CS 406: Operating Systems

Spring 2016

Lafayette College

Programming Project 2: Schedulers

Final Report

Submitted by: Renan Dincer and Ivan Evtimov

Course Instructor: Prof. Amir Sadovnik

Contents

Introduction

One of the key functions of an operating system is to allocate hardware resources among competing processes in an efficient way. The Central Processing Unit is probably the most important of these resources and access to it is controlled by schedulers. These pieces of kernel code determine which member out of a set of processes ready to perform computation gets to run on the CPU at each time step. Scheduler algorithms can be designed to optimize different performance variables, but this project will focus on examining waiting time and response time. The former one measures how much time a process spends in the set of ready processes while the second one measures the difference between the time of arrival in the ready set and the time the process first gets to execute.

Scheduling algorithms

This project will explore 7 different scheduling algorithms that try to solve the problem described above. These can generally be divided in two categories. *Preemtpive* algorithms can replace the currently running process if the algorithm deems it necessary. *Non-preemptive* algorithms never interrupt a process running on the CPU before it is complete and only decide the next one to run once the current one is done.

First Come First Serve: This is a *non-preemptive* scheduler that picks the next process to run based on the arrival order.

Shortest Job First: This is a *non-preemptive* scheduler that picks the next process to run based on estimated burst time (processes that are estimated to complete faster go first).

Shortest Remaining Time First : This is a *preemtpive* scheduler. It always selects the process who has less time remaining to complete to run on the CPU. This also applies for the process currently running on the CPU – if it has more time remaining than a process in the ready set, it is switched out.

Priority : This scheduler can be either *preemptive* or *non-preemptive*. In either case, it guarantees that the process with the highest preassigned priority is the one currently running on the CPU.

Round Robin : This scheduler lets each process run only for a prespecified `quantum` (common for all processes). After a process' time on the CPU is up, it is put back in the ready set and a new one is selected to run.

Custom: We implement a custom scheduling algorithm that is *non-preemptive* and picks processes on a Last Come, First Serve basis.

Solution design

General idea

The "meat" of the solution to the problem is the following: At each time step, our scheduler looks at whether the currently running process needs to be switched out (because it is done running or because the algorithm decided the process needs to be preempted). If that is true, a context switch is performed; otherwise, the same process keeps running.

The `Controller` class

The entry point for our program is the `Controller` class. It scans and parses the specified file, keeps track of time and determines when processes need to arrive from a "future queue" `fq` to a ready queue (held in the `Scheduler` class).

A simple `make` command compiles our program, upon which our solution can be run as follows:

```
$ java Controller <scheduling_algorithm> <input_file>
```

The <scheduling_algorithm> field can be any one of these: `fcfs`, `sjf`, `srtf`, `nonpreprior`, `preprior`, `rr` (codes corresponding to names specified in the assignment document). The default case - if the field is not one of the above - would let the program use our custom scheduler, but our program assumes that argument is a non-empty string.

The <input_file> field can be a path to any accessible text file that follows the pattern in the assignment document (no spaces between commas). Our solution makes no guarantees for successful execution if the input file deviates from that format.

The `Controller` creates a subclass of `Scheduler` depending on the input from the user - for example, if we want to run the FCFS scheduler, an `FCFSScheduler` object will be created (more on `Scheduler` design further down).

The `Controller` class keeps track of all the incoming processes by putting them in a `PriorityQueue` called `fq` in which processes are ordered according to their arrival time. In each time period, new processes are passed to the scheduler via its `arrive` method and the scheduler returns a completed process (if a process finished executing in the previous time step). These completed processes are added to an `ArrayList` called `doneQ` which is used in the computation of the final statistics. When the total number of added processes is equal to the number of done processes, the clock stops working and the stats are computed (`calculateStats`) and printed out (`printStats`).[1]

The `Scheduler` class and its subclasses

We have created an abstract `Scheduler` class for all types of schedulers to inherit from. This abstract class provides a shared `run` method that is called in each time step and handles the context switch and updates processes both in the ready queue and in the CPU to reflect how long they have been running or waiting for. The "context switch" is performed in the `preempt` method and the current process (stored in an instance variable of `Scheduler`) is put back on the ready queue while the process at the top of the ready queue gets to "run". The "ready queue" in question is the instance variable `rq` and it is up to subclasses to initialize it with a comparator that fits the needs of the algorithm they are implementing.

