Adam Blank | Lecture 17/18 | Winter 2017

CSE 332

Data Abstractions

Synchronization

Parallelism and Concurrency (Again)

We're done talking about parallelism. Our goal is no longer (necessarily) "to make the program faster".

The `ForkJoin` Framework is great, but it doesn't actually allow us to share resources.

Two threads only interact at birth and death

For the next few lectures, we'll investigate what happens when we lift that restriction.

Two threads can run different algorithms now

Why Use Threads Then?

- Code structure for responsiveness
  - Example: Respond to GUI events in one thread while another thread is performing an expensive computation
- Processor utilization (mask I/O latency)
  - If 1 thread goes to disk, have something else to do
- Failure isolation
  - Convenient structure if want to interleave multiple tasks and do not want an exception in one to stop the other

Very Complicated, Very Quickly...

Concurrent code gets very complicated very quickly. Why?

Concurrency introduces **non-determinism**!

In sequential programming, when you run the same program multiple times, you get the same result (`Random` isn't really random; it just uses a **seed**).

This is no longer true for concurrent programs:

Thread 0
```
print("Hi I'm Thread 0");
```

Thread 1
```
print("Hi I'm Thread 1");
```

Thread 2
```
print("Hi I'm Thread 2");
```

Thread 3
```
print("Hi I'm Thread 3");
```

These threads could run in any order...

Investigating How Things Can Go Wrong

Concurrent code is only correct if **all possible executions** are correct.

Store

```
Store.cash = 1000;
Customer1.wallet = 100;        Customer2.wallet = 200;
```

Customer 1
```
int store = Store.cash + 50;
Customer1.wallet -= 50;
Store.cash = store;
```

Customer 2
```
int store = Store.cash + 20;
Customer2.wallet -= 20;
Store.cash = store;
```

What Happens With This Interleaving?

`int store = Store.cash + 50;`	
	`int store = Store.cash + 20;`
	`Customer2.wallet -= 20;`
`Customer1.wallet -= 50;`	
	`Store.cash = store;`

This is called a **race condition**.

Data Races

Bad interleavings aren't the only way concurrency can go wrong. . . Consider the following concurrent program:

```
private int data = -1;
private int done = false;
```

Thread 1
```
1 data = 4000;
2 done = true;
```

Thread 2
```
1 while (!done) {
2    // wait for data...
3 }
4 System.out.println(data);
```

This code doesn't have any bad interleavings, but. . .

The compiler is allowed to re-order lines 1 and 2 on the left.

To fix it, we insist that lines 1 and 2 happen together. We'll see how to do this later today.

Data Races

Definition (Data Race)

When two threads interact such that:

- One of them writes to a variable x
- And the other reads from or writes to the same variable x

Data Races can be a problem for two reasons:

1. They can lead to race conditions (we'll talk about this case)
2. The compiler/CPU is allowed to re-order the reads/writes which causes errors (we'll ignore this)

Note that two reads from the same variable is **not** a data race.

To fix a data race, we insist that the relevant reads/writes are **atomic** (must happen together).

For the remainder of the course, we will assume that the compiler **does not** re-order lines, but you must avoid data races in practice.

Back To Race Conditions

Definition (Race Condition)

A **race condition** is a concurrency bug that causes the **result of the computation** to depend on the interleaving of threads.
In other words, a program has a race condition if **any** interleaving of instructions in threads results in an incorrect computation.

The result of the store example should be `Store.cash = 170`, but if we look at the interleaving from before:

`int store = Store.cash + 50;`	
	`int store = Store.cash + 20;`
	`Customer2.wallet -= 20;`
`Customer1.wallet -= 50;`	
	`Store.cash = store;`
`Store.cash = store;`	

`Store.cash` will end up equaling 150. Are there any other interleavings that lead to different results?

So, what's the solution?

Atomicity

`Store.cash` changed between reading and writing!

We want **only one thread** to be able to be working with a value of `Store.cash` at a time. (Or we might get a stale value like above)

When there is a section of code that must occur **atomically** to avoid race conditions, we call this a **critical section**.

Store

```
Store.cash = 1000;
Customer1.wallet = 100;        Customer2.wallet = 200;
```

Customer 1
```
atomic {
   int store = Store.cash + 50;
   Customer1.wallet -= 50;
   Store.cash = store;
}
```

Customer 2
```
atomic {
   int store = Store.cash + 20;
   Customer2.wallet -= 20;
   Store.cash = store;
}
```

The idea of only allowing one thread in the critical section at a time is called **mutual exclusion**.

How Do We Get Mutual Exclusion?

We need to make use of a language feature to do this. (If you take CSE451, you will implement your own mutexes.)

