Toward sharing resources (memory)

So far, we have been studying parallel algorithms using fork-join model

- Reduce span via parallel tasks
- Fork-Join algorithms all had a very simple structure to avoid race conditions
- Each thread had memory “only it accessed”
  - Example: each array sub-range accessed by only one thread
  - Result of forked process not accessed until after join() called
  - So the structure (mostly) ensured that bad simultaneous access wouldn’t occur

This strategy won’t work well when:
- Memory accessed by threads is overlapping or unpredictable
- Threads are doing independent tasks needing access to same resources (rather than implementing the same algorithm)

Really sharing memory between Threads

2 Threads, each with own unshared call stack and “program counter”

Heap for all objects and static fields, shared by all threads

Sharing a Queue....

- Imagine 2 threads, running at the same time,
- both with access to a shared linked-list based queue (initially empty)

```
class SumArray extends RecursiveTask<Integer> {
    int lo, hi;
    int[] arr;
    // fields to know what to do
    SumArray(int[] a, int l, int h) { ... }

    protected Integer compute() { // return answer
        if(hi - lo < SEQUENTIAL_CUTOFF) {
            int ans = 0;
            for(int i=lo; i < hi; i++)
                ans += arr[i];
            return ans;
        } else {
            SumArray left = new SumArray(arr,lo,(hi+lo)/2);
            SumArray right = new SumArray(arr,(hi+lo)/2,hi);
            left.fork();
            int leftAns = left.join();
            rightAns = right.compute();
            return leftAns + rightAns;
        }
    }

    static final ForkJoinPool fjPool = new ForkJoinPool();
    int sum(int[] arr) {
        return fjPool.invoke(new SumArray(arr,0,arr.length));
    }
```
Sharing a Queue….

- Imagine 2 threads, running at the same time, both with access to a shared linked-list based queue (initially empty)

```java
enqueue(x) {
    if (back == null) {
        back = new Node(x);
        front = back;
    } else {
        back.next = new Node(x);
        back = back.next;
    }
}
```

- Each thread has its own program counter (and local stack)
- Queue is shared, so both threads indirectly use the same ‘front’ and ‘back’ (which is the whole point of sharing the queue)
- We have no guarantee what happens first between different threads; can (and will) arbitrarily ‘interrupt’ each other
- Many things can go wrong: say, one tries to enqueue “a”, the other “b”, and both verify that back is ‘null’ before other sets back
- Result: One assignment of back will be ‘forgotten’
- In general, any ‘interleaving’ of results is possible if enqueue were called at the same time for both

Concurrent Programming

Concurrency: Allowing simultaneous or interleaved access to shared resources from multiple clients

- Requires coordination, particularly synchronization to avoid incorrect simultaneous access: make somebody block (wait) until the resource is free
  - Java is not what we want
  - Block until another thread is “done using what we need” not “completely done executing”

- Even correct concurrent applications are usually highly non-deterministic:
  - how threads are scheduled affects what operations happen first
  - non-repeatability complicates testing and debugging

Concurrent Examples

What if we have multiple threads:

1. Processing different bank-account operations
   - What if 2 threads change the same account at the same time?
2. Using a shared cache (e.g., hashtable) of recent files
   - What if 2 threads insert the same file at the same time?
3. Creating a pipeline (think assembly line) with a queue for handing work to next thread in sequence?
   - What if enqueuer and dequeuer adjust a circular array queue at the same time?

Why threads?

- Unlike with parallelism, not about implementing algorithms faster

- But threads still useful for:
  - 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 don’t want an exception in one to stop the other

Sharing, again

- It is common in concurrent programs that:
  - Different threads might access the same resources in an unpredictable order or even at about the same time
  - Program correctness requires that simultaneous access be prevented using synchronization
  - Simultaneous access is rare
    - Makes testing difficult
    - Must be much more disciplined when designing / implementing a concurrent program
    - Will discuss common idioms known to work

Canonical example

Correct code in a single-threaded world

```java
class BankAccount {
  private int balance = 0;
  int getBalance() { return balance; }
  void setBalance(int x) { balance = x; }
  void withdraw(int amount) {
    int b = getBalance();
    if(amount > b)
      throw new WithdrawTooLargeException();
    setBalance(b - amount);
  }
  // other operations like deposit, etc.
}
```
Interleaving

Suppose:
- Thread T1 calls `x.withdraw(100)`
- Thread T2 calls `y.withdraw(100)`

If second call starts before first finishes, we say the calls **interleave**
- Could happen even with one processor since a thread can be **pre-empted** at any point for time-slicing
  - e.g. T1 runs for 50 ms, pauses somewhere, T2 picks up for 50 ms

If x and y refer to different accounts, no problem
- “You cook in your kitchen while I cook in mine”
- But if x and y alias, possible trouble...

