arrival time of each process and the limit of the time slot.  
Step 4: Push all processes into the ready queue according to their arrival time. Then execute each process upto time slot and push left over process in queue again for execution.  
Step 5: After a process is completely executed, print its turn around time and waiting time.

**Example:**

Following is the example of round robin scheduling.

|  |  |  |
| --- | --- | --- |
| **Process Id** | **Arrival Time** | **Burst Time** |
| P1 | 0 | 10 |
| P2 | 1 | 8 |
| P3 | 2 | 7 |

**Time Slot** is 5 Sec.

First **P1** is executed for 5 seconds, left burst time is 5 sec

Then **P2** is executed for 5 seconds, left burst time is 3 sec

Then **P3** is executed for 5 seconds, left burst time is 2 sec

Then **P1** is executed for 5 seconds, execution of **P1** is completed.

Then **P2** is executed for 3 seconds, execution of **P2** is completed.

Then **P1** is executed for 2 sec, execution **P3** is completed.

naround time and burst time.

**Waiting Time = Turnaround Time – Burst Time**

#rround robin

#include<stdio.h>

int main()

{

int i, limit, total = 0, x, counter = 0, time\_quantum;

int wait\_time = 0, turnaround\_time = 0, arrival\_time[10], burst\_time[10], temp[10];

float average\_wait\_time, average\_turnaround\_time;

printf("nEnter Total Number of Processes:t");

scanf("%d", &limit);

x = limit;

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

{

printf("nEnter Details of Process[%d]n", i + 1);

printf("Arrival Time:t");

scanf("%d", &arrival\_time[i]);

printf("Burst Time:t");

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

temp[i] = burst\_time[i];

}

printf("nEnter Time Quantum:t");

scanf("%d", &time\_quantum);

printf("nProcess ID\t\tBurst Timet \tTurnaround Time\t Waiting Timen");

for(total = 0, i = 0; x != 0;)

{

if(temp[i] <= time\_quantum && temp[i] > 0)

{

total = total + temp[i];

temp[i] = 0;

counter = 1;

}

else if(temp[i] > 0)

{

temp[i] = temp[i] - time\_quantum;

total = total + time\_quantum;

}

if(temp[i] == 0 && counter == 1)

{

x--;

printf("\nProcess[%d]\t\t%d\t\t %d\t\t\t\t %d", i + 1, burst\_time[i], total - arrival\_time[i], total - arrival\_time[i] - burst\_time[i]);

wait\_time = wait\_time + total - arrival\_time[i] - burst\_time[i];

turnaround\_time = turnaround\_time + total - arrival\_time[i];

counter = 0;

}

if(i == limit - 1)

{

i = 0;

}

else if(arrival\_time[i + 1] <= total)

{

i++;

}

else

{

i = 0;

}

}

average\_wait\_time = wait\_time \* 1.0 / limit;

average\_turnaround\_time = turnaround\_time \* 1.0 / limit;

printf("nnAverage Waiting Time:\t%f", average\_wait\_time);

printf("nAvg Turnaround Time:\t%f\n", average\_turnaround\_time);

return 0;

}