

## Candy Factory Post-mortem, April 26, 2007

0. println should be synchronized when it is reporting the state of synchronized parts of the code. [-2%] Otherwise, even if the program is running correctly, the output may be lying. As a common example, suppose that the class Table stores its values in the following array

```
(*)      private volatile int [] tableValue = new tableValue [2];
```

And the Table class has a toString() method that creates a string to show what is on the table, say for example

```
public String toString() {
    return "" + tableValue[0] /*here*/ + tableValue[1];
}
```

Now if we create a table,

```
Table table = new Table();
```

I claim that at the point in execution labeled /\*here\*/ table could be updated by another thread, so that the state of the table that is reported is not correctly represented in the output either before or after that other thread. The solution is either to call toString() from a method synchronized against table, or to synchronize the toString() method against the table.

One student who wrote the individual values one at a time instead of creating the string first actually observed part of her table being written out and then another message and then the rest of the table being written out. Her reluctance to synchronize the print was knowing that we should synchronize small blocks of code. She probably thought that prints were very time-consuming. But in fact the time-consuming part of a print statement is not writing the thing to be printed to the output buffer, which is after all in RAM, but in getting the buffer to the screen. That can be done by DMA in parallel with computations, as long as what follows does not mess up the order of output. This parallelism behaving properly is a good example of what I called “conflict serializable” in lecture.

If you use relatively long delays to slow down the simulation, and just three client threads, you may never observe the behavior that her program demonstrated.

1. The volatile qualifier must be used. [-2%] Remember that data accessible to multiple threads must be updated as soon as it is changed, and not kept in a register or otherwise cached. We do that in Java by declaring such variables volatile. See the example at (\*) above. Prior to Java 1.4, there were problems with the consistent application of this, especially in multi-processor environments, but JVMs since then have worked.<sup>1</sup> Without the volatile qualifier, you do not know whether you are testing the current values of variables.

2. We want our processes to compete, not to cooperate, according to Criterion C2a of the assignment. [-2%] The operating system is a better judge of what should happen next than the various individual threads are, so I specified that your threads should not communicate in any way, except through the shared data of the table object, and that they should not intentionally cooperate. That includes not using wait() in one thread and notify() or notifyAll() in another because the thread that agrees to wait is trusting that another thread will notify it eventually. That also includes not using yield(), since each thread should try to do as much as it can without needless context switching. This is one part of Computer Science where the greedy approach is actually the best solution. This was especially a problem if you used notifyAll() in clients without a wait() in the agent, because it showed that you didn’t understand what notifyAll() did.

3. Extra credit for N >= 3. [+4% ]

---

<sup>1</sup><http://www-128.ibm.com/developerworks/java/library/j-jtp02244.html>

4. Extra credit for a GUI. [+10%]

5. Avoid busy waits. [-2%] I actually also took off 2% also if you did **not** have a busy wait but thought that you did. A “busy wait” means hogging the CPU doing nothing, spinning your wheels. That’s why a busy wait in an operating system is often called a “spin lock.” When a thread encounters a synchronized block that it cannot enter because the lock is held by another thread, the thread is removed from running, and put in a wait queue. A busy wait is an infinite loop that is executing. Even an infinite loop with a yield() or wait() in it is no longer a busy wait, since no CPU time is wasted running around the loop. Busy waits are most often encountered at a **lower level of abstraction** than “synchronized.” If you are looping, checking to see if a lock has been released over and over again, then you are busy waiting. The Java synchronized qualifier doesn’t do that; semaphores do. So Java’s new class ReentrantLock might busy wait (I don’t know whether it does), or on a uniprocessor, Peterson’s software solution to managing locks might busy wait.

6. Don’t halt production of a client to wait for other threads after the client has its ingredients. Begin making candy immediately. Of course it should take a simulated amount of time to complete the candy making, though. This is related to #2 above, about greed. [varying points off]

7. You synchronized too little. [-2% or more depending on how serious; for example, see #0 above as a not-serious example]

8. You should both prevent starvation, and argue that you did. For  $N < 3$ , starvation is possible. Some people were confused about whether  $N$  needs to be a multiple of 3 more generally when doing  $N \geq 3$ . There’s nothing wrong with 50 clients. Fewer candies will be made than with 51 ( $= 17 * 3$ ) clients, but 50 should still work fine.

Two students (Alan & Chad) in order to prevent starvation assumed an infinite table, modeling it as something like an ArrayList. I did not take off any credit in the code for this per se, but I take off credit if one did not reason carefully in the analysis about the role that a potentially infinite table played in the solution. It **is** unrealistic for a factory to produce infinitely many ingredients without waiting for the rest of the factory to consume them. Dell doesn’t even make a computer until it has an order, so it will have no inventory on hand. More importantly for our course, the wrappers, candy, and chocolate represent computer resources like CD ROMs, tape drives, and files. An operating system doesn’t make infinitely many copies of a file to make sure that it doesn’t deadlock when two procs want to write to the file at the same time.

9. You must synchronize against the same lock for all clients and the agent(s). [-2% if your error is detected by you experimentally and attempt to reason about it; -5% if you are oblivious to the error]

10. Atomic pickup of the 2 ingredients needed prevents deadlock. [-5% if you do not pick up atomically, but say that you have deadlock; -10% if you are oblivious to deadlock.]

11. Need sample i/o. [-10%] Sadly one person didn’t provide output as evidence that the program was working (or not working) as advertised.

12. You synchronized too much. [-2%, possibly more depending on how severe]

13. Some wondered what “robust” means. [-0%] It’s the opposite of “fragile.” A program is fragile if a small change would require a major rewriting because it does not have coherence and simplicity. How easy would it be to add a 2nd agent or 4th client? Does  $N$  need to be a multiple of 3? Another example of fragility is input that is not adequately bullet-proofed. I didn’t take off for any of this.