Definition (Mutex (Lock))

A **mutex** is a way of marking a critical section. If one thread "has the lock", all other threads wait to get the lock before entering the critical section.

We define `Lock` as an ADT:

Lock ADT

`new()`	Creates a new lock which no thread holds
`lock()`	If the lock is not held by any threads, then the calling thread acquires it. Otherwise, the thread waits until the lock is free.
`unlock()`	If the lock is held by this thread, the calling thread releases it. Otherwise, calling this method is an error.

Solving the `Store` Problem?

Store

```
Store.cash = 1000;
Lock lock = new Lock();
Customer1.wallet = 100;        Customer2.wallet = 200;
```

Customer 1
```
lock.lock();
int store = Store.cash + 50;
Customer1.wallet -= 50;
Store.cash = store;
lock.unlock();
```

Customer 2
```
lock.lock();
int store = Store.cash + 20;
Customer2.wallet -= 20;
Store.cash = store;
lock.unlock();
```

Other Solutions? Some Questions. . .

- Can we use different locks for different **Customers**?
  No! If we do this, the customers will all be allowed in their critical sections at the same time!
- Can we use different locks for different **Stores**?
  Yes! The race condition happens due to the `Store.cash` variable, but each store would have its own.
- What happens if we don't release the lock? The customers that haven't gone yet never will. Uh oh!

Another Race Condition Example?

Sum Array

```
sum = 0;
```

Thread 1
```
for (int i = 0; i < MID; i++) {
   sum += input[i];
}
```

Thread 2
```
for (int i = MID; i < NUM; i++) {
   sum += input[i];
}
```

Does this example have a race condition? If so, fix it.
It does! Notice that both threads **reading** `input` is okay!

The problem is that they're both writing to `sum`.

We could put a mutex around each for loop, but we can do better:

Sum Array

```
Lock lock = new Lock();
sum = 0;
```

Thread 1
```
for (int i = 0; i < MID; i++) {
   lock.lock();
   sum += input[i];
   lock.unlock();
}
```

Thread 2
```
for (int i = MID; i < NUM; i++) {
   lock.lock();
   sum += input[i];
   lock.unlock();
}
```

ANOTHER Example!

```
class Stack<E> {
   Lock lock = new Lock();
   void push(E val) {
      lock.lock();
      ...
      lock.unlock();
   }
   E pop() {
      lock.lock();
      ...
      lock.unlock();
   }
   E peek() {
      E ans = pop();
      push(ans);
      return ans;
   }
}
```

```
Stack s = new Stack();
```

Thread 1
```
???
```

Thread 2
```
???
```

This code has a race condition.
Can you find an interleaving of method calls that causes it?

ANOTHER Example!

`E ans = s.pop();`	
	`s.push(5);`
`s.push(ans);`	
`return ans;`	
	`s.pop();`

In this interleaving, the top two elements end up in the wrong order!

Notice that with our definition of locks, we can't just fix this problem by putting `lock` around the contents of `peek` even though that seems like a reasonable solution.

To fix this, we'd need to make private versions of `push` and `pop` that don't lock and use those.

A Note: Reentrant Locks

A **reentrant lock** is a mutex that can be taken multiple times **by the same thread**.
A reentrant lock would also solve our problem.

And Again!

```
class Stack<E> {
   Lock lock = new Lock();
   void push(E val) {
      lock.lock();
      array[++index] = val;
      lock.unlock();
   }
   E pop() {
      lock.lock();
      array[index--] = val;
      lock.unlock();
   }
   E peek() {
      return array[index];
   }
}
```

Do we need a lock around `peek`?

`int newIndex = ++index;`	
	`return array[index];`
`array[newIndex] = val;`	

A Common Race Condition Pattern

We've seen a lot of race conditions that follow this pattern:

```
if (check()) {
   doAction();
}
```

This is called a **Time-Of-Check-To-Time-Of-Use** (TOCTTOU) bug.

More Examples

```
if (validUser()) {
   authenticateUser();
}
```

```
if (!file.exists()) {
   file.create();
}
```

What is the race condition? Why do these violate their specifications?

`if (check)`	
	`deauthenticate();`
`authenticateUser()`	

More Data Structures

Suppose we have a program that implements an LRU Cache (a store of $\leq n$ items that evicts the oldest item when it has to remove something).

How To Implement?

```
HashTable table;
Queue queue;

insert(i) {
   table.insert(i);
   queue.enqueue(i);
}
remove() {
   e = queue.dequeue();
   table.remove(i);
}
contains(i) {
   return table.contains(i);
}
```

We good?