The schedulers also establish custom behavior by overriding the `contextSwitchable`, `updateCurrent`, `updateRQ`, `preempt` methods. Not all methods are used or overridden in all schedulers, but the `Scheduler` abstract class provides the functionality most of them share "out of the box."

The `FCFSScheduler` works by using the `ArrivalTimeComparator`. It establishes process ordering based on arrival time so that each time we `poll` the priority queue (in the `preempt`) method), the next process in order of arrival is put on the CPU. Since FCFS is a non-preemptive scheduling algorithm, the `contextSwithcable` method returns false all the time except when there is no process in the CPU (the CPU is idle). Thus, the `preempt` method only gets called when a process is done running.

In a similar fashion, the `NonpreprioScheduler` uses the `PriorityComparator` and doesn't `preempt` unless a process is done running. One distinguishing feature of that algorithm is aging. This functionality is implemented by overriding the default

[1]This is an area where our code violates good software design principles as both methods work on the global variables `awt` (average waiting time), `awwt` (average weighted waiting time), `art` (average response time) and `awrt` (average weighted response time) rather than passing them around. For `printStats` to print correct statistics, `calculateStats` needs to be called first. We guarantee that by calling `calculateStats` in the first line of execution in `printStats`.

updateRQ method to increase the priority of all processes in the ready queue at each time step.

The PrepriorScheduler also employs the PriorityComparator in building its ready queue, but preempts if there is a higher priority process waiting at the top of the ready queue in each time period. The process that is preempted is added back to the ready queue for later consideration in similar fashion.

In the RRScheduler, a comparator that always returns 0 (which means the processes are equal) is used. This allows the priority queue to run in a simple queue style as the preempted processes are added to the back of the line as long as there are more than one process in the scheduler.

The SJFScheduer is based on the burst time of processes in the ready queue. Since this is a non-preemptive algorithm, the only functionality of the scheduler is to poll the head of the ready queue each time a process is done in order to decide which process runs next.

The SRTFScheduler performs two actions:

- to determine whether a context switch is necessary, it looks at the head of the ready queue rq and compares that process' remaining time to the current running process' remaining time. If the current has less time remaining it is left to run and contextSwitchable returns false. Otherwise, the process at the head of the queue gets to run on the CPU. Note that the process at the head of the queue is guaranteed to have the shortest remaining time among all processes in the ready queue because rq is implemented with a comparator based on timeRemaining.
- performs a regular context switch if necessary

Our CustomScheduler is a Last Come, First Served scheduler, so it employs the AntiBurstTimeComparator to select the most recently arrived process to run next once the previous is done (it is non-preemptive).

All of these schedulers are implemented using priority queues with different comparators that push processes up the queue in different order. Since all comparators compare in one criteria, ties are broken using the order of entry into the queue[2]. All schedulers use one of the following comparators:

- AntiArrivalTimeComparator: Last come highest priority
- ArrivalTimeComparator: First come highest priority
- BullshitComparator: All the same priority (everyone gets a chance) – the ready queue degenerates to a simple queue in this case
- BurstTimeComparator: Smallest job is highest priority
- PriorityComparator: Largest priority has highest priority
- TimeRemainingComparator: Shortest time remaining highest priority

[2]The Java PriorityQueue is essentially a heap so the ordering is based on the processes' location in that heap. That is, the top element will be the one with guaranteed lowest property, but all elements that are "equal" based on the comparator, will be in the order in which they came in the heap.

Finally, the `Process` class is a representation of a process and holds relevant fields such as `waitingTime` and `responseTime`. All of its instance variables can be accessed with accessor methods that carry the same name as the field of the object. However, only custom mutators are defined based on the needs for different algorithms. The method `cpuTime` modifies `responseTime` and `timeRemaining` to simulate execution on a real processor. The `waitForCPU` method increases the process' waiting time while the `increasePriority` "ages" the process.

