CS526 Term Project

Dexter Legaspi

Contents

- Overview
- Class Definitions and Implementation
  - Process
  - Process List
  - Process Scheduler
  - Custom Comparator and the Priority Queue
  - Reading the Input File
  - Logging and Writing to Output File
- Running the Application
- Observations and Lessons Learned
  - Priority Queue Behavior
  - Conclusion

Overview

This is a command-line application that simulates process scheduling using Java's default `PriorityQueue<E>` implementation. It takes an input file that has the process metadata and logs the output of the simulation to standard output and to a file.

The implementation leans heavily on the use of the new features Java 11 (like lambdas and type inference) for code brevity/conciseness and increased readability but still relies on the basic Generic types to highlight what needs to be learned in this term project.

Class Definitions and Implementation

Process

The `Process` class has all the properties that represents the processes to be run, the data in the process_scheduling_input.txt is represented in this class:

- ID
- Priority
- Duration
- Arrival Time

Getters and/or Setters are defined in the fields of class where applicable and marked `final` where appropriate.

It also has the following features:

- Means to keep track of its wait time. This is used for the final computation of the average wait time.
- Means to keep track of its execution start time, used by the `Scheduler` for tracking time when it is run and be able to determin if finished (in conjunction with the duration)

Process List (D)

The `ProcessList` (D) class extends the ArrayDeque generic instead of using it directly to add the following behaviors/features to the subclass:

1. Be able to take in a `Collection<Process>` as a parameter to the constructor. This allows me to do whatever to the list internally before adding to the queue. In this case, it sorts (creates a sorted copy) the list by arrival time before adding them. The sample happens to have the data conveniently sorted by arrival date, but this may not be always the case.
2. Since the list is modified along the way, this class has property that represents the original list size/count for computing the average wait time in `ProcessScheduler`.
3. Add a convenience creation function `getProcessListFromFile` that takes in a filename and generates the `ProcessList` instance. This allows me to make the constructor private because there is no need for it to be public, less risk of bugs for misusing the constructor.
4. Override the `Object::toString` method to return the formatted list of processes for logging.

Process Scheduler

`ProcessScheduler` is the class that ties it all together, the actual object that represents the scheduler. This has the `simulate` method that loads the processes metadata from file to create the `Process` objects, create the `ProcessList` and then creates the internal `PriorityQueue` and simulate the process scheduling based on the provided data.

The following are the main tasks that the scheduler does: - Moves the processes from the ProcessList to the PriorityQueue depending on current time and process arrival time - Moves the process to queue depending on arrival time and current time and update the process wait time accordingly - Keeps track of the running total of the wait times - Computes the average wait times on demand

Custom Comparator and the Priority Queue (Q)

Prior to Java 8, you will need to define a `Comparator<T>` to create the `PriorityQueue<T>`, and for this case this would look like this:

```
public class ProcessComparator implements Comparator<Process> {
    @Override
    public int compare(Process p1, Process p2) {
```

```
        if (p1.getPriority() > p2.getPriority())
            return 1;
        else if (p1.getPriority() < p2.getPriority())
            return -1;
        else
            return 0;
    }
}
```

or something shorter:

```
public class ProcessComparator implements Comparator<Process> {
    @Override
    public int compare(Process p1, Process p2) {
        return p1.getPriority() - p2.getPriority();
    }
}
```

Where we could use it like this:

```
PriorityQueue<Process> pq = new PriorityQueue<>(new ProcessComparator());
```

With Java 8 (or later), lambda constructs eliminate having to define classes, so we just have this:

```
PriorityQueue<Process> pq =
        new PriorityQueue<>(
                (p1, p2) -> p1.getPriority() - p2.getPriority());
```

This can be further simplified by not using lambda constructs at all, by using `comparingInt`:

```
PriorityQueue<Process> pq =
        new PriorityQueue<>(
                Comparator.comparingInt(Process::getPriority));
```

This solution is using the Java 8's `comparingInt` static method to create the priority queue. Use of the static method is defined in `Process::getPriorityComparator`.

Reading the Input File

The `ProcessList::getProcessListFromFile` takes full advantage of Java 8 streams to read the file and converts it to a `List<Process>` to eventually create a `ProcessList`:

```
ps.map(p -> p.split(" "))
    .map(strings -> Arrays.stream(strings)
        .map(Integer::valueOf)
        .collect(Collectors.toList())
        .toArray(new Integer[4])
```

```
    .map(Process::new)
    .collect(Collectors.toList()
```

The constructs reads every line and converts it into array of Integer of size 4, then creates the `Process` instance (for each line processed) using the `Process(Integer... ids)` constructor. The list is then passed as parameter for the `ProcessList` constructor to create the final list.

Logging and Writing to Output File

A static class `Logger` is defined to be able to output to a file (defaults to `process_scheduling_output.txt`) and at the same time log to standard output (the screen). This sort of acts like the logging classes that comes with `java.util.logging` package but much, much simplified by using a FileWriter and PrintWriter.

Running the Application

You can override the input/output file by optionally specifying the path for the input and the output as command-line parameters:

```
java ProcessScheduling input_file.txt output_file.txt
```

Observations and Lessons Learned

Priority Queue Behavior

One of the notable things that I have learned while writing the code for this project is that default Java implementation of `PriorityQueue` sorts elements with the comparator when adding to queue and not from removing; changing the priority of elements once they are in the queue does not reorder them. This makes sense since as this essentially makes the class efficient; both enqueueing/removing from queue has time complexity of O(log n) per the documentation. This is highly efficient when the element priorities are not modified, but adds overhead when the priorities are modified since you will have to remove the element whose priority is modified (removing objects from the queue has time complexity of O(n)) then re-insert in the queue. This behavior can be demonstrated when you run in the `ProcessScheduling::testPriorityQueueBehavior` static method.

In the implementation, to abstract this behavior from the rest of the `ProcessScheduler` code, we override the `PriorityQueue::add(E)` method to check if the element is already in the queue (using `contains`) and if it is we remove then (re-)add it in the queue. We need to check first if it's in the queue because the `PriorityQueue` behavior *allows duplicates*.

```
public static PriorityQueue<Process> createPriorityQueue() {
    // create an anonymous class instance to change the behavior
    // of the default PriorityQueue
    return new PriorityQueue<>(Process.getPriorityComparator()) {
```

```
        /**
         * This forces the re-ordering if the process exists
         * already in the queue
         *
         * @param process the process
         */
        @Override
        public boolean add(Process process) {
            if (contains(process)) {
                remove(process);
            }

            return super.add(process);
        }
    };
}
```

It is also worth noting that when updating the priorities in the queue that you also don't update the queue itself while iterating you will get `ConcurrentModificationException`–i.e., modify the processes first, then re-add those in the queue in another loop.

Lastly, the way `PriorityQueue::remove(Object)` works is it finds it by using the object's `::equals` implementation (by default it does ref equality). If the Element overrides the method, unexpected behavior may occur. Luckily we didn't feel the need to override the `Process::equals` behavior so we never had to deal with the unexpected behavior changes.

Conclusion

The extensive generics provided out of the box by Java is almost always taken for grated nowadays (I'm guilty of such) and this project has certainly helped me better understand and appreciate them more, especially with the `PriorityQueue` which I should probably use more often with my projects when applicable.