OS

# SYStem calls:

## fork()

* It creates a child process from a parent running.
* Then the child and parent runs simultaneously

fork.c

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main(){

fork();

fork();

printf("Hello\n");

return 0;

}

Output  
Hello  
Hello  
Hello  
  
Hello

## exec()

* It creates a child process from a parent runing.
* As the exec() command is hit, the parent is terminated and the child starts running.

exec1.c

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

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

{

printf("Exec1 - Hello1\n");

execl("./exec2", "exec", NULL);

printf("Hello2\n");

return 0;

}

exec2.c

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

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

{

printf("Exec2 - Hello1\n");

//execl("./exec1", "exec", NULL);

printf("Exec2 - Hello2\n");

return 0;

}

Output  
Exec1 - Hello1  
Exec2 - Hello1  
Exec2 - Hello2

## exit()

* When exit() is called, any open file descriptors belonging to the process are closed and any children of the process are inherited by process 1, init, and the process parent is sent a SIGCHLD signal.

### 0 or EXIT\_SUCCESS :- The program has been successfully executed without encountering any error.

### 1 or EXIT\_FAILURE :- The program has encountered an error and could be executed successfully.

exit.c

#include <stdio.h>

#include <stdlib.h>

int main()

{

FILE\* pFile;

pFile = fopen("myfile.txt", "r");

if (pFile == NULL) {

printf("Error opening file");

// terminating the process if the file is not opened

exit(1);

}

else {

/\* file operations here \*/

}

return 0;

}

output  
Error opening file

## kill()

* It is a built-in command which is used to terminate processes manually. kill command sends a signal to a process that terminates the process. If the user doesn’t specify any signal that is to be sent along with the kill command, then a default TERM signal is sent that terminates the process.

### SYNTAX: kill [signal] PID

## wait()

* This system call is used in processes that have a parent-child relationship. It makes a parent process stop its execution till the termination of the child process. We can create a child process using the fork() system call.

wait.c

#include <stdio.h>

#include <sys/wait.h>

#include <stdlib.h>

#include <unistd.h>

int main(){

// create a child process

int child = fork();

int exitStatus;

int childPid;

printf("Child: I am running!!\n\n");

printf("Child: I have PID: %d\n\n", getpid());

// code for the child process

if(child == 0){

// display running message and make child sleep

printf("Child: I am running!!\n\n");

printf("Child: I have PID: %d\n\n", getpid());

sleep(4);

// child exits with code 100

exit(100);

} // code for the parent process

else{

// print parent running message

printf("Parent: I am running and waiting for child to finish!!\n\n");

// call wait system call

childPid = wait(&exitStatus);

// print the details of the child process

printf("Parent: Child finished execution!, It had the PID: %d, Exit Status: %d\n\n", childPid, WEXITSTATUS(exitStatus));

}

return 0;

}

output

Child: I am running!!

Child: I have PID: 7254

Parent: I am running and waiting for child to finish!!

Child: I am running!!

Child: I have PID: 7255

Child: I am running!!

Child: I have PID: 7255

Parent: Child finished execution!, It had the PID: 7255, Exit Status: 100

# Scheduling Algorithms:

## FCFS.cpp

#include<bits/stdc++.h>

usingnamespacestd**;**

struct**Process**{

*int* id**;**

*int* burstTime**;**

*int* waitingTime**;**

*int* turnaroundTime**;**

}**;**

*void*fcfs(**vector**<**Process**>*&***processes**){

*int* n **=**processes**.**size()**;**

    vector**<**Process**>**result(n)**;**

    result[0]**.***waitingTime***=**0**;**

    result[0]**.***turnaroundTime***=** processes[0]**.***burstTime***;**

**for**(*int* i **=**1**;** i **<** n**;** i**++**){

        result[i]**.***waitingTime***=** result[i **-**1]**.***turnaroundTime***;**

        result[i]**.***turnaroundTime***=** result[i]**.***waitingTime***+** processes[i]**.***burstTime***;**

