SCHOOL OF COMPUTING, QUEEN'S UNIVERSITY

CISC324: Operating Systems

Lab 5, due March 22

In this lab you will be using Java Monitors for process synchronization. This lab allows you to enrich your knowledge and practical experience in writing synchronization code in Java.

Java Monitors overview.

As described in the course lecture: A monitor is a high-level programming language construct that provides mutually-exclusive access to data declared within the monitor. A monitor collects together all critical section codes related to a particular shared data structure. They are considered as less error-prone to use than semaphores. With semaphores, it is easy to forget the `acquire()` or the `release()` primitive around some critical sections. Monitors automatically provide mutual exclusion. In addition, for signaling among threads, monitors provide signal and wait operations on condition variables.

A monitor with N condition variables has N+1 queues associated with it. The additional queue is the entry queue which holds threads that are ready to execute some code within the monitor but have to wait on that queue as at that time another thread is already within the monitor (holds the monitor lock) executing some code. There is one queue for each declared condition variable. A thread waits on queue `x` when it has executed `x.wait()` on a condition variable named `x`.

In Java, there is only one condition variable per monitor. This condition variable does not have a name. Threads simply say `wait()`, without giving the name of a condition variable.

Only one thread at a time can execute monitor code. In Java, this is implemented using the lock associated with the monitor object. Whatever thread is executing monitor code is "holding the lock". When a thread executes `wait()`, it releases the lock. This allows another thread to execute monitor code. While that thread is executing monitor code, it can execute a `notify()` or `notifyAll()`. The way in which `wait()` releases the lock is analogous to this style of semaphore code:

```
Release(mutex); // Release mutex in preparation for Acquire(carSem).
Acquire(carSem); // Wait for the "carSem" semaphore.
Acquire(mutex); // Grab mutex again, now that we're past the carSem waiting.
```

Similarly in a monitor, a thread releases the lock while executing `wait()`, and then, after `notify()` is received, the thread must again obtain the lock before continuing execution of monitor code. The compiler takes care of implementing this. In Java, when a thread executes `notify()`, it does not give up the lock; it keeps holding on to the lock and executing the statements following the `notify()`. The effect of the notify() is that one thread moves from the condition queue to the entry queue. The effect of `notifyAll()` is that all threads in the condition queue move to the entry queue. If the condition queue is empty, then `notify()` and `notifyAll()` have no effect. Also, in Java the "entry queue" is not necessarily FIFO. When a thread releases the lock (by reaching the end of the monitor method that it executes), then one of the threads in the entry queue will get the lock. Which of the entry-queue threads gets the lock is a matter of luck and timing. Note: some Java implementations might choose to use FIFO.

Java is an object-oriented language, with inheritance. Every object in Java inherits from the "Object" class. The Object class defines a lock and a waiting area for threads. The following Java constructs deal with this lock and waiting area. These constructs apply to any Java object, since they all inherit from Object.

- `synchronized` This is a keyword that can be attached to a method or a block of code. For example, the `rwMonitor.java` file (provided for lab 5) declares `public synchronized void startRead()`. Similar declarations are used for `endRead`, `startWrite`, `endWrite`. The synchronized keyword delimits code that holds the lock for an object. When a thread wants to execute synchronized code, it has to wait until no other thread is executing synchronized code associated with the same object. (The compiler takes care of implementing this waiting.)
- `void wait()` Every Object has a `wait()` method. The `wait()` method releases the lock and waits. It re-grabs the lock when a `notify()` is received.
- `void notify()` Every Object has a `notify()` method. The `notify()` method hangs on to the lock, but allows one waiting thread to switch from wait mode to "try to grab the lock" mode. If there is no thread waiting, then `notify()` has no effect. This is different from semaphores: if no one is waiting for a semaphore, then the semaphore Release is remembered, allowing the next thread to get past Acquire without waiting. In contrast, monitor semantics are that `notify()` is ignored if no one is waiting. Therefore, in a monitor, a thread that executes a `wait()` always has to wait, no matter how many preceding `notify()` calls were made.
- `void notifyAll()` Every Object has a `notifyAll()` method. The `notifyAll()` method hangs onto the lock and allows all waiting threads to switch from wait mode to "try to grab the lock" mode. If there is no thread waiting, then `notifyAll()` has no effect.

The `wait()`, `notify()` and `notifyAll()` methods can only be called by a thread that is holding the lock on this Object. To get the lock, put the `wait()`, `notify()` or `notifyAll()` inside a block or method that is synchronized. Java does not necessarily use FIFO order for `wait()` and `notify()`. If thread T1 executes `wait()` and then thread T2 executes `wait()`, either T1 or T2 might be awakened when thread T3 executes `notify()`. Normally, Java code is written without explicitly naming the object to which synchronize, `wait()`, and `notify()` are being applied. (It's shorthand for saying "this" is the object being used.) In the sample code for lab 5, the `wait()` and `notifyAll()` calls in `rwMontor.java` are being applied to the object rw. This rw object is created in the constructor for class `SharedDataStruct`, by the statement "`rw = new rwMonitor()`".

The wait() code in Java monitors usually takes this form:`while <some test> wait()`

This code is used (instead of if `<some test> wait()`) for three reasons.

1. A thread that executes `notify()` or `notifyAll()` immediately continues execution of the following code. The other threads that are awakened by the `notify()` still have to wait to grab the lock. So by the time they get the lock, the condition for which they got blocked in the first time may have changed. Thus it is necessary to test that the condition still holds, using the while loop.

