Submission instructions are in Section 3.

The following are exercises to tighten your knowledge of concurrent programming in Java. Write the answer to each problem in a separate text file; e.g. for problem 1 you will write 1.txt, etc. Any java code you write should also be included in the text file for the problem you write it for. If you don’t want to use “ASCII graphics” for problems 2 and 3, you can include separate .jpg files for your wait-graphs. Use the name of the problem as the title, e.g. 2.jpg, or 3.1.jpg, 3.2.jpg, etc.

1 Background

Threads can be in one of four thread states: created (after creation, but before calling start), runnable (either running, or eligible to be run), blocked (waiting for some event before execution can resume), and terminated (the thread is finished). Threads can be blocked for many reasons, including waiting to acquire a lock, waiting to be awakened with notify, waiting on I/O, and sleeping for a fixed time.

All objects in Java can behave as locks. Each lock has three conceptual fields: the current thread owner of the lock (if any), the count of the times the owner has re-acquired the lock (0 if there is no owner), and the wait-set, which is a list of threads waiting on the lock, each paired with a count indicating the number of times a thread had reacquired a lock before it started waiting. The values of these three fields are called the lock state of a particular lock.

2 Problems

Problem 1 Consider the classes BoundedBuffer and Prob1. The latter class constructs two threads that insert strings into a shared bounded buffer. Run the code to see what the output looks like. Only one output sequence is possible (why?).

Answer:
Each get result is printed after the corresponding put in the other thread. This means that t1 must print its gets first, since t2 cannot print its gets until t1 has done its puts, which always happens after its gets.

Assume that (1) the threads are run on a single processor, (2) t1 is scheduled first, and (3) the scheduler runs each thread until it blocks before switching to the other thread. For each of the labelled program points t1-1, t2-1, t1-2, etc. in Prob1, describe (1) the thread state of each of the threads t1 and t2, and (2) the lock state of the BoundedBuffer objects buf and buf2.

Answer:

For all program points, perhaps surprisingly, the lock state of both buf and buf2 is always that (1) the lock is not held (owner and count are empty), and (2) no threads are on the wait set. This is because (1) no locks are ever held within Prob1, only within BoundedBuffer, and (2) for each thread to get to the current point, it invokes notifyAll after waking up (at the conclusion of the get or put it was blocked in), which clears the wait sets that had any waiters. Moreover, it is always the case that the current thread is running, and the other thread is still blocked (waiting to reacquire its lock after being awakened by notifyAll). That is, at points t1-X, thread t1 is Runnable and t2 is Blocked, while at points t2-X, thread t2 is Runnable and t1 is Blocked.

Problem 2 Consider the interface Number which defines integer-like numbers and a way to add them, and the classes MyNatural and MyInteger, which implement Number. The class Prob2 creates two threads that add some shared Number objects. Feel free to run the program to see its output.

This program can exhibit a deadlock. (1) Draw a wait-graph of the program showing this deadlock. (2) Indicate which statements in the given classes caused each thread to be blocked in this deadlocked state.

Answer:
The statements are line 9 in MyNatural and line 6 in MyInteger (the calls to getInt).

Finally, fix the program by modifying either MyNatural, MyInteger, or both, to remove the deadlock. Do not introduce any race conditions! You may not use a global lock; each Number must have its own lock. Your solution can slightly modify the semantics of the given program by only holding one lock at a time. For extra credit: you can present a solution that holds two locks (as in the original program), but is deadlock-free by enforcing a fixed ordering. This case should enforce that two threads calling n.add(m) at the same time should end up with n being n + 2*m.

For your submission, include in the 2.txt file the entire changed Java file(s).

Answer:

The simplest thing to do is to change the add methods to only synchronized after having acquired the integer from the other number. For example, the add method of MyNatural would become:

```java
public void add(Number n) {
    int x = n.getInt();
    synchronized (this) {
        value += x;
        if (value < 0)
            value = -value;
    }
}
```

However, this approach can allow one thread to see the intermediate state of a number in mid-computation. A more robust approach is to force an ordering of the lock-acquires; figuring this out is extra credit. This requires changes to the add and getInt methods of MyNatural and MyInteger. You can force an order by using the hashcode of the numbers in question. Here are the changes to MyNatural:

```java
public int getInt() { return value; }
private void doAdd(Number n) {
    int x = n.getInt();
    value += x;
    if (value < 0)
        value = -value;
}
public void add(Number n) {
    int nHc = n.hashCode();
```
int thisHc = hashCode();
if (nHc < thisHc) {
    synchronized (n) {
        synchronized (this) {
            doAdd(n);
        }
    }
} else {
    synchronized (this) {
        synchronized (n) {
            doAdd(n);
        }
    }
}

Problem 3 The class BoundedBuffer2 is the same as the BoundedBuffer class, except that (1) it replaces occurrences of notifyAll with notify; and (2) it replaces put with putArray, which inserts the contents of an array into the buffer, rather than a single item.

The class Prob3 has four variables, set by the command-line flags: p is the number of producer threads, which insert a group of items of size b into a single shared buffer; c is the number of consumer threads, which each remove a single item of the shared buffer; and s is the size of the buffer.

For simplicity, you may assume a single processor, but any possible legal schedule. For each of the following values of these parameters, indicate either (a) all of the tokens added to the buffer will be removed, and all the threads will exit normally; or (b) that some number of the threads are stuck in a blocking state before all tokens are removed from the buffer (e.g., this can happen because of deadlock). For (a), be as precise as possible—e.g., describe an invariant that the program maintains the implies constant progress. For (b), write an execution trace that that shows how threads will fail to exit normally. Refer to the threads as \( C_1 \ldots C_c \) for the consumers, and \( P_1 \ldots P_p \) for the producers.