    }

*double* totalWaitingTime **=**0**;**

*double* totalTurnaroundTime **=**0**;**

    cout **<<**"Process\tBurst Time\tWaiting Time\tTurnaround Time\n"**;**

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

        cout **<<** processes[i]**.***id***<<**"\t\t"**<<** processes[i]**.***burstTime***<<**"\t\t"**<<** result[i]**.***waitingTime***<<**"\t\t"**<<** result[i]**.***turnaroundTime***<<**"\n"**;**

        totalWaitingTime **+=** result[i]**.***waitingTime***;**

        totalTurnaroundTime **+=** result[i]**.***turnaroundTime***;**

    }

*double* averageWaitingTime **=** totalWaitingTime **/** n**;**

*double* averageTurnaroundTime **=** totalTurnaroundTime **/** n**;**

    cout **<<**"\nAverage Waiting Time: "**<<** averageWaitingTime **<<**"\n"**;**

    cout **<<**"Average Turnaround Time: "**<<** averageTurnaroundTime **<<**"\n"**;**

}

*int*main(){

*int* n**;**

    cout**<<**"Enter the number of Processes: "**;**

    cin**>>**n**;**

    vector**<**Process**>**processes(n)**;**

    cout**<<**"Enter burst time for each processes serially: "**<<**endl**;**

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

        processes[i]**.***id***=** i**+**1**;**

        cin**>>**processes[i]**.***burstTime***;**

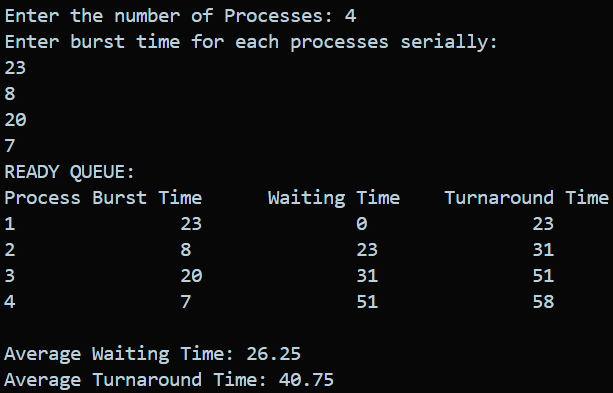
    }

    cout**<<**"READY QUEUE:"**<<**endl**;**

    fcfs(processes)**;**

**return**0**;**

}

OUTPUT

## SJF.cpp

#include<bits/stdc++.h>

usingnamespacestd**;**

struct**Process**{

*intid***;**

*intburstTime***;**

*intwaitingTime***;**

*intturnaroundTime***;**

}**;**

*bool*compareByBurstTime(*const***Process***&***a,***const***Process***&***b**){

**returna.***burstTime***<b.***burstTime***;**

}

*void*sjf(**vector**<**Process**>*&***processes**){

    sort(**processes.**begin()**,processes.**end()**,**compareByBurstTime)**;**

*int* n **=processes.**size()**;**

**processes[**0**].***waitingTime***=**0**;**

**processes[**0**].***turnaroundTime***=processes[**0**].***burstTime***;**

**for**(*int* i **=**1**;** i **<** n**;** i**++**){

**processes[**i**].***waitingTime***=processes[**i **-**1**].***burstTime***+processes[**i **-**1**].***waitingTime***;**

**processes[**i**].***turnaroundTime***=processes[**i**].***waitingTime***+processes[**i**].***burstTime***;**

    }

*double* totalWaitingTime **=**0**;**

*double* totalTurnaroundTime **=**0**;**

    cout **<<**"Process Number\tBurst Time\tWaiting Time\tTurnaround Time\n"**;**

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

        cout **<<processes[**i**].***id***<<**"\t\t"**<<processes[**i**].***burstTime***<<**"\t\t"

