Process Scheduling and Simulation Program

This application basically took a list of jobs as input and it stimulated their execution with different scheduling algorithms and compared their statistics.

We took an object oriented approach to implement this program, where each class would only do what it's assigned to do and every functionality is broken down to small pieces. Although, the application is command line application, this decoupling allows us to reuse classes in a GUI application.

Important class in the application are as following:

- Main: Will parse the command line arguments, and initialize all other classes, and handle the main execution
- JobManager: Will contain the jobs and is in charge of running them and will update the other jobs waiting statistics.
- Algorithm class: They will use jobManager to run each job for a given amount of time with their own scheduling algorithm.
- Job: Will contain the job information and it's statistics. It is in charge of updating its own statistics when waiting and running.

With this decoupling, there's the least amount of code duplication and less bugs. Also it's component is unit tested to make sure there won't be any bugs and behaviour is easily verified.

The main part of the application basically boils down to this(From Main.java):

```
compareAlgorithms(algorithms.stream().map(algorithm -> {
    System.out.println("Running " + algorithm.getName());
    algorithm.run();
    printStatistics(jobManager);
    Statistics.Result result = jobManager.getResult(algorithm.getName());
    executionChartCreator.print();
    jobManager.reset();
    executionChartCreator.reset();
    return result;
}).collect(Collectors.toList()));
```

This will basically first, run each algorithm, print it's statistics and print the execution chart. Then it will reset the statistics and collected data to be used for the next algorithm run, and then compareAlgorithms function will use the function and print a table and charts to compare each algorithm results.

Algorithms class

Since algorithms are just in charge of order of execution and not anything else, they will basically use jobManager.runJob function to execute the job and jobManager will take care of the updating the statistics.

First Come First Served

Here's what the first-come-first-served algorithm work:

```
for (Job job : jobManager.getJobs()) {
    jobManager.runJob(job, Integer.MAX_VALUE);
}
```

As you see the code is very simple. Note that `Integer.MAX_VALUE` is passed as the second argument of runJob which indicates how many can the job be executing for, which in this infinity. The job itself is intelligent enough to only execute for it's own duration. This argument is later used to make the round robin scheduling simpler.

Shortest Job First

The for this algorithm just does what the name suggest, it will sort the jobs by duration and run the from the smallest to longest:

```
ArrayList<Job> jobs = new ArrayList<>(jobManager.getJobs());
jobs.sort((o1, o2) -> o1.getDuration() - o2.getDuration());

for (Job job : jobs) {
    jobManager.runJob(job, Integer.MAX_VALUE);
}
```

As you see the jobs are sorted and simply it will run them.

Here's a sample of the execution chart of shortest job first:

```
_____________________________________________________________________________
_____________
|5 | 2 | 3 | 7 |  9  |  1  |   8   |    6    |    4     |    10    |
_____________________________________________________________________________
_____________
```

As evident in the chart, shortest jobs are executed first.

Round robin

This algorithm implementation is slightly more complicated than the previous algorithms, since it keep switching between jobs.

```
        boolean finished;
        do {
            finished = true;
            for (Job job : jobs) {
                if (!job.isFinished()) {
                    jobManager.runJob(job, quantum);
                }
                if (!job.isFinished()) {
                    finished = false;
                }
            }
        } while (!finished);
```

In this case the round robin will keep running the unfinished jobs till are jobs are finished. Note that the `quantum` is passed as the second argument of runJob which means the job will only be run for the given amount of ticks(quantum). In the previous cases we passed Integer.MAX_VALUE to indicate that there's no limit on the job execution. You can set the quantum time with `--quantum` option.

