ICSI412 – 1 – The Beginning

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. Especially ".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.

You must submit buildable .java files for credit.

Introduction

An operating system is just a program that manages your hardware, dividing up the resources of your computer fairly and reasonably among the multiple programs that you want to run. We will be building a simulator of an operating system – a program that does most of what an operating system does but running in the JVM instead of directly on the hardware.

The problem that we are trying to solve is that we have more than one program that we want to run, but only one CPU. We want all programs to make some forward progress over some reasonable period of time; we don't want one single program to run, and no others get any run time. We want some level of fairness. To get that fairness, each program will get some amount of run time (called a quantum). After that run time is over, another program will get a turn.

Refresher on hardware

A CPU fetches instructions from memory, decodes them (including getting data from registers), executes the instructions, then stores the results back to registers in an (almost) infinite loop. What stops the loop? Interrupts. An interrupt is a signal, generated by an instruction (a soft interrupt) or by outside hardware (a hard interrupt).

CPUs have two (or more) modes – normal mode and privileged mode. Every program runs in normal mode except the operating system. The operating system runs in "privileged" or "ring 0". This mode allows the operating system to bypass all of the security features of the CPU and access anything that it wants. In normal mode, the CPU cannot access devices or memory unless privileged mode has previously granted access to them.

The switching of modes happens by interrupts. When an interrupt (hard or soft) occurs, the CPU stops the fetch-decode-execute-store cycle and transitions to privileged mode where it restarts the cycle, but with the program counter pointing to a piece of code called an interrupt handler. Typically, when the interrupt is handled (and that varies depending on the kind of interrupt), the interrupt handler code switches the CPU back to normal mode.

One final piece of hardware is the hardware timer. This is a separate chip (usually) that can be given a duration. When that duration has elapsed, it signals the CPU with a hard interrupt.

Task Switching

When the computer boots, there is no concept of task-switching; that is an operating system idea. The computer starts and reads the operating system from disk. The operating system does some hardware configuration (things like finding out what disks exist, how much memory do you have). It then picks the first program to run, sets the hardware timer to expire in a time period (called a quantum), then switches to user mode and the userland program runs.

Eventually the quantum elapses, the timer causes an interrupt and the processor switches to privilege mode. The timer interrupt handler calls the operating system's task switch which determines the next program to run. It then resets the hardware timer and starts the program running.

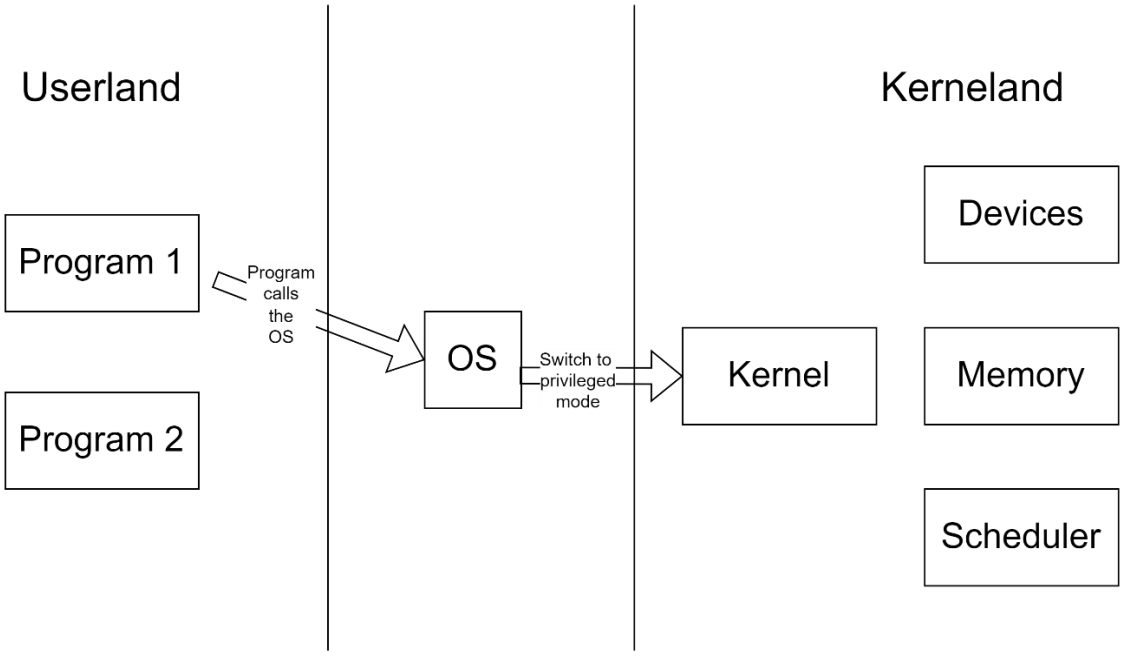
Cooperative Multitasking

Java doesn't really expect people to start and stop threads like operating systems do. Most people start a thread and let it run, because they have work that they want done and the thread does the work. We won't be implementing pre-emptive multitasking. We will be implementing cooperative multitasking. The difference? Each user land program runs until it calls a function "cooperate()" that gives the operating system a chance to switch the thread out for something else. This is not ideal, but some operating systems in the past did use this (original MacOS, for example).