**<<processes[**i**].***waitingTime***<<**"\t\t"**<<processes[**i**].***turnaroundTime***<<**"\n"**;**

        totalWaitingTime **+=processes[**i**].***waitingTime***;**

        totalTurnaroundTime **+=processes[**i**].***turnaroundTime***;**

    }

*double* avgWaitingTime **=** totalWaitingTime **/** n**;**

*double* avgTurnaroundTime **=** totalTurnaroundTime **/** n**;**

    cout **<<**"\nAverage Waiting Time: "**<<** avgWaitingTime **<<**"\n"**;**

    cout **<<**"Average Turnaround Time: "**<<** avgTurnaroundTime **<<**"\n"**;**

}

*int*main(){

*int* n**;**

    cout**<<**"Enter the number of Processes: "**;**

    cin**>>**n**;**

**vector<Process>** processes(n)**;**

    cout**<<**"Enter burst time for each processes serially: "**<<**endl**;**

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

        processes**[**i**].***id***=** i**+**1**;**

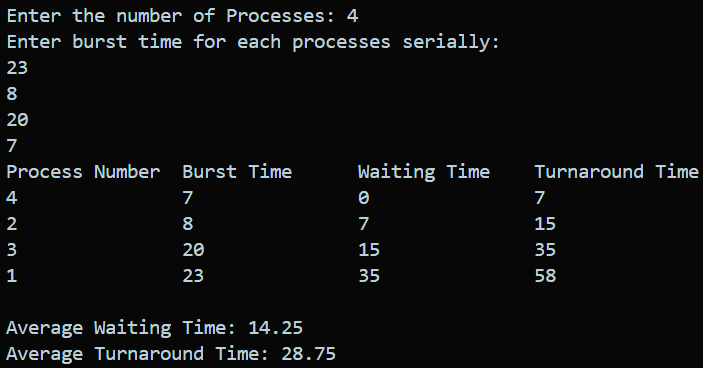
        cin**>>**processes**[**i**].***burstTime***;**

    }

    sjf(processes)**;**

**return**0**;**

}

OUTPUT

## Preemptive-SJF.cpp

#include<bits/stdc++.h>

usingnamespacestd**;**

struct**Process**{

*int* id**;**

*int* burstTime**;**

*int* arrivalTime**;**

*int* waitingTime**;**

*int* turnAroundTime**;**

}**;**

*bool*compareArrivalTime(*const***Process***&***p1,***const***Process***&***p2**){

**return** p1**.***arrivalTime***<** p2**.***arrivalTime***;**

}

*void*psjf(*const***vector**<**Process**>*&***processes**){

**vector<Process>** sortedProcesses **=processes;**

    sort(sortedProcesses**.**begin()**,** sortedProcesses**.**end()**,**compareArrivalTime)**;**

*int* n **=** sortedProcesses**.**size()**;**

**vector<***int***>** remainingTime(n)**;**

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

        remainingTime**[**i**]=** sortedProcesses**[**i**].***burstTime***;**

    }

*int* currentTime **=**0**;**

*int* completed **=**0**;**

**while**(completed **!=** n){

*int* shortestJob **=-**1**;**

*int* shortestTime **=**INT\_MAX**;**

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

**if**(sortedProcesses**[**i**].***arrivalTime***<=** currentTime **&&** remainingTime**[**i**]<** shortestTime **&&** remainingTime**[**i**]>**0){

                shortestJob **=** i**;**

                shortestTime **=** remainingTime**[**i**];**

            }

        }

**if**(shortestJob **==-**1){

            currentTime**++;**

**continue;**

        }

        remainingTime**[**shortestJob**]--;**

**if**(remainingTime**[**shortestJob**]==**0){

            completed**++;**

*int* id **=** sortedProcesses**[**shortestJob**].***id***;**

            sortedProcesses**[**shortestJob**].***turnAroundTime***=** currentTime **+**1**-** sortedProcesses**[**shortestJob**].***arrivalTime***;**

            sortedProcesses**[**shortestJob**].***waitingTime***=** sortedProcesses**[**shortestJob**].***turnAroundTime***-** sortedProcesses**[**shortestJob**].***burstTime***;**

            cout **<<**"Process "**<<** id **<<**":\n"**;**

            cout **<<**"Waiting Time: "**<<** sortedProcesses**[**shortestJob**].***waitingTime***<<**"\n"**;**

            cout **<<**"Turnaround Time: "**<<** sortedProcesses**[**shortestJob**].***turnAroundTime***<<**"\n"**;**

