AASTU
Addis Ababa,
Ethiopia

ADDIS ABABA SCIENCE AND TECHNOLOGY UNIVERSITY
UNIVERSITY FOR INDUSTRY

Department of Software Engineering

Operating System

Process Scheduling Algorithm

Shortest Remaining Job First Scheduler Implementation in Java

PREPARED BY

	Name	ID number
1	Tamirat Dejene	ETS1518/14
2	Tadiyos Dejene	ETS1522/14
3	Tebarek Shemsu	ETS1526/14
4	Tamrat Demse	ETS1523/14
5	Surafel Abera	ETS1507/14
6	Solomon Tadesse	ETS1495/14

05.17.2024 Team INNOV8 Section: E

Submitted to: Inst. Kassahun

Implementing Shortest Remaining Job First (SRJF) Scheduling Algorithm in Java

Introduction

Process scheduling is a fundamental aspect of operating systems, determining the order in which processes are executed by the CPU. Among various scheduling algorithms, the Shortest Remaining Job First (SRJF) is both a preemptive and non preemptive scheduling method where the process with the shortest remaining execution time is selected for execution next. This report outlines the implementation of the SRJF algorithm in Java, including the design considerations, implementation details, and a brief analysis of the results done by Group 3.

Design Considerations

The SRJF scheduling algorithm requires maintaining an up-to-date queue of processes based on their remaining execution times. Key considerations in the design include:

- **Preemptive or Non-preemptive nature**
    - Preemptive: the algorithm preempts the currently running process if a new process with a shorter remaining time arrives.
    - Non-preemptive: the algorithm doesn't pre-empete the currently currently running process even if the the new process has shorter burst time
- **Efficiency**: Efficient data structures are necessary to handle frequent updates and ensure the algorithm runs in a timely manner.

Implementation Details

The implementation involves creating:

- *PCB* (Process control block) class to represent each process
- *Scheduler* (base class) which is extended by a specific scheduling algorithm class to manage the scheduling.
- Execution Snapshot class which is used to show the process under execution.
- ShortestJobFirst class for the implementation of SJF algo.
- TestSRJF class for testing

The PCB Class

The PCB class holds information about each process, including:

- Process id: uniquely represent the process
- Process burst time: represents the cpu time the process requires to complete execution
- Process arrival time: the time at which the process got into the process queue
- Process Priority: priority attached to the process.

The class constructor overloads look like the following:

- If all the parameters are provided:

```
public PCB(String pID, int burstTime, int arrivalTime, int priority)
{
    this.pID = pID;
    this.burstTime = burstTime;
    this.arrivalTime = arrivalTime;
    this.priority = priority;
}
```

- If priority is not set, the scheduler assumes all the process have the same priority

```
public PCB(String pID, int burstTime, int arrivalTime) {
    this.pID = pID;
    this.burstTime = burstTime;
    this.arrivalTime = arrivalTime;
    this.priority = 0;
}
```

- If arrival time is not provided, the scheduler assumes that all process arrived at the same instant time

```
public PCB(int burstTime, int priority, String pID) {
    this.pID = pID;
    this.burstTime = burstTime;
    this.priority = priority;
}
```

- If only process id and burst time are provided, the scheduler assumes that all the processes have the same priority and arrival time.

```
public PCB(String pID, int burstTime) {
    this.pID = pID;
    this.burstTime = burstTime;
    this.arrivalTime = 0;
    this.priority = 0;
}
```

- The class also implements the cloneable interface to help make a deep copy of the process to work on.
- Also has standard getters and setters, with the overridden implementation of `clone` and clone method

The `ExecutionSnapshot` class

This class is used to represent the process under execution in the instant. It has the following fields

- `processId`: used to represent the process under execution uniquely
- `tInitial`: the instant the CPU began executing the process
- `tFinal`: the instant the CPU finished/suspended executing the process

The constructor of this class looks the following

```
public ExecutionSnapshot(String processId, int tInitial, int tFinal)
{
    this.processId = processId;
    this.tInitial = tInitial;
    this.tFinal = tFinal;
}
```

This class also provides the standard getters and setters for the fields.

The `Scheduler` Class

The scheduler class is used as base class for the algorithms which are used to schedule CPU processes and holds the following attributes or fields which are inheritable by the extending classes.