Correctness Results

Input File

For correctness confirmation, we made use of the following short input file:

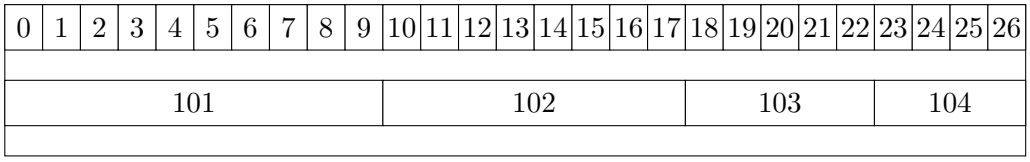
```
101,10,0,1
102,8,0,3
103,5,5,5
104,4,7,10
```

When ran with the respective algorithms, this produced the following results:

Trial Runs

As can be seen from the trial runs below, our predicted scheduling matches our actual outputs, so we can conclude that the program works correctly.

First Come, First Serve



process	wait time	response time
101	0	0
102	8	8
103	13	13
104	18	18
average:	9.75	9.75

```
Time: 0, process 101 running
Time: 1, process 101 running
Time: 2, process 101 running
Time: 3, process 101 running
Time: 4, process 101 running
Time: 5, process 101 running
Time: 6, process 101 running
Time: 7, process 101 running
```

```
Time: 8, process 101 running
Time: 9, process 101 running
Time: 10, process 102 running
Time: 11, process 102 running
Time: 12, process 102 running
Time: 13, process 102 running
Time: 14, process 102 running
Time: 15, process 102 running
Time: 16, process 102 running
Time: 17, process 102 running
Time: 18, process 103 running
Time: 19, process 103 running
Time: 20, process 103 running
Time: 21, process 103 running
Time: 22, process 103 running
Time: 23, process 104 running
Time: 24, process 104 running
Time: 25, process 104 running
Time: 26, process 104 running
Average waiting time is:9.75
Average weighted waiting time is:13.421052631578947
Average response time is:9.75
Average weighted response time is:13.421052631578947
```

Shortest Job First

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
102								104				103							101							

process	wait time	response time
101	17	17
102	0	0
103	7	7
104	1	1
average:	6.25	6.25

```
Time: 0, process 102 running
Time: 1, process 102 running
Time: 2, process 102 running
Time: 3, process 102 running
Time: 4, process 102 running
Time: 5, process 102 running
Time: 6, process 102 running
Time: 7, process 102 running
Time: 8, process 104 running
```

```
Time: 9, process 104 running
Time: 10, process 104 running
Time: 11, process 104 running
Time: 12, process 103 running
Time: 13, process 103 running
Time: 14, process 103 running
Time: 15, process 103 running
Time: 16, process 103 running
Time: 17, process 101 running
Time: 18, process 101 running
Time: 19, process 101 running
Time: 20, process 101 running
Time: 21, process 101 running
Time: 22, process 101 running
Time: 23, process 101 running
Time: 24, process 101 running
Time: 25, process 101 running
Time: 26, process 101 running
Average waiting time is:6.25
Average weighted waiting time is:3.263157894736842
Average response time is:6.25
Average weighted response time is:3.263157894736842
```

Shortest Remaining Time First

Note that with the given input file SRTF "degenerates" to SJF.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
102								104				103					101									

process	wait time	response time
101	17	17
102	0	0
103	7	7
104	1	1
average:	6.25	6.25

```
Time: 0, process 102 running
Time: 1, process 102 running
Time: 2, process 102 running
Time: 3, process 102 running
Time: 4, process 102 running
Time: 5, process 102 running
Time: 6, process 102 running
Time: 7, process 102 running
Time: 8, process 104 running
```

```
Time: 9, process 104 running
Time: 10, process 104 running
Time: 11, process 104 running
Time: 12, process 103 running
Time: 13, process 103 running
Time: 14, process 103 running
Time: 15, process 103 running
Time: 16, process 103 running
Time: 17, process 101 running
Time: 18, process 101 running
Time: 19, process 101 running
Time: 20, process 101 running
Time: 21, process 101 running
Time: 22, process 101 running
Time: 23, process 101 running
Time: 24, process 101 running
Time: 25, process 101 running
Time: 26, process 101 running
Average waiting time is:6.25
Average weighted waiting time is:3.263157894736842
Average response time is:6.25
Average weighted response time is:3.263157894736842
```