        }

        currentTime**++;**

    }

*double* totalWaitingTime **=**0**;**

*double* totalTurnaroundTime **=**0**;**

**for**(*const***Process&**process **:** sortedProcesses){

        totalWaitingTime **+=**process**.***waitingTime***;**

        totalTurnaroundTime **+=**process**.***turnAroundTime***;**

    }

*double* avgWaitingTime **=** totalWaitingTime **/** n**;**

*double* avgTurnaroundTime **=** totalTurnaroundTime **/** n**;**

    cout **<<**"Average Waiting Time: "**<<** avgWaitingTime **<<**"\n"**;**

    cout **<<**"Average Turnaround Time: "**<<** avgTurnaroundTime **<<**"\n"**;**

}

*int*main(){

*int* n**;**

    cout **<<**"Enter the number of processes: "**;**

    cin **>>** n**;**

    cout**<<**"ENter the Burst time and Arrival time for each process serially: "**<<**endl**;**

    vector**<**Process**>**processes(n)**;**

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

        processes[i]**.***id***=** i **+**1**;**

        cin **>>** processes[i]**.***burstTime***;**

        cin **>>** processes[i]**.***arrivalTime***;**

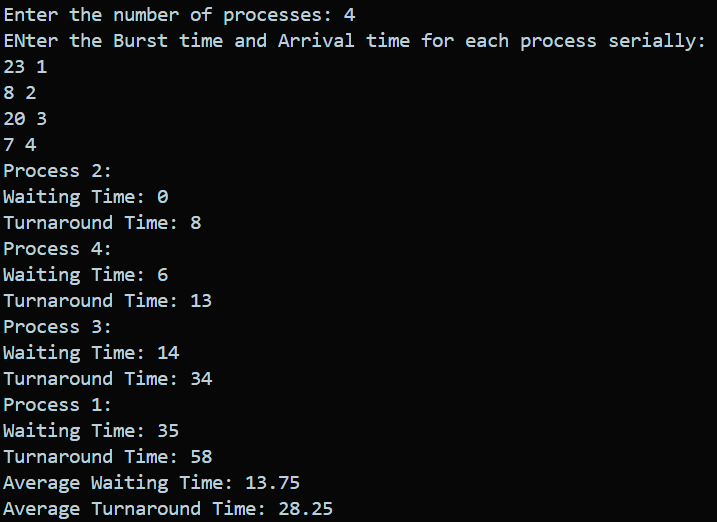
    }

    psjf(processes)**;**

**return**0**;**

}

OUTPUT



## PriorityScheduling.cpp

#include<bits/stdc++.h>

usingnamespacestd**;**

struct**Process**{

*intid***;**

*intpriority***;**

*intburstTime***;**

}**;**

*bool*comparePriority(*const***Process***&***p1,***const***Process***&***p2**){

**returnp1.***priority***<p2.***priority***;**

}

*void*priorityScheduling(*const***vector**<**Process**>*&***processes**){

**vector<Process>** sortedProcesses **=processes;**

    sort(sortedProcesses**.**begin()**,** sortedProcesses**.**end()**,**comparePriority)**;**

*int* currentTime **=**0**;**

*int* totalWaitingTime **=**0**;**

*int* totalTurnaroundTime **=**0**;**

    cout **<<**"Process\tWaiting Time\tTurnaround Time\n"**;**

**for**(*const***Process&**process **:** sortedProcesses){

*int* waitingTime **=** currentTime**;**

*int* turnaroundTime **=** currentTime **+**process**.***burstTime***;**

        cout **<<** process**.***id***<<**"\t"**<<** waitingTime **<<**"\t\t"**<<** turnaroundTime **<<**"\n"**;**

        totalWaitingTime **+=** waitingTime**;**

        totalTurnaroundTime **+=** turnaroundTime**;**

        currentTime **+=**process**.***burstTime***;**

    }

*double* averageWaitingTime **=static\_cast<***double***>**(totalWaitingTime)**/** sortedProcesses**.**size()**;**

*double* averageTurnaroundTime **=static\_cast<***double***>**(totalTurnaroundTime)**/** sortedProcesses**.**size()**;**

    cout **<<**"\nAverage Waiting Time: "**<<** averageWaitingTime **<<**"\n"**;**

    cout **<<**"Average Turnaround Time: "**<<** averageTurnaroundTime **<<**"\n"**;**

}

*int*main(){

*int* n**;**

    cout **<<**"Enter the number of processes: "**;**

    cin **>>** n**;**

    cout**<<**"Enter burst time and priority for each processes serially: "**<<**endl**;**