Stupid Race Conditions. . .

```
HashTable table; Lock tLock;
Queue queue;     Lock qLock;

insert(i) {
   tLock.lock();
   table.insert(i);
   qLock.lock();
   queue.enqueue(i);
   qLock.unlock();
   tLock.unlock();
}
remove() {
   qLock.lock();
   e = queue.dequeue();
   tLock.lock();
   table.remove(i);
   tLock.unlock();
   qLock.unlock();
}
```

Now?

No! Still Not Okay!

A Problem?

```
insert(i) {
   tLock.lock();
   table.insert(i);
   qLock.lock();
   queue.enqueue(i);
   qLock.unlock();
   tLock.unlock();
}
remove() {
   qLock.lock();
   e = queue.dequeue();
   tLock.lock();
   table.remove(i);
   tLock.unlock();
   qLock.unlock();
}
```

tLock.lock();	
	qLock.lock();
table.insert(i);	
	e = queue.dequeue();
qLock.lock();	
	tLock.lock();
queue.enqueue(i);	
	table.remove(i);
qLock.unlock();	
	tLock.unlock();
tLock.unlock();	
	qLock.unlock();

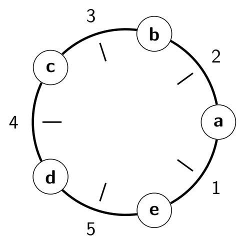
The `insert` is waiting on the queue lock and the `remove` is waiting on the hashtable lock...

These stupid threads are **waiting for each other**!

This is called **deadlock**.

THE Example: Dining Philosophers

Five philosophers go out to eat at a Chinese restaurant. Unfortunately, this restaurant has a shortage of chopsticks. So, they put one chopstick between each place setting:



Every philosopher first grabs the chopstick to her right, then the one to her left. Deadlock.

The Fix

Impose a **global** ordering on the chopsticks:
Always grab the smaller number first

LRU Cache Fix

Broken

```
insert(i) {
   tLock.lock();
   table.insert(i);
   qLock.lock();
   queue.enqueue(i);
   qLock.unlock();
   tLock.unlock();
}
remove() {
   qLock.lock();
   e = queue.dequeue();
   tLock.lock();
   table.remove(i);
   tLock.unlock();
   qLock.unlock();
}
```

Fixed!

```
insert(i) {
   tLock.lock();
   table.insert(i);
   qLock.lock();
   queue.enqueue(i);
   qLock.unlock();
   tLock.unlock();
}
remove() {
   tLock.lock();
   qLock.lock();
   e = queue.dequeue();
   table.remove(i);
   tLock.unlock();
   qLock.unlock();
}
```

Always grab the **table lock** before getting the **queue lock**.

Another Banking Example!

```
class BankAccount {
   Lock lock = new Lock();
   void withdraw(int amt) {
      this.lock.lock();
      this._withdraw(amt);
      this.lock.unlock();
   }
   void deposit(int amt) {
      this.lock.lock();
      this._deposit(amt);
      this.lock.unlock();
   }
   void transfer(int amt, BankAccount
        b) {
      this.lock.lock();
      this._withdraw(amt);
      b.deposit(amt);
      this.lock.unlock();
   }
}
```

Assume that `_withdraw` and `_deposit` are the unlocked versions.

Consider two simultaneous transfers:

- `a.transfer(100, b);`
- `b.transfer(100, a);`

Do you see any problems?

Another Banking Example

a.lock.lock();	
	b.lock.lock();
a._withdraw(amt);	
	b._withdraw(amt);
b.lock.lock();	
	a.lock.lock();
b._deposit(amt);	
	a._deposit(amt);
b.lock.unlock();	
	a.lock.unlock();
a.lock.unlock();	
	b.lock.unlock();

Possible Solutions?

- Solution 1: Don't bother locking transfer at all.
  Bad because clients can see intermediary state.
- Solution 2: Make a special mutex for "all transfers"
  Bad because it doesn't allow multiple transfers at once.
- Solution 3: Order the locks (like with Dining Philosophers)

Banking Example Solution

Order The Locks By `BankAccount` id

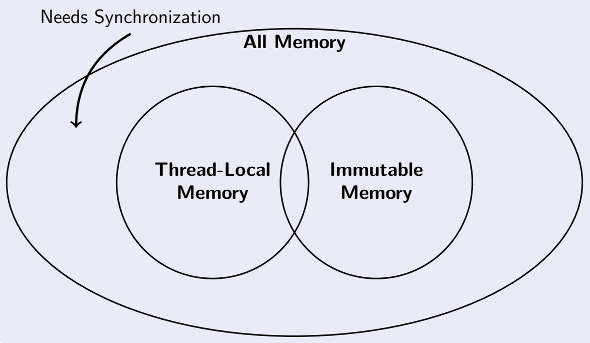
```
void transfer(int amt, BankAccount b) {
   assert(this.id != b.id);
   BankAccount first = this.id < b.id ? this : b;
   BankAccount second = this.id > b.id ? this : b;
   first.lock.lock();
   second.lock.lock();
   this._withdraw(amt);
   b._deposit(amt);
   second.lock.unlock();
   first.lock.unlock();
}
```

Now, we can't get deadlock, because every `transfer` will acquire the locks in the same order.

