Spring 2018

T-444-USTY, Grunnatriði Stýrikerfa

March 26, 2018

Process Scheduling

Report

Edda Steinunn Rúnarsdóttir

kt. 241095-2909

1 The Project

The project controls virtual processors via adaption of implementation of six popular scheduling algorithms, that is *First Come First Served*, *Round Robin*,*Shortest Process Next*, *Shortest Remaining Time*, *Highest Response Ratio Next* and *Feedback*. The project is implemented to interact with a given visualization scheme which shows threads as bars, with their estimated remaining running time percentage colored red and their completed running time as green. The progress (run time) of processes are in their nature different in the visualization scheme with the different algorithms.

The main actions reside within the file *Scheduler.java* which essentially sets up the logic for different process scheduling schemes in project by swapping to and from processes in different manners. The functions shared by all schemes besides the constructor function are the function ***startScheduling*** which handles initialization of the scheduling algorithm however appropriate, the function ***processAdded*** which is called when a process is to be added to ready queue and defines whatever actions are necessary when process is added and the function ***processAdded*** which is called when process is has finished running and is to be removed from ready queue and defines whatever actions necessary when a process finishes.

The functionality of the 'preemptive' scheduling algorithms in the project, ***Round Robin*** and ***Feedback*** (that is scheduling schemes that swap from current process ignoring whether it has finished running or not) is aided by the Runnable class *TimeSlicing.java* which define the behavior of a thread in preemptive priority scheduling manner. The functionality of priority based scheduling algorithms in the project, that is ***Shortest Process Next***, ***Shortest Remaining Time*** and ***Highest Response Ratio Next***, is aided by the Comparator class *ProcessesComparator.java* which compares processes in a scheduling-scheme specific manner in order to sort the process queue, and essentially with it the order of which the process are run. The class *ProcessesMeasurements.java* contains all logic enabling measurements on different types of scheduling schemes for comparison and context, calculating the average response time and the average turnaround time for each scheduling algorithm.

Other files in project are unmodified from original structure from teacher and implement test cases, environment and the visualization scheme for the project. These test cases are run by the file *SchedulingMainProgram.java*. Each test case is the same, but is run on different scheduling algorithms.

2 Scheduling Schemes Implementation Explained

2.1 First Come First Served (FCFS)

The First Come First Served scheduling algorithm uses a regular FIFO (first-in first-out) ready queue for queuing processes (implemented by a Linked-List). Whenever a process is added, it goes to the back of the ready queue and the processes are served in order. Whenever a process is run, it is not swapped out until it has finished due to FCFS being a non-preemptive scheme.

This behaviour is obtained by having the scheme swap to a process when added if and only if that process is the only one in ready queue. Then whenever a process finishes, the scheme swaps to the next process in ready queue in a FIFO manner (unless of course ready queue is empty and no process awaits). That way, processes are served fully in order of their arrival to the ready queue.

2.2 Round Robin (RR)

The Round Robin scheduling algorithm uses a FIFO (first-in first-out) Linked-List ready queue just like the FCFS algorithm. However in contrast to FCFS, Round Robin is preemptive and swaps out current process regardless of whether it finished running. This is due to Round Robin's **time-slicing nature**; all processes get their turn, in order, to be run in some equal amount of time called ***quantum*** before being swapped out for another process and moved the back of the queue to await their turn again. This time-slice behavior is implemented by having a helper thread run in a specific way, i.e. via a custom *Runnable* class which resides in *TimeSlicing.java* whose implementation is described in detail in section 2.2.1. The time-slicing thread is started immediately by the Round Robin algorithm in the function *startScheduling* in *Scheduler.java*. Apart from this preemptive behavior the scheme functions again just like FCFS in the sense whenever a process is added to an empty ready queue, the scheme swaps to that process and whenever a process finishes the scheme swaps to the next process in ready queue. That way, processes get their turn to run in quantum time in order of their arrival to the ready queue.

