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 get 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...

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

This is called a race condition.
Data Races

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

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

Thread 1
data = 4000;
done = true;
while (!done) {
  // wait for data...
  System.out.println(data);
}
```

Thread 2

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

The compiler is allowed to reorder 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.

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:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td><code>Thread 1</code></td>
<td><code>Thread 2</code></td>
</tr>
<tr>
<td><code>lock.lock();</code></td>
<td><code>lock.lock();</code></td>
</tr>
<tr>
<td><code>Customer1.wallet = 100;</code></td>
<td><code>Customer1.wallet = 100;</code></td>
</tr>
<tr>
<td><code>Thread 1</code></td>
<td><code>Thread 2</code></td>
</tr>
<tr>
<td><code>lock.lock();</code></td>
<td><code>lock.lock();</code></td>
</tr>
<tr>
<td><code>Customer2.wallet = 200;</code></td>
<td><code>Customer2.wallet = 200;</code></td>
</tr>
</tbody>
</table>

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

So, what’s the solution?

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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>new()</code></td>
<td>Creates a new lock which no thread holds.</td>
</tr>
<tr>
<td><code>lock()</code></td>
<td>If the lock is not held by any threads, then the calling thread acquires it. Otherwise, the thread waits until the lock is free.</td>
</tr>
<tr>
<td><code>unlock()</code></td>
<td>If the lock is held by this thread, the calling thread releases it. Otherwise, calling this method is an error.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Customer 1</th>
<th>Customer 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lock.lock();</code></td>
<td><code>lock.lock();</code></td>
</tr>
<tr>
<td><code>lock.lock();</code></td>
<td><code>lock.lock();</code></td>
</tr>
<tr>
<td><code>Customer1.wallet = 100;</code></td>
<td><code>Customer1.wallet = 100;</code></td>
</tr>
<tr>
<td><code>Customer2.wallet = 200;</code></td>
<td><code>Customer2.wallet = 200;</code></td>
</tr>
</tbody>
</table>

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

Solving the Store Problem?

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

<table>
<thead>
<tr>
<th>Customer 1</th>
<th>Customer 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lock.lock();</code></td>
<td><code>lock.lock();</code></td>
</tr>
<tr>
<td><code>lock.lock();</code></td>
<td><code>lock.lock();</code></td>
</tr>
<tr>
<td><code>Customer1.wallet = 100;</code></td>
<td><code>Customer1.wallet = 100;</code></td>
</tr>
<tr>
<td><code>Customer2.wallet = 200;</code></td>
<td><code>Customer2.wallet = 200;</code></td>
</tr>
</tbody>
</table>

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

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();
}

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

ANOTHER Example!

1 class Stack<E> {
2     Lock lock = new Lock();
3     void push(E val) {
4         lock.lock();
5         array[++index] = val;
6         lock.unlock();
7     }
8     E pop() {
9         lock.lock();
10        E ans = pop();
11        push(ans);
12        return ans;
13     }
14 }

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?

And Again!

1 class Stack<E> {
2     Lock lock = new Lock();
3     void push(E val) {
4         lock.lock();
5         array[++index] = val;
6         lock.unlock();
7     }
8     E pop() {
9         lock.lock();
10        E ans = pop();
11        push(ans);
12        return ans;
13     }
14 }

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?

A Common Race Condition Pattern

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

1 if (check) {
2     doAction();
3 }

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

More Examples

1 if (validUser()) {
2     authenticateUser();
3 }

1 if (!file.exists()) {
2     file.create();
3 }

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

If (check)

authenticateUser();

dauthenticate();

More Data Structures

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

How To Implement?

1 HashTable table;
2 Queue queue;
3
4 insert(i) {
5     table.insert(i);
6     queue.enqueue(i);
7 }
8 remove() {
9     e = queue.dequeue();
10     table.remove(e);
11 }
12 contains(i) {
13     return table.contains(i);
14 }

We good?

Now?
### The Fix

Impose a global ordering on the chopsticks:

Always grab the smaller number first.

---

### Another Banking Example!

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?

---

### BANKING EXAMPLE SOLUTION

#### Order The Locks By BankAccount id

```java
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.

---

### 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.

---

### BANKING EXAMPLE SOLUTION 22

#### Another Banking Example!

```java
class BankAccount {
    Lock lock = new Lock();
    void withdraw(int amt) {
        // Assume the lock and... versions.
        Consider two simultaneous transfers:
        a.transfer(100, b);
        b.transfer(100, a);
        Do you see any problems?
        
        ---
        
        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)
```
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
Needs Synchronization
All Memory

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.

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

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

Definition (Fine-Grained)
More Locks (fewer objects/lock)
- Lock per data element (e.g., array index)
- Lock per bank account

Fine-Grained Advantages
- More simultaneous access
- 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 size 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.