1. \( p = 1, c = 1, s = 1, b = 1 \).
2. \( p > 0, c = p, s = 1, b = 1 \).
3. \( p = 1, c = 4, s = 4, b = 4 \).
4. \( p = 2, c = 4, s = 5, b = 2 \).

Hint: think about the kinds of threads that can be on the buffer’s wait-set, and what that implies with regard to deadlock.

Answer:
In all cases, there is only ever one lock to acquire, so there is no possibility of deadlock due to circularities in wait-graphs. The only possibility of deadlock is due to a producer or consumer being on a wait-set forever (missed by a notify operation).

1. Will complete normally. It is only possible for the consumer to be on the buffer's wait-set, and in this case, we are sure that the waiter will be awakened by the notify of the other thread.

2. Will not complete normally. Consider when \( p = 2 \) and \( c = 2 \), where \( P1 \) and \( P2 \) are the producers and \( C1 \) and \( C2 \) are the consumers. Here's a trace:
   - \( C1: \text{buffer.get()} \) blocks on buffer
   - \( C2: \text{buffer.get()} \) blocks on buffer
   - \( P1: \text{buffer.put()} \) fills buffer and calls \text{buffer.notify()};
     \( C1 \) is taken off buffer's wait-set
   - \( P2: \text{buffer.put()} \) blocks on buffer (since it's full)
   - \( C1: \) reacquires buffer's lock, removes item, and calls \text{buffer.notify()}
   - \( C2: \) gets lock, buffer.size() is 0, so waits again.
   At this point \( C2 \) are and \( P2 \) are on buffer's wait-set, never to wake up.

3. Will complete normally. There can only ever be consumers on the wait-set, since there is always space for the sole producer to put his items. In the case that there are multiple consumers, the notify will wake one of them, which when it removes its item from the buffer will wake the next with its notify, etc. until all waiting threads are awakened.

4. Will complete normally. Once again, there can only be consumers on the wait-set, since the producers will always be able to put their items in the buffer. Each time a producer does this, one consumer on the wait set will be awakened, which will awaken another, etc. as above.

**Problem 4** The *Eater* class defines a variant of the famous “dining philosophers problem” (if you don’t know what that is, look it up on the web for fun). The idea is that in order to eat, each eater must acquire a chair, then a plate, and then the food. The food is put on the table by a separate chef thread (the *main* method of the *Eater* class). After an Eater eats, it consumes the food and gives up the chair and the plate. But: eating is all or nothing: if an eater grabs the plate, and then the chair, but there is no food on the table, it must release both and give someone else a chance.

The class that we have provided defines the chair and the plate as Resources; to grab a resource, you set its gotIt flag to true. Since resources are
shared, setting this flag requires you hold the `Resource`'s lock. The food is a `Consumable`, and operates similarly. Unfortunately, there is a problem with this class. (1) Indicate the problem. (2) Sketch a way that you could fix the problem without adding any new static or instance fields to the `Eater` class. For extra credit, provide a complete implementation that fixes the problem, without using busywaiting, and implements the above spec.

**Answer:**

The problem is that when an `Eater` waits because it cannot acquire the food, it does not release the plate and the chair, as the specification above requires. In particular, it neither releases the locks associated with the resources, nor does it reset the `gotIt` flags. This results in a likelihood that only one `Eater` ever getting the food, which will be the first one that runs and grabs the plate.

To fix this problem, an `Eater` needs to give up the plate and the chair when doing the wait. However, you still need to acquire the lock on the resource to set its flag, and need to avoid the nesting of locks. To do this, you can extend the `Resource` class to include methods for acquiring and releasing the resource, e.g.:

```java
static class Resource {
    private boolean gotIt = false;
    synchronized void acquire() {
        while (gotIt)
            try { wait(); } catch (InterruptedException e) { }
        gotIt = true;
    }
    synchronized void release() {
        if (gotIt == false) throw new RuntimeException("oops!");
        gotIt = false;
        notifyAll();
    }
}
```

Now, rather than synchronizing on the resource in `eat`, and then setting its `gotIt` flag, we simply call `acquire`. Assuming no untoward exceptions being thrown, `eat` becomes:

```java
private void eat() {
    while (true) {
        plate.acquire();
        chair.acquire();
        synchronized (food) {
            if (!food.ready) {
                System.out.println(this.getName() +
```
" says, "Who hid my food?" ");

try {
    chair.release();
    plate.release();
    food.wait();
} catch (InterruptedException e) {} 

else {
    System.out.println(this.getName() + 
        " says, " That’s right. Crunch Crunch");
    food.ready = false;
    break;
}

}

} 

chair.release();
plate.release();

The result is that access to the food will be fair among the different eaters, since as soon as one thread fails to get the food, a different thread will be able to get the chair and the plate. There are other solutions.

3 Submission Instructions

To submit your homework, put all of the relevant files (1.txt, 2.txt, etc.) into a directory hw1/ and then tar and gzip this directory into the file hw1.tgz. On Linuxlab, starting from the parent directory of hw1/ you would do

tar zcvf hw1.tgz hw1

Now, submit this file, hw1.tgz, using the submit system, using project number "1a" as in

java -jar Submit.jar 1a hw1.tgz

(assuming your personal Submit.jar is in the same directory as your hw1.tgz file).