Priority with Aging (non-preemptive)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
102								104				103					101									

process	wait time	response time
101	12	12
102	0	0
103	17	7
104	1	1
average:	7.5	7.5

```
Time: 0, process 102 running
Time: 1, process 102 running
Time: 2, process 102 running
Time: 3, process 102 running
Time: 4, process 102 running
Time: 5, process 102 running
Time: 6, process 102 running
Time: 7, process 102 running
Time: 8, process 104 running
Time: 9, process 104 running
```

```
Time: 10, process 104 running
Time: 11, process 104 running
Time: 12, process 101 running
Time: 13, process 101 running
Time: 14, process 101 running
Time: 15, process 101 running
Time: 16, process 101 running
Time: 17, process 101 running
Time: 18, process 101 running
Time: 19, process 101 running
Time: 20, process 101 running
Time: 21, process 101 running
Time: 22, process 103 running
Time: 23, process 103 running
Time: 24, process 103 running
Time: 25, process 103 running
Time: 26, process 103 running
Average waiting time is:7.5
Average weighted waiting time is:5.631578947368421
Average response time is:7.5
Average weighted response time is:5.631578947368421
```

Priority with Aging (preemptive)

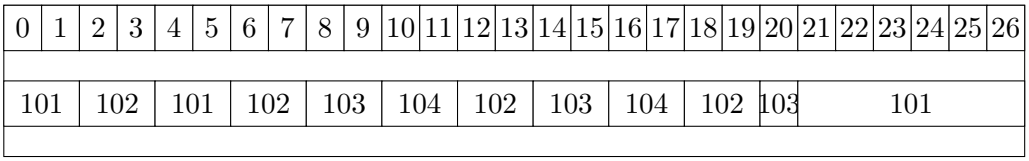
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
2		1	2	1	2	3		4			3	2	1	2	1	3	1	3	2	3	2		1			

process	wait time	response time
101	17	2
102	14	0
103	11	1
104	0	0
average:	10.5	0.75

```
Time: 0, process 102 running
Time: 1, process 102 running
Time: 2, process 101 running
Time: 3, process 102 running
Time: 4, process 101 running
Time: 5, process 102 running
Time: 6, process 103 running
Time: 7, process 104 running
Time: 8, process 104 running
Time: 9, process 104 running
Time: 10, process 104 running
```

```
Time: 11, process 103 running
Time: 12, process 102 running
Time: 13, process 101 running
Time: 14, process 102 running
Time: 15, process 101 running
Time: 16, process 103 running
Time: 17, process 101 running
Time: 18, process 103 running
Time: 19, process 102 running
Time: 20, process 103 running
Time: 21, process 102 running
Time: 22, process 101 running
Time: 23, process 101 running
Time: 24, process 101 running
Time: 25, process 101 running
Time: 26, process 101 running
Average waiting time is:10.5
Average weighted waiting time is:6.0
Average response time is:0.75
Average weighted response time is:0.3684210526315789
```

Round Robin with Quantum 2



process	wait time	response time
101	17	0
102	12	2
103	11	3
104	7	3
average:	11.75	2.00

```
Time: 0, process 101 running
Time: 1, process 101 running
Time: 2, process 102 running
Time: 3, process 102 running
Time: 4, process 101 running
Time: 5, process 101 running
Time: 6, process 102 running
Time: 7, process 102 running
Time: 8, process 103 running
Time: 9, process 103 running
Time: 10, process 104 running
Time: 11, process 104 running
```

```
Time: 12, process 102 running
Time: 13, process 102 running
Time: 14, process 103 running
Time: 15, process 103 running
Time: 16, process 104 running
Time: 17, process 104 running
Time: 18, process 102 running
Time: 19, process 102 running
Time: 20, process 103 running
Time: 21, process 101 running
Time: 22, process 101 running
Time: 23, process 101 running
Time: 24, process 101 running
Time: 25, process 101 running
Time: 26, process 101 running
Average waiting time is:11.75
Average weighted waiting time is:9.368421052631579
Average response time is:2.0
Average weighted response time is:2.6842105263157894
```

