Lecture 34: Concurrency IV

CS 62

Spring 2018

Alexandra Papoutsaki & William Devanny

Some slides based on those fom Dan Grossman, U. of Washington

Race Conditions

- A *race condition* occurs when the computation result depends on scheduling (how threads are interleaved)
  - If T1 and T2 happened to get scheduled in a certain way, things go wrong
  - Since we do not control scheduling, we need to write programs that work *independent of scheduling*
- Race conditions are bugs that exist only due to concurrency
  - No interleaved scheduling problems with only 1 thread.
- Typically, problem is that some *intermediate state* can be seen by another thread; screws up other thread.

Data Races vs Bad Interleavings

- We will make a big distinction between these terms
- Both are kinds of race-condition bugs
- Confusion often results from not distinguishing these or using the ambiguous "race condition" to mean only one

Data races (briefly)

- A ***data race*** is a specific type of *race condition* that can happen in 2 ways:
    - Two different threads *potentially* write a variable at the same time
    - One thread *potentially* writes a variable while another reads the variable
- Not a race: simultaneous reads provide no errors
- "Potentially" is important
    - We claim the code itself has a data race independent of any particular actual execution
- Data races are bad, but we can still have a race condition, and bad behavior, when no data races are present...through ***bad interleavings*** (what we will discuss now).

Stack Example

```
class Stack<E> {
      private E[] array;
      private int index = 0;
      Stack(int size) {
            array = (E[]) new Object[size];
      }
      synchronized boolean isEmpty() {
            return index==0;
      }
      synchronized void push(E val) {
            if(index==array.length)
                  throw new StackFullException();
            array[index++] = val;
      }
      synchronized E pop() {
            if(index==0)
                  throw new StackEmptyException();
            return array[--index]; } }
```

Let's implement peek()

```
synchronized E peek() {
    if(index==0)
        throw new StackEmptyException();
    return array[index-1];
}
```

correct

```
class C {
    static <E> E myPeekHelper(Stack<E> s) {
        synchronized (s) {
            E ans = s.pop();
            s.push(ans);
            return ans;
    } } }
```

Weird, but correct

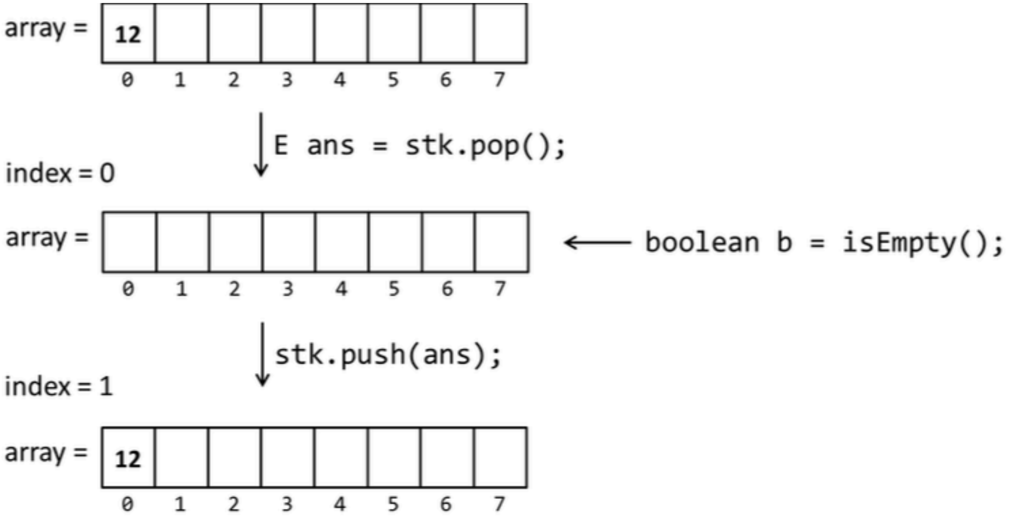
Example of race condition, not data race

```
class C {
    static <E> E myPeekHelper(Stack<E> s) {

        E ans = s.pop();
        s.push(ans);
        return ans;
    } }
```

