Operating Systems and Concurrency

Week 4 lab exercises

Introduction

In this set of exercises you will investigate the control of concurrent access using *semaphores*.

One advantage of using semaphores for concurrency control is that they are themselves thread safe (in the Java implementation, at least). This means you don't have to worry about multiple threads concurrently accessing the semaphores themselves because they have been designed to be inherently thread safe.

Another point to note is that semaphores can be used like locks. The lecture covers *counting semaphores*, which use an integer variable to control access to some resource(s). The maximum value of the semaphore can be set by the programmer. If you set the maximum value to 1 then you get a *binary semaphore* which can behave the same way as a lock because it will only allow a maximum of one thread to acquire the semaphore at a time.

Semaphores can be used in the absence of locks, but also have some uses of their own. As stated in the lecture, they can allow more than one thread to access some resource and can also be used to share sections of a buffer, as will be seen in this lab session.

Task 1

For this task you will convert the dining philosopher solution to use semaphores.

Look at the *dining philosophers* problem and its solution using `ReetrantLock` in the lecture slides for this week.

Either by converting that solution, or creating your own from scratch, change it so that it uses binary semaphores to control access to the forks (we still require that only one philosopher can use a particular fork at a time). This does not require many changes but think about where you will put the semaphores. You can change the classes any way you feel you need to.

Add some code to your program to demonstrate that your solution is allowing each of the philosophers to both think and eat.

Task 2

For this task you will create a system that follows the *producer-consumer* pattern. The producer-consumer pattern involves one or more producer objects that *produce* objects and one or more consumer objects that *consume* those objects. The producer produces objects and places them into a *buffer*. The consumer takes objects from the buffer and processes them in some way.

Many processes could be said to follow this pattern. On a factory production line, some workers (producers) might prepare an object in some way and then put it onto a conveyor belt, whilst other workers

(consumers) take those objects off the conveyor and do something else with them. If there are more producers than consumers then the conveyor belt might get too full. If there are more consumers than producers then it might be empty most of the time. Of course, the factory administrators would try and avoid either of these situations.

In the conveyor belt example, the conveyor belt is the buffer. Some things are clear about the producers and consumers sharing this conveyor belt. For example, consumers can't take objects from the conveyor if there are no objects on it; producers can't add objects to the conveyor if it is full; consumers can't take an object before a producer has finished putting it on the conveyor; different producers can't put objects in the same place; etc.

For this task you can use the classes provided on Keats as the basis for your solution, or you can create your own. The provided classes simply encapsulate the objects that will be produced and consumed.

The system you create should function as follows. The overall purpose of the system is to evaluate basic arithmetical operations. It does this by means of producers that produce the operations and consumers that evaluate them. Producers and consumers share the same buffer. Concurrent access is to be controlled using semaphores. For the purposes of this task, producers can simply produce random (but legal) operations.

The object that producers put into the buffer is a *computation*. Thus, the object that a consumer consumes is also, therefore, a *computation*. A computation consists of an *operation* and an *evaluator*[1]. See the classes provided for this even if you don't intend to use them.

There is only one buffer but, as in previous exercises, we will consider it as being 'sliced'. This time, however, the slices will each be controlled by a semaphore. If there are N slices then there will be N semaphores.

This approach allows for literal concurrent access of the buffer by different threads. Before each thread adds or takes from the buffer, it must attempt to acquire the semaphore for a slice of the buffer. At any given moment, some slices will be being used by other threads and some might not be. If it is granted the semaphore, then that thread (producer or consumer) can access that slice of the buffer. Of course, it must release the semaphore when it is finished with the slice.

The most important part of this system is the buffer itself. This should be encapsulated in a class and should control the semaphores. That is, the threads themselves don't need to know about the semaphores. The buffer should provide a method called `add(Computation)` for producers and a method called `take()` for the consumers. The operation of the buffer and semaphores will be 'hidden' inside those methods.

This implies that producers might produce a *computation* but not be able to add it to the buffer. That is fine. If that happens, they should discard that computation, sleep for a short while and then try and add another one. This can also happen if the buffer is full. Consumers might also not be able to take a computation. This could happen if there is too much *contention* for the semaphores or if the buffer is empty. Again, this is fine. The consumer thread should simply sleep for a short while and then try again. We will assume for the purposes of this exercise that the type of task we are implementing does not rely on every single object being processed.

The only output required from this system is the number of computations that were *consumed*. This gives some measure of the throughput of the system. The actual computation results themselves can be discarded. You can then experiment with parameters such as the size of the buffer, the number of

[1] These classes are very similar to the ones we talked about in a previous lecture.

semaphores, the number of producer threads vs consumer threads, etc. to see what impact they have on the throughput.

Suggestions

If you feel you don't need suggestions on the implementation of Task 2 then you can ignore this section.

In addition to the classes you are provided with you will almost certainly want classes for the following:

- The buffer (which will include the semaphore manipulation). This should have a fixed size and a fixed number of slices. Each slice will be guarded by a semaphore.
- The producer(s): this should implement `Runnable`. As stated above, the producer(s) will produce the computations and, for the purposes of this task, can simply create random ones. This involves simply generating two operands (I suggest restricting the range to, say. -100 to +100) and a random operator (one of $+, -, *, /$). Make sure only legal operations are created.
- The consumer(s);this should implement `Runnable`. This class will simply try and get a computation from the buffer and then evaluate it. The result of the evaluation can be discarded. The consumer class should have an instance variable that records how many computations it evaluated.
- A main method that creates the producers and consumers, starts them, and then makes them quit after, say, 30 seconds. The main method should get the number of computations performed by each consumer (as in the previous bullet) and then simply print out the sum.
- Experiment with different buffer sizes, numbers of slices, numbers of producers and consumers, etc.