What is the balance at the end?

Two threads both trying to `withdraw(25)` from the **same account**:
- Assume initial balance 100

```java
class BankAccount {
    private int balance = 0;
    int getBalance()  { return balance; }
    void setBalance(int x) { balance = x; }
    void withdraw(int amount) {
        int b = getBalance();
        if(amount > b)
            throw new WithdrawTooLargeException();
        setBalance(b – amount);
    }
    // other operations like deposit, etc.
}
```
- **Thread 1**: `x.withdraw(25)`
- **Thread 2**: `x.withdraw(25)`

Another example: a bad interleaving

Two threads both trying to `withdraw(100)` from the **same account**:
- Assume initial balance 150
- This **should cause a WithdrawTooLarge** exception

```java
int b = getBalance();
if(amount > b)
    throw new …;
setBalance(b – amount);
```

Thread 1

```
int b = getBalance();
if(amount > b)
    throw new …;
setBalance(b – amount);
```

Thread 2

A bad fix, Another bad interleaving

Two threads both trying to `withdraw(100)` from the **same account**:
- Assume initial balance 150
- This **should cause a WithdrawTooLarge** exception

```java
if(amount > getBalance())
    throw new …;
setBalance(b – amount);
```

Thread 1

```
int b = getBalance();
if(amount > getBalance())
    throw new …;
setBalance(b – amount);
```

Thread 2

Still a bad fix, Another bad interleaving

Two threads both trying to `withdraw(100)` from the **same account**:
- Assume initial balance 150
- This **should cause a WithdrawTooLarge** exception

```java
if(amount > getBalance())
    throw new …;
setBalance(getBalance() - amount);
```

Thread 1

```
if(amount > getBalance())
    throw new …;
setBalance(getBalance() - amount);
```

Thread 2

Incorrect “fix”

It is tempting and almost always **wrong** to fix a bad interleaving by rearranging or repeating operations, such as:

```java
void withdraw(int amount) {
    if(amount > getBalance())
        throw new WithdrawTooLargeException();
    // maybe balance changed
    setBalance(getBalance() – amount);
}
```

This fixes nothing!
- Narrows the problem by one statement
- (Not even that since the compiler could turn it back into the old version because you didn’t indicate need to synchronize)
- And now a negative balance is possible – why?
**Mutual exclusion**

The same fix: At most one thread withdraws from account A at a time

- Exclude other simultaneous operations on A too (e.g., deposit)

Called **mutual exclusion**: One thread doing something with a resource (here: an account) means another thread must wait

- Define critical sections; areas of code that are mutually exclusive

Programmer (you!) must implement critical sections

- "The compiler" has no idea what interleavings should or shouldn’t be allowed in your program
- But you need language primitives to do it!

**Why this Wrong?**

Why can’t we implement our own mutual-exclusion protocol?

- Say we tried to coordinate it ourselves using a boolean variable – "busy"

```java
class BankAccount {
    private int balance = 0;
    private boolean busy = false;
    void withdraw(int amount) {
        while(busy) { /* "spin-wait" */ }
        busy = true;
        int b = getBalance();
        if(amount > b)
        throw new WithdrawTooLargeException();
        setBalance(b - amount);
        busy = false;
    }
}
```

Still just moved the problem!

While (busy) { }

Busy is initially = false

```
Thread 1
while (busy) { }
busy = true;
int b = getBalance();
if(amount > b)
    throw new …;
setBalance(b - amount);
```

```
Thread 2
while (busy) { }
busy = true;
int b = getBalance();
if(amount > b)
    throw new …;
setBalance(b - amount);
```

"Lost withdraw" – unhappy bank

Time does elapse between checking ‘busy’ and setting ‘busy’; can be interrupted there

**What we need:**

- There are many ways out of this conundrum, but we need help from the language
- One basic solution: **Locks**
  - Still on a conceptual level at the moment, ‘Lock’ is not a Java class
- We will define **Lock** as an ADT with operations:
  - **new**: make a new lock
  - **acquire**: If lock is "not held", makes it "held"
    - blocks if this lock is already currently "held"
    - Checking & setting happen together, and cannot be interrupted
    - Fixes problem we saw before
  - **release**: makes this lock "not held"
    - if >= 1 threads are blocked on it, exactly 1 will acquire it

