Java Monitor Objects:
Motivating Example & Common Concurrency Problems

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 synchronization problems in a buggy concurrent Java program

Concurrent calls to offer() & poll() corrupt internal state in the BuggyQueue fields
Learning Objectives in this Part of the Lesson

• Understand synchronization problems in a buggy concurrent Java program
• Recognize some common complexities of concurrent programs
A Buggy Producer /Consumer App
A Buggy Producer/Consumer App

• A concurrent producer/consumer app that attempts to pass messages via an “BuggyQueue” class

See github.com/douglascraigschmidt/POSA/tree/master/ex/M3/Queues/BuggyQueue
A Buggy Producer/Consumer App

- The BuggyQueue class is modeled on the Java ArrayBoundedQueue class

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBoundedQueue.html](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBoundedQueue.html)
A Buggy Producer/Consumer App

- UML class diagram showing the design of the BuggyQueue

A Buggy Producer/Consumer App

- UML sequence diagram of the BuggyQueue producer/consumer unit test

A Buggy Producer/Consumer App

- UML sequence diagram of the BuggyQueue producer/consumer unit test
A Buggy Producer/Consumer App

• UML sequence diagram of the BuggyQueue producer/consumer unit test

main() -> buggyQueue : BuggyQueue:

producer : Thread -> run()

consumer : Thread -> new()

buggyQueue : BuggyQueue -> new()

BuggyQueueTest : BuggyQueueTest

start() -> run()
A Buggy Producer/Consumer App

- UML sequence diagram of the BuggyQueue producer/consumer unit test

```
main() -> buggyQueue : BuggyQueue
    new()
    start()
```

```
consumer : Thread
    new()
    run()
```

```
producer : Thread
    new()
    run()
    offer("…")
    poll()
```
A Buggy Producer/Consumer App

- UML sequence diagram of the BuggyQueue producer/consumer unit test

Since the offer() & poll() methods aren’t synchronized chaos & insanity will result when this app and/or unit test is run!!
The BuggyQueue Implementation
The BuggyQueue Implementation

• The BuggyQueue class is a simply wrapper around Java’s LinkedList class

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

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

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

    ...
}
```

The BuggyQueue Implementation

• The BuggyQueue class is a simply wrapper around Java’s LinkedList class

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

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

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

    ...
```
The BuggyQueue Implementation

The BuggyQueue class is a simply wrapper around Java’s LinkedList class

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

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

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

    ...
}
```

See docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html
The BuggyQueue Implementation

- The BuggyQueue class is a simply wrapper around Java’s LinkedList class

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

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

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

    ...  
```

Non-synchronized public methods
static class BuggyQueue<E> implements BoundedQueue<E> {
    private LinkedList<E> mList = new LinkedList<>(); ...

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

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

    ...

See en.wikipedia.org/wiki/Robot_B-9
Evaluating the Buggy Producer/Consumer
Evaluating the Buggy Producer/Consumer

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

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)
Evaluating the Buggy Producer/Consumer

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

See stackoverflow.com/questions/499634/how-to-detect-and-debug-multi-threading-problems
Concurrent programs are hard to develop & debug, due to various complexities, e.g.

Deadlock

$\text{Occurs when two or more competing actions are each waiting for the other to finish, & thus none ever do}$

Evaluating the Buggy Producer/Consumer

- Concurrent programs are hard to develop & debug, due to various complexities, e.g.
  - Deadlock
  - Starvation
    - A thread is perpetually denied necessary resources to process its work

See en.wikipedia.org/wiki/Starvation_(computer_science)
Concurrent programs are hard to develop & debug, due to various complexities, e.g.

- **Deadlock**
- **Starvation**
- **Race conditions**
  
  - *Arise when an application depends on the sequence or timing of threads for it to operate properly*

Evaluating the Buggy Producer/Consumer

- Concurrent programs are hard to develop & debug, due to various 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
Evaluating the Buggy Producer/Consumer

- Some of these problems can be fixed by applying Java built-in monitor object mechanisms

```
SimpleBlockingQueue
synchronized put()
synchronized poll()
synchronized offer()
synchronized poll()
```

```
Producer
offer() → SimpleBlockingQueue
Wait Queue
<<contains>> 1
```

```
SimpleBlockingQueue
offer() → Consumer
Entrance Queue
<<contains>> 1
```

There are also helpful techniques for debugging concurrent software.

See www.drdobbs.com/cpp/multithreaded-debugging-techniques/199200938
End of Java Monitor Objects: Motivating Example