2.2.1 Round Robin Time-Slicing Implementation

Below shows how the logic of Round Robin time-slicing behavior is implemented (this code fraction is a little shortened for clarification and resides in file *TimeSlicing.java* of the project):

```
while (true) {
    ....
  // Process running preserved
  Integer currentProcess = scheduler.processes.peek();

  // Have thread sleep for the quantum time
  // Thread sleep period ensures process runs for a quantum time before swap
  Thread.sleep(scheduler.quantum);

  // Check if process running now is the same as the process running
  // before sleep period because that process could have finished running
  // and exited queue (in which case we don't want to move it to the back
  // of the queue)
  if(currentProcess == scheduler.processes.peek()) {

    // Move process to the back of queue since now it should be last of
    // processes in queue to run as it just got it's 'turn'
    Integer moveProcessBack = scheduler.processes.poll();

    if ( moveProcessBack != null) {

      // Process moved from font of queue to back of queue
       scheduler.processes.add(moveProcessBack);

     // Let next process have it's go
       scheduler.processExecution.switchToProcess(scheduler.processes.peek());
    }
  }
    ....
}
```

Now to further clarify for a non-empty ready queue, we have a thread sleep for a specified **quantum** amount of time while a process is running. After that if the process that was running

has not finished, we move it to the back of the ready queue so that it can have it's turn again. When queue has been corrected this way we swap to the next process in queue to have it's turn and so on and so forth. This ensures the Round Robin behavior of 'token-passing' i.e. processes getting their turn in quantum time and in order of arrival.

2.3 Shortest Process Next (SPN)

The Shortest Process Next scheduling scheme uses a **priority queue** which is ordered by a custom *Comparator*, meaning that this *Comparator* orders the processes in the ready queue based on some element comparison, in this case based on processes' total service time. The comparison scheme used for implementing the SPN algorithm is explained thoroughly in section 2.3.1. The processes are served in the order of this custom priority queue as a ready queue and each process is served in full, i.e. the SPN algorithm is non-preemptive meaning that processes that are served get to finish running completely before the scheme swaps to another process.

Therefore whenever a process is added, it is placed in the custom priority queue in some slot based on it's total service time comparison. However note that even though new process goes straight to the front of queue it is not swapped to until the current process running finishes since the algorithm is non-preemptive, i.e. serves processes in full before swapping. Just like in FCFS and RR, whenever a process is added to an empty queue the scheme swaps to that process and whenever a process finishes the scheme swaps to the next process in the ready queue (in this case the priority queue). This ensures that processes are served in the order of their total service time.

2.3.1 Priority Queue Comparator Used

Below shows how the custom Comparator of Shortest Process Next algorithm works and how it orders the priority queue (this code fraction is a little shortened for clarification and resides in file *ProcessComparator.java* of the project):

```
1  public class ProcessesComparator implements Comparator<Integer> {
2      .....
3      @Override public int compare(Integer lhs, Integer rhs) {
4          ....
5          // Get information on both left-hand side process
6          // and right-hand side process being compared
7          ProcessInfo p1 = this.scheduler.processExecution.getProcessInfo(lhs);
8          ProcessInfo p2 = this.scheduler.processExecution.getProcessInfo(rhs);
9
10         // Whichever process that has more total service time is considered
11         // "greater" than the other
12         if (  p1.totalServiceTime > p2.totalServiceTime) return 1;
13         else if (p1.totalServiceTime < p2.totalServiceTime) return -1;
14         else return 0;
15         ....
16
17     }
18     ....
19 }
```

By initializing the priority queue using this Comparator above as follows:

```
1 this.processes = new PriorityQueue<Integer>(50, new ProcessesComparator(this));
```

We're essentially ordering the queue by the total service time of processes, which orders the ready queue just as it should be. This is what ensures that the queue works as it should and processes are served in order of their service time, shortest first, which enables the algorithms' behaviour.

2.4 Shortest Remaining Time (SRT)

The Shortest Remaining Time scheduling scheme uses a **priority queue** as well as SPN which is also ordered by a custom *Comparator*. this *Comparator* orders the processes in the ready queue based on some element comparison, in this case based on processes' remaining time. The comparison scheme used for implementing the SRT algorithm is explained thoroughly in section 2.4.1. The processes are served in the order of this custom priority queue as a ready queue just as in the SPN algorithm, ***however with one major difference*** which is that the SRT algorithm is preemptive whereas SPN algorithm is not, meaning that whenever another process that has shorter remaining time than the current process running, the scheme swaps to that process regardless of whether currently running process has finished running or not.

Therefore whenever a process is added, it is placed in the custom priority queue in some slot based on it's total service time comparison and if that new process goes straight to the front of queue the scheme swaps out current process and runs the new process. Just like in FCFS, RR and SPN, whenever a process is added to an empty queue the scheme swaps to that process and whenever a process finishes the scheme swaps to the next process in the ready queue (in this case the priority queue). This ensures that processes are served in the order of their remaining time.

