ICSI412 – 2 – Priority Scheduler

This assignment is extremely important – (nearly) every assignment after this one uses this one!

If you have bugs or missing features in this, you will need to fix them before you can continue on to new assignments. This is very typical in software development outside of school.

You must submit .java files. Any other file type will be ignored. <u>Especially</u> ".class" files.

You must not zip or otherwise compress your assignment. Brightspace will allow you to submit multiple files.

You must submit every file for every assignment.

<u>You must submit buildable .java files for credit.</u>

"All animals are equal, but some animals are more equal than others."
Animal Farm – George Orwell

Introduction

We did a lot of "setup" work in our previous assignment. Now we will start adding functionality to turn this into a more realistic model.

Every process needs a kernel-based data structure to:

1) Keep track of its resources (memory, open files, etc.)
2) Keep statistics and information about the process

This kernel-based data structure is called the Process Control Block (abbreviated PCB). Previously, we scheduled processes by having a queue of UserlandProcess. Now, we will have queues (multiple) of PCBs. Each PCB will have a reference to the UserlandProcess as a member.

So far, our scheduler has been very simple – everyone gets an equal turn. But if you think about your experience using your computer, you don't really want that to be true. You want some things (like Netflix) to have high priority so that it gets all of the computing resources that it needs. We will refer to these as **<u>real-time</u>** processes. Some processes need to be scheduled quickly, but not absolutely top priority. An example might be your web browser. These are **<u>interactive</u>** processes. Finally, some processes can happily run in the **<u>background</u>** whenever they can get some time. An example of this is getting weather updates.

You might naively start out by saying that when we need a process to run, we will do anything on the real-time list, then anything on the interactive list, then anything on the background list. The problem with that is that some processes will never get run time. This can lead to deadlocks and unresponsive user interfaces. We will be implementing a probabilistic model. If there are real-time processes, then 6/10 we will run a real-time process, 3/10 we will run an interactive process (if there is one) otherwise 1/10 we will run a background process. Otherwise, if there are interactive process(es), we will ¾ run interactive and ¼ run background. If there are only background, then use the first of those.

You might notice that a user can abuse this system – just mark **their** process as "real-time". Then they get more runtime than others. We are going to fix that by watching real-time and interactive processes. If they run to timeout more than 5 times in a row, we will downgrade their priority (real-time → interactive, interactive → background). That downgrade is permanent for the life of the process, although I am sure you can see how we could "promote" them back up later.

For this to make sense, though, we need some way for a process to not run until its time is up. For that, we will add a new system call: Sleep(). Sleep() will put the process to sleep for a specified number of milliseconds. This is very useful for applications that want to "wake up" every so often to do a task (for example – check the weather or render a new frame of an animation).

Detail

Create a PCB class. This class manages the process from the kernel's perspective and is not visible from userland. That makes it secure. You will need a few members: a static "nextpid" int and a an int pid member that holds a process id. Methods are:

```
PCB(UserlandProcess up) /* creates thread, sets pid */
void stop() /* calls userlandprocess' stop. Loops with Thread.sleep() until
     ulp.isStopped() is true.  */
boolean isDone() /* calls userlandprocess' isDone() */
void run() /* calls userlandprocess' start() */
```

Create the Sleep() method in OS (static, as before), calling Sleep() in the Kernel which will call Sleep() in the Scheduler.

```
void Sleep(int milliseconds)
```

Use all the steps that we discussed in Project 1 – the Enum, putting the parameters in the list, switching to kernel, getting the return value, etc.

Inside the Scheduler, we will need to know what time it is. We could try to keep track of this based on the Timer, but that will fail later. Instead, we can use Java's Clock class to get a clock with millisecond accuracy. Do note – this makes our program dependent on real time, which will make using the debugger harder.

When a process calls sleep, we can add the requested time to sleep to the current clock value – that will be the minimum time to wake up. We could be awoken after that time, but not before. Sleep() is not a guarantee that we will wake up right then, just not before. When a process calls Sleep(), we should change the currently running process (which is logical – they want to go to sleep; we don't want them to be awake). Sleep should not put the process back on our one queue, it should put the process in a separate list for sleeping processes.

The other half of sleep is waking up processes that were sleeping. On a task switch, we need to find processes that should be awakened and give them a chance to run. This can be done efficiently if you are smart about the data structure.

For priorities, add a new version of CreateProcess to OS, Kernel and Scheduler that includes both UserlandProcess and a new enum for Priority. Overload it so that the old calls still work with a default to Interactive priority.

You will need to remove our old "processes" job queue and make three new ones (one for each priority). Inside SwitchProcess, there are a few changes to make. When you stop a process, you now must put the process onto the correct queue (hint – we probably need to store the correct queue for the process somewhere). Don't forget to demote processes, as we talked about. We also need to use Java's Random() to figure out what queue to get the next process to run from. I found writing helper methods to be very helpful.

Testing all of this is a little bit tough. You will want to create processes that are each of the priorities. You need to create a real-time process that intentionally runs for a long time (to see demotion), but also one that intentionally calls Sleep() so that it doesn't get demoted.

Rubric	Poor	OK	Good	Great
Code Style	Few comments, bad names (0)	Some good naming, some necessary comments (3)	Mostly good naming, most necessary comments (6)	Good naming, non-trivial methods well commented, static only when necessary, private members (10)
PCB class	None (0)	One of: Holds ULP, holds and assigns PID, is used in all queues (2)	Two of: Holds ULP, holds and assigns PID, is used in all queues (4)	All of: Holds ULP, holds and assigns PID, is used in all queues (5)
Sleep Functionality – awake time storage	None (0)	Implemented in a wrong location (3)		Implemented in a logical place (5)
Sleep functionality - implementation	None (0)	Function updates the storage (5)		Function updates the storage, puts the process in a separate queue (10)
Sleep – awakening	None (0)	Processes are awakened early or very late (2)	Processes are awakened in a timely way, inefficiently(4)	Processes are awakened in a timely way, efficiently (5)
CreateProcess – new method	None (0)	Exists somewhere, enum is replicated or ad-hoc sharing (3)	In OS, Kernel, Scheduler OR enum exists and is shared (6)	In OS, Kernel, Scheduler; enum exists and is shared (10)
Process stores priority	None (0)	Implemented in a wrong location (3)		Implemented in a logical place (5)
Queues for Priorities	None (0)			All three exist (5)
Demotion	None (0)		Exists but is incorrect (4)	Exists and demotes correctly (10)
ProcessChange put process in proper queue	None (0)			When process is stopped, it is put in the correct queue (10)
ProcessChange gets process from correct queue	None (0)			Process change uses a random number to determine the correct queue to pull from and removes first item from the queue (15)
Testing	None (0)	Few tests (3)	Mostly tested (6)	Test processes that show all functionality working (10)