CSC 4320 Operating Systems

Project 1: Process Scheduling Simulation

Brock Freiberger
Giada Giacobbe
Jay Patel

Overview

This project focuses on process scheduling, a critical concept in all operating systems that optimizes performances by managing task execution. For our program specifically, we utilized two fundamental scheduling algorithms: First Come First Serve (FCFS) and Shortest Job First (SJF). In this report, we will explore how FCFS, a simpler queue-based approach, may differ from SJF, a more involved method that prioritizes processes' burst time.

Along with CPU scheduling, our program will also demonstrate file handling and be able to produce a simple Gantt chart. This report provides an overview on the algorithms' implementations, the associated outputs, as well as any challenges our team may have faced.

Procedure

```
private static List<Process> readProcesses(String fileNAme) {
  List<Process> list = new ArrayList<>():
  try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
      string line;
      //SKIPS HEADER
      boolean skipHeader = true;
      while ((line = br.readLine()) != null) {
        if (skipHeader) {
            //CHANGES VALUE
            skipHeader = false;
            continue;
        }
        //SPLIT LINE
        String[] parts = line.trim().split("\\s+");
        if (parts.length < 4) {
            continue;
        }

        int pid = Integer.parseInt(parts[0]);
        int arrival = Integer.parseInt(parts[1]);
        int burst = Integer.parseInt(parts[2]);
        int priority = Integer.parseInt(parts[3]);

        list.add(new Process(pid, arrival, burst, priority));
      }
  ]

  catch (IOException e)  {
      system.out.println("Read error");
  }
  return list;
}
```

First, we implemented a read function to see the process data and use the values to implement schedulers and data charts. It first skips the top line, assuming it isn't data. The buffer opens the file and pushes it into the file reader. Once opened, a flag is created to split code by whitespace and checks that there are the minimum required data entries. If the minimum amount of data is shown, the variables correspond to the four types: PID, arrival time, birth time, and priority. The values are added to the new list called Process, and the list is pushed.

```
1  private static void fcfsSchedule(List<Process> processes) {
2    processes.sort((p1,p2) -> Integer.compare(p1.arrivalTime, p2.arrivalTime));
3
4    int currentTime = 0;
5    for (Process p : processes) {
6        if (currentTime < p.arrivalTime) {
7          currentTime = p.arrivalTime;
8        }
9      p.startTime = currentTime;
10     p.finishTime = p.startTime + p.burstTime;
11     p.waitingTime = p.startTime - p.arrivalTime;
12     p.turnaroundTime = p.finishTime - p.arrivalTime;
13     currentTime = p.finishTime;
14   }
15 }
```

For the FCFS algorithm, the program begins comparing and sorting by process arrival time. Then, in lines 5-8, it iterates through every process and if the CPU is idle, it replaces the *currentTime* with *arrivalTime* to simulate that process having run. Lines 9-10 update the start and finish time, taking into account the current time. Finally, lines 11-13 calculate waiting and turnaround times, as well as update *currentTime* for that process.

```
1  private static void sjfSchedule(List<Process> processes) {
2      processes.sort((p1,p2) -> {
3        if (p1.arrivalTime != p2.arrivalTime) {
4            return Integer.compare(p1.arrivalTime, p2.arrivalTime);
5        }
6        return Integer.compare(p1.burstTime, p2.burstTime);
7      });
8
9      int n = processes.size();
10     bootlean[] visited = new boolean[n];
11     int finished = 0;
12     int currentTime = 0;
13
14     while (finished < n) {
15       int idx = -1
16       int minBurst = Integer.MAX_VALUE;
17       for (int i = 0; i < n; i++) {
18           Process p = processes.get(i);
19           if (!visited[i] && p.arrivalTime <= currentTime) {
20               if (p.burstTime < minBurst) {
21                   minBurst = p.burstTime;
22                   idx = i;
23               }
24           }
25       }
26
```

```
26
27          if (idx == -1) {
28              int earliestArrival = Integer.MAX_VALUE;
29              int earliestIdx = -1;
30              for (int i = 0; i < n; i++) {
31                  if (!visited[i] && processes.get(i).arrivalTime < earliestArrival) {
32                      earliestArrival = processes.get(i).arrivalTime;
33                      earliestIdx = i;
34                  }
35              }
36              currentTime = processes.get(earliestIdx).arrivalTime;
37              idx = earliestIdx;
38          }
39          visited[idx] = true;
40          Process p = processes.get(idx);
41
42          p.startTime = currentTime;
43          p.finishTime = p.startTime + p.burstTime;
44          p.waitingTime = p.startTime - p.arrivalTime;
45          p.turnaroundTime = p.finishTime - p.arrivalTime;
46
47
48          currentTime = p.finishTime;
49          finished++;
50        }
51    }
```