2.4.1 Priority Queue Comparator Used

Below shows how the custom Comparator of Shortest Remaining Time algorithm works and how it orders the priority queue (this code fraction is a little shortened for clarification and resides in file *ProcessComparator.java* of the project). Note that the remaining time is naturally calculated by the formula:

$$process\ remaining\ time\ =\ total\ service\ time\ of\ process\ -\ elapsed\ execution\ time\ of\ process$$

```
1  public class ProcessesComparator implements Comparator <Integer> {
2      .....
3      @Override public int compare (Integer lhs, Integer rhs) {
4          ....
5
6          // Get information on both left-hand side process
7          // and right-hand side process being compared
8          ProcessInfo p1 = this.scheduler.processExecution.getProcessInfo(lhs);
9          ProcessInfo p2 = this.scheduler.processExecution.getProcessInfo(rhs);
10
11         // Whichever process that has more remaining service time is considered
12         // "greater" than the other
13         if (  p1.totalServiceTime-p1.elapsedExecutionTime >
14               p2.totalServiceTime-p2.elapsedExecutionTime)
15                 return 1;
16         else if (p1.totalServiceTime-p1.elapsedExecutionTime <
17                  p2.totalServiceTime-p2.elapsedExecutionTime)
18                 return -1;
19         else
20             return 0;
21         ....
22
23     }
24     ....
25 }
```

By initializing the priority queue using this Comparator above as follows:

```
1 this.processes = new PriorityQueue<Integer>(50, new ProcessesComparator(this));
```

We're essentially setting the priority as shortest remianing time, i.e. ordering the queue by the shortest remaining time of processes, which orders the ready queue just as it should be. This is what ensures that the queue works as it should and processes are served in order of their remaining time, shortest first, which enables the algorithms' behaviour. Then the preemptive behavior (i.e. that if a process is added with shorter remaining time than process currently running the scheme swaps to the new process) is ensured by constantly reevaluating queue when a process is added to it.

2.5 Highest Response Ratio Next (HRRN)

The Highest Response Ratio Next scheduling scheme uses a **priority queue** as well as SRT and SPN which is also ordered by a custom *Comparator*. this *Comparator* orders the processes in the ready queue based on some element comparison, in this case based on response ratio. The comparison scheme used for implementing the HRRN algorithm is explained thoroughly in section 2.5.1. The processes are served in the order of this custom priority queue as a ready queue and each process is served in full, i.e. the HRRN algorithm is just like SPN algorithm non-preemptive meaning that processes that are served get to finish running completely before the scheme swaps to another process.

Therefore whenever a process is added, it is placed in the custom priority queue in some slot based on it's response ratio comparison. However note that even though new process goes straight to the front of queue it is not swapped to until the current process running finishes since the algorithm is non-preemptive, i.e. serves processes in full before swapping. Just like in FCFS and RR, whenever a process is added to an empty queue the scheme swaps to that process and whenever a process finishes the scheme swaps to the next process in the ready queue (in this case the priority queue). This ensures that processes are served in the order of their highest response ratio.

2.5.1 Priority Queue Comparator Used

Below shows how the custom Comparator of Highest Response Ratio Next works and how it orders the priority queue (this code fraction is a little shortened for clarification and resides in file *ProcessComparator.java* of the project). Note: the result ratio is calculated by the formula:

$$Response\ Ratio\ =\ \frac{process\ waiting\ time\ +\ process\ estimated\ run\ time}{estimated\ run\ time}$$

```
1  public class ProcessesComparator implements Comparator <Integer> {
2      .....
3      @Override public int compare (Integer lhs, Integer rhs) {
4          ....
5          // Get information on both left-hand side process
6          // and right-hand side process being compared
7          ProcessInfo p1 = this.scheduler.processExecution.getProcessInfo(lhs);
8          ProcessInfo p2 = this.scheduler.processExecution.getProcessInfo(rhs);
9
10         // Get result ratio of both values being compared
11         double resRatio1 = ((double)(p1.elapsedWaitingTime+p1.totalServiceTime)/
12                             (double)p1.totalServiceTime);
13         double resRatio2 = ((double)(p2.elapsedWaitingTime+p2.totalServiceTime)/
14                             (double)p2.totalServiceTime);
15
16         // Whichever process that has more result ratio is considered
17         // "lower" than the other (because we want HIGHEST ratio value first)
18         if (resRatio1 < resRatio2)      { return 1; }
19         else if (resRatio1 > resRatio2) { return -1; }
20         else return 0;
21         ....
22     }
23     ....
24 }
```

By initializing the priority queue using this Comparator above as follows:

```
this.processes = new PriorityQueue<Integer>(50, new ProcessesComparator(this));
```

We're essentially setting the priority as highest response ratio, i.e. ordering the queue by the highest response ratio of processes, which orders the ready queue just as it should be. This is what ensures that the queue works as it should and processes are served in order of their response ratio, highest first, which enables the algorithms' behaviour.