JobManager and Job classes

Job class is will basically have `run` and `wait` method which be called to update their statistics properly in the simulation:

```
    /**
     * Called when the scheduler decides it's time for this job to be running for a
     * specified number of ticks
     * @return the number of ticks to be processed.
     */
    public int run(int maxTicks) {
        if (remaining < maxTicks) {
            maxTicks = remaining;
        }
        remaining -= maxTicks;
        periodRunning += maxTicks;
        return maxTicks;
```

```
    }

    /**
     * Must be called when the job was waiting for a certain number of ticks
     */
    public void onWait(int ticks) {
        if (remaining != 0) {
            waitingTime += ticks;
            periodWaiting += ticks;
        }
    }
```

JobManager will just handle the execution and waiting of jobs:

```
public class JobManager extends Statistics {
...
    /**
     * Will run a given job with a specified amount of max ticks.
     * @param job
     * @param maxTicks
     * @return returns the number of ticks that was processed.
     */
    public int runJob(Job job, int maxTicks) {
        LOGGER.setLevel(Level.ALL);
        int processed = job.run(maxTicks);
        for (Job j : jobs) {
            if (j != job) {
                j.onWait(processed);
            }
        }
        if (onJobRunCallback != null) {
            onJobRunCallback.call(job, processed, maxTicks);
        }

        if (verbose) {
            System.out.println("Job " + job.getId() + " ran for " + processed + "
ticks");
        }

        return processed;
    }
...
```

```
}
```

This organization of the application helps us to easily reason about execution flow and ensure edge cases are handled well.

Other Features

To make the application more usable we have added a few more features.

Tables

Tables were generated using “ASCII Table” library(Van Der Meer 2016). This library helped saved us from writing a lot of code and it generated nice tables. Here’s an example of a code that creates the table:

```
V2_AsciiTable at = new V2_AsciiTable();
at.addRule();
at.addRow("Algorithm", "Average Turnaround Time", "Average Waiting Time");
for (Statistics.Result result : results) {
    at.addRule();
    at.addRow(result.name, decimalFormat.format(result.turnAroundTime),
            decimalFormat.format(result.waitingTime));
}
at.addRule();

System.out.println(rend.render(at));
```

Basically addRule will create a table line and addRow will insert a row with some cells.

Charts

We could not find any library to help us render the charts, so we wrote our own classes to help us.

Gantt/Execution Chart

This was divided into two parts. ExecutionChartCreator will basically extract the executions data from the JobManager callback and then it will pass it GanttChartCreator. GanttChartCreator will first calculate the smallest piece of the chart so it will scale the other pieces with the same size:

```
        // We first calculate the smallest piece of the chart
        int biggestText = 0;
        int smallestWidth = Integer.MAX_VALUE;
        for (GanttChartCreator.Entry entry : entries) {
            biggestText = Math.max(biggestText, entry.text.length());
```

```
            smallestWidth = Math.min(smallestWidth, entry.width);
        }
        double cellInChars = Math.max(smallestWidth, biggestText);
        double cellTicks = smallestWidth;
```

Once that's down, we will render every entry:

```
        // Build the strings so then later we can do wrapping.
        for (Entry entry : entries) {
            int chars = (int) Math.round(((double) entry.width) *
                    cellInChars / cellTicks);
            assert entry.text.length() <= chars;
            for (int i = 0; i < (chars + 1); i++) {
                top.append("—");
                bottom.append("—");
            }
            middle.append(StringUtils.center(entry.text, chars))
                  .append("|");
         }
```

And then print to console with proper wrapping.

```
        do {
            int len = Math.min(top.length(), (int) width);
            System.out.println(top.substring(0, len));
            top.delete(0, len);
            System.out.println(middle.substring(0, len));
            middle.delete(0, len);
            System.out.println(bottom.substring(0, len));
            bottom.delete(0, len);
        } while (top.length() > 0);
```

This chart also has a verbose mode where it will include the number of ticks executed, which can be enabled by `--verbose-chart` command line option. This is disabled by default as in most cases it will make the chart harder to comprehend.

Bar Chart

This was created to just visually show the user how the averages were different.

```
Waiting time chart:
round-robin             | ############################################################
shortest-job-first      | ################################
first-come-first-served | ################################################

Turnaround time chart:
round-robin             | ############################################################
shortest-job-first      | ####################################
first-come-first-served | ################################################
```

```
int maxKey = map.keySet().stream().mapToInt(String::length).max().getAsInt();
double maxLen =
        map.values().stream().mapToDouble(value -> value).max().getAsDouble();
int rest = width - maxKey;
map.forEach((key, length) -> {
    System.out.print(StringUtils.rightPad(key, maxKey) + " | ");
    int barWidth = (int) Math.round(length / maxLen * rest);
    for (int i = 0; i < barWidth; i++) {
        System.out.print("#");
    }
    System.out.println();
});
```