★ List of processes, to represent processes to be executed (The process queue)
★ Turnaround time of each process scheduled to be executed
★ Waiting time used to store the amount of time the process waiting before getting into the ready queue

- ★ Completion time is also the time at which the process finishes execution.
- ★ Response time, the time gap between the arrival time and the first execution time
- ★ Schedule table(Snapshot) is used to keep track of the execution snapshot of the process while the cpu executes.
- ★ Average turnaround time
- ★ Average waiting time
- ★ Average response time
- ★ Throughput

The above fields are represented as follows using Java.

```
private List<PCB> processes;
private Map<String, Integer> turnAroundTime;
private Map<String, Integer> waitingTime;
private Map<String, Integer> completionTime;
private Map<String, Integer> responseTime;
private LinkedList<ExecutionSnapshot> scheduleTable;
private double averageTurnAroundTime;
private double averageWaitingTime;
private double averageResponseTime;
private double throughput;
```

The constructor of the class Scheduler looks like the following

- The constructor takes one parameter which is list of processes to be executed

```
public Scheduler(List<PCB> processes) {
    this.processes = processes;
    this.turnAroundTime = new HashMap<>();
    this.waitingTime = new HashMap<>();
    this.completionTime = new HashMap<>();
    this.responseTime = new HashMap<>();
    this.scheduleTable = new LinkedList<>();
    this.averageTurnAroundTime = 0D;
    this.averageWaitingTime = 0D;
    this.averageResponseTime = 0D;
    this.throughput = 0D;
}
```

- With the provision of standard getters and setters for all the above fields, this class has the following helper methods with the key method class `schedule()` which is used to implement the actual algorithm for scheduling. Let's discuss each method.

Four Overloads of the method print

`print(LinkedList<ExecutionSnapshot> scheduleTable)`

- Helps to print the snapshot of the process executed and takes a list of execution snapshot.
    - scheduleTable: represents the list of the execution snapshot

`print(String header, Map<String, Integer> mapData)`

- Takes two parameters and is used to display the information such as completion time, turnaround time and waiting time of each process.
    - `header`: represents the name of the information displayed
    - `mapData`: represents the data to be displayed

`print(String header, double timeUnits)`

- To display time either average waiting time or average turnaround times and takes two parameters
    - header: represents information to be displayed
    - timeUnits: the time

`print(List<PCB> processes)`

- To display list of processes and takes list of process as a parameter
    - processes: list of processes to be printed on the console.

`getProcess(String pID)` : retrieve the PCB of the given process id

`saveSnapshot(String pId, int tInit, int tFinal)`: stores the executed process

`computeResponseTime()`: computes the response time and average resp time

`computeCompletionTime()`: sets the completion time for each process

`computeTurnAroundTime()`: computes the tat and average tat

`computeWaitingTime()`: computes wt and average wt

`computeThroughput()`: calculates the throughput of the CPU

The `schedule` method

The whole purpose of this class is to provide the implementation which can be easily customised to fit for other algorithms. This method provides the actual implementation of the scheduling algorithm which can be overridden by the extending process to implement another algorithm and takes three parameters.

1. `Comparator<PCB> processQueueComparator`

- This comparator is provided by the user of the algorithm and is used to sort the processes in the process queue

2. `Comparator<PCB> readyQueueComparator`

- This comparator is also provided by the user of this method and is used to manage the order of the processes in the ready queue.

3. `boolean isPreemptive`

- The third parameter tells the method whether the scheduler uses the preemptive or non-preemptive, a value of true means the scheduler uses the preemptive algorithm.

