

**Mawlana Bhashani Science and Technology University**

**Lab-Report**

Report No:10

Course code: ICT-3110

Course title: Operating System Lab

Date of Performance:10/09/2020

Date of Submission:10/09/2020

**Submitted by Submitted To**

Nazrul Islam

Assistant Professor

Dept. of ICT

MBSTU.

Name:Md Asikur Rahman

ID:IT-18025

3rd  year 1st semester

Session: 2017-2018

Dept. of ICT

MBSTU.

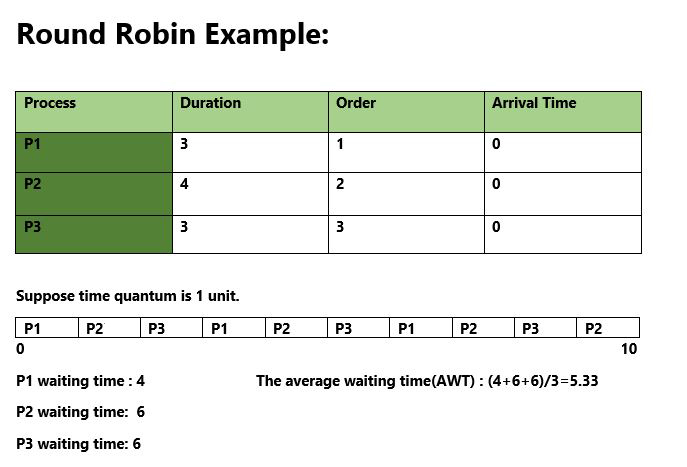
# Experiment No : 10

# Experiment Name : Implementation of Round Robin Scheduling Algorithm

# Theory:

The name of this algorithm comes from the round-robin principle, where each person gets an equal share of something in turns. It is the oldest, simplest scheduling algorithm, which is mostly used for multitasking.

In Round-robin scheduling, each ready task runs turn by turn only in a cyclic queue for a limited time slice. This algorithm also offers starvation free execution of processes.

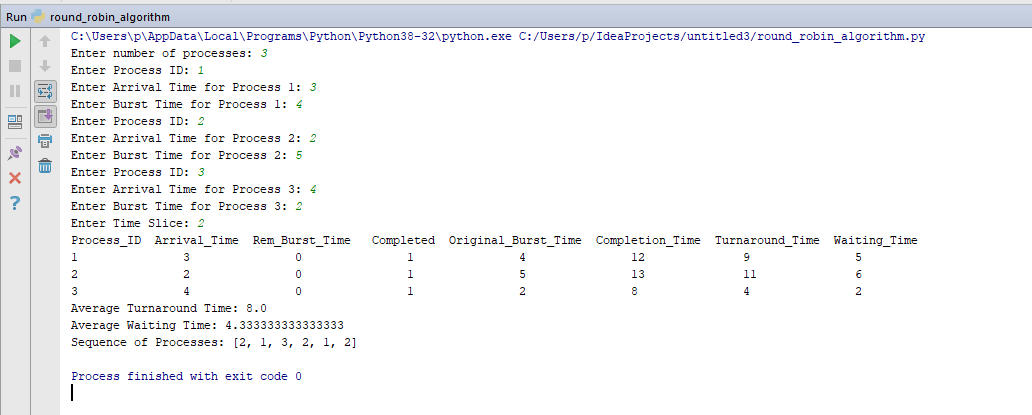


# Working Procedure:

Coding implementation with python ….

*''' Created by asik  
 date:03/09/2020  
'''***class** RoundRobin:  
  
 **def** processData(self, no\_of\_processes):  
 process\_data = []  
 **for** i **in** range(no\_of\_processes):  
 temporary = []  
 process\_id = int(input(**"Enter Process ID: "**))  
  
 arrival\_time = int(input(**f"Enter Arrival Time for Process {process\_id}: "**))  
  
 burst\_time = int(input(**f"Enter Burst Time for Process {process\_id}: "**))  
  
 temporary.extend([process\_id, arrival\_time, burst\_time, 0, burst\_time])  
 **'''  
 '0' is the state of the process. 0 means not executed and 1 means execution complete  
   
