CS5250

Assignment 4

Zhang Shuhao

shuhao.zhang@comp.nus.edu.sg

A0120258N

Abstract

In this assignment, I use my preferred programming language—Java to finish tasks.
All the source code are public available at
https://github.com/ShuhaoZhangTony/ProcessSchedulingSimulator.
Please contact me if there is anything missing in the folder.
Different scheduling schemes (implemented as a Java class, e.g., RR.class) extends a common abstract Java class—*Scheduler*.

Taking round robin scheme (RR) as an example, the calling of RR is as follows.

```
input = new ProcessInput("input.txt");
LOG.info("simulating RR ----");
scheduler = new RR(input, 1);
output = new ScheduleOutput(scheduler);
LOG.info("printing output ---- (current time, process id)");
output.write_output("RR.txt");
```

Task 1:

1. Implement round robin scheme.

```
package schedule;

import Input.Process;
import Input.ProcessInput;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import schedule.impl.Queue;

/**
 * First come first service
 */
public class RR extends Scheduler {
   private static final Logger LOG = LoggerFactory.getLogger(RR.class);
   private final Queue<Process> q = new Queue();
   private int last_slice = 1;
   private Process current_process;

   public RR(ProcessInput input, int quantum) {
      do {
         boolean new_process = false;
         //update the queue during the last quantum.
         for (Process process : input.process_list) {
            if (arrive_in_lastSlices(process, current_time, last_slice)) {
               q.addLast(process);
               new_process = true;
```

```
            }
         }

         //terminate case
         if (!new_process && q.size() == 0) {
            if (all_finished_process(input.process_list)) {
               LOG.debug("RR finished scheduling for all process!");
               break;
            }
         }

         if (q.size() > 0) {
            //schedule the queue
            final Process process = q.removeFirst();
            if (current_process != null) {
               if (current_process != process) {
                  current_process = process;
                  schedule.add(current_time, process.id());
               }
            } else {
               current_process = process;
               schedule.add(current_time, process.id());
            }

            final int remaining = process.progress(quantum);
            if (remaining <= 0) {
               LOG.debug("Process finishes early than its assigned quantum
expire");
               last_slice = quantum + remaining;
               current_time += last_slice;
               process.finish();
               //clean the queue
               waiting_time += (current_time - process.arrive_time() -
process.burst_time());
            } else {
               last_slice = quantum;
               current_time += last_slice;
               q.addLast(process);//haven't finish computing, add to the end of
the queue.
            }
         } else {
            last_slice = 1;
            current_time += last_slice;//advance the timeslice to wait for next
process
         }
      } while (true);

      average_waiting_time = waiting_time / input.process_list.size();
   }

   private boolean arrive_in_lastSlices(Process process, int current_time, int
timeslice) {
//    return process.arrive_time() >= current_time && process.arrive_time() <
current_time + quantum;
      return process.arrive_time() <= current_time && process.arrive_time() >
current_time - timeslice;
   }
}
```

A few implementation details are highlighted here.
First, I use a linked list (``q") to implement the ready queue. The queue is getting updated by scanning the input process list by examine their arrival time, current time and quantum size.

The process that finishes computing are removed from the queue, otherwise, it will be added to the end of the list.
Second, processes that finish earlier than its assigned quantum expire will immediately be removed and allow other processes to get scheduled.

Given the sample input:
0 0 4
1 2 3
and set quantum to be 1, the output from RR will be:

```
printing output ---- (current time, process id)
(0,0)
(3,1)
(4,0)
(5,1)
average waiting time:1
```

