Race Conditions!

CSE 332 Spring 2021

Instructor: Hannah C. Tang

Teaching Assistants:

Aayushi Modi	Khushi Chaudhari	Patrick Murphy
Aashna Sheth	Kris Wong	Richard Jiang
Frederick Huyan	Logan Milandin	Winston Jodjana
Hamsa Shankar	Nachiket Karmarkar	

gradescope

gradescope.com/courses/256241

- Why does a lock require help from the operating system for a correct implementation?

Announcements

- P3 released, contact your partner soon!

- Mini projects ("para") due tonight!

Lecture Outline

- **Re-entrancy**
- Locking in Java
- Race Conditions: **Data Race**s vs. **Bad Interleaving**s

Lecture questions: pollev.com/cse332

Review: The Lock ADT

- A **Lock** ADT with operations **`new`**, **`acquire`**, **`release`**
  - Only one executor may acquire an instance of the lock at one time
  - Given simultaneous acquires/releases, a "correct thing" will happen
    - Specifically: if we have two acquires: one will "win" and one will block

- Needs hardware and O/S support
  - Needs special "check if held; if not, make held" single operation
    - See computer-architecture or operating-systems course
  - In CSE 332, we take this as a primitive and use it

- Used by threads to synchronize access to critical sections
  - Therefore, must be accessible to multiple threads

BankAccount Example

```
class BankAccount {
  private int balance = 0;
  private Lock lk = new Lock();

  protected int  getBalance()       { return balance; }
  protected void setBalance(int x) { balance = x; }

  public void withdraw(int amount) {
    lk.acquire();  // may block
    int b = getBalance();
    if (amount > b)
      throw new WithdrawTooLargeException();
    setBalance(b - amount);
    lk.release();
  }

  // deposit() would also acquire/release lk
}
```

Critical section

Note: 'Lock' is not an actual Java class

gradescope

gradescope.com/courses/256241

1. Where is the critical section?

2. How many locks do we need?
   a) One lock per BankAccount object? ✓
   b) Two locks per BankAccount object?
      - i.e., one for withdraw() and one for deposit()
   c) One lock for the entire Bank
      - Bank contains multiple BankAccount instances

3. There is a bug in withdraw(), can you find it?
   early exit due to throw

4. Do we need locks for:
   a) getBalance?
   b) setBalance?

Some Common Locking Mistakes (1 of 2)

- A lock is very primitive; up to you to use correctly
- **Incorrect**: different locks for **`withdraw`** and **`deposit`**
  - Mutual exclusion works only when sharing same lock
  - **`balance`** field is the shared resource being protected
- **Poor performance**: same lock for entire Bank
  - No simultaneous operations on *different* accounts

Some Common Locking Mistakes (2 of 2)

- **Bug**: forgot to release a lock when exiting early
  - Can block other threads forever if there's an exception

```
if (amount > b) {
  lk.release(); // hard to remember!
  throw new WithdrawTooLargeException();
}
```

Remembering to release() before every exit is challenging!

- What about **`getBalance`** and **`setBalance`**?
  - Assume now that they are **`public`** (which may be reasonable)
  - If they *do not* acquire the same lock, then **`setBalance`** and **`withdraw`** could interleave badly and produce a wrong result
  - If they *do* acquire the same lock, then **`withdraw`** would block forever because it tries to acquire a lock it already has!

One (Not Very Good) Possibility

- Have two versions of setBalance!
  - withdraw() calls a non-locking version of setBalance() (since it already has the lock)
  - Outside world calls the locking version of setBalance()
- Could work if adhered to, but inconvenient
- Alternately, we can modify the meaning of the Lock ADT to support **re-entrant locks**
  - Java does this
  - Then just always use the locking version of setBalance()

```
private int setBalanceNoLock(
    int x) {
  balance = x;
}

public int setBalance(int x) {
  lk.acquire();
  setBalanceNoLock(x)
  lk.release();
}
public void withdraw(int amount) {
  lk.acquire();
  …
  setBalanceNoLock(b – amount);
  lk.release();
}
```

Re-entrancy

- A **re-entrant lock** (a.k.a. **recursive lock**)
  - Once acquired, the lock is held *by the executor*,
  - Subsequent acquire calls *in that executor* won't block

- Example:
  - withdraw() can acquire the lock
  - Then, withdraw() can call setBalance(), which also acquires the lock
  - Because they're *in the same executor* and it's a *re-entrant* lock, the inner acquire won't block!

