

Homework 8

Due 03 December

Handout 12
CS242: Autumn 2008
19 November

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.securingsjava.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. Stack Inspection

One component of the Java security mechanism is called *stack inspection*. This problem asks you some general questions about activation records and the run-time stack then asks about an implementation of stack inspection that is similar to the one used in Netscape 3.0. Some of this problem is based on the book *Securing Java*, by Gary McGraw and Ed Felten.

Parts of this problem will ask about the following functions, written in a Java-like pseudocode. In the stack used in this problem, activation records will contain the usual data (local variables, arguments, control and access links, etc.) plus a *privilege flag*. The privilege flag is part of our security implementation and will be discussed later. For now, we will just mention that `SetPrivilegeFlag()` sets the privilege flag for the current activation record.

```
void url.open(string url) {
    int urlType = GetUrlType(url); // Gets the type of URL

    SetPrivilegeFlag();

    if (urlType == LOCAL_FILE)
        file.open(url);
}

void file.open(string filename) {
    if (CheckPrivileges())
    {
        // Open the file
    }
    else
    {
        throw SecurityException;
    }
}

void foo() {
    try {
        url.open("confidential.data");
    }
}
```

```

    } catch (SecurityException) {
        System.out.println("Curses, foiled again!\n");
    }
    // Send file contents to evil competitor corporation
}

```

- (c) Assume that the URL `confidential.data` is indeed of type `LOCAL_FILE`, and that `sys.main` calls `foo()`. Fill in the missing data in the following illustration of the activation records on the run-time stack just before the call to `CheckPrivileges()`. For convenience, ignore the activation records created by calls to `GetUrlType()` and `SetPrivilegeFlag()`. (They would have been destroyed by this point anyway.)

	<i>Activation Records</i>		<i>Closures</i>	<i>Compiled Code</i>
(1)	Principal	SYSTEM		
(2)	Principal	UNTRUSTED		
(3)	control link	(2)	⟨ (), • ⟩	
	access link	(1)		
	url.open	•		
(4)	control link	(3)	⟨ (), • ⟩	code for url.open
	access link	(3)		
	file.open	•		
(5)	control link	(4)	⟨ (), • ⟩	code for file.open
	access link	(2)		
	foo	•		
(6) sys.main	control link	(5)	⟨ (), • ⟩	code for foo
	access link	(1)		
	privilege flag	NOT SET		
(7) foo	control link	()		
	access link	()		
	privilege flag			
(8) url.open	control link	()		
	access link	()		
	privilege flag			
	url	"_____"		
(9) file.open	control link	()		
	access link	()		
	privilege flag			
	filename	"_____"		

- (b) As part of stack inspection, each activation record is classified as either `SYSTEM` or `UNTRUSTED`. Functions that come from system packages are marked `SYSTEM`. All other functions (including user code and functions coming across the network) are marked `UNTRUSTED`. `UNTRUSTED` activation records are not allowed to set the privilege flag.

Effectively, every package has a global variable `Principal` which indicates whether the package is `SYSTEM` or `UNTRUSTED`. Packages which come across the network have this variable set to `UNTRUSTED` automatically on transfer. Activation records are classified as `SYSTEM` or `UNTRUSTED` based on the value of `Principal`, which is determined according to static scoping rules.

List all activation records (by number) that are marked `SYSTEM` and list all activation records (by number) that are marked `UNTRUSTED`. (Hint : Follow access link for each activation record)

- (c) `CheckPrivileges()` uses a dynamic-scoping approach to decide whether the function corresponding to the current activation record is allowed to perform privileged operations. The algorithm looks at all activation records on the stack, from most recent one up, until:

- It finds an activation record with the privilege flag set. In this case it returns `TRUE`. Or,

- It finds an activation record marked UNTRUSTED. In this case it returns FALSE. (Remember that it is not possible to set the privilege flag of an untrusted activation record.) Or,
- It runs out of activation records to look at. In this case it returns FALSE.

What will `CheckPrivileges()` return for the stack shown above (resulting from the call to `foo()` from `sys.main`? Please answer “True” or “False.”

- (d) Is there a security problem in this code? (I.e., will something undesirable or “evil” occur when this code is run?)
- (e) Suppose that `CheckPrivileges()` returned FALSE and thus a `SecurityException` was thrown. Which activation records from part (a) will be popped off the stack before the handler is found? List the numbers of the records.