- *No overall* effect on the shared data. State should be the same at the end
- But the way it is implemented creates an inconsistent ***intermediate state***
- There is still a *race condition* though. This intermediate state should not be exposed → ***bad interleavings***

peek() and isEmpty()

```
Thread 1                     Thread 2 (calls myPeekHelperWrong)
--------                     ----------------------------------

                             E ans = stk.pop();
boolean b = isEmpty();
                             stk.push(ans);
                             return ans;
```



array = | 12 | | | | | | | |
0 1 2 3 4 5 6 7

E ans = stk.pop();

index = 0

array = | | | | | | | | |
0 1 2 3 4 5 6 7

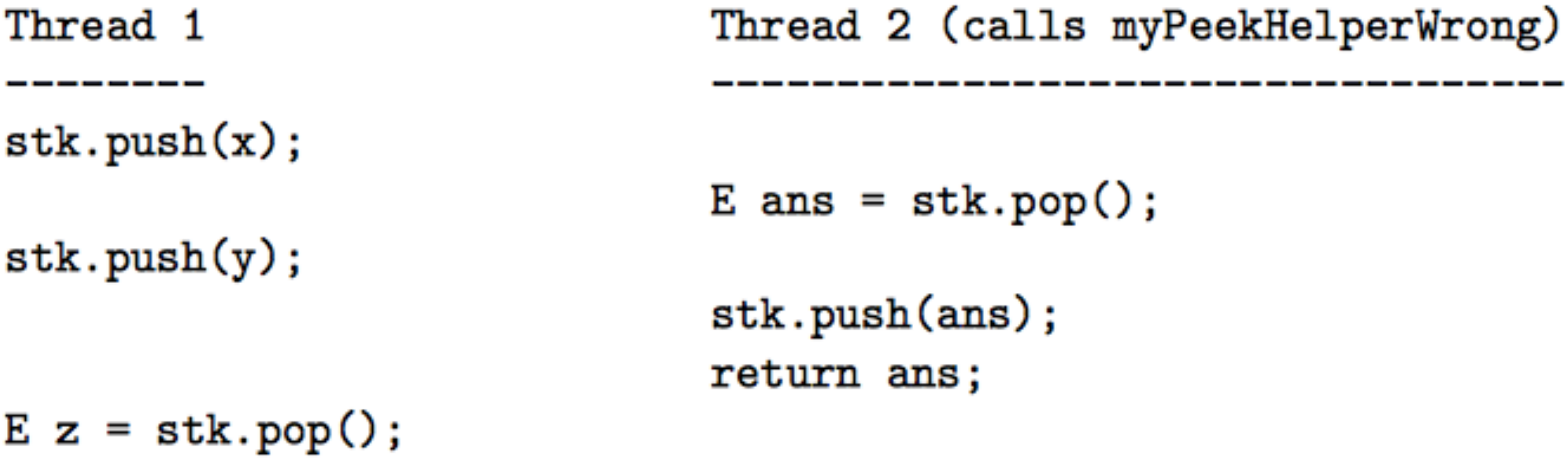
← boolean b = isEmpty();

stk.push(ans);