2. Implement SRTF with given burst time.

SRTF with given burst time gives the optimal scheduling results.
The following is the *SRTF.java* implements the SRTF scheme.

```java
package schedule;

import Input.Process;
import Input.ProcessInput;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import schedule.impl.Queue;

/**
 * Shortest remaining task first.
 */
public class SRTF extends Scheduler {
   private static final Logger LOG = LoggerFactory.getLogger(SRTF.class);
   private final Queue<Process> q = new Queue();
   Process current_process = null;

   public SRTF(ProcessInput input) {

      do {
         boolean new_process = false;
         //update the queue during the last interval.
         for (Process process : input.process_list) {
            if (arrive_in_interval(process, current_time)) {
               q.addLast(process);
               new_process = true;

            }
         }

         //terminate case
         if (!new_process && q.size() == 0) {
            if (all_finished_process(input.process_list)) {
               LOG.debug("SRTF finished scheduling for all process!");
               break;
            }
         }

         //schedule
         if (q.size() > 0) {
            //schedule the queue
            final Process process = q.SRJ();
            if (current_process != null) {
               if (current_process != process) {
```

```
                current_process = process;
                schedule.add(current_time, process.id());
             }
          } else {
             current_process = process;
             schedule.add(current_time, process.id());
          }
          final int remaining = process.progress(1);
          if (remaining == 0) {
             current_time += 1;
             process.finish();
             //clean the queue
             waiting_time += (current_time - process.arrive_time() -
process.burst_time());
          } else {
             current_time += 1;
             q.addLast(process);//haven't finish computing, add to the end of the
queue.
          }
       } else {
          current_time += 1;//advance the timeslice to wait for next process
       }
    } while (true);
    average_waiting_time = waiting_time / input.process_list.size();
 }


 private boolean arrive_in_interval(Process process, int current_time) {
    return process.arrive_time() <= current_time && process.arrive_time() >
current_time - 1;
 }

}
```

The implementation of SRTF is very similar to RR, the key differences are highlighted as follows.

First, the quantum to progress in SRTF is essentially the time unit (i.e., 1), that is, current_time will be incremented by one (instead of *quantum size* used in RR) every time the scheduler progress.

Second, instead of always pick first item from the queue, it picks the job with least remaining time (the SRJ function).

This is the SRJ method used in SRTF that retrieves the job with least remaining time.

```
public Process SRJ() {
   Process SJ = new Process();
   for (Process e : this) {
      final int i = e.getRemaining_time();
      if (i < SJ.getRemaining_time()) {
         SJ = e;
      }
   }
   remove(SJ);
   return SJ;
}
```

Given the sample input:

0 0 4
1 2 3

, the output from SRTF will be:

```
printing output ---- (current time, process id)
(0,0)
(4,1)
average waiting time:1
```

3. Implement SJF with estimated burst time.

The main differences of SJF is that it can not rely on the knowledge of given burst time of process. Instead, it has to estimate the burst time.

```
package schedule;

import Input.Process;
import Input.ProcessInput;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import schedule.impl.Estimator;
import schedule.impl.Queue;

/**
 * Shortest Job First
 */
public class SJF extends Scheduler {
   private static final Logger LOG = LoggerFactory.getLogger(SJF.class);
   private int last_interval = 1;//allow the first task to be added.
   private final Queue<Process> q = new Queue();

   public SJF(ProcessInput input) {
      Estimator estimator = new Estimator();
      do {
         boolean new_process = false;
         //update the queue during the last interval.
         for (Process process : input.process_list) {
            if (arrive_in_interval(process, current_time)) {
               q.addLast(process);
               new_process = true;

            }
         }

         //terminate case
         if (!new_process && q.size() == 0) {
            if (all_finished_process(input.process_list)) {
               LOG.debug("SJF finished scheduling for all process!");
               break;
            }
         }

         if (q.size() > 0) {
            //schedule the queue
            final Process process = q.ESJ(estimator);//pick the estimated
shortest job.
            schedule.add(current_time, process.id());
            waiting_time += (current_time - process.arrive_time());
            last_interval = process.burst_time();
            process.finish();//do the work.
            q.remove(process);
            estimator.update_burst(process.id(), process.burst_time());
         } else {
            last_interval = 1;//advance time by 1 time slice.
         }
         current_time += last_interval;

      } while (true);
      average_waiting_time = waiting_time / input.process_list.size();
   }
```

```
    private boolean arrive_in_interval(Process process, int current_time) {
        return process.arrive_time() <= current_time && process.arrive_time() >
current_time - last_interval;
    }


}
```

Thanks to modular programming, clear decomposition and OOP design, the implantation of SJF is again very similar to SRTF and RR. The major differences are highlighted as follows.
First, as the question asks for non-preemptive version of SJF, we do not need to remember current process as did in SRTF and RR, as the current process is going to finish in its iteration.
Second, the progress of scheduler is non-determined and I use ``last_interval" to record it. This is an indicator of how long has it pass since the scheduling of the last process.
Third, ``ESJ()" method is used to determine which process to pick up in each iteration. Each time a process finishes, it updates the estimator by its actual burst time such that estimator can provide better estimation when encounter the same process next time.