2. In Java, only one condition variable is used: all threads wait on the same object. Thus a thread executes `notifyAll()` and leaves it to each awakened thread to determine whether or not its particular condition is really satisfied. If the condition is not satisfied, the while loop makes this thread `wait()` once again.

3. Java threads can have spurious wakeups. The semantics of Java wait() are as follows: the thread executing `wait()` can only be woken up because some other thread executes `notify()` or `notifyAll()`. However, for efficiency reasons, Java implementations are allowed to have spurious wakeups, where a thread wakes even though there was no execution of `notify()` or `notifyAll()`. Because of this, a Java thread that is awoken must allow for the situation that this wakeup was spurious. Surrounding the `wait()` with a while loop does just that.

The Wikipedia entry for Spurious wakeup provides more information about this disappointing aspect of Java concurrency. The existence of spurious wakeups was an intentional design decision: "on some multiprocessor systems, making condition wakeup completely predictable might substantially slow all condition variable operations." This `while <some test> wait()` code looks similar to busy waiting, but only a small amount of repeated testing is involved. Threads spend most of their time waiting in a queue to grab the lock of the object (and not using CPU during this time). Occasionally they become awakened because another thread executes `notify()` or `notifyAll()`. The awakened thread executes its test, and if it fails the thread calls `wait()` again. In contrast, the really CPU-intensive form of busy waiting uses code such as "`while <test> do {nothing}`": this wastes a lot of CPU time because the thread constantly tests and retests the same condition, never getting suspended on a queue. It is possible to explicitly name the Java objects to which wait() and notify() are applied. Remember to do this in a synchronized block of code: it is necessary in order to get the lock for an object before executing wait() or notify(). For example:

```
Object wrt;
...
synchronized (wrt)
{
wrt.wait();
// wrt.wait() and wrt.notify() must be in a synchronized (wrt) block.
...
}
```

Writing code with several named conditions can be tricky, because every Java object (acting like a condition variable) has its own lock. In contrast, in a traditional monitor (as defined by Hoare Lampson/Redell), the monitor itself provides a lock, and allows any number of condition variables to be declared inside the monitor. The designers of the Java language only provide a simplified approximation to these traditional monitors. Because every Java object has its own lock, deadlock can easily result from code such as the following.

```
Object wrt, rd;
// declare two objects that act as condition variables
synchronized (wrt)
{
// some code here; perhaps using wrt.wait() or wrt.notify()
synchronized (rd)
{
rd.wait(); // There is a problem here.  At this point the thread holds a lock
}
// on object wrt as well as on object rd.  The rd.wait() releases
}

// the lock on rd and gets into the queue associated with rd.
// However, the whole time that this thread is waiting in the
// rd queue, it will still hold the lock on wrt.  Therefore, no
// other thread can get into a ‘‘synchronized (wrt)’’ block, and
// we might get deadlock.
```

In your lab 5 programs, it is easier if you stick to using one unnamed condition per monitor.

Exercise 1.

A sample monitor, for the readers and writers problem (readers having priority), is provided for you in directory lab 5. Study the structure of this code carefully. Understand how monitors are used: look at class `rwMonitor`, used in `SharedDataStructure`. Also understand the software engineering reasons behind the organization of the `SharedDataStructure` class. This class declares the shared data to be private, and provides the access methods dataRead and dataWrite that perform the needed synchronization. This organization allows external users (reader and writer threads) to access the data without having to worry about the synchronization - maybe even without being aware that there is any synchronization going on. This ensures that external users cannot by mistake access the shared data directly: they have to go through the access methods, and are thereby guaranteed to get proper synchronization.

Study the readers/writers monitor given above. Execute the monitor code, and study the output to convince yourself that it is working. We have discussed during the lecture that both solutions “readers having priority” and “writers having priority” (in lab 3) are not starvation free solutions. In this first exercise, you are asked to modify the given code to transform the solution into a starvation free solution.

What should be submitted for this exercise

- Place your codes (`MainMethod, Reader, Writer, SharedDataStruct, and rwMonitor`) in one folder.
- Your codes should be well commented. You must add comments to explain why you have used a particular variable and what each piece of code performs.
- Run your program for a certain number of readers and writers and save your output in a text file (copy and past). Perform at least three different runs (three different output files). Do not run your program for more than 10 readers and writers.
- Use a ReadMe1.txt file to explain the settings that you have used to produce each output file (e.g., for 10 readers and 10 writers, the outputs are in file out1-10-10.txt).

Exercise 2

In this exercise, you will write synchronization code for the producer and consumer problem a.k.a., bounded buffer problem. You are given the following Java files: `MainThread.java` containing the main class, `Producer.java` defining the producer class, `Consumer.java` defining the consumer class, and `Buffer.java` which defines the buffer class.

1. Compile and execute the program then explain the output.
2. In the main thread, swap the two lines `p.start()` and `c.start()`, compile and execute then explain the output.
3. The program seems to be not working properly. Complete the program so that the producer thread produces letters that are consumed later on by the consumer thread.

What should be submitted for this exercise

- Place your codes (`MainThread, Producer, Consumer, and Buffer`) in one folder.
- Your codes should be well commented. You must add comments to explain why you have used a particular variable and what each piece of code performs.
- Run your final program and save your output in a text file (out2.txt).
- Answer to the question and write them on a ReadMe2.txt file.

Place every exercise folder in one folder and compress the file using zip. Submit the zipped file onto OnQ before the deadline.