index = 1

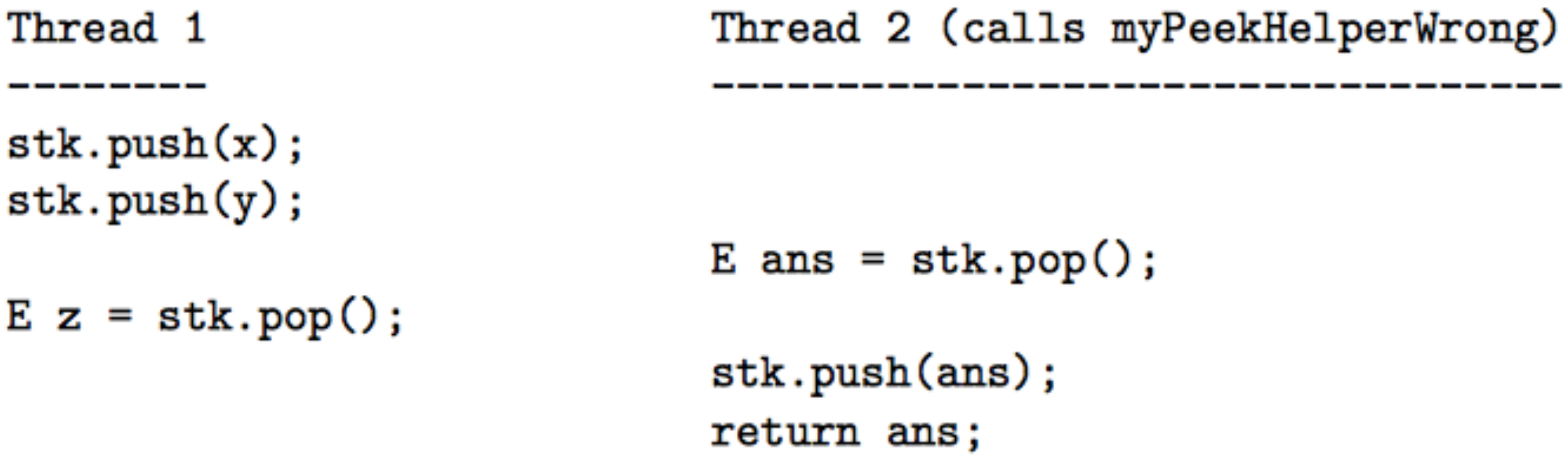
array = | 12 | | | | | | | |
0 1 2 3 4 5 6 7

peek() and push()



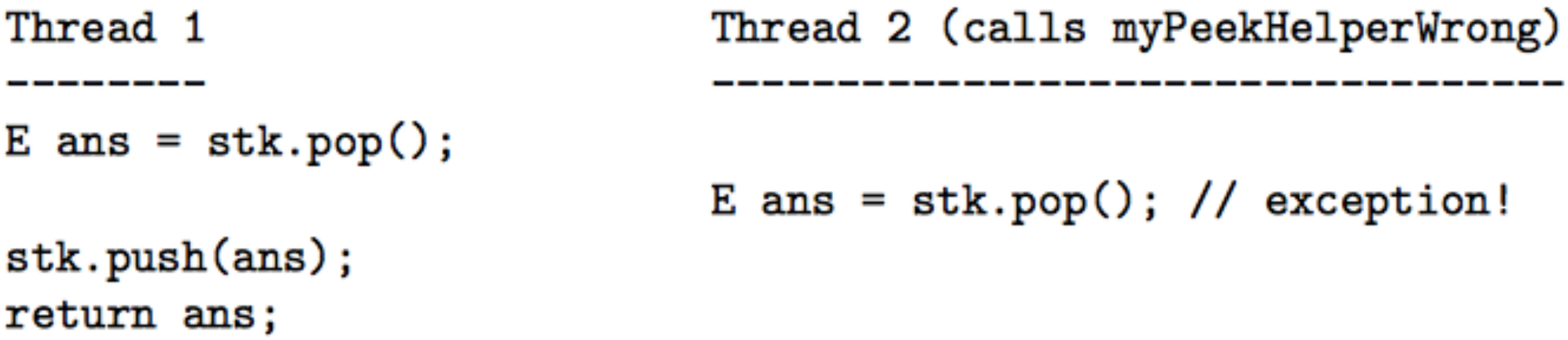
```
Thread 1                      Thread 2 (calls myPeekHelperWrong)
---------                     ----------------------------------
stk.push(x);
                              E ans = stk.pop();
stk.push(y);
                              stk.push(ans);
                              return ans;

E z = stk.pop();
```

peek() and pop()



