CSc 360 Assignment #2

Due date: 11:55pm, Wednesday, 19 June 2013

Objectives

In this assignment, you will learn the following issues:

1. the priority issue in the classic readers/writers problem;
2. synchronization issue in a multicast media stream problem;
3. the design and implementation of POSIX Condition variables.

Problems

There are three parts to this assignment. Each is independent of the other. They are increasingly more difficult. You should try them in the order listed below. Part (a) and (b) will be done in Java. Part (c) will be done in C.

a) The Readers/Writers Problem [5%]

The Readers/Writers problem is defined here.

There is a shared data area used by many processes. Some processes only **read** the shared data, called the Readers; some only **write**, called the Writers. Any number of Readers may simultaneously read the shared data, i.e., they may overlap while reading. Only a single writer may write to the shared data. While writing, no other writers or readers are allowed to proceed. Thus, reading and writing, and writing and writing, are mutually exclusive. Only reading may proceed concurrently.

There are commonly two cases to be considered:

1. (**Readers have priority**) While reading, any number of readers may join in; as a result, if readers continuously overlap their readings, writers may starve.
2. (**Writers have priority**) Whenever a writer is waiting, it is allowed to write as soon as possible. If writers wait continuously for writing, readers may starve.

In folder *code/a*, a partial solution is given where no preference is given to readers or writers. A single condition queue is used for both readers and writers. For part (a), you are required to modify code in order to enforce a **writers priority** using POSIX Mutexes and Condition variables.

1. Look in folder *code/a*.
    - *Database.java*: An interface for a thread-safe, reader-writer shared resource.
    - *DatabaseRW.java*: The shared resource; you will be modifying this for writer preference.
    - *Writer.java*: A thread that writes data to a Database.

   - *Reader.java*: A thread that reads data from a Database.
   - *TestBasicRW.java*: An initial test for showcasing 1 writer, 3 readers.
   - *TestPreference.java*: Test for writer preference.
2. Run *../soln/rose.exe 10 TestBasicRW.bin* and fiddle with it to get a feel for what is going on.
3. Run *../soln/rose.exe 10 TestPreference.bin* to see how the readers are getting in.
4. Run *cd ../soln; ./rose.exe 10 TestPreference.bin* to see what your solution should resemble and how the writers are dominating order.

You are only required to modify this solution (in *DatabaseRW.java*) to reflect a preference to writers. Your solution must use our Conditions and Mutexes supported by ROSE. **REMEMBER**: Use *code/soln/rose.exe* to test your solution -- later you can use your part (c) rose.exe solution to test again.

HINT: In *Reader.java* and *Writer.java*, there are a few lines of code commented out. Uncommenting these out may or may not be useful to help you debug.

---

b) Synchronous Readers-Writer Problem [25%]

Consider the problem of multicasting a live media (e.g., MP3) stream to several users. Let us assume there is a single writer (or producer) which produces the media stream, and there are three readers (or consumers) which consume the data, i.e., playing the stream.

There is a bounded buffer shared by all readers and a single writer. Every item in the stream must be delivered in order and guaranteed. Thus, the buffer behaves like a FIFO queue. If the buffer is empty, the readers must wait; if the buffer is full, the writer must wait.

In a single-producer-single-consumer bounded buffer, the consumer reads an item and immediately advances the "out" pointer to the next item. But, in a single-producer-multiple-consumer bounded buffer for streaming media, the consumers (or readers) must be synchronized on their reads. That is, reading is synchronized; a reader/consumer is not allowed to read the next item until all readers have read the current item. A faster reader must wait for a slower reader. This way the timing of playing back the media stream is synchronized in all readers.

For part (b) you are going to implement your solution in folder **code/b**.

- *SBuffer.java*: An interface for a synchronized buffer.
- *BoundedBuffer.java*: Thread safe, synchronized bounded buffer; you will be adding to this to make it synchronize with multiple readers.
- *Producer.java*: Very similar to Writer in part (a); there is some random delay included to crudely simulate writing time.
- *Consumer.java*: Very similar to Reader in part (a), but Consumer requests the the index of the data it wants to read. Its output includes this to help w/ debugging.
- *TestP1C1.java*: An initial test for one producer and one consumer.
- *TestP1C3.java*: A test to verify that readers are synchronized.

Your solution will be to modify **BoundedBuffer.java** and ensure that all readers read the same item before advancing.

Use *TestP1C3.java* to test your code. Again, there are some commented out lines of code in *Producer.java* and *Consumer.java* that may be helpful for debugging.

---

c) POSIX Condition Variables [40%]

The POSIX standard defines Mutexes and Condition variables to support the concept of Monitors. There are three primitives defined on Mutexes:

```
public class Mutex {
    public Mutex() {...} ;
    public void Lock() {...} ;
    public void UnLock() {...} ;
}
```

A Mutex is basically a binary semaphore. It is initally free (or unlocked). The operations Lock() and UnLock() are indivisible. Lock() waits until the Mutex is free, then locks it. UnLock() unlocks the Mutex, and as a result may resume a waiting process. A Mutex has ownership; only the process that locks the Mutex may unlock it.

