

Race Conditions: Data Races vs. Bad Interleavings

We will make a big distinction between:

data races and *bad interleavings*

- Both are kinds of **race-condition** bugs
- Confusion often results from not distinguishing these or using the ambiguous “race condition” to mean only one

2/18/2022

4

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).

2/18/2022

5

Stack Example (pseudocode)

```
class Stack<E> {
    private E[] array = (E[])new Object[SIZE];
    private int index = -1;
    synchronized boolean isEmpty() {
        return index==-1;
    }
    synchronized void push(E val) {
        array[++index] = val;
    }
    synchronized E pop() {
        if(isEmpty())
            throw new StackEmptyException();
        return array[index--];
    }
}
```

2/18/2022

6

Example of a Race Condition, but not a Data Race

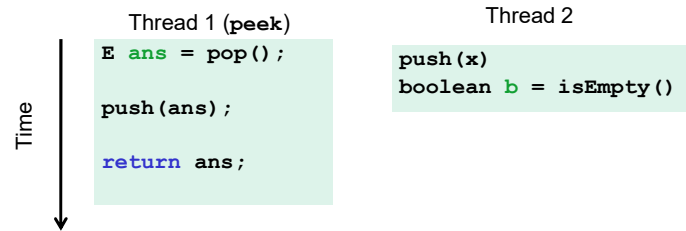
```
class Stack<E> {
    ... // state used by isEmpty, push, pop
    synchronized boolean isEmpty() { ... }
    synchronized void push(E val) { ... }
    synchronized E pop() {
        if(isEmpty())
            throw new StackEmptyException();
        ...
    }
    E peek() { // this is wrong
        E ans = pop();
        push(ans);
        return ans;
    }
}
```

2/18/2022

7

Example 1: peek and isEmpty

- **Property we want:** If there has been a `push` (and no `pop`), then `isEmpty` should return `false`
- With `peek` as written, property can be violated – how?

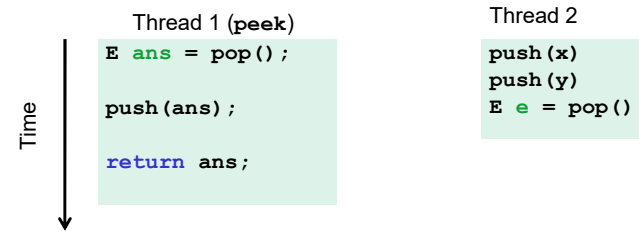


2/18/2022

10

Example 2: peek and push

- **Property we want:** Values are returned from `pop` in LIFO order
- With `peek` as written, property can be violated – how?

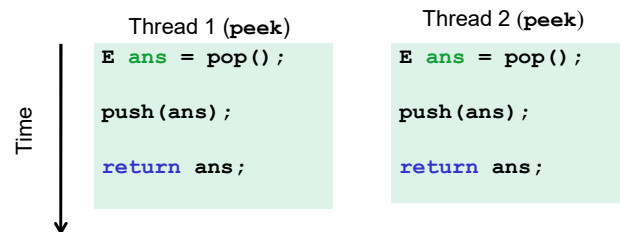


2/18/2022

13

Example 4: peek and peek

- **Property we want:** `peek` doesn't throw an exception unless stack is empty
- With `peek` as written, property can be violated – how?



2/18/2022

16

The fix

- In short, `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;
    }
}
}
```

2/18/2022

18