The following is the implementation of ESJ method. Each time, it tries to pick up the estimated shortest task from the queue.

```
public Process ESJ(Estimator estimator) {
    Process SJ = new Process();
    estimator.update_burst(SJ.id(), Integer.MAX_VALUE);

    for (Process e : this) {
        final int i = estimator.estimate_burst(e.id());
        if (i < estimator.estimate_burst(SJ.id())) {
            SJ = e;
        }
    }
    return SJ;
}
```

The estimation function is implemented in the Estimator class (Estimator.java) as follows.

```
public int estimate_burst(int pid) {
    final int estimate = (int) (a * get_preburst(pid) + (1 - a) *
get_preestimate(pid));
    estimate_burst.put(pid, estimate);
    return estimate;
}
```

``a" is set to 0.5 initially. By adjusting ``a", we can tune SJF.

Given the sample input:

```
0 0 4
1 2 3
```

and set ``a" to be 0.5, the output from SJF will be:

```
printing output ---- (current time, process id)
(0,0)
(4,1)
average waiting time:1
```

Task 2

Question 1:

1. Test with the given input with RR.

Quantum=1

```
 printing output ---- (current time, process id)
(0,0)
(2,1)
(3,0)
(4,2)
(5,1)
(6,0)
(7,2)
(8,3)
(9,1)
(10,0)
(11,3)
(12,1)
(13,0)
(14,1)
(15,0)
(16,1)
(17,0)
(18,1)
(19,0)
(20,1)
(30,3)
(32,1)
(33,3)
(34,2)
(35,1)
(36,3)
(37,2)
(38,3)
(39,2)
(40,0)
(41,2)
(42,0)
(43,2)
(44,0)
(45,2)
(46,0)
(60,2)
(63,0)
(64,2)
(65,0)
(66,2)
(67,1)
(68,3)
(69,2)
(70,1)
(71,3)
(72,2)
(73,1)
(74,3)
(90,1)
(96,0)
(97,1)
(98,0)
(99,1)
(100,2)
(101,0)
(102,3)
(103,1)
```

```
(104,2)
(105,0)
(106,3)
(107,1)
(108,2)
(109,0)
(110,3)
(111,2)
(112,0)
(113,3)
(114,2)
(115,0)
(116,3)
(117,2)
(118,0)
(119,3)
(120,2)
(121,0)
(122,3)
(123,2)
(124,0)
(125,3)
(126,2)
average waiting time:8
```

Quantum=2

```
printing output ---- (current time, process id)
(0,0)
(4,1)
(6,2)
(8,0)
(10,1)
(12,3)
(14,0)
(16,1)
(18,0)
(19,1)
(30,3)
(34,1)
(36,2)
(38,3)
(39,2)
(41,0)
(43,2)
(45,0)
(60,2)
(64,0)
(66,2)
(68,1)
(70,3)
(72,2)
(73,1)
(74,3)
(90,1)
(98,0)
(100,1)
(102,2)
(104,0)
(106,3)
(108,2)
(110,0)
(112,3)
(114,2)
(116,0)
(118,3)
```

```
(120,2)
(122,0)
(124,3)
(126,2)
average waiting time:8
```

Quantum=4

```
printing output ---- (current time, process id)
(0,0)
(8,1)
(12,2)
(14,0)
(15,3)
(17,1)
(30,3)
(35,1)
(37,2)
(43,0)
(60,2)
(67,0)
(69,1)
(72,3)
(90,1)
(100,0)
(104,2)
(108,3)
(112,0)
(116,2)
(120,3)
(124,0)
(126,2)
average waiting time:7
```

Quantum=8

```
printing output ---- (current time, process id)
(0,0)
(9,1)
(17,2)
(19,3)
(30,3)
(35,1)
(37,2)
(43,0)
(60,2)
(67,0)
(69,1)
(72,3)
(90,1)
(100,0)
(108,2)
(116,3)
(124,0)
(126,2)
average waiting time:7
```

