# CS 4348 Project 3

**The goal**: Simulate CPU scheduling
**Due**: May 7th, 2024
**GitHub Classroom Link**: https://classroom.github.com/a/zsGDE1dK

## The Setup

As before, all you need to get started will come from GitHub. You will be implementing 4 scheduling algorithms, each of which has a shell class and a JUnit test class. They are:

1. First-Come, First-Served (`FirstComeFirstServedScheduler`)
2. Priority (`PriorityScheduler`)
3. Round-Robin (`RRScheduler`)
4. Multilevel Feedback Queue (`MultilevelFeedbackQueueScheduler`)

Instead of "threads", we're using CPU "bursts" for the tasks. Each of these tasks is separate; you won't use past performance to try to predict anything.

Each CPU burst is a `CPUBurst` object, which the JUnits create for you. The methods in it are:

- `start()`: where you start the task. Pass `this` to it for the callback when the burst is done.
- `stop()`: where you stop the task (if it is preempted/timer runs out)
- `timePasses()`: don't call this one; the JUnits will do it for you. (It's used to trigger the `done()` callback below when the task finishes.)
- `getters`: you know what they're for :)

Note that there is no way to get the time remaining in a burst. This is by design; a real thread can't tell you hold it will be before it's done. 😀

For each of the four schedulers you will need to implement these methods (all marked with a TODO):

- `addTask()`: where you are given a new task to schedule. If nothing is running, start the task. Depending on the algorithm, you might preempt the running burst (stop it) to start this new one. Otherwise, put it in a data structure that will help you pick the task at the right time.
- `done()`: called when a burst completes. It passes the burst that completed, but you probably won't need it. Pick the next task and start it.
- `hasTasks()`: checks to see if the CPU is still working. Return true if there's something on the CPU and/or there are tasks still waiting to run. (I found it easiest to keep track of the currently running task for use here.)

## OPTIONAL: Running All the JUnits at Once

Here's how to run all the JUnits at once from within IntelliJ

1. Go to `Run->Edit Configurations` (If I remember right, in Windows the menu is hiding at the top of the window near the name of the project)
2. Press the '+' button and select "`JUnit`"
3. Give it a name 😀
4. For the module, any of the dropdown options should be fine
5. For "`-cp`" select `cs-4348-project-3.test`
6. Change "Class" to "All in Package" and type in `edu.utdallas.cs4348`
7. Your configuration should now show up towards the top-right; the play button will run them.

## What You'll Do (in the order you should do them)

1. `FirstComeFirstServedScheduler`
    1. The easiest: everything gets run in the order they come in.
2. `PriorityScheduler`:
    1. Each CPUBurst that arrives will have its priority set, with a higher number meaning higher priority. You should run the tasks by order of priority (and first-come, first-served within a priority level). I used a map of priorities to lists of tasks of that priority, but that's not the only way to do it.
    2. This scheduler should be preemptive: if a new CPUBurst arrives with a *higher* priority than the current one, stop the current one and place it *at the front* of its priority list (so it will finish before anything of the same priority runs).

3. `RRScheduler`
    1. You'll get the time quantum to use in the constructor. The tasks run in order (no priority here), but with a timer based on the time quantum.
    2. When you start a task, create a new `Timer` object (the abstract `edu.utdallas.cs4348.Timer` class) and have the `timerExpired()` method stop the current task, move it to the back of the line, and start the next task.
    3. Give this timer to `TimedTasks` (`TimedTasks.addTimer()`) so my code can advance the timer as time passes and notify you when the timer expires.
    4. When a task completes normally (`done()` is called in `RRScheduler`), cancel the timer by calling `TimedTasks.clearTimer()`.
4. `MultilevelFeedbackQueueScheduler`
    1. This is a 3-level queue, similar to the one in the slides. The time quantums for the first two levels are given to you (3 and 6), and the last level is first-come, first-served, with no timer (i.e. all bursts run to completion).
    2. I suggest you keep three lists, one for each level. Bursts in the 3ms queue get priority over ones in the 6ms queue, which have priority over the last queue.
    3. For the first two levels create a `Timer` just like you did in round-robin (steps 2-4 above).