Okay, Race Conditions and Deadlock Suck. . .

This stuff is **hard**. How do we go about programming concurrent code in practice? For the rest of the lecture, we'll talk about best practices and conventional wisdom.

Three Choices



Thread-Local Memory

Definition (Thread-Local Memory)

Thread-Local Memory is only used by a single thread. All the instance variables in the `RecursiveTasks` we've been writing are Thread-Local Memory.

If a thread has its own **copy** of a resource, then we don't have to worry about other threads updating it.

This only works if the threads don't need to share the resource.

Example

- `Random` Objects don't need to be shared
- In p3, you will **make a copy** of the `Board` object for each thread to work with.

Immutable Memory

Definition (Immutable Memory)

Immutable Memory is memory that never changes. All the **input** arrays we've been passing to `RecursiveTasks` have been Immutable Memory.

If none of the threads write to the location, then they can all share the same copy!

Example

- Generally, input data structures won't be mutated.
- In p3, you will share `Move` objects between threads (since they never change).

What's Left?

If it's not possible to use Thread-Local Memory or Immutable Memory for a task, we're stuck with using synchronization.

Try as much as possible to minimize this category

If we must use synchronization, . . .

Guideline #0: Avoid Data Races

Never allow two threads to write/write or read/write to the same location at the same time. (Avoid this with **mutexes**.)

Guideline #1: Use Consistent Locking

For each location needing synchronization, have a lock that is always held when reading or writing the location.

- Use the same lock to guard multiple locations when it makes sense
- Clearly document what each lock is for
- Conceptually partition shared-and-mutable locations into "which lock"

But how much should each lock be responsible for?

Lock Granularity

Guideline #2: Start With Fewer Locks

Start with fewer locks (coarse-grained) and move to more (fine-grained) only if contention on the locks becomes an issue.

Definition (Coarse-Grained)

Fewer Locks (more objects/lock)

- Lock for entire data structure (e.g., array)
- Lock for all bank accounts

Definition (Fine-Grained)

More Locks (fewer objects/lock)

- Lock per data element (e.g., array index)
- Lock per bank account

Coarse-Grained Advantages

- Simpler to implement
- Faster/easier to implement operations that access multiple locations
- Much easier for operations that modify data-structure shape

Fine-Grained Advantages

- More simultaneous access
- Can make multi-node operations more difficult: say, rotations in an AVL tree

Lock Granularity Example: Separate Chaining Hashtable

Lock for entire hashtable (coarse) vs. Lock per bucket (fine)

- Which supports more concurrency for insert and lookup?
  Fine-grained; allows simultaneous access to different buckets
- Which makes implementing resize easier?
  Coarse-grained; just grab one lock and proceed
- If a hashtable has a `numElements` field, maintaining it will destroy the benefits of using separate locks for each bucket, why?
  Updating it each insert without a lock would be a data race

Critical Section Granularity

Guideline #3: Keep Critical Sections Small

Don't do expensive computations or I/O in critical sections, but also don't introduce race conditions. In other words, keep critical sections as small as possible without being incorrect.

- If critical sections are too long, it's a huge **performance loss**.
- If critical sections are too short, we get **race conditions** (visible intermediary state).

Atomicity & Libraries

Guideline #4: Think About Atomicity

Think in terms of what operations need to be atomic and make critical sections just long enough to preserve atomicity. Then, design the locking protocol to implement the critical sections correctly.

Guideline #5: Use Libraries

Avoid implementing data structure synchronization as much as possible. Most languages have built-in libraries to handle frequent needs.

For example, `ConcurrentHashMap` is written by world-experts who know what they are doing. It would be silly to write your own.

Some Final Words on Synchronization

Java has a special syntax called `synchronized` for locking:

Our Code

```
method() {
   lock.lock();
   doStuff();
   lock.unlock();
}
```

In Java

```
method() {
   synchronized(this) {
      doStuff();
   }
}
```

This code treats **the actual object** as the lock.

Other Synchronization Primitives

- **Condition Variables** have **signal** and **wait** methods. `wait` allows a thread to wait until some condition is true. `signal` wakes the thread up at the right time.
- **Reader-Writer Locks** allow threads to declare if they are reading or writing to a resource. These allow multiple readers at the same time.
- . . . and others.