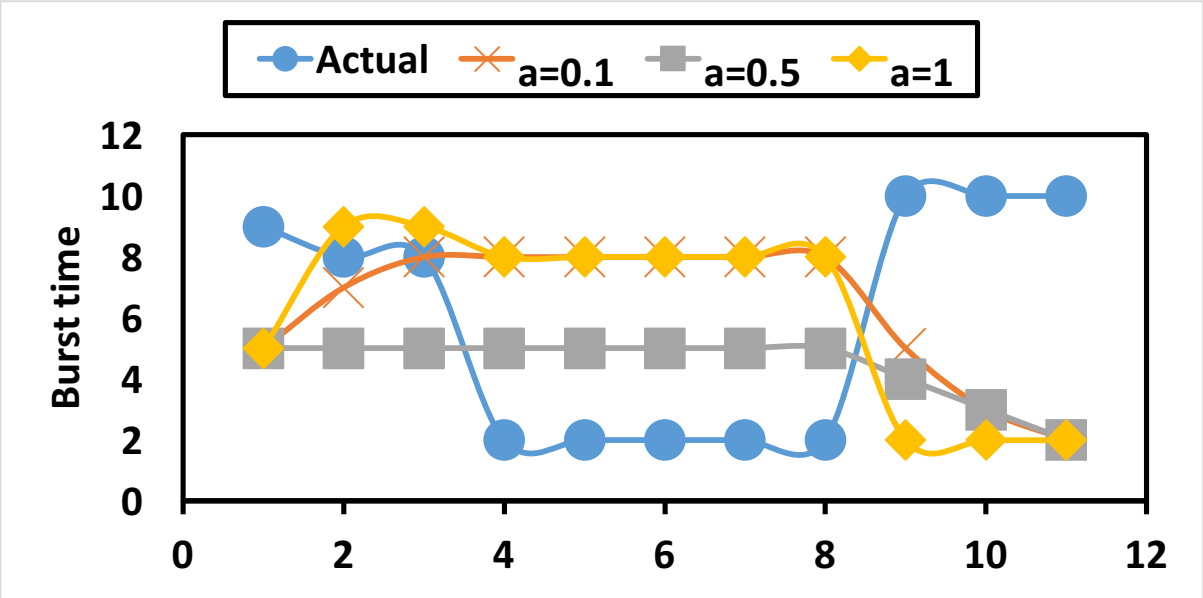
2. Test with given input with SJF

For the given input, SJF performs the same under varying α.

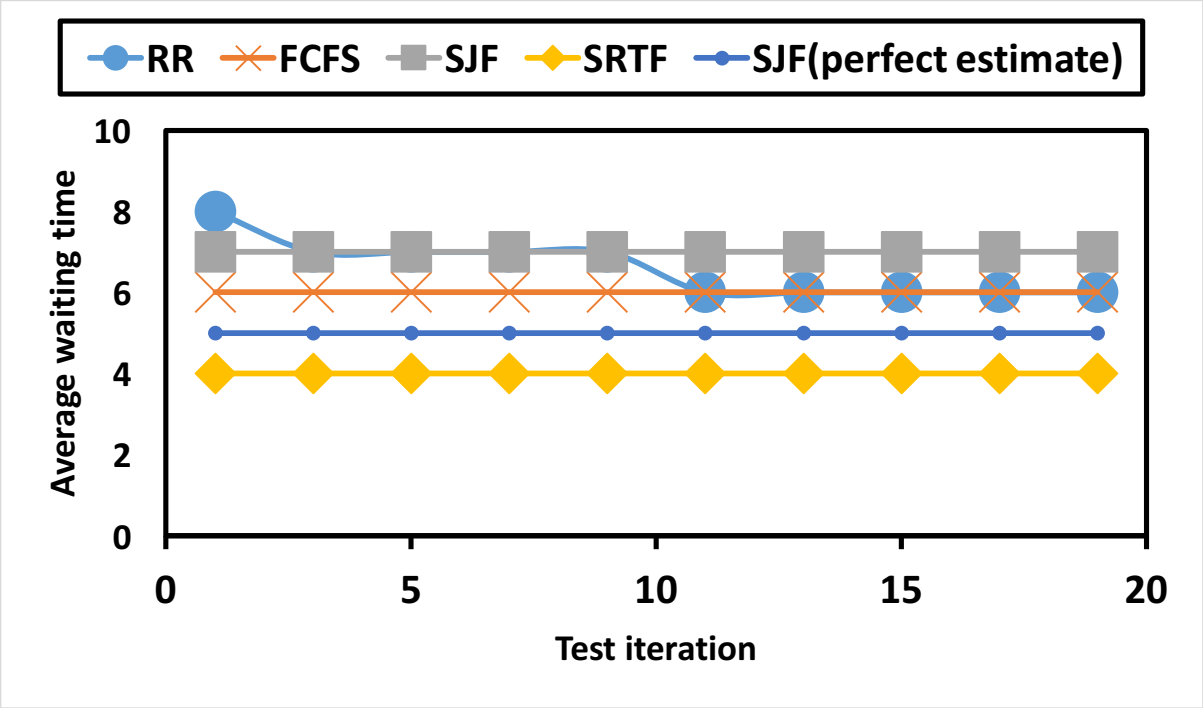
```
printing output ---- (current time, process id)
(0,0)
(9,1)
(17,2)
(19,3)
(30,3)
```

```
(35,2)
(41,1)
(43,0)
(60,2)
(67,1)
(70,3)
(78,0)
(90,1)
(100,0)
(110,2)
(119,3)
average waiting time:7
```

