|  |  |  |  |
| --- | --- | --- | --- |
| **Course Name:** | **Operating Systems and Compilers** | **Semester:** | **VI** |
| **Date of Performance:** | **21 / 03 / 2025** | **Batch No.:** | **B - 2** |
| **Faculty Name:** | **Prof. Nilesh Lakade** | **Roll No.:** | **16014022050** |
| **Faculty Sign & Date:** |  | **Grade/Marks:** | **\_\_\_ / 25** |

**Experiment No.: 4**

**Title:** **Implementation of Basic Process management algorithms – Pre emptive (SRTN, RR)**

|  |
| --- |
| **Aim and Objective of the Experiment:** |
| To implement basic Process management algorithms (Round Robin, SRTN). |

|  |
| --- |
| **COs to be achieved:** |
| **CO2:** Describe the problems related to process concurrency and the different synchronization mechanisms available to solve them. |

|  |
| --- |
| **Theory:** |
| 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 the queue, in the operating system keeps a list of all processes that are ready to run and are not blocked on some I/O or other system request, such as a semaphore. The entries in this list are pointers to the process control blocks (PCBs), which store all information and the 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 where the scheduler needs to step in and make this decision:   1. The current process moves 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 completes 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 are 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: i. Be fair – give each process a fair share of the CPU and allow each process to run within a reasonable amount of time.   1. Be efficient – keep the CPU busy all the time. 2. Maximize throughput – service the largest possible number of jobs in a given amount of time and minimize the amount of time users must wait for their results. 3. Minimize response time – interactive users should experience good performance. 4. Minimize overhead – avoid wasting resources by keeping scheduling time and context-switch time at a minimum. 5. Maximize resource use – favor processes that will utilize underutilized resources. There are two motives for this:    * Most devices are slow compared to CPU operations, so keeping devices busy will improve system throughput.    * A process may be holding a key resource that other (possibly more important) processes need, and giving it more CPU time may free up that resource more quickly. 6. Avoid indefinite postponement – ensure that every process eventually gets a chance to run. |
| **Implementation details:** |
| **Round Robin Algorithm:**  #include <iostream>  #include <queue>  #include <vector>  #include <algorithm>  #include <iomanip>  using namespace std;  struct Process {      string id;      int arrival\_time, burst\_time, remaining\_time, waiting\_time, turnaround\_time, completion\_time;  };  bool compareArrival(Process a, Process b) {      return a.arrival\_time < b.arrival\_time;  }  void roundRobinScheduling(vector<Process>& processes, int time\_quantum) {      int n = processes.size();      queue<Process\*> q;      int current\_time = 0, completed = 0;      int total\_waiting\_time = 0, total\_turnaround\_time = 0;        sort(processes.begin(), processes.end(), compareArrival);        int index = 0;      q.push(&processes[index++]);      while (completed < n) {          if (q.empty()) {              current\_time = processes[index].arrival\_time;              q.push(&processes[index++]);          }          Process\* p = q.front();          q.pop();          int execution\_time = min(p->remaining\_time, time\_quantum);          current\_time += execution\_time;          p->remaining\_time -= execution\_time;          while (index < n && processes[index].arrival\_time <= current\_time) {              q.push(&processes[index++]);          }          if (p->remaining\_time == 0) {              p->completion\_time = current\_time;              p->turnaround\_time = p->completion\_time - p->arrival\_time;              p->waiting\_time = p->turnaround\_time - p->burst\_time;              total\_waiting\_time += p->waiting\_time;              total\_turnaround\_time += p->turnaround\_time;              completed++;              cout << "Process " << p->id << " completed at time " << p->completion\_time << endl;          } else {              q.push(p);          }      }      cout << "\n------------------------------------------------------------\n";      cout << "| Process | AT  | BT  | CT  | WT  | TAT |\n";      cout << "------------------------------------------------------------\n";      for (auto &p : processes) {          cout << "| " << setw(7) << p.id << " | "               << setw(3) << p.arrival\_time << " | "               << setw(3) << p.burst\_time << " | "               << setw(3) << p.completion\_time << " | "               << setw(3) << p.waiting\_time << " | "               << setw(3) << p.turnaround\_time << " |\n";      }      cout << "------------------------------------------------------------\n";      