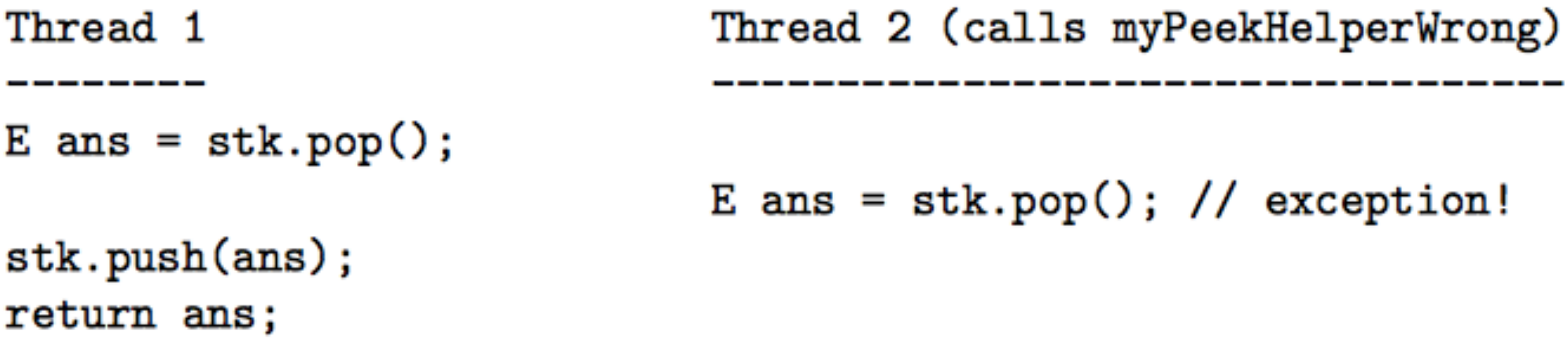
```
Thread 1                    Thread 2 (calls myPeekHelperWrong)
--------                    ----------------------------------
stk.push(x);
stk.push(y);
                            E ans = stk.pop();
E z = stk.pop();
                            stk.push(ans);
                            return ans;
```

`peek()` and `peek()` on 1 element



```
Thread 1                    Thread 2 (calls myPeekHelperWrong)
--------                    ----------------------------------
E ans = stk.pop();
                            E ans = stk.pop(); // exception!
stk.push(ans);
return ans;
```

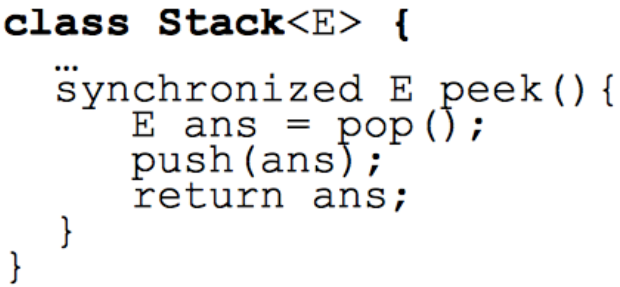
peek() and peek() on > 1 element



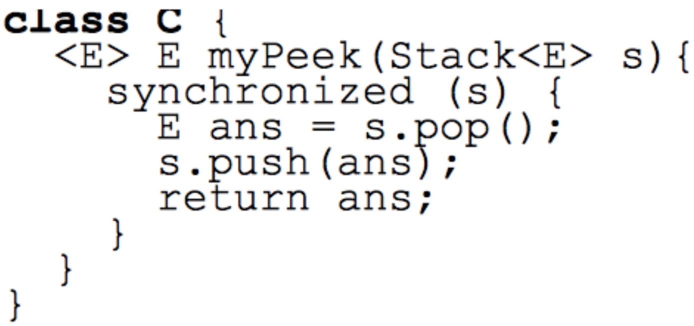
```
Thread 1                    Thread 2 (calls myPeekHelperWrong)
--------                    ----------------------------------
E ans = stk.pop();
                            E ans = stk.pop(); // exception!
stk.push(ans);
return ans;
```