There are four primitives defined on Condition variables:

```
public class Condition {
    public Condition() {...};
    public Wait( Mutex m ) {...};
    public Signal() {...};
    public Broadcast() {...};
}
```

Condition variables MUST be used in conjuction with Mutexes. A Condition variable is associated with a predicate (a condition that may be true or false). The testing of the predicate requires the reading of some shared variables. (Note: We use Condition to denote a variable of type **Condition**. We use *condition* to denote the predicate associated with a Condition variable.)

For example, "buffer is not full" is a *condition*. This requires a test "count < MAX" where count is a shared integer variable. Therefore, testing this *condition* is a critical section. We need a Mutex to protect this critical section. If the *condition* is evaluated to be false, the process MUST then wait until the *condition* has changed.

For example,

```
private Mutex m;
private Condition nonfull; // count < MAX

m.Lock();
// if buffer is full, then wait until not full
while (count == MAX) nonfull.Wait(m);
m.UnLock();
```

The *condition* we desire is "not full", i.e., count < MAX. The test "count == MAX" is true if "not full" does not hold. As a result, we must wait until the *condition* "not full" has changed. For this type of synchronization, we need Condition variables.

A Condition variable is conceptually a queue. The process waits on a Condition will be blocked in the Condition queue. Later, when some other process signals this Condition, one of the processes in the Condition queue will be resumed. To maintain fairness, we insist the Condition queue be first-come-first-served.

After gaining the lock to a Mutex, if this process then waits on a Condition variable, then in order to allow other processes to proceed, this process MUST unlock the Mutex and enter the Condition queue at the same time. Thus, the c.Wait(m) call includes a parameter "m" which is the Mutex.

When a process is resumed from a c.Signal(), it MUST relock its Mutex and resume after the c.Wait(m) statement. As a result, it is guaranteed to be safe inside its critical section.

Signal() on a Condition variable resumes a single process. Broadcast() resumes all waiting processes. To prevent starvation, the ordering of resumption MUST follow the same ordering of waiting. That is, a process that waits first will be the first process to be resumed after a Broacast(). A process resumed from a Condition Signal() or Broadcast() must relock the enclosing Mutex. To prevent "barging", we insist that processes resumed from a Condition have priority of locking the Mutex over other processes which just wait for the Mutex.

For part (c), you are required to implement the Wait(), Signal() and Broadcast() primitives in C inside ROSE as specified above. Specifically, you are only required to fill the details in *j_thread.c* for:

- *void CondWait()*
- *void CondSignal()*
- *void CondBroadcast()*

Hints:

Study the *MutexLock()* and *MutexUnLock()*, *SemWait()* and *SemSignal()* code carefully. You need to understand these well in order to implement the Condition variables primitives (i.e. your solution will likely resemble aspects of their solution).

There is a folder **code/c**, with a bin that can be used to run a basic Condition.broadcast() test.

Take note of the thread states in jvm.h:

- *BLOCK_ON_MUTEX*
- *BLOCK_ON_COND*

You will probably need to use them.

In *jvm.h*, the structure **thread_T** defines an additional field called **mutex_t *relock**. This is very important! You'll need it.

Support

A directory called "soln" contains binaries for:

- **TestBasicRW.bin** and **TestPrefence.bin**: Bin solutions for Part (a).
- **TestP1C1.bin** and **TestP1C3.bin**: Bin solutions for Part (b).
- **rose.exe**: a solution to Part (c) and should be used for intial testing of Part (a) and (b).

Run everything and see what they do.

For part a and b, please use our solution "rose.exe" to test your Java code. After completing Part c, you may then test your Part a and b using your solution to Part c.

Evaluation

The assignment is to be done in teams of two. Each member will receive the same mark. Each team member needs to understand all aspects of the assignment -- midterms will be based around understanding aspects of the assignments. It is **HIGHLY** recommended that team members do the assignment together.

Your evaluation will be based mostly on the correctness of your solutions. Testing is one way to evaluate correctness. Code inspection is another. Even if your code runs, there may still be errors which can only be uncovered by inspection. Obscured programming tricks and habits will make programs harder to understand. Keep everything simple and elegant!

- Code [70%]
- Documentation [30%]
  - Doxygen and coding style (The solution should not be overly convoluted) [10%]
  - Descriptions of tests performed [10%]
  - Answer to questions [10%]

Submission

Answer the questions in doxygen. If you do not answer the questions on the mainpage, please provide a link from the mainpage to the answers for each question (helpful for the marker and you).

Please include all necessary makefiles in each directory to generate the executables and make sure that "make clean; make" will rebuild your solutions to each part in each subdirectories.

Submit a zipped file with the name *assignment2.tgz*.

assign2 should keep the structure of

- assign2
  - code/
  - html/
  - Doxyfile

To do this go into cygwin and navigate to the path above assign2 and enter:

```
    tar czvf assignment2.tgz assign2
```

Please submit this file (**assignment2.tgz**) electronically through Connex before midnight on the due date. **Only one team member needs to submit.**