2. Fairness

The guarded-command looping construct

```
do
    Condition  $\Rightarrow$  Command
    ...
    Condition  $\Rightarrow$  Command
od
```

involves nondeterministic choice, as explained in the text. An important theoretical concept related to potentially nonterminating nondeterministic computation is *fairness*. If a loop repeats indefinitely, then fair nondeterministic choice must eventually select each command whose guard is true. For example, in the loop

```
do
    true  $\Rightarrow$  x := x+1
    true  $\Rightarrow$  x := x-1
od
```

both commands have guards that are always true. It would be *unfair* to execute `x := x+1` repeatedly without ever executing `x := x-1`. Most language implementations are designed to provide fairness, usually by providing a bounded form. For example, if there are n guarded commands, then the implementation may guarantee that each enabled command will be executed at least once in every $2n$ or $3n$ times through the loop. Since the number $2n$ or $3n$ is implementation dependent, though, programmers should only assume that each command with a true guard will eventually be executed.

- (a) Suppose that an integer variable `x` can only contain an integer value with absolute value less than `INTMAX`. Will the `do ... od` loop above cause overflow or underflow under a fair implementation? What about an implementation that is not fair?
- (b) What property of the following loop is true under a fair implementation but false under an unfair implementation?

```
go := true;
n := 0;
do
    go  $\Rightarrow$  n := n+1
    go  $\Rightarrow$  go := false
od
```

- (c) Is fairness easier to provide on a single-processor language implementation or a multiprocessor? Discuss briefly.

3. Actor computing

The Actor mail system provides asynchronous buffered communication and does not guarantee that messages (*tasks* in Actor terminology) are delivered in the order they are sent. Suppose actor *A* sends tasks t_1, t_2, t_3, \dots to actor *B* and we want actor *B* to process tasks in the order *A* sends them.

- (a) What extra information could be added to each task so that *B* can tell whether it receives a task out of order? What should *B* do with a task when it first receives it, before actually performing the computation associated with the task?
- (b) Since the Actor model does not impose any constraints on how soon a task must be delivered, a task could be delayed an arbitrary amount of time. For example, suppose actor *A* sends tasks $t_1, t_2, t_3, \dots, t_{100}$ and actor *B* receives the tasks t_1, t_3, \dots, t_{50} without receiving task t_2 . Since *B* would like to proceed with some of these tasks, it makes sense for *B* to ask *A* to resend task t_2 . Describe a protocol for *A* and *B* that will add resend requests to the approach you described in part (a) of this problem.
- (c) Suppose *B* wants to do a final action when *A* has finished sending tasks to *B*. How can *A* notify *B* when *A* is done? Be sure to consider the fact that if *A* sends *I'm done* to *B* after sending task t_{100} , the *I'm done* message may arrive before t_{100} .

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.

- (a) 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 such a list. To provide clarity, write a Haskell statement that, given a list `L`, removes the second element. (*Hint*: Your statement should use only the functions `head`, `tail`, and the infix operator `:`)

- (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.

5. Java Concurrency

NPR's Car Talk radio show, in which Click and Clack, the Tappet brothers, give opinionated advice on all kinds of automotive ailments, wants to expand online. Their Computer Hardware Specialist, C. Colin Backslash, is trying to write a serverside Java program that can reply to user queries about car problems with auto-generated back and forth dialogue between the two show

hosts. Early on in the project, Colin encounters a puzzling concurrency problem and asks for your help. His code exhibits unpredictable deadlocks:

```
public class CarTalkGuy extends Thread {
    private String name;
    private CarTalkGuy brother;
    private static SummerIntern gladysOvernow = new SummerIntern();

    public CarTalkGuy(String name) {
        super();
        this.name=name;
    }
    public void setBrother(CarTalkGuy b) {
        brother=b;
    }
    public synchronized void dispenseAdvice() {
        String brokenPart = gladysOvernow.researchAnswer();
        System.out.print(name+": Clearly, your car needs a "+brokenPart+".");
        brother.add2Cents();
    }
    public synchronized void add2Cents() {
        System.out.print(name+": And remember: don't drive like my brother.");
        brother.haveLastWord();
    }
    public synchronized void haveLastWord() {
        System.out.println(name+": No, don't drive like *my* brother!");
    }
    public void run() {
        dispenseAdvice();
    }
    public static void main(String[] args) {
        CarTalkGuy click = new CarTalkGuy("Click");
        CarTalkGuy clack = new CarTalkGuy("Clack");
        click.setBrother(clack);
        clack.setBrother(click);
        click.start();
        clack.start();
    }
}
```

When the program executes successfully, it produces output like the following:

```
Click: Clearly, your car needs a new radiator. Clack: And remember:
don't drive like my brother. Click: No, don't drive like *my*
brother!
```

```
Clack: Clearly, your car needs a right blinker bulb. Click: And
remember: don't drive like my brother. Clack: No, don't drive like
*my* brother!
```

Reminder: In Java, you can create a thread by extending the `Thread` class and overriding the `Thread.run()` method. A call to `Thread.start()` returns immediately in the calling thread and executes `Thread.run()` in a newly created thread.