 '''** process\_data.append(temporary)  
  
 time\_slice = int(input(**"Enter Time Slice: "**))  
  
 RoundRobin.schedulingProcess(self, process\_data, time\_slice)  
  
 **def** schedulingProcess(self, process\_data, time\_slice):  
 start\_time = []  
 exit\_time = []  
 executed\_process = []  
 ready\_queue = []  
 s\_time = 0  
 process\_data.sort(key=**lambda** x: x[1])  
 **'''  
 Sort processes according to the Arrival Time  
 '''  
 while** 1:  
 normal\_queue = []  
 temp = []  
 **for** i **in** range(len(process\_data)):  
 **if** process\_data[i][1] <= s\_time **and** process\_data[i][3] == 0:  
 present = 0  
 **if** len(ready\_queue) != 0:  
 **for** k **in** range(len(ready\_queue)):  
 **if** process\_data[i][0] == ready\_queue[k][0]:  
 present = 1  
 **'''  
 The above if loop checks that the next process is not a part of ready\_queue  
 '''  
 if** present == 0:  
 temp.extend([process\_data[i][0], process\_data[i][1], process\_data[i][2], process\_data[i][4]])  
 ready\_queue.append(temp)  
 temp = []  
 **'''  
 The above if loop adds a process to the ready\_queue only if it is not already present in it  
 '''  
 if** len(ready\_queue) != 0 **and** len(executed\_process) != 0:  
 **for** k **in** range(len(ready\_queue)):  
 **if** ready\_queue[k][0] == executed\_process[len(executed\_process) - 1]:  
 ready\_queue.insert((len(ready\_queue) - 1), ready\_queue.pop(k))  
 **'''  
 The above if loop makes sure that the recently executed process is appended at the end of ready\_queue  
 '''  
 elif** process\_data[i][3] == 0:  
 temp.extend([process\_data[i][0], process\_data[i][1], process\_data[i][2], process\_data[i][4]])  
 normal\_queue.append(temp)  
 temp = []  
 **if** len(ready\_queue) == 0 **and** len(normal\_queue) == 0:  
 **break  
 if** len(ready\_queue) != 0:  
 **if** ready\_queue[0][2] > time\_slice:  
 **'''  
 If process has remaining burst time greater than the time slice, it will execute for a time period equal to time slice and then switch  
 '''** start\_time.append(s\_time)  
 s\_time = s\_time + time\_slice  
 e\_time = s\_time  
 exit\_time.append(e\_time)  
 executed\_process.append(ready\_queue[0][0])  
 **for** j **in** range(len(process\_data)):  
 **if** process\_data[j][0] == ready\_queue[0][0]:  
 **break** process\_data[j][2] = process\_data[j][2] - time\_slice  
 ready\_queue.pop(0)  
 **elif** ready\_queue[0][2] <= time\_slice:  
 **'''  
 If a process has a remaining burst time less than or equal to time slice, it will complete its execution  
 '''** start\_time.append(s\_time)  
 s\_time = s\_time + ready\_queue[0][2]  
 e\_time = s\_time  
 exit\_time.append(e\_time)  
 executed\_process.append(ready\_queue[0][0])  
 **for** j **in** range(len(process\_data)):  
 **if** process\_data[j][0] == ready\_queue[0][0]:  
 **break** process\_data[j][2] = 0  
 process\_data[j][3] = 1  
 process\_data[j].append(e\_time)  
 ready\_queue.pop(0)  
 **elif** len(ready\_queue) == 0:  
 **if** s\_time < normal\_queue[0][1]:  
 s\_time = normal\_queue[0][1]  
 **if** normal\_queue[0][2] > time\_slice:  
 **'''  
 If process has remaining burst time greater than the time slice, it will execute for a time period equal to time slice and then switch  
 '''** start\_time.append(s\_time)  
 s\_time = s\_time + time\_slice  
 e\_time = s\_time  
 exit\_time.append(e\_time)  
 executed\_process.append(normal\_queue[0][0])  
 **for** j **in** range(len(process\_data)):  
 **if** process\_data[j][0] == normal\_queue[0][0]:  
 **break** process\_data[j][2] = process\_data[j][2] - time\_slice  
 **elif** normal\_queue[0][2] <= time\_slice:  
 **'''  
 If a process has a remaining burst time less than or equal to time slice, it will complete its execution  
 '''** start\_time.append(s\_time)  
 s\_time = s\_time + normal\_queue[0][2]  
 e\_time = s\_time  
 exit\_time.append(e\_time)  
 executed\_process.append(normal\_queue[0][0])  
 **for** j **in** range(len(process\_data)):  
 **if** process\_data[j][0] == normal\_queue[0][0]:  
 **break** process\_data[j][2] = 0  
 process\_data[j][3] = 1  
 process\_data[j].append(e\_time)  
 t\_time = RoundRobin.calculateTurnaroundTime(self, process\_data)  
 w\_time = RoundRobin.calculateWaitingTime(self, process\_data)  
 RoundRobin.printData(self, process\_data, t\_time, w\_time, executed\_process)  
  
