
Reading

1. Some Java security issues are discussed in section 13.5 of the textbook.
2. Additional information at approximately the right level of detail for this course may be found at <http://www.securingjava.com/chapter-three/chapter-three-5.html> and the next two pages reachable from this page.
3. Read chapter 14 on concurrency, but skip the section on the Java memory model.
4. For a summary of the current Java memory model, see <http://www.cs.umd.edu/users/pugh/java/memoryModel/jsr-133-faq.html>

Problems

1. Concurrency in Lisp

The concept of *future* was popularized by R. Halstead's work on the language Multilisp for concurrent Lisp programming. Operationally, a future consists of a location in memory (part of a cons cell) and a process that is intended to place a value in this location at some time "in the future." More specifically, the evaluation of `(future e)` proceeds as follows:

- i. The location ℓ that will contain the value of `(future e)` is identified (if the value is going to go into an existing cons cell) or created if needed.
- ii. A process is created to evaluate `e`.
- iii. When the process evaluating `e` completes, the value of `e` is placed in the location ℓ .
- iv. The process that invoked `(future e)` continues in parallel with the new process. If the originating process tries to read the contents of location ℓ while it is still empty, then the process blocks until the location has been filled with the value of `e`.

Other than this construct, all other operations in this problem are defined as in Pure Lisp. For example, if expression `e` evaluates to the list `(1 2 3)`, then the expression

```
(cons 'a (future e))
```

produces a list whose first element is the atom `'a` and whose tail becomes `(1 2 3)` when the process evaluating `e` terminates. The value of the `future` construct is that the program can operate on the `car` of this list while the value of the `cdr` is being computed in parallel. However, if the program tries to examine the `cdr` of the list before the value has been placed in the empty location, then the computation will block (wait) until the data is available.

- (c) Assuming an unbounded number of processors, how much time would you expect the evaluation of the following `fib` function to take, on positive integer argument `n`?

```
(defun fib (n)
  (cond ((eq n 0) 1)
        ((eq n 1) 1)
        (T (plus (future (fib (minus n 1)))
                  (future (fib (minus n 2)))))))
```

We are only interested in time up to a multiplicative constant; you may use “big Oh” notation if you wish. If two instructions are done at the same time by two processors, count that as one unit of time.

- (b) At first glance, we might expect that two expressions

```
(...e ...)  
(...(future e) ...)
```

which differ only because an occurrence of a subexpression e is replaced by `(future e)`, would be equivalent. However, there are some circumstances when the result of evaluating one might differ from the other. More specifically, side effects may cause problems. To demonstrate this, find a subexpression e in a larger expression `(...e ...)` so that when e is changed to `(future e)`, the larger expression’s value or behavior might be different because of side effects, and explain why. Do not be concerned with the efficiency of either computation or the degree of parallelism.

- (c) Side effects are not the only cause for different evaluation results. In Pure Lisp, find a subexpression e in a larger expression `(...e ...)` so that when e is changed to `(future e)`, the expression’s value or behavior might be different. For concreteness, imagine typing the two expressions in read-eval-print loops. Explain why the expression behave differently. (For credit, evaluating your expression must cause the introduced `future` to be evaluated.)
- (d) Suppose you are part of a language design team that has adopted futures as an approach to concurrency. The head of your team suggests an error handling feature called a “try block.” The syntactic form of a try block is

```
(try e  
  (error-1 handler-1)  
  (error-2 handler-2)  
  ...  
  (error-n handler-n))
```

This construct would have the following characteristics:

- i. Errors are programmer defined, and occur when an expression of the form `(raise error-i)` is evaluated inside e , the main expression of the try block.
- ii. If no errors occur, then `(try e (error-1 handler-1) ...)` is equivalent to e .
- iii. If the error named `error-i` occurs during the evaluation of e , the rest of the computation of e is aborted, the expression `handler-i` is evaluated, and this becomes the value of the try block.

The other members of the team think this is a great idea and, claiming that it is a completely straightforward construct, ask you to go ahead and implement it. You think the construct might raise some tricky issues. Name two problems or important interactions between error handling and concurrency that you think need to be considered. Give short code examples or sketches to illustrate your point(s). (*Note:* You are not asked to solve any problems associated with futures and try blocks, just identify the issues.) Assume for this part that we are using Pure Lisp (no side effects).