Calling the Operating System

When a user program needs to call the operating system (which, of course, needs to run in privilege mode), it can't just call it like it would a function; that wouldn't switch processor modes. That transition happens though a soft interrupt. Remember that this is just an instruction (like LOAD, STORE or ADD) except that it causes the CPU to switch to privilege mode. Typically, the user program will store an id in a pre-defined register and then use the INTERRUPT instruction. The interrupt handler will look at that register and call some operating system function. Other registers will hold parameters to the OS function.

Our Software Architecture



We will have two different areas – I call them "kerneland" and "userland". Kerneland is the privileged area and userland is where our traditional programs are run. There is a bridge between them. Every publicly accessible function in kerneland a bridge function in a class called "OS".

In terms of organizing our code, we will have an OS class and a Kernel class. The OS class, for our simulator, will call the Kernel class. **Our userland programs may not call the Kernel class or any other class that resides in kerneland except by calling a function in OS that does the work for them**; this simulates the soft interrupt approach that happens in CPUs. The Kernel class, for some of the simple examples, will call a matching function in one of the classes in kerneland. That doesn't always have to be the case, though – the Kernel function might call several different functions. Consider creating a process – the process needs memory, and it needs to be scheduled, so CreateProcess needs to call two different areas. The Kernel class serves as a single reference point for every kernel call.

Threads allow you to run more than one thing "at the same time". That could be literally at the same time (on a multi-core computer) or using time-sharing (one thread gets to run, and then the other). We will be using Java Threads (https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/lang/Thread.html) to simulate … threads and processes in our operating system. A Java Thread accepts a Runnable (https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/lang/Runnable.html) class and runs it in a managed thread.

Java has a class (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Timer.html) that can call a method after a set period of time. We will use this to simulate the hardware-based timer that can interrupt the CPU and allow us to switch between processes.

The last "big" concept is semaphores.
https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/Semaphore.html

These are actually very simple – they are counters, like integers. You can increment and decrement them. But there are two big differences:

1) Two threads changing a normal int at the same time might overwrite each other. Semaphores are protected against that.
2) The most important thing – if you try to decrement past 0, your thread will **stop** until the semaphore is incremented. Think of this like the number of people allowed in a club; if the club can have 50 people inside and you are person #51, you have to wait until you can get in. Semaphores just measure that "backward" – the semaphore is the number still allowed in. If you decrement when the semaphore is 0, you wait.

Detail

Create a new Java project. We will use this one project for the whole semester. Make sure that you use a fairly new JDK (I used Java 19, but anything close will be fine).

Userland Process

Create a new **abstract** class "UserlandProcess" that implements Runnable. This will be the base class for every test program that we write. Your userland process should have a Java Thread and a Semaphore. The thread will allow your user program to run, the semaphore will stop in when we should cooperate. One more member – a Boolean that indicates that our quantum is expired.

You will need some methods:

```
void requestStop() – sets the boolean indicating that this process' quantum has expired
abstract void main() – will represent the main of our "program"
boolean isStopped() – indicates if the semaphore is 0
boolean isDone() – true when the Java thread is not alive
void start() – releases (increments) the semaphore, allowing this thread to run
void stop() – acquires (decrements) the semaphore, stopping this thread from running
void run() – acquire the semaphore, then call main
void cooperate() – if the boolean is true, set the boolean to false and call OS.switchProcess()
```

Test Programs

Create two new classes derived from UserlandProcess – HelloWorld and GoodbyeWorld. You will have to create a "main" method (because of Runnable). In each, make an infinite loop that just prints "Hello World" or "Goodbye world". Make sure that each calls cooperate() inside the loop. If you forget this, your OS will never switch processes.

We will also create a Userland process called the idle process. It runs an infinite loop of cooperate() and Thread.sleep(50).

OS Class

Create the OS class. Everything in the OS class will be static, so we never need to create an instance of it. It will have a **private static reference to the one and only instance** of the Kernel class.

The OS class has a fundamental problem to solve – it is the gateway between the userland thread and the kernel thread. Consider what has to happen if your user program calls a kernel function.

1) The userland thread calls the OS code.
2) ???
3) OS has to start the kernel and stop the user program

But how does the kernel, an independent thread, know what to do? How does it know if you want to open() a file or fork()? The answer is that we will have a shared data area. OS will "leave a note" for the kernel – "here is what I want you do to".

Let's lay out that data area. We need:

1) An enum of what function to call (I called this "CallType")
2) A static instance of that enum (I called this "currentCall")
3) A static array list of parameters to the function; we don't know what they will be, so we will make it an arraylist of Object.
4) The return value. In a similar way, we don't know what the return value type will be, so make it a static Object.