 **def** calculateTurnaroundTime(self, process\_data):  
 total\_turnaround\_time = 0  
 **for** i **in** range(len(process\_data)):  
 turnaround\_time = process\_data[i][5] - process\_data[i][1]  
 **'''  
 turnaround\_time = completion\_time - arrival\_time  
 '''** total\_turnaround\_time = total\_turnaround\_time + turnaround\_time  
 process\_data[i].append(turnaround\_time)  
 average\_turnaround\_time = total\_turnaround\_time / len(process\_data)  
 **'''  
 average\_turnaround\_time = total\_turnaround\_time / no\_of\_processes  
 '''  
 return** average\_turnaround\_time  
  
 **def** calculateWaitingTime(self, process\_data):  
 total\_waiting\_time = 0  
 **for** i **in** range(len(process\_data)):  
 waiting\_time = process\_data[i][6] - process\_data[i][4]  
 **'''  
 waiting\_time = turnaround\_time - burst\_time  
 '''** total\_waiting\_time = total\_waiting\_time + waiting\_time  
 process\_data[i].append(waiting\_time)  
 average\_waiting\_time = total\_waiting\_time / len(process\_data)  
 **'''  
 average\_waiting\_time = total\_waiting\_time / no\_of\_processes  
 '''  
 return** average\_waiting\_time  
  
 **def** printData(self, process\_data, average\_turnaround\_time, average\_waiting\_time, executed\_process):  
 process\_data.sort(key=**lambda** x: x[0])  
 **'''  
 Sort processes according to the Process ID  
 '''** print(**"Process\_ID Arrival\_Time Rem\_Burst\_Time Completed Original\_Burst\_Time Completion\_Time Turnaround\_Time Waiting\_Time"**)  
 **for** i **in** range(len(process\_data)):  
 **for** j **in** range(len(process\_data[i])):  
  
 print(process\_data[i][j], end=**" "**)  
 print()  
  
 print(**f'Average Turnaround Time: {average\_turnaround\_time}'**)  
  
 print(**f'Average Waiting Time: {average\_waiting\_time}'**)  
  
 print(**f'Sequence of Processes: {executed\_process}'**)  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 no\_of\_processes = int(input(**"Enter number of processes: "**))  
 rr = RoundRobin()  
 rr.processData(no\_of\_processes)

# Output:



# Discussion:

We learn characteristic and advantages of round robin algorithm

1.Round robin is a pre-emptive algorithm

2.The CPU is shifted to the next process after fixed interval time, which is called time quantum/time slice.

3.The process that is preempted is added to the end of the queue.

4.Round robin is a hybrid model which is clock-driven

5.Time slice should be minimum, which is assigned for a specific task that needs to be processed. However, it may differ OS to OS.

6.It is a real time algorithm which responds to the event within a specific time limit.

7.Round robin is one of the oldest, fairest, and easiest algorithm.

8.Widely used scheduling method in traditional OS.

9.It doesn't face the issues of starvation or convoy effect.

10.All the jobs get a fair allocation of CPU.

11.It deals with all process without any priority