K. J. Somaiya College of Engineering, Mumbai-77
(A Constituent College of Somaiya Vidyavihar University)
Department of Computer Engineering

Batch: C2	**Roll No.:** 16010121110
Experiment No. 04	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

TITLE: Implementation of Basic Process management algorithms - Preemptive (SRTN, RR, priority)

AIM: To implement basic Process management algorithms (Round Robin,SRTN, Priority)

Expected Outcome of Experiment:

CO 2. To understand the concept of process, thread and resource management.

Books/ Journals/ Websites referred:

1. **Silberschatz A., Galvin P., Gagne G. "Operating Systems Principles", Willey Eight edition.**
2. **Achyut S. Godbole , Atul Kahate "Operating Systems" McGraw Hill Third Edition.**
3. **William Stallings, "Operating System Internal & Design Principles", Pearson.**
4. **Andrew S. Tanenbaum, "Modern Operating System", Prentice Hall.**

Pre Lab/ Prior Concepts:

Most systems have a large number of processes with short CPU bursts interspersed between I/O requests and a small number of processes with long CPU bursts. To provide good time-sharing performance, we may preempt a running process to let another one run. The ready list, also known as a run queue, in the operating system keeps a list of all processes that are ready to run and not blocked on some I/O or other system request, such as a semaphore. Then entries in this list are pointers to the process control block, which stores all information and state about a process.

When an I/O request for a process is complete, the process moves from the *waiting* state to the *ready* state and gets placed on the run queue.

The process scheduler is the component of the operating system that is responsible for deciding whether the currently running process should continue running and, if not, which process should run next. There are four events that may occur where the scheduler needs to step in and make this decision:

1. The current process goes from the *running* to the *waiting* state because it issues an I/O request or some operating system request that cannot be satisfied immediately.

2. The current process terminates.

3. A timer interrupt causes the scheduler to run and decide that a process has run for its allotted interval of time and it is time to move it from the *running* to the *ready* state.

4. An I/O operation is complete for a process that requested it and the process now moves from the *waiting* to the*ready* state. The scheduler may then decide to preempt the currently-running process and move this *ready* process into the *running* state.

The decisions that the scheduler makes concerning the sequence and length of time that processes may run is called the scheduling algorithm (or scheduling policy). These decisions are not easy ones, as the scheduler has only a limited amount of information about the processes that are ready to run. A good scheduling algorithm should:

1. Be fair – give each process a fair share of the CPU, allow each process to run in a reasonable amount of time.
2. Be efficient – keep the CPU busy all the time.

3. Maximize throughput – service the largest possible number of jobs in a given amount of time; minimize the amount of time users must wait for their results.
4. Minimize response time – interactive users should see good performance
5. Minimize overhead – don't waste too many resources. Keep scheduling time and context switch time at a minimum.
6. Maximize resource use – favor processes that will use underutilized resources. There are two motives for this. Most devices are slow compared to CPU operations. We'll achieve better system throughput by keeping devices busy as often as possible. The second reason is that a process may be holding a key resource and other, possibly more important, processes cannot use it until it is released. Giving the process more CPU time may free up the resource quicker.
7. Avoid indefinite postponement – every process should get a chance to run eventually.

Description of the application to be implemented:

Round Robin Algorithm

Round Robin is a CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way. It is basically the preemptive version of First come First Serve CPU Scheduling algorithm.

1) Round Robin CPU Algorithm generally focuses on Time Sharing technique.
2) The period of time for which a process or job is allowed to run in a preemptive method is called time quantum.
3) Each process or job present in the ready queue is assigned the CPU for that time quantum, if the execution of the process is completed during that time then the process will end else the process will go back to the waiting table and wait for its next turn to complete the execution.

Shortest Remaining Time First Algorithm :

This Algorithm is the preemptive version of SJF scheduling. In SRTF, the execution of the process can be stopped after certain amount of time. At the arrival of every process, the short term scheduler schedules the process with the least remaining burst time among the list of available processes and the running process.

Once all the processes are available in the ready queue, No preemption will be done and the algorithm will work as SJF scheduling. The context of the process is saved in the Process Control Block when the process is removed from the execution and the next process is scheduled. This PCB is accessed on the next execution of this process.

Priority scheduling:

This Algorithm is the preemptive version of SJF scheduling. In SRTF, the execution of the process can be stopped after certain amount of time. At the arrival of every process, the short term scheduler schedules the process with the least remaining burst time among the list of available processes and the running process. Once all the processes are available in the ready queue, No preemption will be done and the algorithm will work as SJF scheduling. The context of the process is saved in the Process Control Block when the process is removed from the execution and the next process is scheduled. This PCB is accessed on the next execution of this process.

