159.355: Concurrent Systems
Lab 3: The `synchronized` Keyword and Locks

Callum Lowcay, Stephen Marsland

1 Introduction

This week we are going to introduce some concurrency control to our threads. We are going to look at two aspects of the same problem, the readers and writers (or producers and consumers) problem and solve it using both the `synchronized` keyword, and explicit locks.

2 Your Tasks

2.1 Synchronize the ornamental gardens

Take your solution to last week's ornamental gardens problem. Rewrite it to use `synchronized` instead of the `Bakery` class.

2.2 The producer/consumer problem

Suppose that there are a set of threads, some of which want to write into a shared buffer, and some of which want to read from that buffer. The problem is to synchronise the reading and writing so that the writer is never overwriting something that has not been read yet, and the reader is never reading something that is currently being written.

2.2.1 Implement the producer and consumer threads

Download the code for this lab from Stream and import it into Eclipse (or whatever editor you favour). Implement the `Consumer` and `Producer` classes. The `Producer` class should write a number of items into the buffer, then terminate. The `Consumer` class reads some number of items from the buffer, then terminates. Incorporate a delay (using `Thread.sleep`) into the consumer class so the producers have time to fill the buffer.

Implement the `main` method in the `InfBuffer` class to start multiple consumer and producer threads. Make sure there are enough producers to keep the buffer full for the consumers.

2.2.2 Implement the buffer using `synchronized`

The Buffer class as given has no synchronisation, see if you can make the system fail. You may need several producers and consumers. Then, use `synchronized` methods to maintain mutual exclusion.

2.2.3 Implement the buffer using `ReentrantReadWriteLocks`

Make a second buffer class that uses explicit locks. Ensure that the locks are always given back correctly, and make the scope of the locks as small as possible.

3 Things to Think About

- There is a race condition where a consumer may attempt to read the buffer only to find it empty, because the producer thread has not had a turn yet. How can you prevent this problem? Can you prevent it without busy waiting?
- Could performance be increased by using read/write locks (with write locks for the producers and read locks for the consumers)?
- Does it make any difference if you use fair locks? Or priorities?
- Could you implement this buffering algorithm using atomic variables and compare-and-set?