Re-entrant Lock Implementation

- Contains the following state:
  - the thread (if any) that currently holds it and a count
- When the lock goes from not-held to held:
  - remembers the thread and sets count = 0
- If the *current holder* calls acquire() again:
  - it does not block and count++
- If the *current holder* calls release():
  - if count > 0 and count--
  - if count == 0, the lock "forgets" the thread

Lecture Outline

- Re-entrancy
- **Locking in Java**
- Race Conditions: **Data Race**s vs. **Bad Interleaving**s

Java's Re-entrant Lock

- Java doesn't have the "plain" lock we discussed earlier; it only has re-entrant locks

- `java.util.concurrent.locks.ReentrantLock`
  - Has methods lock() and unlock()

Locking Best Practices in Java

- Remember our bug in withdraw()?

- Need to guarantee that locks are always released
  - Recommend something like this:

```
mylock.lock();
try { /* method body */ }
finally { mylock.unlock(); }
```

  - The code in `finally` will always execute afterwards
    - Regardless of exceptions, returns, or "normal" completion

`synchronized`: A Java Convenience

- Or use `synchronized` statement instead of explicitly instantiating a `ReentrantLock` + try/catch/finally blocks

```
synchronized (expression) {
  statements
}
```

- `synchronized` statement:
  - Evaluates *expression* to an object
    - Every object (but not primitive types) can be a lock in Java
  - Acquires the lock, blocking if necessary
    - "If you get past the {, you have the lock"
  - Releases the lock "at the matching }", even if throw, return, etc.
    - So it's impossible to forget to release the lock

Version #1: Correct, But Can Be Improved

```
class BankAccount {
  private int balance = 0;
  private Object lk = new Object();

  protected int getBalance()
    { synchronized (lk) { return balance; } }
  protected void setBalance(int x)
    { synchronized (lk) { balance = x; } }

  public void withdraw(int amount) {
    synchronized (lk) {
      int b = getBalance();
      if (amount > b)
        throw …
      setBalance(b – amount);
    }
  }

  // deposit() would also use synchronized(lk)
}
```

Improving Version #1

- As written, the lock is **private**
  - Seems like a good idea … ?
  - But prevents other classes from synchronizing with BankAccount operations

- More idiomatic is to synchronize on **`this`**
  - Also more convenient: no need to have an extra object!

Version #2: Still Improvable

```
class BankAccount {
  private int balance = 0;

  protected int getBalance()
    { synchronized (this){ return balance; } }
  protected void setBalance(int x)
    { synchronized (this){ balance = x; } }

  public void withdraw(int amount) {
     synchronized (this) {
       int b = getBalance();
       if(amount > b)
         throw …
       setBalance(b – amount);
     }
  }

  // deposit() would also use synchronized(this)
}
```

Improving Version #2: Syntactic Sugar

- There is a shorter way to say the same thing as version #2

- Putting **synchronized** before a method declaration means the entire method body is surrounded by

```
synchronized(this){…}
```

- Version #3 is *identical to version #2*, but more concise, more standard, and therefore better style

Version #3: Final Version

```
class BankAccount {
  private int balance = 0;

  synchronized protected int getBalance()
    { return balance; }
  synchronized protected void setBalance(int x)
    { balance = x; }

  synchronized public void withdraw(int amount) {
     int b = getBalance();
     if(amount > b)
       throw …
     setBalance(b – amount);
  }

  // deposit() would also use synchronized
}
```

A Few Final Thoughts

- Our `synchronized`-less BankAccount pseudocode needs **`java.util.concurrent.locks.ReentrantLock`** and **`try { … } finally { … }`** blocks
  - Or just used `synchronized` ☺
- Don't have time to cover these highly-relevant lock variants: *readers/writer locks* and *condition variables*
  - See Grossman notes for more info
- Java provides many other features and details. See also:
  - Chapter 14 of CoreJava, Volume 1 by Horstmann/Cornell
  - Java Concurrency in Practice by Goetz et al

Lecture Outline

- Re-entrancy
- Locking in Java
- **Race Conditions: Data Races vs. Bad Interleavings**

Race Conditions

- A ***race condition*** occurs when the computation result depends on scheduling (ie, how threads are interleaved)
  - i.e.: if T1 and T2 are scheduled in a certain way, things go “wrong”
  - Only exist due to concurrency: no interleaving with only 1 thread!