**vector<Process>** processes(n)**;**

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

        processes**[**i**].***id***=** i **+**1**;**

        cin **>>** processes**[**i**].***burstTime***;**

        cin **>>** processes**[**i**].***priority***;**

    }

    priorityScheduling(processes)**;**

**return**0**;**

}

OUTPUT

## Preemptive-priorityScheduling.cpp

#include<bits/stdc++.h>

usingnamespacestd**;**

struct**Process**{

*intid***;**

*intarrivalTime***;**

*intburstTime***;**

*intpriority***;**

*intremainingTime***;**

}**;**

*bool*compareArrivalTime(*const***Process***&***p1,***const***Process***&***p2**){

**returnp1.***arrivalTime***<p2.***arrivalTime***;**

}

*bool*comparePriority(*const***Process***&***p1,***const***Process***&***p2**){

**returnp1.***priority***<p2.***priority***;**

}

*void*preemptivePriorityScheduling(**vector**<**Process**>*&***processes**){

    sort(**processes.**begin()**,processes.**end()**,**compareArrivalTime)**;**

*int* n **=processes.**size()**;**

**vector<***int***>** completionTime(n)**;**

**vector<***int***>** turnaroundTime(n)**;**

**vector<***int***>** waitingTime(n)**;**

**vector<***bool***>**isCompleted(n**,**false)**;**

*int* currentTime **=**0**;**

*int* completedProcesses **=**0**;**

**while**(completedProcesses **<** n){

*int* highestPriority **=**INT\_MAX**;**

*int* selectedProcess **=-**1**;**

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

**if**(**!**isCompleted**[**i**]&&processes[**i**].***arrivalTime***<=** currentTime **&&processes[**i**].***priority***<** highestPriority){

                highestPriority **=processes[**i**].***priority***;**

                selectedProcess **=** i**;**

            }

        }

**if**(selectedProcess **==-**1){

            currentTime**++;**

**continue;**

        }

**processes[**selectedProcess**].***remainingTime***--;**

**if**(**processes[**selectedProcess**].***remainingTime***==**0){

            completionTime**[**selectedProcess**]=** currentTime **+**1**;**

            turnaroundTime**[**selectedProcess**]=** completionTime**[**selectedProcess**]-processes[**selectedProcess**].***arrivalTime***;**

            waitingTime**[**selectedProcess**]=** turnaroundTime**[**selectedProcess**]-processes[**selectedProcess**].***burstTime***;**

            isCompleted**[**selectedProcess**]=**true**;**

            completedProcesses**++;**

        }

        currentTime**++;**

    }

    cout **<<**"Process\tArrival Time\tBurst Time\tPriority\tCompletion Time\tTurnaround Time\tWaiting Time\n"**;**

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

        cout **<<processes[**i**].***id***<<**"\t\t"**<<processes[**i**].***arrivalTime***<<**"\t\t"**<<processes[**i**].***burstTime***<<**"\t\t"**<<processes[**i**].***priority***<<**"\t\t"**<<** completionTime**[**i**]<<**"\t\t\t"**<<** turnaroundTime**[**i**]<<**"\t\t\t"**<<** waitingTime**[**i**]<<**"\n"**;**

    }

}

*int*main(){

*int* n**;**

    cout **<<**"Enter the number of processes: "**;**

    cin **>>** n**;**

    cout**<<**"Enter burst time and priority and arrival time for each process serially: "**<<**endl**;**

**vector<Process>** processes(n)**;**

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

        processes**[**i**].***id***=** i **+**1**;**

        cin **>>** processes**[**i**].***burstTime***;**

        cin **>>** processes**[**i**].***priority***;**

        cin **>>** processes**[**i**].***arrivalTime***;**

        processes**[**i**].***remainingTime***=** processes**[**i**].***burstTime***;** *// Initialize remainingTime*