Implementation details:

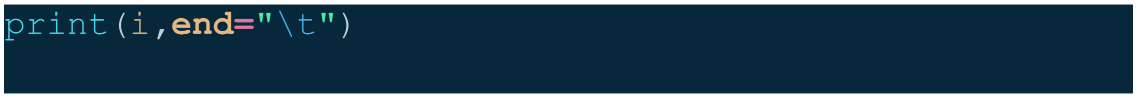
Round Robin

```
k=int(input("Please enter number of processes "))

arrivalTime=[int(input("Please enter arrival time for
process "+str(i)+" ")) for i in range(k)]

reqTime=[int(input("Please enter estimated time for
process "+str(i)+" ")) for i in range(k)]
```

```
timeQuantaDecided =int(input("Please enter the time quanta "))


'''
#Test case

k=5
arrivalTime=[0,2,4,6,7]
reqTime=[2,3,4,5,6]
timeQuantaDecided =2
'''
processes=range(k)
timeQuanta = timeQuantaDecided
reqTimeCopy=[i for i in reqTime]
queue=[]
answer=[]
waiting=[0 for i in range(k)]


for i in range(100):
```

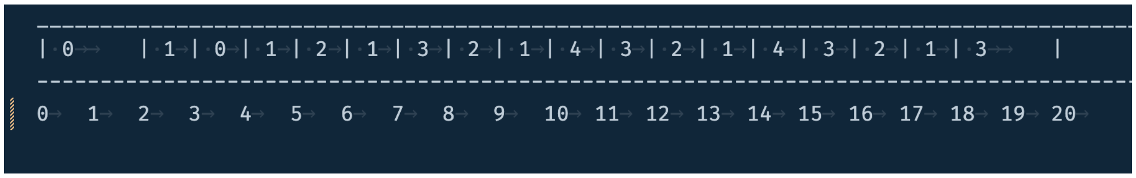
```
for process in processes: #check new processes comming

if(arrivalTime[process]==i):

queue.append(process)

if(reqTime[queue[0]]==0):# if process still over next

timeQuanta= timeQuantaDecided

queue.pop(0)

elif(timeQuanta==0):

timeQuanta= timeQuantaDecided

queue.append(queue[0])

queue.pop(0)

if(len(queue)==0 and i!=0):

break

currentProcess=queue[0]

print(i,"\t", currentProcess,"\t\t\t",queue)

for waitingProcess in processes:

# if process arrived and waiting and present in
queue(not terminated)
```

```
if(waitingProcess!=currentProcess and
arrivalTime[waitingProcess]<=i and waitingProcess in
queue):

waiting[waitingProcess]+=1

answer.append(currentProcess)

reqTime[currentProcess]-=1

timeQuanta-=1

print("Total time spent waiting for processes is given
by", waiting)

print("Average waiting
Time",sum(waiting)/len(waiting))

turnaroundTimeArray=[int(waiting[i])+int(reqTimeCopy[i
]) for i in processes]

print("Total turnaround time for processes is given
by",turnaroundTimeArray )

print("Average turnaround
Time",sum(turnaroundTimeArray)/len(turnaroundTimeArray
))

print("\n")

print("\n______________________________________________
__________________________________________________________
______________________________________________________
________")



temp=-1

for i in range(0,len(answer)):

if(i==0):

print("|",answer[i],end="\t")

temp=answer[i]

else:

if(temp==answer[i]):

print("", end="\t")

continue

else:

print("|",answer[i],end="\t")

temp=answer[i]

print("|\n--------------------------------------------
------------------------------------------------------
------------------------------------------------------
---------")

for i in range(0,len(answer)+1):
```



```
print(i,end="\t")
```

OUTPUT



```
-------------------------------------------------------------------------------------------
| 0       | 1 | 0 | 1 | 2 | 1 | 3 | 2 | 1 | 4 | 3 | 2 | 1 | 4 | 3 | 2 | 1 | 3        |
-------------------------------------------------------------------------------------------
0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20
```

SRTF shortest remaining task first

```
'''

k=int(input("Please enter number of processes "))


arrivalTime=[int(input("Please enter arrival time for
process "+str(i)+" ")) for i in range(k)]


reqTime=[int(input("Please enter estimated time for
process "+str(i)+" ")) for i in range(k)]



timeQuantaDecided =int(input("Please enter the time
quanta "))


'''

#Test case


k=5

arrivalTime=[0,2,4,6,7]

reqTime=[12,1,3, 4, 9]
```

```
timeQuantaDecided =1


processes=range(k)

timeQuanta = timeQuantaDecided

reqTimeCopy=[i for i in reqTime]

queue=[]

answer=[]

waiting=[0 for i in range(k)]


for i in range(100):

for process in processes: #check new processes comming

if(arrivalTime[process]==i):

queue.append(process)

if(reqTime[queue[0]]==0):# if process still over next

timeQuanta= timeQuantaDecided

queue.pop(0)

elif(timeQuanta==0):

timeQuanta= timeQuantaDecided
```

