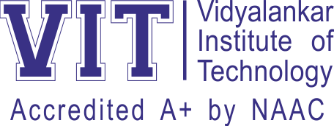
J

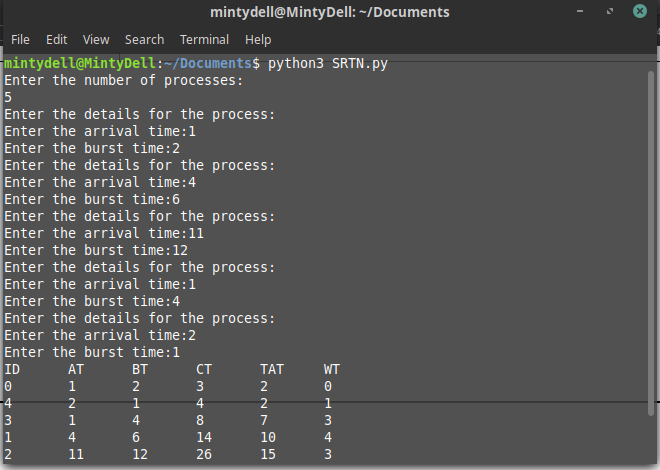
 ***DEPARTMENT OF COMPUTER ENGINEERING***

5

Experiment No.

|  |  |
| --- | --- |
| Semester | S.E. Semester IV – Computer Engineering |
| Subject | Operating System |
| Subject Professor In-charge | SNA |
| Assisting Teachers | Ms. Rasika Ransing |
| Laboratory | M310B – Computer Engineering Laboratory |

|  |  |  |
| --- | --- | --- |
| Student Name | Amir Shaikh | |
| Roll Number | 18102A0053 | |
| Grade and Subject Teacher’s Signature |  |  |