2. Java synchronized objects

This question asks about the following Java implementation of a bounded buffer. A bounded buffer is a FIFO data structure that can be accessed by multiple threads.

```
class BoundedBuffer {  
  // designed for multiple producer threads and  
  // multiple consumer threads  
  protected int numSlots = 0;
```

```

protected int[] buffer = null;
protected int putIn = 0, takeOut = 0;
protected int count = 0;

public BoundedBuffer(int numSlots) {
    if (numSlots <= 0)
        throw new IllegalArgumentException("numSlots <= 0");
    this.numSlots = numSlots;
    buffer = new int[numSlots];
}

public synchronized void put(int value)
    throws InterruptedException {
    while (count == numSlots) wait();
    buffer[putIn] = value;
    putIn = (putIn + 1) % numSlots;
    count++;
    notifyAll();
}

public synchronized int get()
    throws InterruptedException {
    int value;
    while (count == 0) wait();
    value = buffer[takeOut];
    takeOut = (takeOut + 1) % numSlots;
    count--;
    notifyAll();
    return value;
}
}

```

- (a) What is the purpose of `while (count == numSlots) wait()` in `put`?
- (b) Synchronized objects in Java have wait/notify semantics that were first introduced in the Mesa programming language developed in the late 1970s at Xerox PARC. With Mesa semantics, a call to `notify()` or `notifyAll()` does not guarantee that a waiting thread is executed immediately. Instead, notify is only considered a hint that some thread that issued a `wait()` for the synchronized object will resume at a convenient future time. In contrast, with Hoare semantics (after C.A.R. Hoare, who introduced monitors), one single thread waiting on a condition will run immediately after another thread signals that condition. Given Mesa semantics, why does the call to `wait()` in `put` have to be enclosed in a while loop? What could go wrong if you replaced it with a single test: `if (count == numSlots) wait()`?
- (c) What does `notifyAll()` do in this code? Could it be replaced with a call to `notify()`? Explain briefly.
- (d) Describe one way that the buffer would fail to work properly if all synchronization code is removed from `put`.
- (e) Suppose a programmer wants to alter this implementation so that one thread can call `put` at the same time as another calls `get`. This causes a problem in some situation but not in others. Assume that some locking may be done at entry to `put` and `get` to make sure the concurrent-execution test is satisfied. You may also assume that increment or decrement of an integer variable is atomic and that only one call to `get` and one call to `put` may be executed at any given time. What test involving `putIn` and `takeOut` can be used to decide whether `put` and `get` can proceed concurrently?
- (f) The changes in part (d) will improve performance of the buffer. List one reason that leads to this performance advantage. Despite this win, some programmers may choose to use the original method anyway. List one reason why they might make this choice.

3. Concurrency and the Observer Pattern

The observer design pattern defines a one-to-many dependency between objects so that when one object (value holder) changes state, all its dependents (listeners) are notified and updated automatically. In a typical implementation, the target of observation implements a valueHolder interface, and the observers implement a Listener interface.

The following Java code is an implementation of the observer pattern, showing two methods where an invocation of the setValue() method triggers notification of the new value by calling the valueChanged() method of any objects that were registered by a call to addListener(). Note that each listener can implement their the valueChanged() method arbitrarily.

```
public class ValueHolder {
    private List listeners = new LinkedList();
    private int value;

    public interface Listener {
        public void valueChanged(int newValue);
    }
    public void addListener(Listener listener) {
        listeners.add(listener);
    }
    public void setValue(int newValue) {
        value = newValue;
        Iterator i = listeners.iterator();
        while(i.hasNext()) {
            ((Listener)i.next()).valueChanged(newValue);
        }
    }
}
```

(a) This Java implementation of the Observer pattern is not thread safe. Why? (You only need a few lines to explain why)

(b) A good way to make code thread safe is to use synchronization. A Stanford student comes up with the following solution to make the Observer implementation thread safe. He thinks that he should just add the Java keyword synchronized to both the setValue() and addListener() method definitions.

What can go wrong here? Explain what can go wrong by giving an example AND explaining why it goes wrong. (Hint: think about the valueChanged method.)

(c) A Cal student comes up with the following more complicated solution to make the Observer implementation thread safe.

```
1  public class ValueHolder {
2      private List listeners = new LinkedList();
3      private int value;
4
5      public interface Listener {
6          public void valueChanged(int newValue);
7      }
8      public synchronized void addListener(Listener listener) {
9          listeners.add(listener);
10     }
11     public void setValue(int newValue) {
12         List copyOfListeners;
13         synchronized(this) {
14             value = newValue;
15             copyOfListeners = new LinkedList(listeners);
16         }
17     }
18 }
```

```

17         Iterator i = copyOfListeners.iterator();
18         while(i.hasNext()) {
19             ((Listener)i.next()).valueChanged(newValue);
20         }
21     }
22 }

```

Is this implementation thread safe? Answer yes or no.