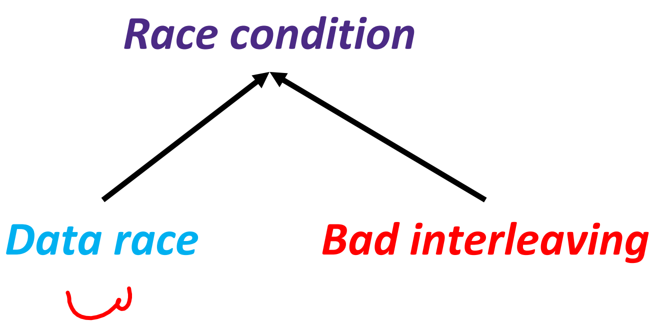
- We, as programmers, cannot control scheduling of threads
  - Thus we must write programs that work independent of scheduling

Data Races vs. Bad Interleavings

- We will make a big distinction between:

  data races and ***bad interleavings***

- Both are types of ***race conditions***
  - Confusion often results from not distinguishing these, or using the term "race condition" to refer to only one of these two



read + write = data race
write + write = data race
read + read = not data race
(but possibly bad interleaving)

Very Briefly: Data Races

- A ***data race*** is a type of ***race condition*** that can happen when:
  1. Different threads ***potentially*** write a variable *at the same time*
  2. One thread ***potentially*** writes a variable *at the same time* another thread reads it

- Two threads *reading* the same variable at the same time is not a **data race** and doesn't create an error
  - The key is that one of the threads must be *writing* to the variable

- The 'potentially' is important!
  - Code has a **data race** independent of any particular actual execution

Bad Interleavings

- Easy to see why **data races** are bad

- However, we can still have a **race condition** (and bad behavior) even without **data races**, thanks to **bad interleavings**
  - Different threads' reads and writes are "interleaved" without simultaneity

- Warning sign: intermediate/temporary state visible to a concurrently executing thread
  - E.g.: partial insert in a linked list: 'front' field updated with new node, but 'count' not yet updated

A Race Condition but Not a Data Race

```
class Stack<E> {
  private E[] array = (E[])new Object[SIZE];
  private int index = -1;

  synchronized public boolean isEmpty() {
    return index == -1;
  }
  synchronized public void push(E val) {
    array[++index] = val;
  }
  synchronized public E pop() {
    if(isEmpty())
      throw new StackEmptyException();
    return array[index--];
  }
  public E peek() {
    E ans = pop();
    push(ans);
    return ans;
  }
}
```

peek, Sequentially Speaking

- In a sequential world, this code is of questionable *style* but unquestionably *correct*
  - Imagine this is the only way to add peek functionality to an existing class or interface

```
interface Stack<E> {
  boolean isEmpty();
  void push(E val);
  E pop();
}

class C {
  public static <E> E myPeek(Stack<E> s){
     …
  }
}
```

Concurrency Problems with `peek`

```
public E peek() {
   E ans = pop();
   push(ans);
   return ans;
}
```

- `peek` has no ***overall*** effect on the shared data
  - It is a “reader” not a “writer”; state should be the same before and after it executes
- `peek`’s calls to `push` and `pop` are synchronized
  - So there are no ***data races*** on the underlying array/index
- But the way it is implemented creates a ***race condition***
  - `peek` has an ***intermediate state*** that shouldn’t be exposed to other threads
  - If exposed to other threads, `peek`’s intermediate state can lead to ***bad interleavings***

Bad Interleaving #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?