For each kernel call, we will have to:

1) Reset the parameters.
2) Add the new parameters to the parameter list.
3) Set the currentCall.
4) Switch to the kernel (more on this later)
5) Cast and return the return value.

Create two methods in OS:

```
public static int CreateProcess(UserlandProcess up)
      Make an enum entry for CreateProcess, and follow the steps above
public static void Startup(UserlandProcess init)
      Creates the Kernel() and calls CreateProcess twice – once for "init" and once for
the idle process.
```

Kernel Class

Create a Kernel class with a member of type Scheduler (we will make this later). Kernel will have some things very similar to the Userland process – the thread and semaphore and the start() method. Write an appropriate constructor to initialize these, then call thread.start().

The run() method is an infinite loop:

```
      while (true)
            mySemaphore.acquire() – to see if I should be running
            switch on OS.currrentCall – for each of these, call the function that implements them
            call run() on the next process to run (we will see this in the scheduler).
```

Implement the run() method. The only "currentCall" values we are expecting is CreateProcess and SwitchProcess (more on these in the scheduler, below).

Scheduler Class

Finally, create the Scheduler. The scheduler will have a private LinkedList<UserlandProcess> to hold the list of process that the scheduler knows about, a private instance of the Timer class (java.util.Timer – there are a few timer classes in Java) and a **public** reference to the UserlandProcess that is currently running. The constructor should schedule (using the timer) the interrupt for every 250ms. Inside the interrupt, call "requestStop()" on the currently running process.

The scheduler needs two methods:

```
public int CreateProcess(UserlandProcess up)
public void SwitchProcess()
```

CreateProcess should add the userland process it to the list of processes and, if nothing else is running, call switchProcess() to get it started.

SwitchProcess' job, overall, is to take the currently running process and put it at the end of the list. It then takes the head of the list and runs it. Two possible corner cases:

1) Nothing is currently running (we are at startup). We just don't put null on our list.
2) The user process is done() – we just don't add it to the list.

You might ask "runs it"? That's actually easy – we are just going to set "currentlyRunning" to the new process.

Back to the Kernel

In the run() method, we said: `call run() on the next process to run`

Now, we know how to do that – call run() on scheduler.currentlyRunning.

Back to OS

In OS, we said: Switch to the kernel (more on this later)

Now, we can implement this.

First, we call start() on the kernel.

Then if the scheduler (you might need an accessor here) has a currenlyRunning, call stop() on it.

Note – this could create a weird problem. When you call Init(), we are going to jump to the kernel to create our first process, but there is nothing running! That's OK. If nothing is running, create an infinite loop that calls Thread.sleep(10).

Main

Last thing – we need a main() – make a new class for it. Call OS.Startup() with a new HelloWorld then CreateProcess() with a new GoodbyeWorld().

Run this and ensure that your console alternates between (a bunch of) hello world and goodbye world. I found the output a little intimidating, so I added a sleep to my hello world and goodbye world:

```
try {
    Thread.sleep(50); // sleep for 50 ms
} catch (Exception e) { }
```

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)
Userland Process	Doesn't exist (0)			Exists, is abstract and implements Runnable (5)
Userland Process – Mechanisms	Don't Exist (0)		One of: members are correct, methods are correct (5)	All of: members are correct, methods are correct (10)
HelloWorld	Doesn't exist(0)			Exists, derives from Userland process, implement run() with an infinite loop of printing (5)
Goodbye World	Doesn't exist(0)			Exists, derives from Userland process, implement run() with an infinite loop of printing (5)
Idle Process	Doesn't exist(0)			Exists, derives from Userland process, implement run() with an infinite loop of waiting (5)
OS	Doesn't exist (0)		Two of: Creates one and only one Kernel, Startup() instantiates kernel calls its CreateProcess. CreateProcess calls Kernel's CreateProcess (3)	Creates one and only one Kernel, Startup() instantiates kernel calls its CreateProcess. CreateProcess calls Kernel's CreateProcess (5)
OS invoke Kernel	Doesn't exist(0)			Starts kernel, stops running process, sleeps if no running process (5)
Kernel	Doesn't exist(0)			Creates a private scheduler, has CreateProcess which calls the scheduler's version (10)
Kernel – run	Doesn't exist(0)	Two of: loop, acquires, dispatches calls, runs next process (3)	Three of: loop, acquires, dispatches calls, runs next process (7)	All of: loop, acquires, dispatches calls, runs next process (10)
Scheduler	Doesn't exist (0)			Has list of processes, a timer and tracks the currently running process (10)
Scheduler Create Process	Doesn't exist (0)	Two of: Creates a new kerneland process, adds it to the list, starts it if it is	Three of: Creates a new kerneland process, adds it to the list, starts it if it is	Creates a new kerneland process, adds it to the list, starts it if it is the first one, returns pid (10)

		the first one, returns pid (3)	the first one, returns pid (6)	
Scheduler SwitchProcess – start new	Doesn't exist (0)	One of: Gets first process from the queue, runs the process (4)		Gets first process from the queue, runs the process (10)