**ROUND ROBIN**

// C++ program for implementation of RR scheduling

#include<iostream>

using namespace std;

// Function to find the waiting time for all

// processes

void findWaitingTime(int processes[], int n,

             int bt[], int wt[], int quantum)

{

    // Make a copy of burst times bt[] to store remaining

    // burst times.

    int rem\_bt[n];

    for (int i = 0 ; i < n ; i++)

        rem\_bt[i] =  bt[i];

    int t = 0; // Current time

    // Keep traversing processes in round robin manner

    // until all of them are not done.

    while (1)

    {

        bool done = true;

        // Traverse all processes one by one repeatedly

        for (int i = 0 ; i < n; i++)

        {

            // If burst time of a process is greater than 0

            // then only need to process further

            if (rem\_bt[i] > 0)

            {

                done = false; // There is a pending process

                if (rem\_bt[i] > quantum)

                {

                    // Increase the value of t i.e. shows

                    // how much time a process has been processed

                    t += quantum;

                    // Decrease the burst\_time of current process

                    // by quantum

                    rem\_bt[i] -= quantum;

                }

                // If burst time is smaller than or equal to

                // quantum. Last cycle for this process

                else

                {

                    // Increase the value of t i.e. shows

                    // how much time a process has been processed

                    t = t + rem\_bt[i];

                    // Waiting time is current time minus time

                    // used by this process

                    wt[i] = t - bt[i];

                    // As the process gets fully executed

                    // make its remaining burst time = 0

                    rem\_bt[i] = 0;

                }

            }

        }

        // If all processes are done

        if (done == true)

          break;

    }

}

// Function to calculate turn around time

void findTurnAroundTime(int processes[], int n,

                        int bt[], int wt[], int tat[])

{

    // calculating turnaround time by adding

    // bt[i] + wt[i]

    for (int i = 0; i < n ; i++)

        tat[i] = bt[i] + wt[i];

}

// Function to calculate average time

void findavgTime(int processes[], int n, int bt[],

                                     int quantum)

{

    int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

    // Function to find waiting time of all processes

    findWaitingTime(processes, n, bt, wt, quantum);

    // Function to find turn around time for all processes

    findTurnAroundTime(processes, n, bt, wt, tat);

    // Display processes along with all details

    cout << "Processes "<< " Burst time "

         << " Waiting time " << " Turn around time\n";

    // Calculate total waiting time and total turn

    // around time

    for (int i=0; i<n; i++)

    {

        total\_wt = total\_wt + wt[i];

        total\_tat = total\_tat + tat[i];

        cout << " " << i+1 << "\t\t" << bt[i] <<"\t "

             << wt[i] <<"\t\t " << tat[i] <<endl;

    }

    cout << "Average waiting time = "

         << (float)total\_wt / (float)n;

    cout << "\nAverage turn around time = "

         << (float)total\_tat / (float)n;

}

// Driver code

int main()

{

    // process id's

    int processes[] = { 1, 2, 3};

    int n = sizeof processes / sizeof processes[0];

    // Burst time of all processes

    int burst\_time[] = {5, 8, 10};

    // Time quantum

    int quantum = 8;

    findavgTime(processes, n, burst\_time, quantum);

    return 0;

}

**FCFS**

// C++ program for implementation of FCFS

// scheduling

#include<iostream>

using namespace std;

// Function to find the waiting time for all

// processes

void findWaitingTime(int processes[], int n,

int bt[], int wt[])

{

// waiting time for first process is 0

wt[0] = 0;

// calculating waiting time

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

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

}

// Function to calculate turn around time

void findTurnAroundTime( int processes[], int n,

int bt[], int wt[], int tat[])

{

// calculating turnaround time by adding

// bt[i] + wt[i]

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

tat[i] = bt[i] + wt[i];

}

//Function to calculate average time

void findavgTime( int processes[], int n, int bt[])

{

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

//Function to find waiting time of all processes

findWaitingTime(processes, n, bt, wt);

//Function to find turn around time for all processes

findTurnAroundTime(processes, n, bt, wt, tat);

//Display processes along with all details

cout << "Processes "<< " Burst time "

<< " Waiting time " << " Turn around time\n";

// Calculate total waiting time and total turn

// around time

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

{

total\_wt = total\_wt + wt[i];

total\_tat = total\_tat + tat[i];

cout << " " << i+1 << "\t\t" << bt[i] <<"\t "

<< wt[i] <<"\t\t " << tat[i] <<endl;

}

cout << "Average waiting time = "

<< (float)total\_wt / (float)n;

cout << "\nAverage turn around time = "

<< (float)total\_tat / (float)n;

}

// Driver code

int main()

{

//process id's

int processes[] = { 1, 2, 3};

int n = sizeof processes / sizeof processes[0];

//Burst time of all processes

int burst\_time[] = {5, 8, 10};

findavgTime(processes, n, burst\_time);

return 0;

}

#include<stdio.h>

#define N 10

typedef struct

{

int process\_id, arrival\_time, burst\_time, priority;

int a, start;

}process\_structure;

int Queue(int t1)

{

if(t1 == 0 || t1 == 1 || t1 == 2 || t1 == 3)

{

return 1;

}

else

{

return 2;

}

}

int main()

{

int limit, count, temp\_process, time, j, y;

process\_structure temp;

printf("Enter Total Number of Processes:\t");

scanf("%d", &limit);

process\_structure process[limit];

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

{

printf("\nProcess ID:\t");

scanf("%d", &process[count].process\_id);

printf("Arrival Time:\t");

scanf("%d", &process[count].arrival\_time);

printf("Burst Time:\t");

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

printf("Process Priority:\t");

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

temp\_process = process[count].priority;

process[count].a = Queue(temp\_process);

process[count].start = 0;

}

time = process[0].burst\_time;

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

{

for(count = y; count < limit; count++)

{

if(process[count].arrival\_time < time)

{

process[count].start = 1;

}

}

for(count = y; count < limit - 1; count++)

{

for(j = count + 1; j < limit; j++)

{

if(process[count].start == 1 && process[j].start == 1)

{

if(process[count].a== 2 && process[j].a== 1)

{

temp = process[count];

process[count] = process[j];

process[j] = temp;

}

}

}

}

for(count = y; count < limit - 1; count++)

{

for(j = count + 1; j < limit; j++)

{

if(process[count].start == 1 && process[j].start == 1)

{

if(process[count].a == 1 && process[j].a == 1)

{

if(process[count].burst\_time > process[j].burst\_time)

{

temp = process[count];

process[count] = process[j];

process[j] = temp;

}

else

{

break;

}

}

}

}

}

printf("\nProcess[%d]:\tTime:\t%d To %d\n", process[y].process\_id, time, time + process[y].burst\_time);

time = time + process[y].burst\_time;

for(count = y; count < limit; count++)

{

if(process[count].start == 1)

{

process[count].start = 0;

}

}

}

return 0;

}