The return of this method will be linked list of execution snapshot and the implementation is provided as follows with proper documentation

```
/**
   * @param processQueueComparator Comparator to sort the process queue
   * @param readyQueueComparator   The comparator used to order ready queue inside the priority queue
   * @param isPreemptive  tells the scheduler whether to use preemptive or nonpreemptive scheduling
   * @return schedule(execution) snapshot as a LinkedList of {@code ExecutionSnapshot }
   */
  public LinkedList<ExecutionSnapshot> schedule(Comparator<PCB> processQueueComparator, Comparator<PCB>
readyQueueComparator, boolean isPreemptive) {
    // To not mutate the original process queue: make deep copy of the queue to work with
    var processesCopy = new LinkedList<PCB>();
    for (PCB process : processes) processesCopy.add(process.clone());

    // Check if the process queue is empty
    if (processesCopy.size() == 0) {
      scheduleTable.addLast(new ExecutionSnapshot("--", 0, 0));
      return scheduleTable;
    }

    // Order the process queue based on the queue comparator
    processesCopy.sort(processQueueComparator);

    /**
     * Make a ready queue (process priority queue), remove the first process in the process queue and add
     * it to the ppq. ppq is a priority queue in which it is always guaranteed that element with highest
     * priority is at the top/front (depends on the readyQueueComparator) provided
     *
     * timer: used as clock to keep track of the cpu time
     **/
    var ppq = new PriorityQueue<PCB>(readyQueueComparator);
    ppq.add(processesCopy.removeFirst());
    var timer = 0;

    // As long as ready-queue is not empty: the processor keeps executing
    while (!ppq.isEmpty()) {
      /**
       * If the process at the front of the top of the ready queue arrived late meaning arrived after the
timer
       * has started, there is cpu cycle wastage we need to keep track of.
       * Example if the first process arrives 1 second after the timer started
```

```
       */
      if (ppq.peek().getArrivalTime() > timer) {
        saveSnapshot("--", timer, ppq.peek().getArrivalTime());
        timer = ppq.peek().getArrivalTime();
        continue;
      }

      /** Take the front process inside the ready queue for execution */
      var currentProcess = ppq.poll();

      /**
       * In case of preemptive scheduling: Until the next processe's arrival time, the currently executing
process
       * goes on execution unless it is done executing. We do this because we need to make sure if the next
process
       * has lower burst time/higher priority it will pre-empete the currently executing process.
       *
       * let's find the next arrival time :
       *    If the processCopy is empty we have no next process
       *    Else we will take the process at the front inside the process queue and get it arrival time
       **/
      var nextProcessArrivalTime = processesCopy.isEmpty() ? null :
processesCopy.getFirst().getArrivalTime();

      /**
       * Then:
       *  - If the scheduling is not preemptive: the next coming process will not preempt the currently
executing process
       *  - Or If there is no more process to be executed
       *  - Or If the currently executing process gets done executing before the next process arrives
       * Execution can just continue and
       *    # Save the snapshot of the execution
       *    # Advance the timer by burst time of the executed process (time it took to execute the current
process)
       *    # Save - completion time and update the timer
       */
      if (!isPreemptive || processesCopy.isEmpty() || timer + currentProcess.getBurstTime() <=
nextProcessArrivalTime) {
        saveSnapshot(currentProcess.getPID(), timer, timer + currentProcess.getBurstTime());
        timer += currentProcess.getBurstTime();
      }
      /** Else
       * The scheduling is preemptive and there is next process to be executed and the current process will
not be done executed,
       * thus we let the current process execute partly  or till the next process arrives and then push it
back to ppq or the
       * ready queue with the remaining burst time
       *
       * - The remaining burst time for the process is calculated as
       *      # timer + current process burst time - next process arrival time
       * - And advance the timer to the next processe's arrival time
       */
      else {
        saveSnapshot(currentProcess.getPID(), timer, nextProcessArrivalTime);
        currentProcess.setBurstTime(currentProcess.getBurstTime() - (nextProcessArrivalTime - timer));
        timer = nextProcessArrivalTime;
        ppq.add(currentProcess);
      }

      /**
       * If there are other processes arrived till now, we move them from the process queue to the ready
the ready queue
       **/
```

```
      while (!processesCopy.isEmpty() && processesCopy.getFirst().getArrivalTime() <= timer)
        ppq.add(processesCopy.removeFirst());

      /**
       * Check if processCopy is not empty and readyQueue is empty: this means that the process at the
front of the process queue
       * has not arrived yet, thus we have to forcefully push it inside the ready queue: otherwise the
execution will cease before finishing
       */
      if (!processesCopy.isEmpty() && ppq.isEmpty())
        ppq.add(processesCopy.removeFirst());
    }
    /**
     * Finally calculate completion, turnaround, waiting, and response times
     * And the throughput of the cpu
     */

    computeCompletionTime();
    computeTurnAroundTime();
    computeWaitingTime();
    computeResponseTime();
    computeThroughput();
    /* and return the execution snapshot */
    return this.scheduleTable;
  }
```

The `ShortestJobFirst` class

This class is used to implement one of the CPU task scheduling algorithms which is the shortest remaining job first, which means the process with the shortest burst time will get executed first.