cout << "\nAverage Waiting Time: " << fixed << setprecision(2) << (float)total\_waiting\_time / n << endl;      cout << "Average Turnaround Time: " << fixed << setprecision(2) << (float)total\_turnaround\_time / n << endl;  }  int main() {      int n, time\_quantum;      cout << "Enter number of processes: ";      cin >> n;        vector<Process> processes(n);        cout << "\nEnter process details (ID, Arrival Time, Burst Time):\n";      for (int i = 0; i < n; i++) {          cout << "Process " << (i + 1) << ": ";          cin >> processes[i].id >> processes[i].arrival\_time >> processes[i].burst\_time;          processes[i].remaining\_time = processes[i].burst\_time;      }        cout << "\nEnter Time Quantum: ";      cin >> time\_quantum;      roundRobinScheduling(processes, time\_quantum);      return 0;  }    **Shortest Remaining Time First Algorithm:**  #include <bits/stdc++.h>  using namespace std;  struct Process {      string id;      int arrival\_time, burst\_time, remaining\_time, waiting\_time, turnaround\_time, completion\_time, start\_time;      bool is\_completed;  };  bool compareArrival(Process a, Process b) {      return a.arrival\_time < b.arrival\_time;  }  void srtfScheduling(vector<Process>& processes) {      int n = processes.size();      int current\_time = 0, completed = 0;      int total\_waiting\_time = 0, total\_turnaround\_time = 0;      for (auto &p : processes) {          p.remaining\_time = p.burst\_time;          p.is\_completed = false;          p.start\_time = -1;      }      while (completed < n) {          int idx = -1, min\_remaining\_time = INT\_MAX;          for (int i = 0; i < n; i++) {              if (!processes[i].is\_completed && processes[i].arrival\_time <= current\_time && processes[i].remaining\_time < min\_remaining\_time) {                  min\_remaining\_time = processes[i].remaining\_time;                  idx = i;              }          }          if (idx == -1) {              current\_time++;              continue;          }          Process &p = processes[idx];          if (p.start\_time == -1) {              p.start\_time = current\_time;          }          p.remaining\_time--;          current\_time++;          if (p.remaining\_time == 0) {              p.completion\_time = current\_time;              p.turnaround\_time = p.completion\_time - p.arrival\_time;              p.waiting\_time = p.turnaround\_time - p.burst\_time;              p.is\_completed = true;              total\_waiting\_time += p.waiting\_time;              total\_turnaround\_time += p.turnaround\_time;              completed++;              cout << "Process " << p.id << " completed at time " << p.completion\_time << endl;          }      }      cout << "\n------------------------------------------------------------\n";      cout << "| Process | AT  | BT  | CT  | WT  | TAT |\n";      cout << "------------------------------------------------------------\n";      for (auto &p : processes) {          cout << "| " << setw(7) << p.id << " | "               << setw(3) << p.arrival\_time << " | "               << setw(3) << p.burst\_time << " | "               << setw(3) << p.completion\_time << " | "               << setw(3) << p.waiting\_time << " | "               << setw(3) << p.turnaround\_time << " |\n";      }      cout << "------------------------------------------------------------\n";      cout << "\nAverage Waiting Time: " << fixed << setprecision(2) << (float)total\_waiting\_time / n << endl;      cout << "Average Turnaround Time: " << fixed << setprecision(2) << (float)total\_turnaround\_time / n << endl;  }  int main() {      int n;      cout << "Enter number of processes: ";      cin >> n;        vector<Process> processes(n);        cout << "\nEnter process details (ID, Arrival Time, Burst Time):\n";      for (int i = 0; i < n; i++) {          cout << "Process " << (i + 1) << ": ";          cin >> processes[i].id >> processes[i].arrival\_time >> processes[i].burst\_time;      }      srtfScheduling(processes);      return 0;  } |

|  |
| --- |
| **Post Lab Subjective/Objective type Questions:** |
| 1. **What is Starvation.**   Starvation is a condition in process scheduling where a process waits indefinitely and never gets CPU time to execute. This typically happens when higher-priority processes keep getting selected by the scheduler, and lower-priority processes are left waiting. Over time, the lower-priority process may never get a chance to run. One common solution to prevent starvation is aging, which increases the priority of a waiting process the longer it stays in the queue.   1. **Compare Round robin and SRTF algorithm.** |

|  |
| --- |
| **Conclusion:** |
| The implementation of the Round Robin and SRTF Algorithm was successfully completed, and its output was thoroughly compared and validated against theoretical results. |

|  |
| --- |
| **Signature of faculty in-charge with Date:** |