- (d) If your answer in the previous part is yes, explain why the code is threadsafe. If your answer is no, explain what could go wrong by giving an example.
- (e) This question illustrates the difficulty of implementing even well studied patterns in a multi-threaded environment. With a slight change in semantics, we can solve the difficulties encountered above. Suppose we take the code that the Cal student wrote and changed the following lines :
- i. Line 3 : `private int value` changed to `private volatile int value`,
 - ii. Line 19 : `((Listener)i.next()).valueChanged(newValue);`
changed to `((Listener)i.next()).valueChanged(value);`.

How does the semantics of this implementation of the Observer class change from the original implementation?

4. Java ConcurrentHashMap

ConcurrentHashMap allows concurrent access to a hash table with minimal locking. Without going into all of the details, this problem asks you to think about some aspects of the design. The relatively tricky design of ConcurrentHashMap is intended to allow read access to parts of the data structure that may be written concurrently, in a way that causes the read to try again if the state seems inconsistent or incomplete.

The hash table implementation uses a resizable array of hash buckets, each consisting of a linked list of Map.Entry elements. As with other hash table implementations you may be familiar with, the hashcode of an entry determines its bucket; when several entries hash to the same bucket, they are placed in a linked list. Here is part of the definition of a Map.Entry class.

```

protected static class Entry implements Map.Entry {
    protected final Object key;
    protected volatile Object value;
    protected final int hash;
    protected final Entry next;
    ...
}

```

The volatile modifier asks the Java Virtual Machine to order accesses to the shared copy of the variable so that its most current value is always read.

Instead of a single lock governing access to the entire collection, ConcurrentHashMap uses a lock over each segment of buckets. The linked list used by ConcurrentHashMap is designed so that the implementation can detect that its view of the list is inconsistent or stale. If it detects that its view is inconsistent or stale, or simply does not find the entry it is looking for, it then synchronizes on the appropriate bucket lock and searches the chain again.

- (c) What does the use of final in the Map.Entry class tell you about the way a linked list of Map.Entry elements may change when a ConcurrentHashMap is updated? (Don't think too deeply - the purpose of this question is to point out something that is important for later questions.)

- (b) There is one straightforward way to remove an item from a linked list of `Map.Entry` objects. Describe the steps involved in removing the second item from a list using a short Lisp statement to provide clarity to your removal. Assume the pointer to the beginning of the list is mutable. (*Hint: Pure Lisp.*)
- (c) The `ConcurrentHashMap` retrieval operations first find the head pointer for the desired bucket. This is done without locking, so the value of the head pointer could be stale. The operation then traverses the linked list representing the bucket starting from the head pointer, without acquiring the lock for that bucket. If the operation does not find the value it is looking for, it acquires the lock for the bucket and tries again. Here is the code, in case you want to look at it; you may be able to answer the question without reading the code.

```
int hash = hash(key); // throws null pointer exception if key is null

// Try first without locking...
Entry[] tab = table;
int index = hash & (tab.length - 1);
Entry first = tab[index];
Entry e;

for (e = first; e != null; e = e.next) {
    if (e.hash == hash && eq(key, e.key)) {
        Object value = e.value;
        // null values means that the element has been removed
        if (value != null)
            return value;
        else
            break;
    }
}

// Recheck under synch if key apparently not there or interference
Segment seg = segments[hash & SEGMENT_MASK];
synchronized(seg) {
    tab = table;
    index = hash & (tab.length - 1);
    Entry newFirst = tab[index];
    if (e != null || first != newFirst) {
        for (e = newFirst; e != null; e = e.next) {
            if (e.hash == hash && eq(key, e.key))
                return e.value;
        }
    }
    return null;
}
```

Why do the second traversal? What advantage does this two-pass algorithm have over simply locking the linked list the first time and doing only one traversal?

- (d) Removing an element from a `ConcurrentHashMap` poses several problems. First, because a thread could see stale values for the link pointers in a hash chain, simply removing an element from the chain would not be sufficient to ensure that other threads will not continue to see the removed value when performing a lookup. However, there's a clue to how the implementation works in the `get` code above – the appropriate `Entry` object is found and its `value` field is set to `null`. Then the algorithm you may have discovered in part (b) is used. Explain why declaring the `value` field as `volatile` is useful here.