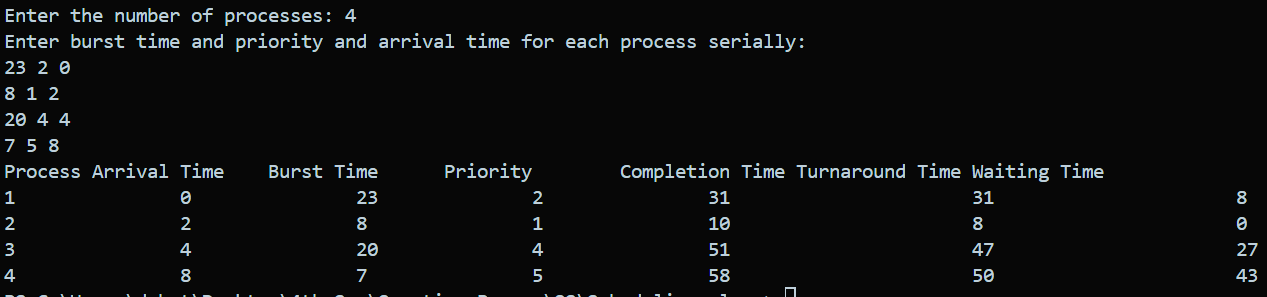
    }

    preemptivePriorityScheduling(processes)**;**

**return**0**;**

}

OUTPUT



## RoundRobinScheduling.cpp

#include<iostream>

#include<queue>

#include<vector>

usingnamespacestd**;**

struct**Process**{

*intpid***;**

*intburstTime***;**

*intarrivalTime***;**

*intremainingTime***;**

*intturnaroundTime***;**

*intwaitingTime***;**

}**;**

*void*roundRobinScheduling(**vector**<**Process**>*&***processes,***int***n,***int***quantum**){

**queue<Process>** readyQueue**;**

*int* currentTime **=**0**;**

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

**processes[**i**].***remainingTime***=processes[**i**].***burstTime***;**

    }

*int* completedProcesses **=**0**;**

**while**(completedProcesses **<n**){

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

**if**(**processes[**i**].***arrivalTime***<=** currentTime **&&processes[**i**].***remainingTime***>**0){

**if**(**processes[**i**].***remainingTime***<=quantum**){

                    currentTime **+=processes[**i**].***remainingTime***;**

**processes[**i**].***remainingTime***=**0**;**

**processes[**i**].***turnaroundTime***=** currentTime **-processes[**i**].***arrivalTime***;**

**processes[**i**].***waitingTime***=processes[**i**].***turnaroundTime***-processes[**i**].***burstTime***;**

                    completedProcesses**++;**

                }**else**{

                    currentTime **+=quantum;**

**processes[**i**].***remainingTime***-=quantum;**

                }

                cout **<<**"Process "**<<processes[**i**].***pid***<<**" executed for "**<<**min(**quantum,processes[**i**].***burstTime*)**<<**" units."**<<**endl**;**

            }

        }

    }

    cout **<<**"Process\tTurnaround Time\tWaiting Time"**<<**endl**;**

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

        cout **<<processes[**i**].***pid***<<**"\t"**<<processes[**i**].***turnaroundTime***<<**"\t\t"**<<processes[**i**].***waitingTime***<<**endl**;**

