Synchronization

Thread is a double-edged sword

Shared data

- All threads can access shared data directly
- Atomicity Issues

Race condition

```
public class UnsafeSequence {
    private int value = 0;
    public int getNextID() {
        return value++;
    }
}
```

- Thread A & B call getNext
- Duplicated ID

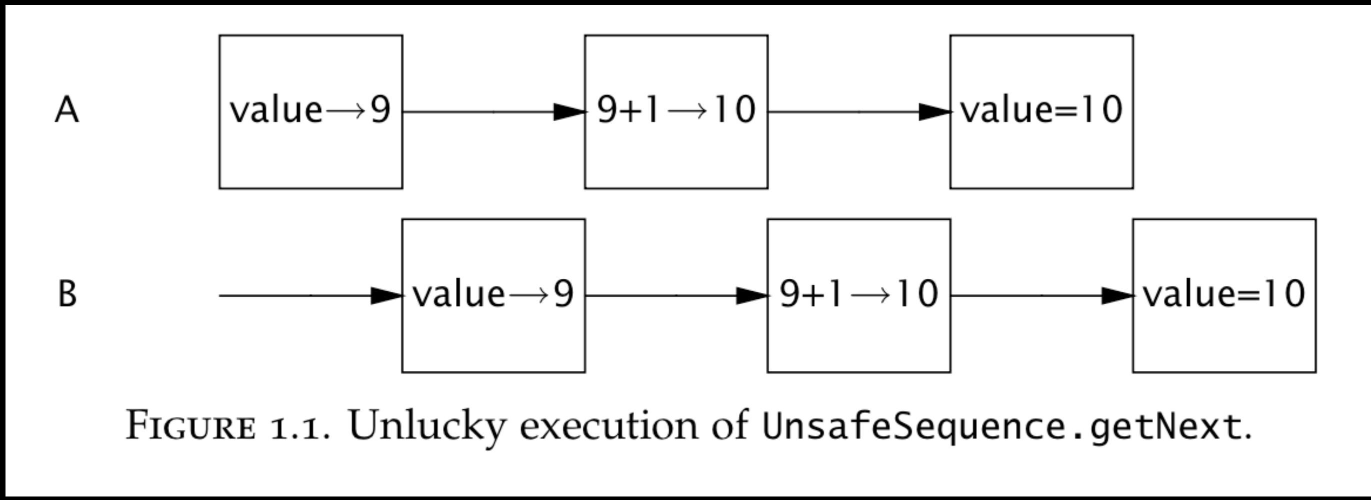

FIGURE 1.1. Unlucky execution of UnsafeSequence.getNext.

Check and act

```
if (!list.contains(element)) {
    list.add(element);
}
```

Inconsistence

Object is composed multiple parts can be in inconsistent state if race condition happens.

1. Double = 64 bits => needs 2 operations.
2. (d1, d2) <- (a1, a2)
3. (d1, d2) <- (b1, b2)
4. Race condition => (d1 == a1) and (d2 == b2) ???

Sequence of operations needs to be `atomic`

Solution 1: Atomic Variable

```java
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicSequence {
    private AtomicInteger counter = new AtomicInteger(0);
    public int getNext() {
        return counter.incrementAndGet();
    }
}
```

Solution 2: Intrinsic Lock

Built in lock on all Java object
- Synchronized statements
- Synchronized Method

Synchronized block

```
public void addIfNotExist(Element e) {
    synchronized(this) {
        if (!this.contains(element)) {
            this.add(element);
        }
    }
}
```

- Locks current object this and check and act

Synchronized method

```
public void addIfNotExist(Element e) {
    synchronized(this) {
        if (!this.contains(element)) {
            this.add(element);
        }
    }
}

public synchronized void addIfNotExist(Element e) {
    if (!this.contains(element)) {
        this.add(element);
    }
}
```

Synchronized Notes

- Select appropriate `object` to lock
- `synchronized` method is re-entrant
- Re-entrant means one thread is able to acquire same lock object many times
- Synchronized solves `Visibility` issue

Common mistakes

- Only `lock` write operations
- Select incorrect object to lock (synchronized)
- Group of statements to lock is too small or too big

Visibility Demo

Not always failed

```java
class Reader extends Thread {
  public boolean completed = false;
  public int result;

  public void run() {
    while(!completed){ Thread.yield(); }
    System.out.println("Result: " + result);
  }

  public static void main(String[] args) {
    Reader reader = new Reader();
    reader.start();
    reader.result = 10;
    reader.completed = true;
  }
}
```

Visibility Issues

- Updating values may not be visible
- Reordering

Declare variable with `volatile` to force JVM to make changes visiable

Taking advantage of multipleprocessors