 ***DEPARTMENT OF COMPUTER ENGINEERING***

4

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 | Chinmay Tiwari | |
| Roll Number | 18102A0066 | |
| Grade and Subject Teacher’s Signature |  |  |

|  |  |  |
| --- | --- | --- |
| Experiment Number | 4 | |
| Experiment Title | To implement Non-Pre-emptive Scheduling (FCFS, SJF) | |
| Resources / Apparatus Required | Hardware: Computer | Software: Linux Ubuntu Terminal |
| Objectives  (Skill Set / Knowledge Tested / Imparted) | Scheduling algorithms | |
| Theory | **FCFS:**  **First Come First Serve (FCFS)** is an operating system scheduling algorithm that automatically executes queued requests and processes in order of their arrival. It is the easiest and simplest CPU scheduling algorithm. In this type of algorithm, processes which requests the CPU first get the CPU allocation first. This is managed with a FIFO queue. The full form of FCFS is First Come First Serve.  As the process enters the ready queue, its PCB (Process Control Block) is linked with the tail of the queue and, when the CPU becomes free, it should be assigned to the process at the beginning of the queue.  **SJF:`**  Shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN is a non-preemptive algorithm.   * Shortest Job first has the advantage of having a minimum average waiting time among all scheduling algorithms. * It is a Greedy Algorithm. * It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of aging. * It is practically infeasible as Operating System may not know burst time and therefore may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times. SJF can be used in specialized environments where accurate estimates of running time are available. | |
| Code | **FCFS:**  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.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 == list()  def dequeue(self):  return self.ProcessQueue.pop(0)  def queueSort(self):  self.ProcessQueue = sorted(self.ProcessQueue, key=operator.attrgetter('arrival\_time', 'ppid'))  def acceptProcess(queue):  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)  queue.enqueue(new\_process)  queue.queueSort()  def FCFS(queue, processor\_time):  x = queue.dequeue()  print("ID\tAT\tBT\tCT\tTAT\tWT")  while(True):  if (x.arrival\_time > processor\_time):  processor\_time += 1  continue  else:  x.wait\_time = processor\_time - x.arrival\_time  processor\_time += x.burst\_time  x.completion\_time = processor\_time  x.turn\_around\_time = processor\_time - x.arrival\_time  print(  f"{x.ppid+1}\t{x.arrival\_time}\t{x.burst\_time}\t{x.completion\_time}\t{x.turn\_around\_time} \t{x.wait\_time}")  if(queue.isEmpty()):  break  x = queue.dequeue()  queue = simpleQueue()  acceptProcess(queue)  processor\_time = 0  FCFS(queue, processor\_time)  **SJF:**  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.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(  'arrival\_time', 'burst\_time', 'ppid'))  def peek(self):  return self.ProcessQueue[0]  def queueSortSJF(self):  self.ProcessQueue = sorted(  self.ProcessQueue, key=operator.attrgetter('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 SJF(process\_list):  process\_list = sorted(process\_list, key=operator.attrgetter(  'arrival\_time', 'burst\_time', 'ppid'))  processor\_time = 0  processor\_queue = simpleQueue()  print("ID\tAT\tBT\tCT\tTAT\tWT")  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.queueSortSJF()  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)  SJF(process\_list) | |
| Output | **FCFS:**    **SJF:** | |
| Conclusion | Thus, we have successfully implemented non-pre-emptive scheduling algorithm | |