This was created by just calculating the maximum number of characters available and also the maximum length of any bar:

```
int maxKey = map.keySet().stream().mapToInt(String::length).max().getAsInt();
double maxLen =
        map.values().stream().mapToDouble(value -> value).max().getAsDouble();
int rest = width - maxKey;
map.forEach((key, length) -> {
    System.out.print(StringUtils.rightPad(key, maxKey) + " | ");
    int barWidth = (int) Math.round(length / maxLen * rest);
    for (int i = 0; i < barWidth; i++) {
        System.out.print("#");
    }
    System.out.println();
});
```

Command line options

We made most things in the application configurable via command line options. This was achieved using JCommander library(Beust 2012). You can view the command line options via `--help` option, which would basically show you the following:

```
Usage: <main class> [options]
  Options:
    --help
       You're looking at what it does.
       Default: false
    --job
       duration[:priority], specifies a job with duration given in ticks and a
       an optional priority which by default is 1. You should specify this argument
       multiple times to have multiple jobs. For example: `--job 1:2 --job 3:3`
       Default: []
    --max-duration
       Maximum allowed duration for the randomly generated jobs
       Default: 10
    --max-priority
       Maximum priority for the randomly generated jobs
       Default: 10
    --quantum
       Time quantum for round robin algorithm
       Default: 3
    --random
       Will make the application to randomly generate jobs. By default it will
       generate 10 random jobs, with max duration of 10 and max priority of 10. You can
       customize these with --total-jobs, --max-duration and --max-priority options, each
       expects an integer value
       Default: false
    --total-jobs
       The number of jobs to be randomly generated.
       Default: 10
    --verbose
       Some verbose logging to see what it does internally
       Default: false
    --verbose-chart
       Make the execution chart contain more information.
       Default: false
```

Thanks to JCommander this was very easily done. Here's an example of definition of one of the options

```
@Parameter(names = "--random", description = "Randomly generate the resources that"
+
        "each job want")
private boolean random;
```

JCommander later will populate the fields of the main class by the options provided.

Conclusion

Creating and using this application helped to understand the advantages and disadvantages the different algorithms. We can see that shortest job first algorithm even though simple, was very efficient since it was having much better statistics compared to others.

Round robin most of the time perform the worst as it's done in a way that will make jobs finish close to each therefore having longer turnaround and waiting times. The advantage of round robin is when the response time of each job is important which was out of scope for this project.
First come first served is only as good as using it to compare other algorithms efficiency and shouldn't be used due to a fact that having a long job in the beginning will block all other jobs.

Sample input/output

You can run the application like the following:

```
$ java -jar G52OSC.jar --job 2:4 --job 8:4 --job 5 --job 3 --job 8
```

Where --job 2:4 would mean a job with a duration of 2 and priority 4. You can also run the app in random mode where data is randomly generated:

```
$ java -jar G52OSC.jar --random
```

Check next page for sample output.

Running first-come-first-served

Job id	Duration	Priority	Turn Around Time	Waiting Time
1	2	4	2.0	0.0
2	8	4	10.0	2.0
3	5	1	15.0	10.0
4	3	1	18.0	15.0
5	8	1	26.0	18.0
Average			14.2	9

Job Execution Chart:

```
|1 |   2    |  3  | 4 |   5    |
```

Running shortest-job-first

Job id	Duration	Priority	Turn Around Time	Waiting Time
1	2	4	2.0	0.0
2	8	4	18.0	10.0
3	5	1	10.0	5.0
4	3	1	5.0	2.0
5	8	1	26.0	18.0
Average			12.2	7

Job Execution Chart:

```
|1 | 4 |  3  |   2    |   5    |
```

Running round-robin

Job id	Duration	Priority	Turn Around Time	Waiting Time
1	2	4	2.0	0.0
2	8	4	24.0	16.0
3	5	1	19.0	14.0
4	3	1	11.0	8.0
5	8	1	26.0	18.0
Average			16.4	11.2