Thread A (`peek`)

```
E ans = pop();
push(ans);
return ans;
```

Thread B

```
push(x);
boolean b = isEmpty();
```

Time ↓

Answer #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?

Time ↓

Thread A (`peek`)	Thread B
	`push(x);`
`E ans = pop();`	
	`boolean b = isEmpty();`
`push(ans);`	
`return ans;`	

Bad Interleaving #2: peek and push

- **Property we want:** Values are **push**()'ed in LIFO order
- With **peek** as written, property can be violated – how?

Thread A (peek)

```
E ans = pop();
push(ans);
return ans;
```

Thread B

```
push(x);
push(y);
```

Time

Answer #2: peek and push

- **Property we want:** Values are **`push`**()'ed in LIFO order
- With **`peek`** as written, property can be violated – how?

Time ↓

Thread A (`peek`)

```
E ans = pop();

push(ans);
return ans;
```

Thread B

```
push(x);

push(y);
```

Bad Interleaving #3: peek and pop

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

Time ↓

Thread A (`peek`)

```
E ans = pop();
push(ans);
return ans;
```

Thread B

```
push(x);
push(y);
E e = pop();
```

Answer #3: peek and pop

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

Time ↓

Thread A (`peek`)	Thread B
	`push(x);`
	`push(y);`
`E ans = pop();`	
	`E e = pop();`
`push(ans);`	
`return ans;`	

Bad Interleaving #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?

Time

Thread A (`peek`)

```
E ans = pop();
push(ans);
return ans;
```

Thread B (`peek`)

```
E ans = pop();
push(ans);
return ans;
```

Answer #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?

Thread A (`peek`)

```
E ans = pop();
push(ans);
return ans;
```

Thread B (`peek`)

```
E ans = pop();



push(ans);
return ans;
```

Time ↓

The Fix: Disallow Interleavings

- **`peek`** needs synchronization to disallow interleavings
  - *Enlarging the critical section* will protect `peek`'s intermediate state
  - 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 public
      E peek() {
    E ans = pop();
    push(ans);
    return ans;
  }
}
```

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

The Wrong "Fix": Read-only Interleavings

- **Problem so far**: `peek` does writes which yield an incorrect intermediate state

- **Tempting but wrong**: if `peek` (or `isEmpty`) doesn't write anything, maybe we can skip the synchronization?
  - Unfortunately, does NOT work due to ***data races*** with `push` and `pop`

Turning a Bad Interleaving Into a Data Race

```
class Stack<E> {
  private E[] array = (E[])new Object[SIZE];
  private int index = -1;

  public boolean isEmpty() {  // unsynchronized; wrong?!
    return index == -1;
  }
  synchronized public void push(E val) {
    array[++index] = val;
  }
  synchronized public E pop() {
    return array[index--];
  }
  public E peek() {  // unsynchronized and wrong!
    return array[index];
  }
}
```

How Could This Be a Data Race? (1 of 2)

- It looks like `isEmpty` and `peek` can "get away with this" since `push` and `pop` adjust the state "in one tiny step"

- But this is an unsafe assumption about implementation details!
  - What looks like "tiny steps" in code may actually be multiple steps in the implementation:
    - `array[++index] = val` probably takes at least two steps
    - Compiler optimizations may modify "simple code" in unanticipated ways

How Could This Be a Data Race? (2 of 2)

- Since `push` and `pop` (ie, methods which *write*) probably require >=2 steps, an unsynchronized *read* (eg, `isEmpty` and `peek`) will create a ***data race***
  - See Grossman notes for more details

- **Moral**: Do not introduce a ***data race***, even if every interleaving you can think of is correct

Summary

- Java locks are re-entrant
  - Use `finally` blocks or `synchronized` to ensure locks are released
- ***"Race condition"*** refers to different things, but both are the result of a lack of synchronization:
  - ***Data races***: Simultaneous read/write or write/write of the same memory location
    - Always an error
    - Original `peek` example had no data races, but we introduced one later
  - ***Bad interleavings***: Exposing intermediate state to other threads
    - Not all interleavings are "bad"
    - Original `peek` had several bad interleavings