Principles of Software Construction:
Concurrency, Part 2

Josh Bloch Charlie Garrod

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

- Homework 5a due now
- Homework 5 framework goals:
  - Functionally correct
  - Well documented and easy to understand
  - Interesting
- 2nd midterm exam returned today, after class

Key concepts from Tuesday…

- `Runnable` interface represents work to be done
- To create a thread: `new Thread(Runnable)`
- To start thread: `thread.start();`
- To wait for thread to finish: `thread.join();`
- One `sychronized` static method runs at a time
- `volatile` – communication sans mutual exclusion
- ***Must*** synchronize access to shared mutable state
  – Else program will suffer safety and liveness failures

Outline

I. Discrete Event Simulation exam question
II. `Wait/Notify` - primitives for cooperation
III. The dangers of over-synchronization

DES specification summary

- Simulator steps through executing events
  - Time usually represented as an integer
- Events can do these things:
  - Change the simulated system state
  - Create and schedule new events to occur in future
  - Cancel a future event, given a reference to event
  - Stop the simulation
- Framework is sequential – no concurrency
  - Events scheduled for same time can run in any order

Minimal API for event and simulator

Event implementation

Simulator implementation (1)

Simulator implementation (2)

Zombie invasion spec summary

- Initial population: humans = 10^6, zombies = 4
- On first day, each zombie goes hunting
- When a zombie hunts, one of these things happen
  - p = .2, zombie infects human: zombies++, humans--
  - p = .2, zombie is destroyed: zombies--
  - p = .6, nothing happens (populations unchanged)
- If zombie survives, sleeps 1-10 days & hunts again
- Newly-infected zombie hunts day after infected
- Run till humans gone, zombies gone, or 100 years

Zombie invasion (1)

Zombie invasion (2)

Key design decisions

- **No class to represent state explicitly**
  - State is merely the variables shared by events
  - Eliminates need for generics
  - Occam's Razor / "When in doubt, leave it out"
- **Events have a `Runnable`, not a `run` method**
  - Enables use of anonymous class or lambda
  - "Favor composition over inheritance" [EJ Item 16]
- Pending events represented as `PriorityQueue`
  - Nice code and good performance

Outline

I. Discrete Event Simulation exam question
II. `Wait/Notify` - primitives for cooperation
III. The dangers of over-synchronization

The basic idea is simple…

- State (fields) protected by lock (`synchronized`)
- Sometimes, thread can't proceed till state is right
  - So it waits with `wait`
  - Automatically drops lock while waiting
- Thread that makes state right wakes waiting thread(s) with `notify`
  - Waking thread must hold lock when it calls `notify`
  - Waiting thread automatically gets lock when woken

But the devil is in the details
Never invoke wait outside a loop!

- Loop tests condition before and after waiting
- Test before skips wait if condition already holds
  - Necessary to ensure **liveness**
  - Without it, thread can wait forever!
- Testing after waiting ensure **safety**
  - Condition may not be true when thread wakens
  - If thread proceeds with action, it can destroy invariants!

All of your waits should look like this

```
synchronized (obj) {
    while (<condition does not hold>) {
        obj.wait();
    }

    ... // Perform action appropriate to condition
}
```


institute for SOFTWARE RESEARCH

Why can a thread wake from a `wait` when condition does not hold?

- Another thread can slip in between `notify` & wake
- Another thread can invoke `notify` accidentally or maliciously when condition does not hold
  - This is a flaw in java locking design!
  - Can work around flaw by using private lock object
- Notifier can be liberal in waking threads
  - Using `notifyAll` is good practice, but causes this
- Waiting thread can wake up without a `notify`(!)
  - Known as a *spurious wakeup*

Example: read-write locks (API)

Also known as shared/exclusive mode locks

```
private final RwLock lock = new RwLock();

lock.readLock();
try {
    // Do stuff that requires read (shared) lock
} finally {
    lock.unlock();
}

lock.writeLock();
try {
    // Do stuff that requires write (exclusive) lock
} finally {
    lock.unlock();
}
```