The fix

- `peek` needs synchronization to disallow interleavings
  - The key is to make a *larger critical section*
  - That intermediate state of `peek` needs to be protected
- Use re-entrant locks; will allow calls to `push` and `pop`
- Code on right is example of a peek external to the `Stack` class



```
class Stack<E> {
  …
  synchronized E peek(){
     E ans = pop();
     push(ans);
     return ans;
  }
}
```

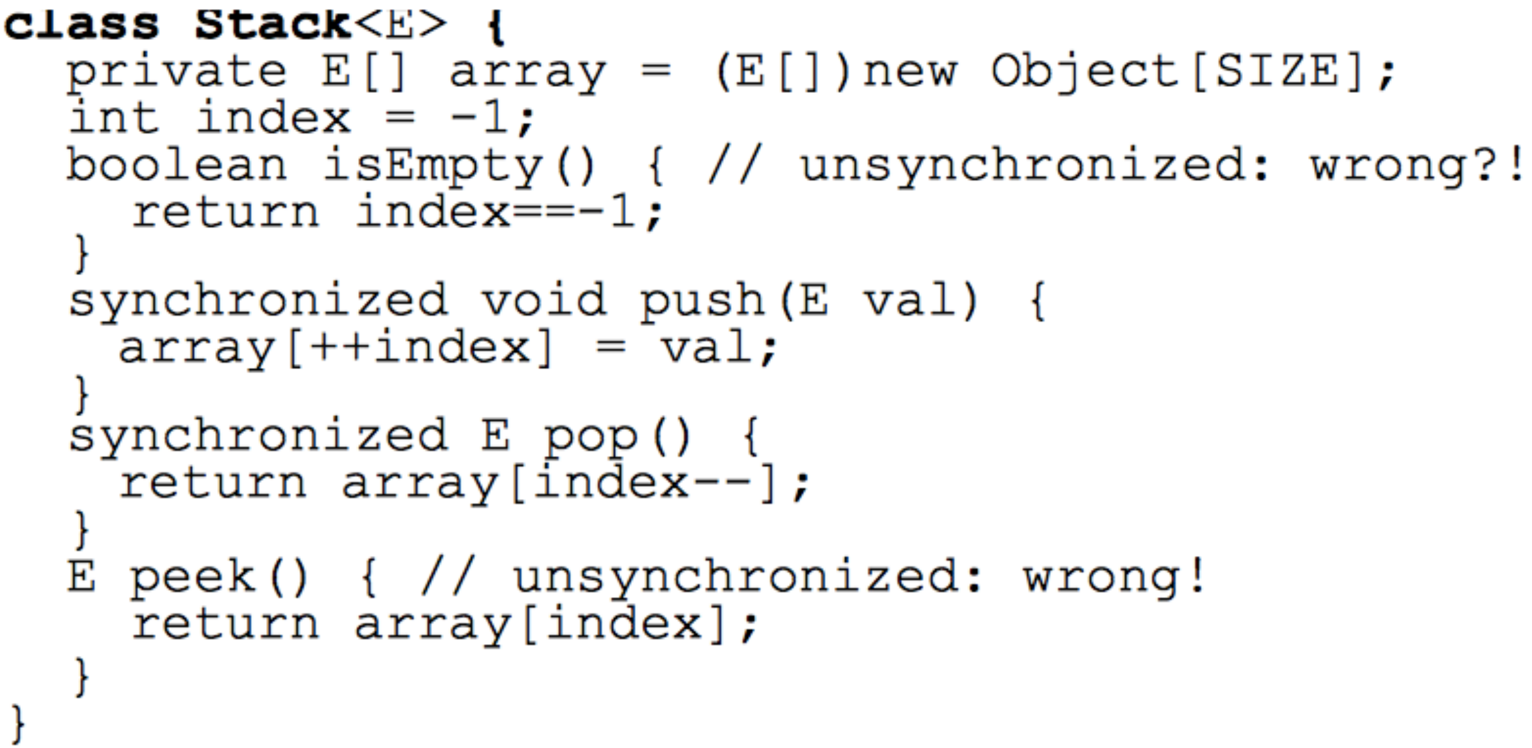


```
class C {
  <E> E myPeek(Stack<E> s){
    synchronized (s) {
      E ans = s.pop();
      s.push(ans);
      return ans;
    }
  }
}
```