- (C) Locks protecting synchronized objects in Java are reentrant: threads that already own a particular lock are allowed to acquire the lock more than once. Which function calls, during a successful execution as given above, are only possible because of reentrant locks?

- (b) Sometimes the given program deadlocks, especially for complicated cases when the call to `researchAnswer()` may take a while to return. Explain, in terms of threads, locks, and synchronized objects, why deadlock may occur here. Your answer should not make any assumptions about class `SummerIntern`.
- (c) What will the program have output when the deadlock occurs?
- (d) After some thought, Colin suggests taking some pressure off the `SummerIntern` by passing in a hint after calling `researchAnswer()` and presents the following code, in which the `SummerIntern` waits until the hint is given:

```
public class SummerIntern {
    String hint=null;
    public synchronized String researchAnswer() {
        //...
        while (hint==null) wait();
        //...lookup and return answer
    }
    public synchronized void provideHint(String hint) {
        this.hint=hint;
        notify();
    }
}
```

To provide the hint, class `CarTalkGuy` gets an extra synchronized method :

```
public synchronized void provideHint() {
    gladysOvernow.provideHint("look under the hood");
}
```

Also, the last two lines of `main` are changed so that `click` will call `researchAnswer()` in one thread and also provide the hint in another thread:

```
click.start();
click.provideHint();
```

This change avoids the previous deadlock problem, since `Clack`'s thread is never started. To Colin's surprise though, the new program sometimes deadlocks without printing anything. In which lines of code do `Click`'s threads get stuck now during a deadlock?

- (e) Explain in one paragraph why the threads are deadlocked.

6. Race Conditions

The `DoubleCounter` class defined below has methods `incrementBoth` and `getDifference`. Assume that `DoubleCounter` will be used in multi-threaded applications.

```
class DoubleCounter {
    protected int x = 0, y = 0;

    public int getDifference() {
        return x - y;
    }
    public void incrementBoth() throws InterruptedException {
        x++;
        y++;
    }
}
```

There is a potential data race between `incrementBoth` and `getDifference` if `getDifference` is called between the increment of `x` and the increment of `y`. Assume for the questions below that one or more threads concurrently execute code contain calls to both `incrementBoth` and `getDifference`.

- (a) What are the possible return values of `getDifference` if there are 2 threads?
- (b) What are the possible return values of `getDifference` if there are n threads?
- (c) Data races can be prevented by inserting synchronization primitives. One option is to declare

```
public synchronized int getDifference() {...}
public int incrementBoth() {...}
```

This will prevent two threads from executing method `getDifference` at the same time. Is this enough to ensure that `getDifference` always returns 0? Explain briefly.

- (d) Is the following declaration

```
public int getDifference() {...}
public synchronized int incrementBoth() {...}
```

sufficient to ensure that `getDifference` always returns 0? Explain briefly.

- (e) What are the possible values of `getDifference` if the following declarations is used?

```
public synchronized int getDifference() {...}
public synchronized int incrementBoth() {...}
```

- (f) Atomicity is another concurrency control concept that is used in some newer language designs. If a block is declared atomic, then the implementation guarantees that the program output is equivalent to either completing the atomic block without interference from any other threads, or not running the block at all. Atomic blocks can be implemented by a variety of methods, including locking and/or mechanisms that allow a thread to rollback to a previous state if needed. However, one goal of atomicity as a language construct is to separate reasoning about correctness from details of how atomicity is achieved in a particular instance.

Using atomic blocks, we could declare `incrementBoth` atomic as follows:

```
public void incrementBoth() throws InterruptedException {
    atomic{
        x++;
        y++;
    }
}
```

If a thread calling `incrementBoth` is interrupted in the middle by a change to the value of `x` or `y`, the runtime system can rollback to the state before entering the atomic block. Then when the thread runs again it will continue from the beginning of the atomic block.

Will declaring `incrementBoth` atomic ensure that `getDifference` always returns 0? Explain.