Java Monitor Objects:
Evaluating the Motivating Example

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

- Understand what monitors are & know how Java built-in monitor objects can ensure mutual exclusion & coordination between threads
- Note a human-known use of monitors
- Recognize common synchronization problems in concurrent Java programs
- Be aware of common complexities in concurrent programs like BuggyQueue
Evaluating the Buggy Producer/Consumer
Evaluating the Buggy Producer/Consumer

- Key question: what’s the output & why?

```
consumer : Thread

producer : Thread

buggyQueue : BuggyQueue

main()

QueueTest

new()

start()

run()

(offer("...")

poll()

new()

new()

start()

run()
```
Evaluating the Buggy Producer/Consumer

- Key question: what’s the output & why?

```
Exception in thread "Thread-1" java.lang.NullPointerException
at java.util.LinkedList.unlink(LinkedList.java:211)
at java.util.LinkedList.remove(LinkedList.java:526)
at edu.vandy.buggyqueue.model.BuggyQueue.poll(BuggyQueue.java:52)
at edu.vandy.BuggyQueueTest$Consumer.run(BuggyQueueTest.java:104)
at java.lang.Thread.run(Thread.java:745)
```

Depending on the implementation of the BuggyQueue class & the underlying LinkedList the app & test program may simply “hang”
Evaluating the Buggy Producer/Consumer

- Key question: what’s the output & why?

```java
static class BuggyQueue<E> implements BoundedQueue<E> {
    private LinkedList<E> mList = new LinkedList<E>();

    public boolean offer(E e) {
        if (!isFull()) { mList.add(e); return true; }
        else return false;
    }

    public E poll() {
        if (!isEmpty()) return mList.remove(0); else return false;
    }
}
```

There’s no protection against critical sections being run by multiple threads concurrently

```java
public E poll() {
    if (!isEmpty()) return mList.remove(0); else return false; }
...}
```

**Note that this implementation is not synchronized.** If multiple threads access a linked list concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements; merely setting the value of an element is not a structural modification.)

See [docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html](docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html)
Common Complexities in Concurrent Programs
Common Complexities in Concurrent Programs

- Concurrent programs are hard to develop & debug, due to various inherent & accidental complexities

See stackoverflow.com/questions/499634/how-to-detect-and-debug-multi-threading-problems
Common Complexities in Concurrent Programs

- Concurrent programs are hard to develop & debug, due to various inherent & accidental complexities, e.g.
  - Deadlock
    - *Occurs when two or more competing actions are each waiting for the other to finish, & thus none ever do*

See en.wikipedia.org/wiki/Deadlock
Concurrent programs are hard to develop & debug, due to various inherent & accidental complexities, e.g.

- **Deadlock**
- **Starvation**
  - A thread is perpetually denied necessary resources to process its work

See [en.wikipedia.org/wiki/Starvation_(computer_science)](en.wikipedia.org/wiki/Starvation_(computer_science))
Common Complexities in Concurrent Programs

- Concurrent programs are hard to develop & debug, due to various inherent & accidental complexities, e.g.
  - Deadlock
  - Starvation
  - Race conditions
    - Arise when an application depends on the sequence or timing of threads for it to operate properly

See [en.wikipedia.org/wiki/Race_condition](en.wikipedia.org/wiki/Race_condition)
Common Complexities in Concurrent Programs

- Concurrent programs are hard to develop & debug, due to various inherent & accidental complexities, e.g.
  - Deadlock
  - Starvation
  - Race conditions
  - Tool limitations
    - e.g., behavior in the debugger doesn’t reflect actual behavior

See en.wikipedia.org/wiki/Heisenbug

The act of observing a system can alter its state
Common Complexities in Concurrent Programs

- Some concurrency complexities can be fixed by applying Java built-in monitor object mechanisms

```
Producer offer()  > SimpleBlocking Queue
  synchronized put()  
  synchronized poll()  
  synchronized offer() 
  synchronized poll() 
Consumer poll()  < Wait Queue
  1 <<contains>>  
  1 <<contains>>  
  Entrance Queue
```

- Wait Queue
- Entrance Queue
Common Complexities in Concurrent Programs

- There are also helpful techniques for debugging concurrent software

Common Complexities in Concurrent Programs

- There are also helpful techniques for debugging concurrent software, e.g.
  - Use well-established concurrency & synchronization patterns & frameworks

Common Complexities in Concurrent Programs

• There are also helpful techniques for debugging concurrent software, e.g.
  • Use well-established concurrency & synchronization patterns
  • Conduct code reviews

See en.wikipedia.org/wiki/Code_review
Common Complexities in Concurrent Programs

- There are also helpful techniques for debugging concurrent software, e.g.
  - Use well-established concurrency & synchronization patterns
  - Conduct code reviews
  - Apply automated analysis tools

Static Analysis Tools for Concurrency

- **FindBugs** – works on Java. In the list of bugs detected all of the “Multithreaded correctness” bugs are relevant to concurrency. Command-line interface or eclipse plugin (eclipse plugin update site: http://findbugs.cs.umd.edu/eclipse/)
- **Lint** – a UNIX tool for C
- **JLint** – a Java version of Lint that is available as stand alone or eclipse plugin (eclipse plugin update site: http://www.jutils.com/eclipse-update)
- **Parasoft JTest** – commercial tool that combines static analysis and testing. Has capability to check for thread safety in multithreaded Java programs.
- **Coverity Static Analysis and Static Analysis Custom Checkers** – commercial tool that can be used to create custom static analyzers to find concurrency bugs in C/C++ programs.
- **GrammaTech’s CodeSonar** – commercial tool that can detect a special case race condition and locking issues in C/C++ (see datasheet for list of all bugs detected).
- **Chord** – static and dynamic analysis tool for Java (listed above as well).
- **JSure for Concurrency** – a commercial tool from SureLogic that is currently available in early release.
- **ESC/Java 2** – can detect race conditions and deadlocks – requires annotation (more…)
- **Relay** – static race detection
- **RacerX** – uses flow-sensitive static analysis tool for detection race conditions and deadlocks in C [paper] [slides]
- **SyncChecker** – a tool developed by F. Otto and T. Moschyn for finding race conditions and deadlocks in Java. Reduce false positives by combining static analysis with points-to and may-happen-in-parallel (MHP) information.
- **Warlock** – race detection tool for C – requires annotation.

See [www.sqrlab.ca/blog/2012/03/02/static-analysis-tools-for-concurrency](http://www.sqrlab.ca/blog/2012/03/02/static-analysis-tools-for-concurrency)
• There are also helpful techniques for debugging concurrent software, e.g.
  • Use well-established concurrency & synchronization patterns
  • Conduct code reviews
  • Apply automated analysis tools
  • Instrument code with logging & tracing statements

See www.dre.vanderbilt.edu/~schmidt/PDF/DSIS_Chapter_Waddington.pdf
End of Java Monitor Objects: Evaluating the Motivating Example