The wrong fix

- **Focus so far**: problems from `peek` doing writes that lead to an incorrect intermediate state
- **Tempting but wrong**: If an implementation of `peek` (or `isEmpty`) does not write anything, then maybe we can skip the synchronization?
- Does not work due to *data races* with `push` and `pop`...

Example



```
class Stack<E> {
  private E[] array = (E[])new Object[SIZE];
  int index = -1;
  boolean isEmpty() { // unsynchronized: wrong?!
    return index==-1;
  }
  synchronized void push(E val) {
   array[++index] = val;
  }
  synchronized E pop() {
   return array[index--];
  }
  E peek() { // unsynchronized: wrong!
    return array[index];
  }
}
```

Why wrong?

- It *looks like* `isEmpty` and `peek` can “get away with this” since `push` and `pop` adjust the state “in one tiny step”
- But this code is still *wrong* and depends on language-implementation details you cannot assume
  - Even “tiny steps” may require multiple steps in the implementation:
- `array[++index] = val;` probably takes at least two steps
  - Code has a data race, allowing very strange behavior
- Moral: Do not introduce a data race, even if every interleaving you can think of is correct

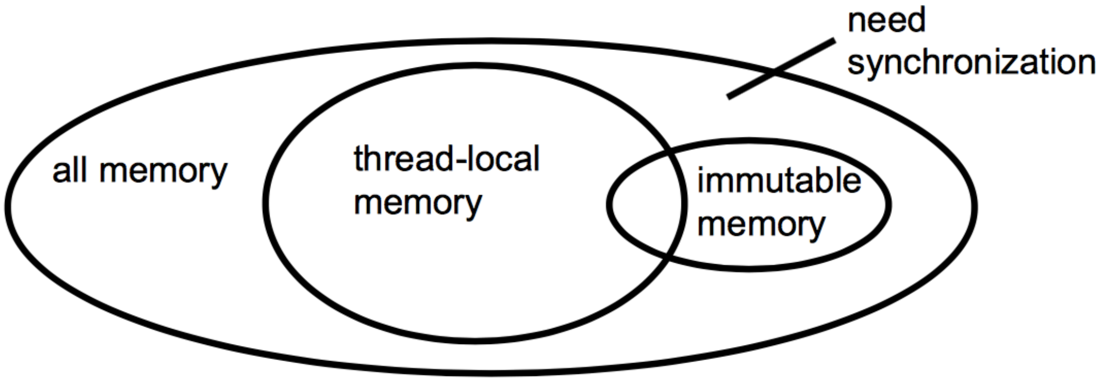
Getting it right

- Avoiding race conditions on shared resources is difficult
  - What "seems fine" in a sequential world can get you into trouble when multiple threads are involved.
  - Decades of bugs have led to some *conventional wisdom*: general techniques that are known to work
- Next we discuss this conventional wisdom!