Round Robin with Quantum 5

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
101					102					103					104					102			101			

process	wait time	response time
101	17	0
102	14	5
103	5	5
104	8	8
average:	11	4.5

```
Time: 0, process 101 running
Time: 1, process 101 running
Time: 2, process 101 running
Time: 3, process 101 running
Time: 4, process 101 running
Time: 5, process 102 running
Time: 6, process 102 running
Time: 7, process 102 running
Time: 8, process 102 running
Time: 9, process 102 running
Time: 10, process 103 running
Time: 11, process 103 running
Time: 12, process 103 running
```

```
Time: 13, process 103 running
Time: 14, process 103 running
Time: 15, process 104 running
Time: 16, process 104 running
Time: 17, process 104 running
Time: 18, process 104 running
Time: 19, process 102 running
Time: 20, process 102 running
Time: 21, process 102 running
Time: 22, process 101 running
Time: 23, process 101 running
Time: 24, process 101 running
Time: 25, process 101 running
Time: 26, process 101 running
Average waiting time is:11.0
Average weighted waiting time is:8.631578947368421
Average response time is:4.5
Average weighted response time is:6.315789473684211
```

Last Come, First Serve

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
101					103		104				103			101					102								

process	wait time	response time
101	9	0
102	19	19
103	4	0
104	0	0
average:	8	4.75

```
Time: 0, process 101 running
Time: 1, process 101 running
Time: 2, process 101 running
Time: 3, process 101 running
Time: 4, process 101 running
Time: 5, process 103 running
Time: 6, process 103 running
Time: 7, process 104 running
Time: 8, process 104 running
Time: 9, process 104 running
Time: 10, process 104 running
Time: 11, process 103 running
Time: 12, process 103 running
Time: 13, process 103 running
```

```
Time: 14, process 101 running
Time: 15, process 101 running
Time: 16, process 101 running
Time: 17, process 101 running
Time: 18, process 101 running
Time: 19, process 102 running
Time: 20, process 102 running
Time: 21, process 102 running
Time: 22, process 102 running
Time: 23, process 102 running
Time: 24, process 102 running
Time: 25, process 102 running
Time: 26, process 102 running
Average waiting time is:8.0
Average weighted waiting time is:4.526315789473684
Average response time is:4.75
Average weighted response time is:3.0
```

