Semaphores

Last updated: January 21st 2019, at 4.35pm

1 The Code

This week you will be implementing semaphores in Java in various ways. This week's code package contains some classes to get you started.

1.1 The Bounded Buffer Package

The `BoundedBuffer` package contains an implementation of a bounded buffer using semaphores, and is provided to enable you to test your semaphore implementations. It consists of the following classes: `Producer`, `Consumer`, `Factory`, `IntegerFactory`, `Buffer`, `BufferError`, `BufferSystem`, and `BufferSystemTest`. The most important of these classes are:

`Buffer`⟨`T`⟩ a buffer for holding items of type `T`

`Producer`⟨`T`⟩ produces items of type `T` and sends them to the buffer

`Consumer`⟨`T`⟩ collects items of type `T` from the buffer and consumes them

The `BufferError` class provides a basic exception class for managing buffer errors, and the `BufferSystem` class provides an example of using the three main classes, with some basic tests for it in the `BufferSystemTest` class. The two "factory" classes are used by the `Producer` and `BufferSystem` classes. These are:

`Factory`⟨`T`⟩ is an interface. `Factory`⟨`T`⟩ objects should have a `make()` method which manufactures and returns an object of type `T`

`IntegerFactory` is a specific implementation of `Factory`⟨`T`⟩ for which the type `T` is `Integer`

1.2 The Semaphore Package

The `Semaphore` package holds the semaphore implementations. Currently it contains the following interface and classes:

`SemaphoreInterface` Every implementation of semaphores should implement the `SemaphoreInterface` interface (see, e.g., `LocalSemaphore`).

`LocalSemaphore` This is a wrapper class for Java's `Semaphore` class.

`BlockingSemaphore` This is an implementation of a blocking bounded semaphore, as presented in the lecture.

`BinarySemaphore` A binary semaphore is a special case of a bounded semaphore, with an upper limit of one. This implementation chooses to use a blocking approach.

`SemaphoreLimitError` This class is not currently used in the package, but could be useful in implementing crashing semaphores (see question 4).

`Semaphore` The `Semaphore` class is provided as a wrapper class for the various semaphore implementations. Currently it is defined to "wrap" the `LocalSemaphore` class, but simply changing the class declaration so that it "wraps" a different implementation allows that implementation to be tested in the buffer package without any need to change any of the buffer code.

2 Exercises

Note: There are three (related) logbook questions this week — the first three questions below. These are not primarily programming exercises, but require you to think about the behaviour of semaphores, and about identifying problems that careless use of semaphores can use. Please think carefully about how the identified problem(s) can arise, and make sure that your explanation of the reasons is clear and succinct.

1. **Logbook questions.** For the logbook answer the three following questions. You can observe the behaviour of the buffer system by running the `BufferSystem` class's `main` method. You may want to change the programme arguments to influence this behaviour — e.g. by making the consumer faster than the producer, or vice versa.

   The four programme arguments are:

   The buffer size. This should be a positive non-zero intger value.

   The run time. This should be a positive integer or decimal number. It specifies the number of seconds the buffer system should run for. After this number of seconds the system wwill shut down.

   The producer delay. This should also be a positive integer or decimal number, or one of the predefined values **slow**, **medium** and **fast**. The producer will pause between each access of the buffer for a random time between zero seconds and the number of seconds specified

```
public void producer() {
   while (true) {
      ...;
      criticalSection.P();
      buffer.put(elem);
      criticalSection.V();
      noOfElements.V();
      ...;
   }
}
```

```
public void consumer() {
   while (true) {
      ...;
      noOfElements.P();
      criticalSection.P();
      buffer.take(elem);
      criticalSection.V();
      ...;
   }
}
```

Figure 1: The `producer` and `consumer` methods from the lecture.

by this argument. The delays corresponding to the predefined **slow**, **medium** and **fast** arguments are $2 \cdot 0$, $1 \cdot 0$ and $0 \cdot 5$ seconds, respectively.

The consumer delay. As the producer delay, but for the consumer.

To change these parameters, select `BufferSystem` as the class to run, then edit the run configuration — either under the **Run** menu, or in the pull-down options at the top right next to the “run” symbol. You can then enter the desired parameters in the “Program arguments” field.

- **Logbook question.** In the lecture it was said that in the implementation of bounded buffers using semaphores (see figure 1 on page 3) the order of the `criticalSection.P()` and `noOfElements.P()`, in the `Buffer` class’s `get` method, was essential, but that the order of `criticalSection.V()` `noOfElements.V()`, in the same class’s `put(item)` method, was not. Identify the corresponding piece of code in the `Buffer` class provided and make the change. Can you produce an error situation? *Note:* You may see “error” messages about attempts to access a closed buffer. This is not the error you are looking for.
- **Logbook question.** Why does the error situation arise when the code is changed as described in question 1? Why does it not arise in the original code?
- **Logbook question.** Is the order of the calls of `P()` in the `Buffer` class’s `put` method also essential?

2. Now implement your own version of (unbounded) semaphores, and use these in the bounded buffer implementation, rather than Java’s `Semaphores`, and check that the code still works.

3. **Model question.** Now implement absorbing bounded semaphores. . .

4. . . . and crashing bounded semaphores.
5. Try each of the three possible implementations of bounded semaphores in the bounded buffer code and observe their behaviour. What happens (or can happen) in each case?
6. Sometimes only binary semaphores are available. Implement unbounded semaphores (i.e. create a new implementation of the `SemaphoreInterface` interface) using only binary semaphores (i.e. instances of the `BinarySemaphore` class provided) to manage the synchronisation and communication between your semaphores (i.e. you may *not* make use of Java's **sychronized** mechanism). Your implementation is likely to start with something like:

```
public class SemaphoreSemaphore implements SemaphoreInterface {
    // Binary semaphore used to control access to this semaphore
    private BinarySemaphore access;
    ⋮
}
```

Note: there is a potential conflict between Java's wish to make use of reuseable code through inheritance, and the wish to ensure that a semaphore's value is purely internal to each semaphore. It is probably best, in these exercises, to implement each semaphore type independently.

End of semaphores tutorial