CI583 Assignment Tasks

Semester 2 2023/24

Module registration scheduler-implementation

This assignment is about simulating different scheduling strategies, for the module registration to students, during the enrolment on a course, at the University of Brighton. When a student is enrolled for an academic year, they are registered for a number of modules they will study. A request to register a student to a module can come at any time and It is possible that many students will need to be registered at the same time. So, in order to handle a large number of requests, the Students Registration System(SRS) needs to use a strategy to decide how each process (code) for module registration will be executed by the computer system.

The possible strategies are:

***Round Robin*:** manage a single list of module registration processes, taking the module registration process at the head of the list and allowing it to run for a fixed amount of time (the *quantum*). When the process is complete, remove it from the list.

Priority Scheduling: threads are created with a priority (*HIGH*, *MED* or *LOW*) and stored in a priority queue. Whilst scheduling, take the highest priority process from the queue and allow it to run. When a process is complete, remove it from the queue.

***Multi-level Feedback Queue*:** manage two separate lists of processes, which are to be considered, *YOUNG* and *OLD*. New processes are added to the *YOUNG* list. Whilst scheduling, if the *YOUNG* lists is not empty, take the process at the head of the list and allow it to run. Then, add it to the *OLD* list. If the *YOUNG* list is empty, take a process from the *OLD* list, allow it to run then add it to the *YOUNG* list. When both lists are empty, we are done.

THE CLASSES IN THE JAVA PROJECT

The **ModuleRegister** class is a wrapper around a Java **Thread**, which does the module registration. There are two ways to create a *ModuleRegister* process: The first way is to give it a name and specifying the amount of "work" to do (a length of time in milliseconds):

```
ModuleRegister m = new ModuleRegister("P1",2000;
```
//A ModuleRegister called P1 that will work for 2 seconds

Optionally, we can give the ModuleRegister process a priority, which will be ignored by schedulers other than the ***Priority Scheduler*.**

```
ModuleRegister m = new ModuleRegister("P1",4000, ModuleRegister.PRIORITY.HIGH);
```
//A high priority process that will work for 4 seconds

Now that we have a process we can start it running:

```
m.start();
```

Every ModuleRegister process has a *status* accessed by `m.getStatus()`. The status is either `NEW`, meaning it hasn't started running yet, `TERMINATED`, meaning that it has finished, or some other value that indicates it is running, or waiting for something. See[Thread.State](https://docs.oracle.com/javase/9/docs/api/java/lang/Thread.State.html) for more details. If a process is waiting for something we can "wake it up" by calling `m.interrupt()`.

The `**ModRegReceiver**` class is an abstract, module registration scheduling class that declares some methods that every module registration subclass must implement.
It includes some code that is common to all registration schedulers.
For each ModRegReceiver scheduler subclass, you need to implement:
the constructor, the `**enqueue**` method and the `**startRegistration**` method. A description of what each of them needs to do follows.

##Round Robin strategy (**RRReceiver**)

The constructor of this scheduler needs to call the constructor of the superclass, passing in the `quantum` argument:

```
  super(quantum);
```

Then you need to initialise the list (such as an `ArrayList`) that will hold the module registration processes. This should be a
class-level field, so declare the list at the top of the class:

```
    public class RRReceiver extends ModRegReceiver {
      private ArrayList< ModuleRegister > queue;
      //...
```

then initialise it inside the constructor by typing `queue = new ArrayList<>()`.

The `**enqueue**` method simply needs to add processes to the queue.

The `startRegistration` method needs to start by creating an empty list which will hold the completed processes. This will be the return value of the method. Then, while the queue is not empty, do the following:

+ take the next process from the queue and get its State.
+ if the state is NEW, start the process then sleep for QUANTUM milliseconds, then put the process at the back of the queue.
+ if the state is TERMINATED, add it to the results list.
+ if the state is anything else then interrupt the process to wake it up then sleep for QUANTUM milliseconds, then put the process at the back of the queue.
+ when the queue is empty, return the list of completed processes.

##Priority Registration strategy (**PRReceiver**)

The only real difference between this scheduler and the *Round Robin* scheduler is that the processes each have a priority (***HIGH***, *MED* or ***LOW***) and are stored in a priority queue. Import the class

import java.util.List.PriorityQueue;

and declare a class-level field with the type `PriorityQueue<ModuleRegister>`.
As before you will initialise this in the constructor. Before doing that, you need to create a `Comparator` that will determine the order of the queue. This is a function that takes two processes, `m1` and `m2`, and returns -1 if the priority of `m1` is less than the priority of `m2`, and 1 if the priority of `m1` is greater than `m2`.
The usual thing to do if the priorities are equal is to return 0, but in this case your function should return -1. This is so that when a process is added to the queue, it ends up behind any other processes with the same priority.

The easiest way to define the comparator is by using a lambda expression:

```
Comparator<Process> c = (m1, m2) -> {
 //compare the priorities of m1 and m2 and return an int
};
```

Then supply the comparator to the constructor of the `PriorityQueue`.

Other than that, the `startRegistration` method will be the same as for the *Round Robin* registration scheduler, except that you should use the `poll` method to get the highest priority item from the queue.

##Multi-Level Feedback Queue strategy (MLFQReceiver)

This module registration scheduler will manage two lists, which are considered "*young*" and "*old*". To schedule the next module registration process it does the following:

+ If the list of young processes is not empty, take the next process, allow it to run then put it at the end of the list of old processes (unless the state of process is TERMINATED, in which case add the process to the list of results).

+ If the list of young processes is empty, take the next process from the list of old processes, allow it to run then put it at the end of the list of young processes (as before, unless the state of process is TERMINATED).

More guidance is given in the source code comments.