Job Execution Chart:

```
|1 | 2 | 3 | 4 | 5 | 2 |3 | 5 |2 |5 |
```

Algorithm	Average Turn Around Time	Average Waiting Time
first-come-first-served	14.2	9
shortest-job-first	12.2	7
round-robin	16.4	11.2

```
Waiting time chart:
round-robin             | ##########################################################
shortest-job-first      | ##################################
first-come-first-served | ###########################################

Turnaround time chart:
round-robin             | ##########################################################
shortest-job-first      | ########################################
first-come-first-served | ################################################
```

Deadlock Detection and Recovery Simulation

This application simulated a situation where multiple processes tries to allocate one resource at a time till reaching allocating all the resources they need and then exiting, but since resources are shared and limited there will times a process has to wait for another to finish, which might cause a deadlock in some cases. In this application we will detect the deadlock when it happens and then try to resolve it by killing the process, releasing its resources and then restarting it again, once another process was done with the resources.

This application has two threads. The main thread which does the allocation and the other thread which handles deadlock detection and recovery.

Main Thread

```
        while (true) {
            ArrayList<Process> allocatableProcesses = new ArrayList<>();
            if (Arrays.stream(processes).allMatch(Process::fulfilled)) {
                System.out.println("All processes are finished");
                System.exit(0);
                return;
            }
            for (Process process : processes) {
                if (process.wants()) {
                    allocatableProcesses.add(process);
                }
            }

            if (allocatableProcesses.size() == 0) {
                System.out.println("We are in a deadlock. Waiting for deadlock
detector" +
                        " thread to resolve this.");
            } else {
                Process process = Utils.sample(allocatableProcesses);
                process.allocate();
                if (process.fulfilled()) {
                    System.out.println("Reviving all killed processes");
                    Arrays.stream(processes).filter(Process::isKilled)
                            .forEach(Process::reset);
                }
            }
```

```
            long sleep = (long) (Math.random() * maxSleep * 1000);
            Thread.sleep(sleep);
        }
```

The main thread on each loop, will find all processes that can do allocation(ie. Not waiting for resources to be available) and then randomly choose one. Which it will allocate one random resource from that process. Then it will check if the process was finished or not, in case if it was then it will revive killed processes so they can finish their work.

Deadlock detection thread

```
        while (true) {
            ArrayList<Process> allocatableProcesses = new ArrayList<>();
            if (Arrays.stream(processes).allMatch(Process::fulfilled)) {
                System.out.println("All processes are finished");
                System.exit(0);
                return;
            }
            for (Process process : processes) {
                if (process.wants()) {
                    allocatableProcesses.add(process);
                }
            }

            if (allocatableProcesses.size() == 0) {
                System.out.println("We are in a deadlock. Waiting for deadlock
detector" +
                        " thread to resolve this.");
            } else {
                Process process = Utils.sample(allocatableProcesses);
                process.allocate();
                if (process.fulfilled()) {
                    System.out.println("Reviving all killed processes");
                    Arrays.stream(processes).filter(Process::isKilled)
                            .forEach(Process::reset);
                }
            }

            long sleep = (long) (Math.random() * maxSleep * 1000);
            Thread.sleep(sleep);
```

```
}
```

This thread will basically keep checking if there are any processes left that is not waiting(meaning not deadlocked) and then also it will check with bankers algorithm to see if there's going to be any deadlock in the future or not.

In case if any deadlock was detected, then a recovery will be attempted which will kill the jobs holding most resources:

```
List<Process> sorted = Arrays.stream(processes)
        .sorted((a, b) ->
                Integer.compare(b.getTotalHeld(), a.getTotalHeld()))
        .collect(Collectors.toList());
for (Process process : sorted) {
    process.kill();
    System.out.println("Killing " + process.getId() + " to recover");
    for (Process p : sorted) {
        p.resetWaiting();
    }
    if (!detectDeadlock()) {
        return true;
    }
}
return false;
```

Conclusion

Running this simulation showed us detecting and trying to recover from a deadlock in this nature will be very slow(due to multiple attempts and timeouts). Real applications should try to prevent deadlocks before they happen.

Sample input/output

BEUST, C., 2012. *JCommander.* Available from: http://www.jcommander.org/.
VAN DER MEER, S., 2016. *asciitable.* Available from: https://github.com/vdmeer/asciitable.