- This class extends the `Scheduler` class
- The constructor of this class takes one parameter which is the list of processes to be executed and looks like the following

```
public ShortestJobFirst(List<PCB> processes) {
    super(processes);
}
```

- Provides a method `shortestRemainingJobFirstScheduler(boolean isPreemptive)`
    - This method facilitates for the use of algorithm presented in the parent class and it takes one parameter as used to tell whether to use preemptive or nonpreemptive scheduling
        - isPreemptive: true value of this parameter tells the scheduler to use the preemptive scheduling scheme, otherwise it uses the non-preemptive scheme.
    - The return value of this function will be the snapshot of the executed processes as a linked list
    - The signature of the method looks like the following

```
public LinkedList<ExecutionSnapshot>
shortestRemainingJobFirstScheduler(boolean isPreemptive)
```

- This method calls the schedule method provided by the parent class and provides the comparators used to order or manage the order of processes used by the schedule method. The implementation of this method looks like the following.

```
/**
   *
   * @param isPreemptive : tell the scheduler to whether the coming process can
preempete the currently executing process or not, default value is true
   * @return snapshot of the process execution as a string linked list
   */
  public LinkedList<ExecutionSnapshot> shortestRemainingJobFirstScheduler(boolean
isPreemptive) {
    /**
     * The process queue will be in the order of their arrival time and if two
     * process arrives at the same time, their burst time will be used as a
comparator
     */
    Comparator<PCB> processQueueComparator = (PCB p1, PCB p2) -> {
      return p1.getArrivalTime() == p2.getArrivalTime() ? p1.getBurstTime() -
p2.getBurstTime()
          : p1.getArrivalTime() - p2.getArrivalTime();
    };

    /**
     * The ready queue will be order based on the burst time, since the job with
shortest
     * burst time need to be at the front of the process priority queue
     */
    Comparator<PCB> readyQueueComparator = (PCB p1, PCB p2) -> p1.getBurstTime() -
p2.getBurstTime();
    return schedule(processQueueComparator, readyQueueComparator, isPreemptive);
  }
```

Overall the implementation of the algorithm is well documented and the user can refer to it.

The TestSRJF class

This class is a test class, which performs some testing on the implemented scheduling algorithm. Below we will discuss a main test case which covers edge cases with the output and the expected results. The rest of test can be referred to by the user in the implemented file

Process ID	Burst Time(ms)	Arrival Time(ms)
P1	12	1
P2	4	2
P3	6	3
P4	5	8

Non-preemptive scheduler

Expected result of schedule table:

```
[0 <- - -> 1 <- P1 -> 13 <- P2 -> 17 <- P4 -> 22 <- P3 -> 28]
```

Output:

```
[0 <- - -> 1 <- P1 -> 13 <- P2 -> 17 <- P4 -> 22 <- P3 -> 28]
```

NB: – represents the CPUwasn't utilised during that time, and wasted CPU cycle

Testcase	Expected Result	Output
Average turnaround-time	16.5ms	16.5ms
Average waiting time	9.75ms	9.75ms
Average response time	13.25ms	13.25ms
Throughput	0.1428proc/ms	0.1428proc/ms

Preemptive scheduler

Expected result of schedule table:

```
[0 <- -- -> 1 <- P1 -> 2 <- P2 -> 6 <- P3 -> 12 <- P4 -> 17 <- P1 -> 28]
```

Output:

```
[0 <- -- -> 1 <- P1 -> 2 <- P2 -> 6 <- P3 -> 12 <- P4 -> 17 <- P1 -> 28]
```

Testcase	Expected Result	Output
Average turnaround-time	12.25ms	12.25ms
Average waiting time	5.5ms	5.5ms
Average response time	5.25ms	5.25ms
Throughput	0.1428proc/ms	0.1428proc/ms

- So this test case has successfully passed. To get more test cases and their output can refer to the class TestSRJF java class in the implementation file.

Analysis

The Preemptive SJF scheduling algorithm is effective in minimising the average waiting time for processes compared to non-preemptive algorithms like FCFS (First Come, First Serve). However, the continuous monitoring and updating of the process queue can be computationally intensive, especially with a large number of processes. The implementation ensures that the scheduler dynamically selects the next process with the shortest remaining time, demonstrating the preemptive nature of the SRJF algorithm. While we see the non preemptive part of this algorithm, it is less effective than the preemptive one and has higher average waiting and turnaround time.

Conclusion

This report provides a comprehensive overview of the implementation of the Shortest Remaining Job First scheduling algorithm in Java. The design focuses on maintaining an efficient and dynamic process queue to ensure optimal CPU scheduling. Through the use of classes and appropriate data structures, the implementation accurately reflects the preemptive nature of SRJF, showcasing its benefits and potential performance considerations.

References

- Operating System Concepts 9th Edition
- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts (10th ed.). Wiley.
- Stallings, W. (2018). Operating Systems: Internals and Design Principles (9th ed.). Pearson.