OS Lab Assignment V

Varun Khadayate

NMIMS

# Definition

A round-robin is a CPU scheduling algorithm that shares equal portions of resources in circular orders to each process and handles all processes without prioritization. In the round-robin, each process gets a fixed time interval of the slice to utilize the resources or execute its task called **time quantum or time slice**. Some of the round-robin processes are pre-empted if it executed in each time slot, while the rest of the processes go back to the ready queue and wait to run in a circular order with the scheduled time slot until they complete their task. It removes the starvation for each process to achieve CPU scheduling by proper partitioning of the CPU.

# Characteristics

* Round robin is a pre-emptive algorithm
* The CPU is shifted to the next process after fixed interval time, which is called time quantum/time slice.
* The process that is preempted is added to the end of the queue.
* Round robin is a hybrid model which is clock-driven
* Time slice should be minimum, which is assigned for a specific task that needs to be processed. However, it may differ OS to OS.
* It is a real time algorithm which responds to the event within a specific time limit.
* Round robin is one of the oldest, fairest, and easiest algorithm.
* Widely used scheduling method in traditional OS.

# Part1: All processes arrive at time 0

# Implementation

1. Create an array rem\_bt[] to keep track of remaining burst time of processes. This array is initially a copy of bt[] (burst times array)
2. Create another array wt[] to store waiting times of processes. Initialize this array as 0.
3. Initialize time : t = 0
4. Keep traversing the all processes while all processes are not done. Do following for i'th process if it is not done yet.
   1. If rem\_bt[i] > quantum
      1. t = t + quantum
      2. bt\_rem[i] -= quantum;
   2. Else
      1. t = t + bt\_rem[i];
      2. wt[i] = t - bt[i]
      3. bt\_rem[i] = 0;

# Code

#include<iostream>

#include<conio.h>

using namespace std;

void findWaitingTime(int processes[], int n, int bt[], int wt[], int quantum)

{

int rem\_bt[n];

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

rem\_bt[i] = bt[i];

int t = 0;

while (1)

{

bool done = true;

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

{

if (rem\_bt[i] > 0)

{

done = false;

if (rem\_bt[i] > quantum)

{

t += quantum;

rem\_bt[i] -= quantum;

}

else

{

t = t + rem\_bt[i];

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

rem\_bt[i] = 0;

}

}

}

if (done == true)

break;

}

}

void findTurnAroundTime(int processes[], int n,

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

{

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

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

}

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

int quantum)

{

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

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

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

cout << "Processes "<< " Burst time " << " Waiting time " << " Turn around time\n";

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;

}

int main()

{

int processes[] = { 1, 2, 3, 4, 5, 6};

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

int burst\_time[] = {5, 6, 3, 1, 5, 4};

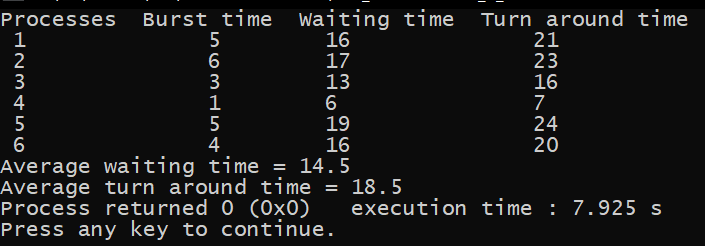
int quantum = 2;

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

return 0;

}

# Output



# Part 2: Different Arrival Time

# Implementation

* We first have a queue where the processes are arranged in first come first serve order.
* A quantum value is allocated to execute each process.
* The first process is executed until the end of the quantum value. After this, an interrupt is generated and the state is saved.
* The CPU then moves to the next process and the same method is followed.
* Same steps are repeated till all the processes are over.

# Code

#include<iostream>

#include<conio.h>

using namespace std;

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\tBurst Time\t Turnaround Time\t Waiting Time\n");

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("\nP%d\t%d\t\t %d\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("\n\nAverage Waiting Time:\t%f", average\_wait\_time);

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

return 0;

}

# Output