The SJF algorithm starts by sorting processes based on their arrival time, and if two processes have the same arrival time, it further sorts them by burst time to prioritize the shortest job. After initializing key variables like visited, finished, and currentTime, the program enters a while loop that runs until all processes are scheduled. Inside the loop, it iterates through all unvisited processes that have already arrived and selects the one with the shortest burst time. If no such process is available, it finds the process with the earliest arrival time among the remaining ones and moves currentTime forward to simulate CPU idling. Once a process is selected, it is marked as visited, and its startTime, finishTime, waitingTime, and turnaroundTime are calculated. The finishTime is determined by adding the burstTime to startTime. At the same time, waitingTime is derived from the difference between startTime and arrivalTime, and turnaroundTime is calculated as the difference between finishTime and arrivalTime. Finally, currentTime is updated to reflect the completion of the process, and finished is incremented. This cycle repeats until all processes are scheduled, ensuring that the shortest available job is always executed next.

```java
private static void printGanttChart(List<Process> processes) {
    processes.sort((p1,p2) -> Integer.compare(p1.startTime, p2.startTime));

    StringBuilder top = new StringBuilder();
    StringBuilder bottom = new StringBuilder();

    for (int i = 0; i < processes.size(); i++) {
      Process p = processes.get(i);
      top.append("| P").append(p.pid).append(" "); // prints "| P# "
      bottom.append(p.startTime).append("    "); // prints spacing for times
      if (i == processes.size() - 1) {
        bottom.append(p.finishTime);
      }
    }
    top.append("|");

    System.out.println(top);
    System.out.println(bottom);
  }
```

This method prints out a legible Gantt chart to display the processes' timeline. Firstly, the program uses StringBuilder to initialize a top and bottom line for the PIDs and the execution times. Using a loop, the program prints every process along with its associated start times. It also guarantees to print the finish time of the final process. Finally, it outputs both lines.

Troubleshooting/Challenges We faced

One challenge we faced was handling the file that contains the process data. The program reads from processes.txt, but if the file is missing, formatted incorrectly, or has incomplete data, it might not read the processes correctly. This could lead to an empty process list, causing the scheduling functions to stop early or give wrong results. Also, if the file has unexpected characters or extra spaces, the program might not be able to process the data properly, leading to errors or skipped processes. So, when we wrote the program, we made sure to include error handling to check for missing or incorrect data and display clear error messages to prevent these issues.

Another challenge we faced was how the sorting processes were handled for the two different scheduling methods. FCFS is simple because it only sorts by arrival time. However, SJF is more complex because if two processes arrive simultaneously, they must be sorted by arrival time and burst time. If the sorting isn't done correctly, a process with a shorter burst time might be scheduled for later. So, when we wrote the program, making sure the sorting was accurate and efficient was important.

For SJF scheduling, keeping track of which processes had already run was very important. We used a visited list to mark any completed processes, but if a process wasn't marked correctly, it could either run again or get skipped. Another issue we ran into was when no process was ready to run at the current time. In that case, the program had to find the next process with the earliest arrival time to keep the CPU from idle too long. If we hadn't handled this properly, the program could have ended up in an infinite loop or scheduled processes incorrectly.

Results

```
FCFS
| P1 | P2 | P3 | P4 | P5 |
0    5    8    16    18    23
PID     Start   Finish  Wait    TAT
1       0       5       0       5
2       5       8       4       7
3       8       16      6       14
4       16      18      12      14
5       18      23      12      17
Avg Waiting: 6.80
Avg TAT: 11.40

SJF
| P1 | P4 | P2 | P5 | P3 |
0    5    7    10    15    23
PID     Start   Finish  Wait    TAT
1       0       5       0       5
2       7       10      6       9
3       15      23      13      21
4       5       7       1       3
5       10      15      4       9
Avg Waiting: 4.80
Avg TAT: 9.40
```

This is the result of running the whole file. First, the FCFS scheduler is printed, and it shows the P values and the respective text based Gantt chart for the scheduler. Finally, the wait times and TAT are both shown at the bottom.

The same output type is then printed for the SJF scheduler with the respective numbers.

Link for the GitHub Repository:
https://github.com/brockeh8/OS_Project