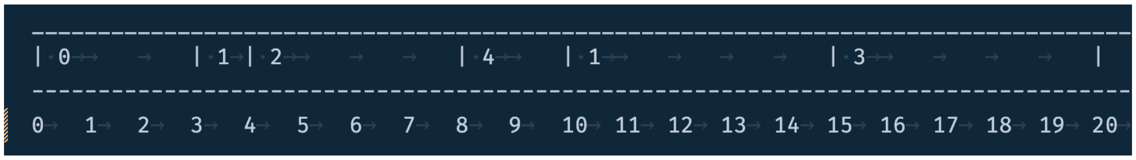
```
queue.append(queue[0])

queue.pop(0)

if(len(queue)==0 and i!=0):

break



currentProcess = reqTime.index(min([reqTime[process]
for process in range(len(processes)) if process in
queue]))

queue.remove(currentProcess)

queue.insert(0,currentProcess)

#if a process has arrived then check lowest time and
put that.

print(i,"\t", currentProcess,"\t\t\t",queue)

for waitingProcess in processes:

# if process arrived and waiting and present in
queue(not terminated)

if(waitingProcess!=currentProcess and
arrivalTime[waitingProcess]<=i and waitingProcess in
queue):

waiting[waitingProcess]+=1
```

```
answer.append(currentProcess)

reqTime[currentProcess]-=1

timeQuanta-=1

print("Total time spent waiting for processes is given by", waiting)

print("Average waiting Time",sum(waiting)/len(waiting))

turnaroundTimeArray=[int(waiting[i])+int(reqTimeCopy[i]) for i in processes]

print("Total turnaround time for processes is given by",turnaroundTimeArray )

print("Average turnaround Time",sum(turnaroundTimeArray)/len(turnaroundTimeArray))

print("\n")

print("\n_________________________________________________________________________________________________________________________________________________________________")



temp=-1
```

```
for i in range(0,len(answer)):

if(i==0):

print("|",answer[i],end="\t")

temp=answer[i]

else:

if(temp==answer[i]):

print("", end="\t")

continue

else:

print("|",answer[i],end="\t")

temp=answer[i]

print("|\n--------------------------------------------
------------------------------------------------------
------------------------------------------------------
---------")

for i in range(0,len(answer)+1):

print(i,end="\t")
```

OUTPUT

K. J. Somaiya College of Engineering, Mumbai-77
(A Constituent College of Somaiya Vidyavihar University)
Department of Computer Engineering



```
----------------------------------------------------------------------------------------------------
| 0           | 1 | 2               | 4       | 1                   | 3                   |
----------------------------------------------------------------------------------------------------
0    1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16   17   18   19   20
```

Priority scheduling

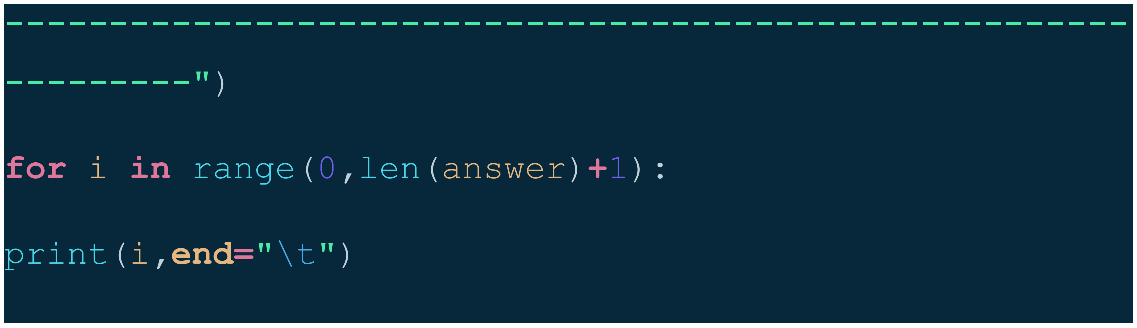
```
'''

Priority premptive

k=int(input("Please enter number of processes "))



arrivalTime=[int(input("Please enter arrival time for
process "+str(i)+" ")) for i in range(k)]



reqTime=[int(input("Please enter estimated time for
process "+str(i)+" ")) for i in range(k)]





timeQuantaDecided =int(input("Please enter the time
quanta "))

priority=[int(input("Please enter priority for process
"+str(i)+" ")) for i in range(k)]





'''

#Test case
```

```
k=5

arrivalTime=[0,2,4,6,7]

reqTime=[12,1,3, 4, 9]

priority=[3,2,1,4,5]


timeQuantaDecided =1


processes=range(k)

timeQuanta = timeQuantaDecided

reqTimeCopy=[i for i in reqTime]

queue=[]

answer=[]

waiting=[0 for i in range(k)]


for i in range(100):

for process in processes: #check new processes comming

if(arrivalTime[process]==i):
```

```
queue.append(process)

if(reqTime[queue[0]]==0):# if process still over next

timeQuanta= timeQuantaDecided

queue.pop(0)

elif(timeQuanta==0):

timeQuanta= timeQuantaDecided

queue.append(queue[0])

queue.pop(0)

if(len(queue)==0 and i!=0):

break



currentProcess = priority.index(min([priority[process]
for process in range(len(processes)) if process in
queue]))

queue.remove(currentProcess)

queue.insert(0,currentProcess)

#if a process has arrived then check lowest time and
put that.

print(i,"\t", currentProcess,"\t\t\t",queue)
```