|  |  |  |
| --- | --- | --- |
| Experiment Number | 5 | |
| Experiment Title | To implement Pre-emptive Scheduling (SRTN, RR) | |
| Resources / Apparatus Required | Hardware: Computer | Software: Linux Ubuntu Terminal |
| Objectives  (Skill Set / Knowledge Tested / Imparted) | Scheduling algorithms | |
| Theory | **SRTN:**  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.  **RR:**  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. | |
| Code | **SRTN:**  import operator  class process:  def \_\_init\_\_(self, ppid, arrival\_time, burst\_time):  self.ppid = ppid  self.arrival\_time = arrival\_time  self.burst\_time = burst\_time  self.remaining\_burst\_time = burst\_time  self.wait\_time = 0  self.turn\_around\_time = 0  self.completion\_time = 0  class simpleQueue:  def \_\_init\_\_(self):  self.ProcessQueue = []  def enqueue(self, process):  self.ProcessQueue.append(process)  def isEmpty(self):  return self.ProcessQueue == []  def dequeue(self):  return self.ProcessQueue.pop(0)  def queueSort(self):  self.ProcessQueue = sorted(self.ProcessQueue, key=operator.attrgetter(  'remaining\_burst\_time', 'ppid'))  def peek(self):  return self.ProcessQueue[0]  def queueSortSRTN(self):  self.ProcessQueue = sorted(  self.ProcessQueue, key=operator.attrgetter('remaining\_burst\_time', 'ppid'))  def acceptProcess(process\_list):  no\_of\_processes = int(input("Enter the number of processes:\n"))  for i in range(no\_of\_processes):  print("Enter the details for the process:")  arrival\_time = int(input("Enter the arrival time:"))  burst\_time = int(input("Enter the burst time:"))  new\_process = process(i, arrival\_time, burst\_time)  process\_list.append(new\_process)  def SRTN(process\_list):  n = len(process\_list)  process\_list = sorted(process\_list, key=operator.attrgetter(  'arrival\_time', 'burst\_time', 'ppid'))  processor\_time = 0  completed\_processes = 0  processor\_queue = simpleQueue()  print("ID\tAT\tBT\tCT\tTAT\tWT")  while(completed\_processes <= n):  while(process\_list != list() and processor\_time >= process\_list[0].arrival\_time):  x = process\_list.pop(0)  processor\_queue.enqueue(x)  processor\_queue.queueSortSRTN()  if(not processor\_queue.isEmpty()):  running\_process = processor\_queue.dequeue()  if running\_process.remaining\_burst\_time == 1:  running\_process.remaining\_burst\_time -= 1  running\_process.wait\_time = processor\_time - running\_process.arrival\_time  processor\_time += running\_process.burst\_time  running\_process.completion\_time = processor\_time  running\_process.turn\_around\_time = processor\_time - running\_process.arrival\_time  completed\_processes += 1  print(f"{running\_process.ppid}\t{running\_process.arrival\_time}\t{running\_process.burst\_time}\t{running\_process.completion\_time}\t{running\_process.turn\_around\_time} \t{running\_process.wait\_time}")  else:  running\_process.remaining\_burst\_time -= 1  processor\_queue.enqueue(running\_process)  processor\_queue.queueSortSRTN()  else:  processor\_time += 1  """  while(1):  while(process\_list != list() and processor\_time >= process\_list[0].arrival\_time):  x = process\_list.pop(0)  processor\_queue.enqueue(x)  processor\_queue.queueSortSRTN()  if(not processor\_queue.isEmpty()):  running\_process = processor\_queue.dequeue()  running\_process.wait\_time = processor\_time - running\_process.arrival\_time  processor\_time += running\_process.burst\_time  running\_process.completion\_time = processor\_time  running\_process.turn\_around\_time = processor\_time - running\_process.arrival\_time  print(f"{running\_process.ppid}\t{running\_process.arrival\_time}\t{running\_process.burst\_time}\t{running\_process.completion\_time}\t{running\_process.turn\_around\_time} \t{running\_process.wait\_time}")  else:  processor\_time += 1  if(process\_list == [] and processor\_queue.isEmpty()):  break  """  process\_list = []  acceptProcess(process\_list)  SRTN(process\_list)  **RR:**  #include <stdio.h>  #include <stdlib.h>  typedef struct  {  int at, bt, ct, wt, tat, remaining\_bt, id;  } process;  typedef struct  {  process q[100];  int front, rear;  } queue;  void enqueue(queue \*process\_queue, process ele)  {  process\_queue->rear++;  process\_queue->q[process\_queue->rear] = ele;  }  int isempty(queue \*process\_queue)  {  if (process\_queue->rear + 1 == process\_queue->front)  return 1;  else  return 0;  }  process dequeue(queue \*process\_queue)  {  process x;  if (isempty(process\_queue))  printf("empty");  x = process\_queue->q[process\_queue->front];  process\_queue->front++;  return x;  }  void acceptProcess(process process\_list[], int n)  {  int i;  for (i = 0; i < n; i++)  {  printf("\nEnter the arrival time for PPID-%d: ", i);  scanf("%d", &process\_list[i].at);  printf("\nEnter the burst time for PPID-%d: ", i);  scanf("%d", &process\_list[i].bt);  process\_list[i].remaining\_bt = process\_list[i].bt;  process\_list[i].id = i;  }  }  void roundRobin(process process\_list[], int n)  {  queue process\_queue;  process temp;  process\_queue.front = 0;  process\_queue.rear = -1;  int completed\_execution, processor\_time = 0, i = 0, quantum;  printf("Enter the time quantum:\n");  scanf("%d", &quantum);  printf("PID\tAT\tBT\tCT\tTAT\tWT\n");  while (process\_list[i].at <= processor\_time && i < n)  {  enqueue(&process\_queue, process\_list[i]);  i++;  }  while (completed\_execution != n)  {  if (!isempty(&process\_queue))  {  temp = dequeue(&process\_queue);  if (temp.remaining\_bt > quantum)  {  temp.remaining\_bt -= quantum;  processor\_time += quantum;  while (process\_list[i].at <= processor\_time && i < n)  {  enqueue(&process\_queue, process\_list[i]);  i++;  }  enqueue(&process\_queue, temp);  }  else  {  processor\_time += temp.remaining\_bt;  temp.remaining\_bt = 0;  temp.ct = processor\_time;  temp.tat = temp.ct - temp.at;  temp.wt = temp.tat - temp.bt;  completed\_execution++;  printf("%d\t%d\t%d\t%d\t%d\t%d\t\n", temp.id, temp.at, temp.bt, temp.ct, temp.tat, temp.wt);  while (process\_list[i].at <= processor\_time && i < n)  {  enqueue(&process\_queue, process\_list[i]);  i++;  }  }  }  else  {  processor\_time++;  while (process\_list[i].at <= processor\_time && i < n)  {  enqueue(&process\_queue, process\_list[i]);  i++;  }  }  }  }  int main()  {  int n;  process process\_list[100];  printf("Enter the number of processes you want to enter:\n");  scanf("%d", &n);  acceptProcess(process\_list, n);  roundRobin(process\_list, n);  return 0;  } | |
| Output |  | |
| Conclusion | Thus, we have successful implemented pre-emptive scheduling algorithm | |