    }

}

*int*main(){

*int* n**;**

    cout **<<**"Enter the number of processes: "**;**

    cin **>>** n**;**

**vector<Process>** processes(n)**;**

    cout **<<**"Enter the Burst Times and Arrival Times"**<<**endl**;**

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

        processes**[**i**].***pid***=** i **+**1**;**

        cin **>>** processes**[**i**].***burstTime***;**

        cin **>>** processes**[**i**].***arrivalTime***;**

    }

*int* quantum**;**

    cout **<<**"Enter the time quantum: "**;**

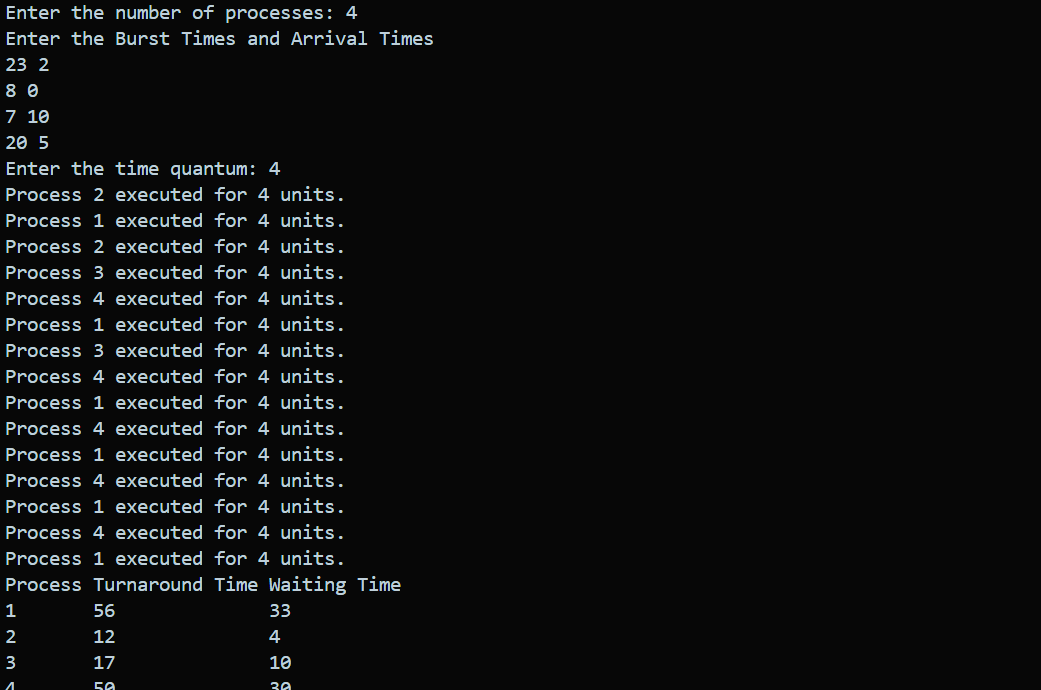
    cin **>>** quantum**;**

    roundRobinScheduling(processes**,** n**,** quantum)**;**

**return**0**;**

}

OUTPUT



## MultilevelQueueScheduling.cpp

#include<iostream>

#include<queue>

#include<vector>

#include<algorithm>

usingnamespacestd**;**

struct**Process**{

*intid***;**

*intburstTime***;**

}**;**

*// Define a comparison operator for Process struct to sort based on burst time*

*bool***operator<**(*const***Process***&***a,***const***Process***&***b**){

**returna.***burstTime***>b.***burstTime***;**

}

*void*multilevelQueueScheduling(**vector**<**Process**>*&***processes**){

**priority\_queue<Process>** highPriorityQueue**;**

**priority\_queue<Process>** mediumPriorityQueue**;**

**priority\_queue<Process>** lowPriorityQueue**;**

*// Sorting processes based on burst time*

    sort(**processes.**begin()**,processes.**end())**;**

**for**(*constauto***&**process **:processes**){

**if**(process**.***burstTime***<=**2){

            highPriorityQueue**.**push(process)**;**

        }**elseif**(process**.***burstTime***<=**4){

            mediumPriorityQueue**.**push(process)**;**

        }**else**{

            lowPriorityQueue**.**push(process)**;**

        }

    }

    cout **<<**"Execution Order: "**;**

**while**(**!**highPriorityQueue**.**empty()**||!**mediumPriorityQueue**.**empty()**||!**lowPriorityQueue**.**empty()){

**if**(**!**highPriorityQueue**.**empty()){

**Process** process **=**highPriorityQueue**.**top()**;**

            highPriorityQueue**.**pop()**;**

            cout **<<** process**.***id***<<**" "**;**

            process**.***burstTime***-=**2**;**

**if**(process**.***burstTime***>**0){

                highPriorityQueue**.**push(process)**;**

            }

        }**elseif**(**!**mediumPriorityQueue**.**empty()){

**Process** process **=**mediumPriorityQueue**.**top()**;**

            mediumPriorityQueue**.**pop()**;**

            cout **<<** process**.***id***<<**" "**;**

            process**.***burstTime***-=**4**;**

**if**(process**.***burstTime***>**0){

                mediumPriorityQueue**.**push(process)**;**

            }

        }**else**{

**Process** process **=**lowPriorityQueue**.**top()**;**

            lowPriorityQueue**.**pop()**;**

            cout **<<** process**.***id***<<**" "**;**

            process**.***burstTime***-=**6**;**

**if**(process**.***burstTime***>**0){

                lowPriorityQueue**.**push(process)**;**

            }

        }

    }

    cout **<<**endl**;**

}

*int*main(){