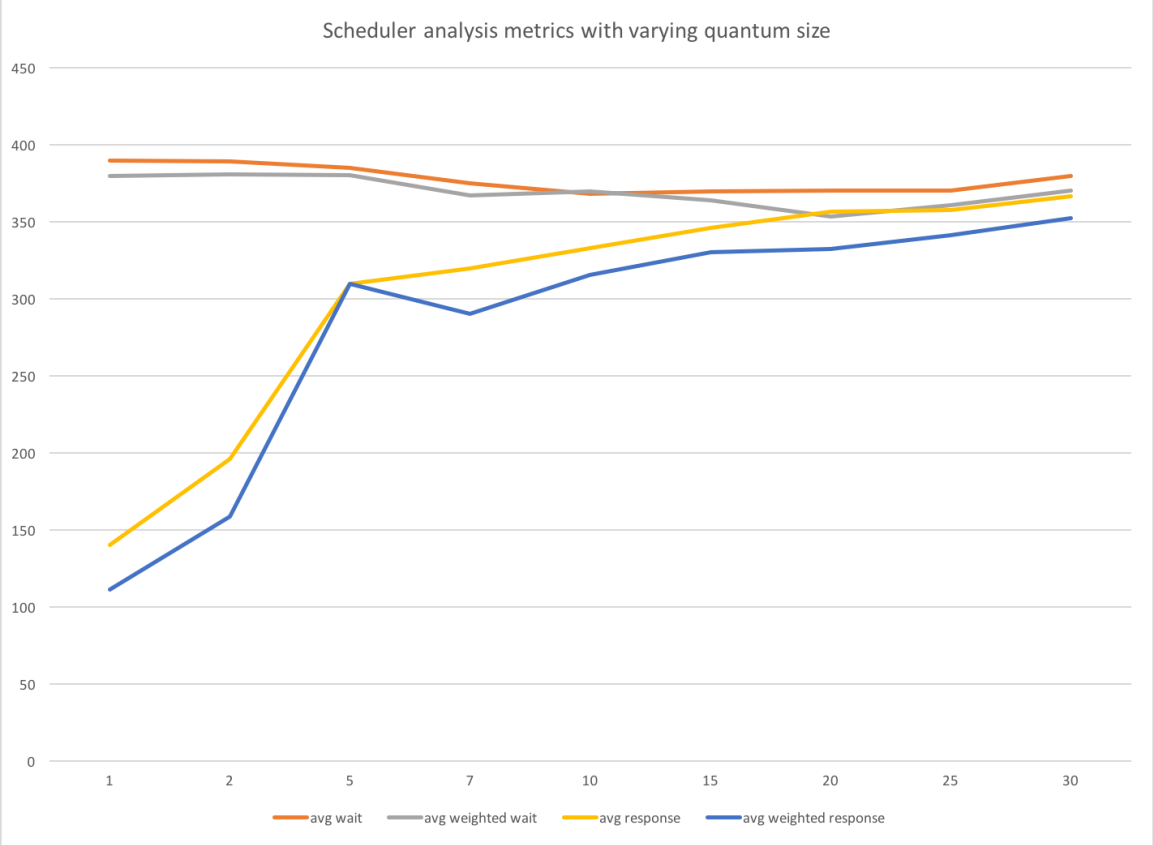
Analysis

The larger input file was generated with the python script stored in `proc_random.py` and is to be found in `bigTrial.txt`. Here are our analysis results:

algorithm	avg. wait	weighted avg. weight	avg. response	weighted avg. response
FCFS	327.14	330.14	327.14	330.14
SJF	233.36	262.46	233.36	262.46
SRTF	233.14	263.56	232.66	263.05
Nonpreprior	332.32	313.37	332.32	313.37
Preprior	390.48	387.96	225.20	209.05
RR q = 2	389.46	380.66	196.24	158.93
RR q = 5	384.90	380.38	309.78	309.70
Custom	382.66	377.43	139.42	175.51

An analysis of scheduling algorithms can be conducted over all of the data we have collected: average waiting time, weighted average wait time, average response time, weighted average response time.
On an average system, weighted average weight would be a good indicator of scheduler performance, however, in an interactive system one should also consider response times as well. Often, a process can produce some output early on and continue operating as the user is reviewing the previous results.
If we assume that the scheduler is designed for an interactive system and focus on response times, the custom last-come-first-served scheduler does the best because of promised least response time. After that, round robin with a shorter quantum does best for similar reasons of faster rotation between processes.
It is important to note the success of Preprior, priority with aging and preemption, especially because of its results in both response columns. Aging on every wait period and preemption has made this algorithm behave similar to a combination of RR and first

come first served.
On non-interactive systems one should focus on turnaround time and wait times. The least wait time is achieved by shortest job first and shortest remaining timem first algorithms, with very close results. The third best result after these algorithms is at least 100 time units more, making these two metrics clear winners. This result is similar in the averaged category. As a result, the best overall result is achieved by SJF and SRTF algorithms and in interactive systems, our custom scheduler or round robin with short quantum should be preferred.



Appendix

We were in charge of all details in the project together and used pair coding (two people on one monitor). We took turns in typing as we went along. We split up when typing up the report and worked on each section separately.

References

No references - thanks to java api