To illustrate the effect of varying α, I show the changes of burst estimation by using process 0 (pid=0) as an example.
The following figure illustrates the estimation of burst time under varying α from 0.1 to 1. As process 0 has a significantly varying burst time, none of the estimation has a good match with the actual.



Finally, the following figure shows the average waiting time of different schemes under varying parameters of Q (from 1 to 20) and α (from 0.1 to 1) .



When Q is set to 6, RR achieves the minimum average waiting time of 6. In fact, RR becomes FCFS when Q is large.

While, for SJF, it always has an average waiting time of 7 no matter how to tune α. As a sanity check, I have tested SJF with a ``perfect'' estimator that always gives correct burst time, and it gives an average waiting time of 5, which is still worse than SRTF (optimal) because of the non-preemptiveness.

Question 2: What is the optimal scheduling scheme (gives minimum average waiting time) for a system with (4 marks):

SRTF will always give the minimum average waiting time for both cases, as it gives the theoretical bounded optimal scheduling. We can use SRTF to measure the optimality of other schemes.

In theory, all schemes are fine for all short processes, while SJF should perform better for a mix of different length of processes.

For this test, I create two methods to generate two synthetic workloads to test.
All_short_process() will generate process with burst time ranging from 1 to 5.

```
public ProcessInput all_short_process() {
   int entry = 50;//entries
   Random r = new Random();
   int current_time = 0;
   process_list.add(new Process(0, 0, 1));

   for (int i = 0; i < entry; i++) {
      int id = r.nextInt(3);
      int arrival_time = current_time + r.nextInt(2);
      current_time = arrival_time;
      int burst_time = 1 + r.nextInt(4);
      process_list.add(new Process(id, arrival_time, burst_time));
   }
   return this;
}
```

While, interleave_proces() will generate processes with burst time ranging from 1 to 101.

```
public ProcessInput interleave_process() {
   int entry = 50;//number entries in the input process list
   Random r = new Random();
   int current_time = 0;
   process_list.add(new Process(0, 0, 1));

   for (int i = 0; i < entry; i++) {
      int id = r.nextInt(3);
      int arrival_time = current_time + r.nextInt(10);
      current_time = arrival_time;
      int burst_time = 1 + r.nextInt(100);
      process_list.add(new Process(id, arrival_time, burst_time));
   }
   return this;
}
```

I tested all schemes with varying parameters (i.e., Q and a) with the synthetic generated input 100 times for each case. In general, the results matches my hypothesis.

For all short process case,
All three schemes have an average waiting time of 1.4~1.44 times of SRTF.

On the other hand, for interleave process case,
FCFS and RR has an average waiting time of ~1.57 times of SRTF.
SJF has an average waiting time of ~1.46 times of SRTF.

Question 3:

As the process only requires burst time on 1 core, it is easy to extend the scheduler. In particular, there are two possible ways to extend.

First is to have one scheduler of each core, where the scheduler is identical to the one I have implemented. The process will then first be dispatched to one of the scheduler, and then get scheduled as usual. This design is simple to extend based on current implementation, but it lefts the question of how to dispatch to which scheduler, and if one core is being assigned too much, it needs some kinds of task stealing mechanism to improve the overall system performance.

Second is to have a modified version of scheduler that recognizes multicore environment. For example, the scheduler will take multiple process and schedule them into multiple cores concurrently. This design would require a bit re-write of the current implementation. In this case, concurrent updates of the ready queue could be a performance bottleneck of the scheduler.