3 choices

- For every memory location (e.g., object field) in your program, you must obey at least one of the following:

1. Thread-local: Do not use the location in > 1 thread
2. Immutable: Do not write to the memory location
3. Shared-and-mutable: Use synchronization to control access to the location



1. Thread-local

- Whenever possible, do not share resources
  - Easier to have each thread have its own **thread-local *copy*** of a resource than to have one with shared updates
  - This is correct only if threads do not need to communicate through the resource
    - That is, multiple copies are a correct approach
- Note: Because each call-stack is thread-local, never need to synchronize on local variables
- *In typical concurrent programs, the vast majority of objects should be thread-local: shared-memory should be rare – minimize it!*

2. Immutable

- Whenever possible, don’t update objects
  - Make new objects instead
- One of key tenets of functional programming
  - You did study this in 52
  - Generally helpful to avoid side-effects
  - Much more helpful in a concurrent setting
- If a location is only read, never written, no synchronization is necessary!
  - Simultaneous reads are not races and not a problem
- *Programmers over-use mutation – minimize it!*

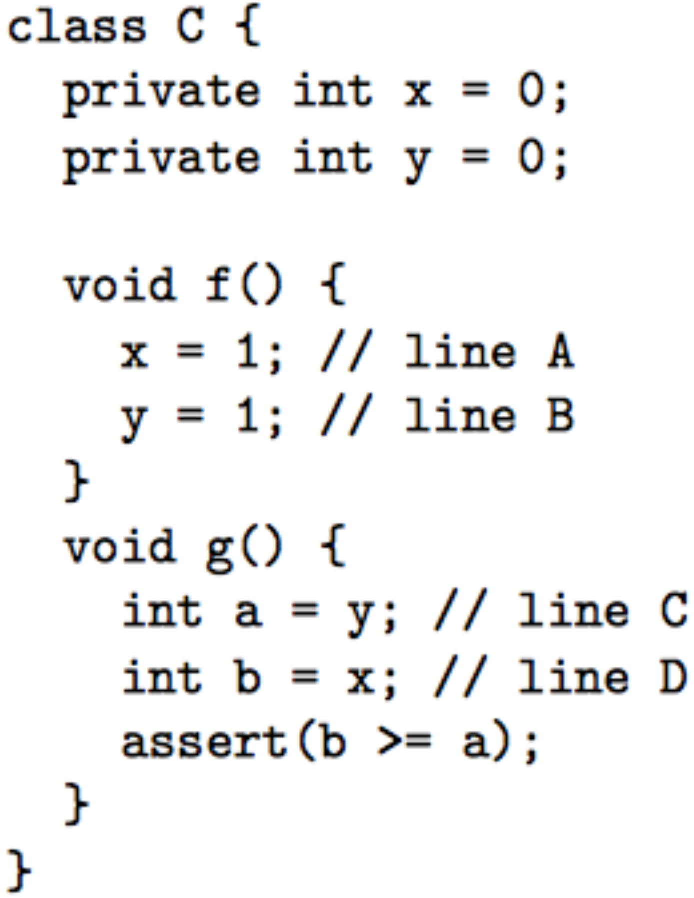
3. The rest: keep it synchronized

- After minimizing the amount of memory that is (1) thread-shared and (2) mutable, we need guidelines for how to use locks to keep other data consistent
- **Guideline:** No data races
  - *Never allow two threads to read/write or write/write the same location at the same time* (use locks!)
  - Even if it 'seems safe'
- Necessary: A Java or C program with a data race is almost always wrong
- *But Not sufficient*: Our `peek` example had no data races, and it's still wrong...

Worse than you think

Assertion always true w/ single threaded.

- Looks always true for multithreaded.
- OK if `f` not called at all
- OK after `f` completes
- Looks OK if in middle of `f`
- But has race condition



```
class C {
  private int x = 0;
  private int y = 0;

  void f() {
    x = 1; // line A
    y = 1; // line B
  }
  void g() {
    int a = y; // line C
    int b = x; // line D
    assert(b >= a);
  }
}
```

Memory reordering

- For performance reasons, compiler and hardware reorder memory operations.
- But, but, ...
  - Compiler/hardware will never perform a memory reordering
  - that affects the result of a single-threaded program
  - The compiler/hardware will never perform a memory reordering that affects the result of a data-race-free multi- threaded program
- So: If no interleaving of your program has a data race, then need not worry: result will be equivalent to some interleaving