2.6 Feedback (FB)

Feedback uses a list of seven queues as it's ready queue instead of a single queue unlike all the other scheduling algorithms covered. Each queue in the list is a FIFO queue. This scheduling algorithm is preemptive and uses ***time-slicing*** just as the Round Robin algorithm to swap between processes after a **quantum** amount of time. Unlike the Round Robin algorithm however the Feedback algorithm does not go in FIFO order of processes allowing them to take turns of running in strict FIFO order, but rather each queue of processes has a priority corresponding to the index of the queue list (that is, queue with index 0 in list has the most priority) and Feedback allows all processes to take turn (in a Round Robin style) in the most-priority queue of list until all processes in that queue have finished before moving to the next queue and repeating the progress.

Therefore scheduling scheme basically partitions the processes into multiple queues, allows processes to move between queues, then finishes processes queue by queue in ascending order of queues, each queue finished in a Round Robin style. To achieve this functionality multiple Feedback-specific helper functions were needed that are described thoroughly in section 2.6.1 and a custom time slicing implementation explained thoroughly in section 2.6.2.

2.6.1 Helper Functions Used and their Purpose

There were four functions added to fully support the functionality of the Feedback scheduling algorithm, three of which have straight-forward functionality. The first of the three, *feedbackFindQueueOfCurrentProcess* finds the queue of current process. This is necessary check to conduct f.x. when switching between processes and moving process. The second is *allFeebackQueuesEmpty* which checks if all queues are empty which is necessary f.x. when deciding what to do when process is added to an empty feedback ready queue. The third is *findFirstNonEmptyFeedbackQueue* which essentially gives us which queue in system has the most priority, i.e. the first non-empty queue in list.

The fourth added helper function for Feedback however is not as straight-forward as the other three. The function is called ***feedbackSwitchProcess*** and conducts the swapping of processes in Feedback scheduling scheme. This action of course given the structural differences of the Feedback algorithm and the other algorithms implemented differs quite a lot and needs it's own different implementation. The function is as follows:

```
1  public void feedbackSwitchProcess() {
2
3    // Find queue containing current process running
4    int queueOfCurrentProcess = this.feedbackFindQueueOfCurrentProcess();
5
6    // If no process is running, discontinue
7    if(queueOfCurrentProcess != -1) {
8
9      // Remove current process from it's queue (if current process has not
10     // finished running at this point, it should be relocated to the next
11     // queue before calling this function)
12     this.feedbackQueues.get(queueOfCurrentProcess).remove();
13
14     // Get queue of highest priority to determine which process to swap to
15     // (should be first process in highest priority queue)
16     int highestPriorityQueue = this.findFirstNonEmptyFeedbackQueue();
17
```

```
18      if(highestPriorityQueue != -1) {
19
20          // Switch to first process in highest priority queue
21          int nextProcess = this.feedbackQueues.get(highestPriorityQueue).peek();
22          this.processExecution.switchToProcess(nextProcess);
23          this.currentProcess = this.feedbackQueues.get(highestPriorityQueue).peek();
24          .....
25       } else { this.currentProcess = -1; }
26    }
27 }
```

Basically, this function needs it's own implementation because the scheme needs to determine from which queue to remove the process it just finished running, and because it needs to determine to which queue to swap to (always swapped to first element in highest priority queue).

2.6.2 Time-Slicing Implementation

Below shows how the logic of Feedback time-slicing behavior is implemented. It is quite similar to the Round Robin time-slicing implementation but needs it's own implementation due to the different structure of ready queues of the algorithms. (This code fraction is a little shortened for clarification and resides in file *TimeSlicing.java* of the project):

```
1  while (true) {
2      ....
3      // Current process running preserved before running quantum time
4      Integer currentProcess = this.scheduler.currentProcess;
5
6      // Have thread sleep for the quantum time
7      // Thread sleep period ensures process runs for a quantum time before swap
8      Thread.sleep(scheduler.quantum);
9
10     if (this.scheduler.currentProcess != -1) {
11
12       // Check if process running now is the same as the process running
13       // before sleep period because that process could have finished running
14       // and exited queue (in which case we don't need to move it)
15       if(currentProcess == this.scheduler.currentProcess) {
16
17         // Find queue of current process
18         int currentProcessQueue = scheduler.feedbackFindQueueOfCurrentProcess();
19
20         // Move current process to next queue where it will await it's turn again
21         int processMove = this.scheduler.currentProcess;
22         if(currentProcessQueue != this.scheduler.FEEDBACKQUEUECOUNT-1) {
23          this.scheduler.feedbackQueues.get(currentProcessQueue+1).add(processMove)
24         } else {
25          this.scheduler.feedbackQueues.get(currentProcessQueue).add(processMove);
26         }
27
28       // After making proper arrangements we swap to next process
29       this.scheduler.feedbackSwitchProcess();
30       }
31     }
32     ....
33 }
```