*int* numProcesses**;**

    cout **<<**"Enter the number of processes: "**;**

    cin **>>** numProcesses**;**

**vector<Process>** processes(numProcesses)**;**

**for**(*int* i **=**0**;** i **<** numProcesses**;** i**++**){

        cout **<<**"Enter burst time for process "**<<** i **+**1**<<**": "**;**

        cin **>>** processes**[**i**].***burstTime***;**

        processes**[**i**].***id***=** i **+**1**;**

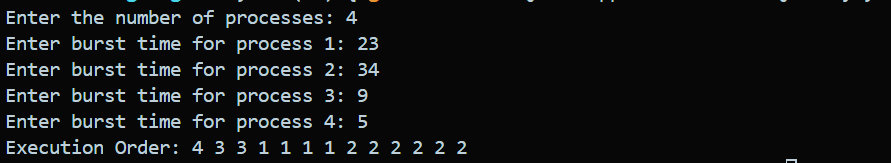
    }

    multilevelQueueScheduling(processes)**;**

**return**0**;**

}

OUTPUT



## MultilevelFeedbackQueueScheduling.cpp

#include<iostream>

#include<queue>

#include<vector>

usingnamespacestd**;**

struct**Process**{

*intid***;**

*intburstTime***;**

}**;**

*void*multilevelFeedbackQueueScheduling(**vector**<**Process**>*&***processes**){

**queue<Process>** highPriorityQueue**;**

**queue<Process>** mediumPriorityQueue**;**

**queue<Process>** lowPriorityQueue**;**

**for**(*auto***&**process **:processes**){

        highPriorityQueue**.**push(process)**;**

    }

*int* timeQuantumHigh **=**2**;**

*int* timeQuantumMedium **=**4**;**

*int* timeQuantumLow **=**6**;**

    cout **<<**"Execution Order: "**;**

**while**(**!**highPriorityQueue**.**empty()**||!**mediumPriorityQueue**.**empty()**||!**lowPriorityQueue**.**empty()){

**if**(**!**highPriorityQueue**.**empty()){

**Process** currentProcess **=** highPriorityQueue**.**front()**;**

            highPriorityQueue**.**pop()**;**

            cout **<<** currentProcess**.***id***<<**" "**;**

**if**(currentProcess**.***burstTime***>** timeQuantumHigh){

                currentProcess**.***burstTime***-=** timeQuantumHigh**;**

                mediumPriorityQueue**.**push(currentProcess)**;**

            }

        }**elseif**(**!**mediumPriorityQueue**.**empty()){

**Process** currentProcess **=** mediumPriorityQueue**.**front()**;**

            mediumPriorityQueue**.**pop()**;**

            cout **<<** currentProcess**.***id***<<**" "**;**

**if**(currentProcess**.***burstTime***>** timeQuantumMedium){

                currentProcess**.***burstTime***-=** timeQuantumMedium**;**

                lowPriorityQueue**.**push(currentProcess)**;**

            }

        }**elseif**(**!**lowPriorityQueue**.**empty()){

**Process** currentProcess **=** lowPriorityQueue**.**front()**;**

            lowPriorityQueue**.**pop()**;**

            cout **<<** currentProcess**.***id***<<**" "**;**

*// No need to push back to lowPriorityQueue as it's the lowest priority*

        }

    }

    cout **<<**endl**;**

}

*int*main(){

*int* numProcesses**;**

    cout **<<**"Enter the number of processes: "**;**

    cin **>>** numProcesses**;**

**vector<Process>** processes(numProcesses)**;**

**for**(*int* i **=**0**;** i **<** numProcesses**;** i**++**){

        cout **<<**"Enter burst time for process "**<<** i **+**1**<<**": "**;**

        cin **>>** processes**[**i**].***burstTime***;**

        processes**[**i**].***id***=** i **+**1**;**

    }

    multilevelFeedbackQueueScheduling(processes)**;**

**return**0**;**

}

OUTPUT