Example: read-write locks (Impl. 1)

```
public class RwLock {
    // State fields are protected by RwLock's intrinsic lock

    /** Num threads holding lock for read. */
    private int numReaders;

    /** Whether lock is held for write. */
    private boolean writeLocked;

    public synchronized void readLock() throws InterruptedException {
        while (writeLocked) {
            wait();
        }
        numReaders++;
    }
```

Example: read-write locks (Impl. 2)

```
    public synchronized void writeLock() throws InterruptedException {
        while (numReaders != 0 || writeLocked) {
            wait();
        }
        writeLocked = true;
    }

    public synchronized void unlock() {
        if (numReaders > 0) {
            numReaders--;
        } else if (writeLocked) {
            writeLocked = false;
        } else {
            throw new IllegalStateException("Lock not held");
        }
        notifyAll(); // Wake any waiters
    }
}
```

Caveat: `RwLock` is just a toy!

- It has poor fairness properties
  - Readers can starve writers!
- `java.util.concurrent` provides an industrial strength `ReadWriteLock`
- More generally, avoid `wait/notify`
  - In the early days it was all you had
  - Nowadays, higher level concurrency utils are better

Outline

I. Discrete Event Simulation exam question
II. `Wait/Notify` - primitives for cooperation
III. The dangers of over-synchronization

Broken Work Queue (1)

```
public class WorkQueue {
    private final Queue<Runnable> queue = new ArrayDeque<>();
    private boolean stopped = false;
    public WorkQueue() {
        new Thread(() -> {
            while (true) { // Main loop
                synchronized (queue) { // Locking on private obj.
                    try {
                        while (queue.isEmpty() && !stopped)
                            queue.wait();
                    } catch (InterruptedException e) {
                        return;
                    }
                    if (stopped) return;  // Causes thread to end
                    queue.remove().run(); // BROKEN - LOCK HELD!
                }
            }
        }).start();
    }
```

Broken Work Queue (2)

```
Broken Work Queue (2)
    public final void enqueue(Runnable workItem) {
        synchronized (queue) {
            queue.add(workItem);
            queue.notify();
        }
    }
    public final void stop() {
        synchronized (queue) {
            stopped = true;
            queue.notify();
        }
    }
}
```

Perverse use of that shows flaw

```
public static void main(String[] args) {
    WorkQueue wq = new WorkQueue();

    // Enqueue task that starts thread that enqueues task...
    wq.enqueue(() -> {
        Thread t = new Thread(() -> {
            wq.enqueue(() -> { System.out.println("Hi Mom!"); });
        });

        // ...and waits for thread to finish
        t.start();
        try {
            t.join();
        } catch (InterruptedException e) {
            throw new AssertionError(e);
        }
    });
}
```

Luckily, it's easy to fix the deadlock

```
public WorkQueue() {
    new Thread(() -> {
        while (true) { // Main loop
            Runnable task = null;
            synchronized (queue) {
                try {
                    while (queue.isEmpty() && !stopped)
                        queue.wait();
                } catch (InterruptedException e) {
                    return;
                }
                if (stopped) return;  // Causes thread to terminate
                task = queue.remove();
            }
            task.run(); // Fixed! "Open call" (no lock held)
        }
    }).start();
}
```

Never do callbacks while holding lock

- It is *over-synchronization*
- We saw it deadlock
- And it can do worse!
  - If the callback goes back into the module holding the lock, it will not block, and can damage invariants!
- So always drop any locks before callbacks
  - You may have to copy the callbacks under lock

Summary

- Discrete Event/Zombie problem was long & hard
  - But sol'n could be short & sweet with good design choices
- **Never use wait outside of a while loop!**
  - Think twice before using it at all
- **Neither an under- nor an over-synchronizer be**
  - Under-synchronization causes safety (& liveness) failures
  - Over-synchronization causes liveness (& safety) failures