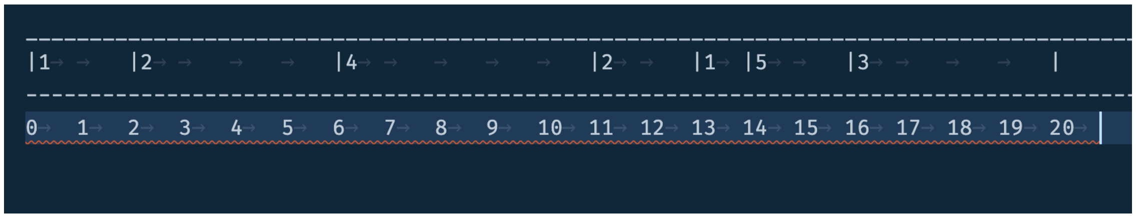
```
for waitingProcess in processes:

# if process arrived and waiting and present in
queue(not terminated)

if(waitingProcess!=currentProcess and
arrivalTime[waitingProcess]<=i and waitingProcess in
queue):

waiting[waitingProcess]+=1

answer.append(currentProcess)

reqTime[currentProcess]-=1

timeQuanta-=1

print("Total time spent waiting for processes is given
by", waiting)

print("Average waiting
Time",sum(waiting)/len(waiting))

turnaroundTimeArray=[int(waiting[i])+int(reqTimeCopy[i
]) for i in processes]

print("Total turnaround time for processes is given
by",turnaroundTimeArray )

print("Average turnaround
Time",sum(turnaroundTimeArray)/len(turnaroundTimeArray
))
```

```
print("\n")

print("\n________________________________________________________________________________________________________________________________________________________________")


temp=-1

for i in range(0,len(answer)):

if(i==0):

print("|",answer[i],end="\t")

temp=answer[i]

else:

if(temp==answer[i]):

print("", end="\t")

continue

else:

print("|",answer[i],end="\t")

temp=answer[i]

print("|\n-------------------------------------------------------------------------------------------------------------
```



```
---------------------------------------------------------------
----------")
for i in range(0,len(answer)+1):
print(i,end="\t")
```

<u>OUTPUT of priority</u>



```
----------------------------------------------------------------------------------
|1      |2                      |4                              |2      |1  |5      |3                      |
----------------------------------------------------------------------------------
0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20
```

<u>Conclusion:</u> Thus we have implemented preemptive algorithms. These algorithms give a chance for every process to run. This is why the processes get less starved. Every process gets a chance to execute and no single process is kept running for a long amount of time.

<u>Post Lab Descriptive Questions</u>

1. Consider three processes, all arriving at time zero, with total execution time of 10, 20 and 30 units, respectively. Each process spends the first 20% of execution time doing I/O, the next 70% of time doing computation, and the last 10% of time doing I/O again. The operating system uses a shortest remaining compute time first scheduling algorithm and schedules a new process either when the running process gets blocked on I/O or when the running process finishes its compute burst. Assume that all I/O operations can be overlapped as much as possible. For what percentage of time does the CPU remain idle?

Ans - Only 2/60 = 3.33% time is spent idle considering efficient scheduling

2. What is Starvation?

Starvation is when a process does not get chance to be executed and is completely ignored. this can occur in non preemptive algorithms when a process requiring high bust time is scheduled for execution. This can also occur in priority based scheduling when higher priority processes are scheduled and lower priority process dont get time to execute.