Now to further clarify for a non-empty ready queue, we have a thread sleep for a specified **quantum** amount of time while a process is running. After that if the process that was running has not finished, we move remove it from it's current queue and move it to next queue with lower priority than the one it was in. There it awaits to have it's turn again to run in quantum time. When our list of queues has been corrected this way we swap to the next process which is the first element in the highest-priority (first non-empty) queue in list and so on and so forth. This ensures the Feedback behavior of processes getting their turn in quantum time in sequences of queues.

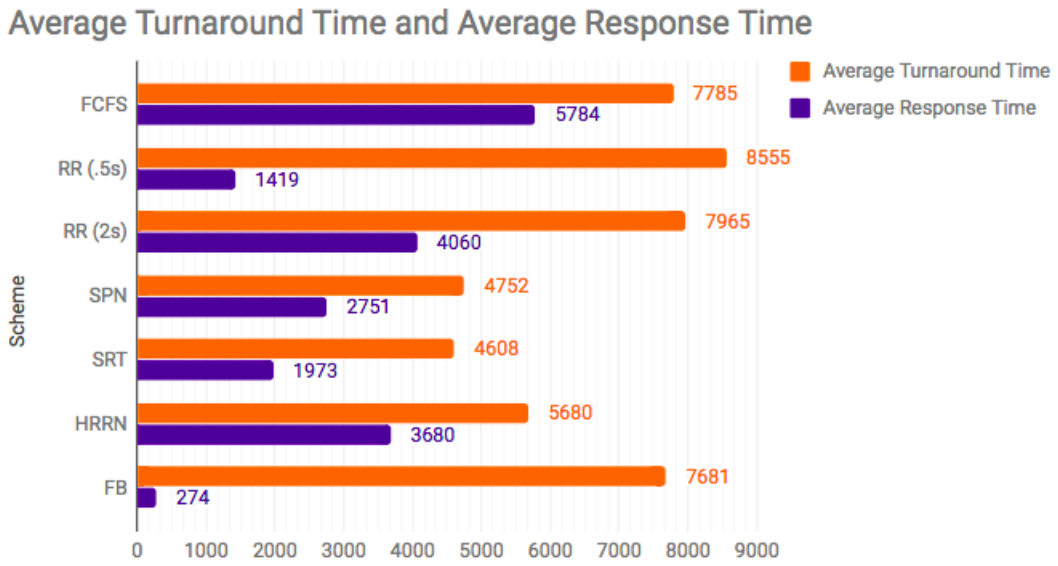
3 Scheduling Schemes Measurements and Deductions

As stated before, the class *ProcessMeasurements.java* conducts a measurement of average respone time and average turnaround time for all processes served by each scheduling algorithm. In table 1 below the result of those measurements are accessible:

Table 1: *Averages of turnaround time and response time for processes served by scheduling algorithms in project*

Process Scheduling Scheme	*Average Turnaround Time (ms)*	*Average Response Time (ms)*
First Come First Served	7785	5784
Round Robin (with quantum .5s)	8555	1419
Round Robin (with quantum 2s)	7965	4060
Shortest Process Next	4752	2751
Shortest Remaining Time	4608	1973
Highest Response Ratio Next	5680	3680
Feedback	7681	274

For a better context of this, consider chart 1 which demonstrates the differences in these measurement values between scheduling schemes:



Note that despite Feedback having the best response time, the turnaround time is high. However, the Shortest Process Next and Shortest Time Remaining scheduling algorithms have a good balance of both values (the SRT algorithm in particular). From this we can therefore deduce that in a system where fast response is imperative but speed isn't, Feedback is the way to go. By intuition Feedback would for example be optimal for a system with several processes that have short estimated run time. But for a system where speed is imperative (small turnaround time), either of the Shortest Process Next or Shortest Time Remaining algorithms are a better fit.