**Why that works**

- A **Lock** ADT with operations **new, acquire, release**
- The lock implementation ensures that given simultaneous acquires and/or releases, a correct thing will happen
  - Example:
    - If we have two acquires: one will "win" and one will block
- How can this be implemented?
  - Need to "check-and-update" all at once
  - Uses special hardware and O/S support
    - See a senior-level course in computer architecture or operating systems
  - In CSE 332, we take this as a primitive and use it

**Almost-correct pseudocode**

```java
class BankAccount {
    private int balance = 0;
    private Lock lk = new Lock();
    void withdraw(int amount) {
        lk.acquire(); /* may block */
        int b = getBalance();
        if(amount > b)
        throw new WithdrawTooLargeException();
        setBalance(b - amount);
        lk.release();
    }
}
```

Note: ‘Lock’ is not an actual Java class
Some mistakes

- A lock is a very primitive mechanism
  - Still up to you to use correctly to implement critical sections
- Incorrect: Forget to release a lock (blocks other threads forever)
  - Previous slide is wrong because of the exception possibility!
    ```java
    if(amount > b) {
      lk.release(); // hard to remember!
      throw new WithdrawTooLargeException();
    }
    ```
- Incorrect: Use different locks for withdraw and deposit
  - Mutual exclusion works only when using same lock
- Poor performance: Use same lock for every bank account
  - No simultaneous withdrawals from different accounts

Other operations

- If withdraw and deposit use the same lock, then simultaneous calls to these methods are properly synchronized
- But what about getBalance and setBalance?
  - Assume they’re public, which may be reasonable
  - If they don’t acquire the same lock, then a race between setBalance and withdraw could 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

```java
int setBalance1(int x) {
  balance = x;
}
int setBalance2(int x) {
  lk.acquire();
  balance = x;
  lk.release();
}
void withdraw(int amount) {
  lk.acquire();
  setBalance1(b - amount);
  lk.release();
}
```

Re-entrant lock idea

A re-entrant lock (a.k.a. recursive lock)

- The idea: Once acquired, the lock is held by the Thread, and subsequent calls to acquire in that Thread won’t block
- Result withdraw can acquire the lock, and then call setBalance, which can also acquire the lock
  - Because they’re in the same thread & it’s a re-entrant lock, the inner acquire won’t block!!

Java’s Re-entrant Lock

```java
java.util.concurrent.locks.ReentrantLock
Has methods lock() and unlock()
As described above, it is conceptually owned by the Thread, and shared within that thread
Important to guarantee that lock is always released!!!
Recommend something like this:
```java
myLock.lock();
try { // method body }
finally { myLock.unlock(); }
```
- Despite what happens in ‘try’, the code in finally will execute afterwards
**Synchronized: A Java convenience**

Java has built-in support for re-entrant locks
- You can use the `synchronized` statement as an alternative to declaring a `ReentrantLock`

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

1. Evaluates expression to an object
   - Every object (but not primitive types) “is a lock” in Java
2. Acquires the lock, blocking if necessary
   - “If you get past the {, you have the lock”
3. Releases the lock “at the matching }”
   - Even if control leaves due to `throw`, `return`, etc.
   - So impossible to forget to release the lock

**Java example (correct but can be improved)**

```java
class BankAccount {
    private int balance = 0;
    private Object lk = new Object();
    int getBalance() {
        synchronized (lk) { return balance; }
    }
    void setBalance(int x) {
        synchronized (lk) { balance = x; }
    }
    void withdraw(int amount) {
        synchronized (lk) {
            int b = getBalance();
            if(amount > b) throw ...
            setBalance(b - amount);
        }
        // deposit would also use synchronized(lk)
    }
}
```

**Improving the Java**

- As written, the lock is `private`
  - Might seem like a good idea
  - But also prevents code in other classes from writing operations that synchronize with the account operations
- More common is to synchronize on `this`
  - Also, it’s convenient; don’t need to declare an extra object!

**Java version #2**

```java
class BankAccount {
    private int balance = 0;
    int getBalance() {
        synchronized (this) { return balance; }
    }
    void setBalance(int x) {
        synchronized (this) { balance = x; }
    }
    void withdraw(int amount) {
        synchronized (this) {
            int b = getBalance();
            if(amount > b) throw ...
            setBalance(b - amount);
        }
        // deposit would also use synchronized(this)
    }
}
```

**Syntactic sugar**

Version #2 is slightly poor style because there is a shorter way to say the same thing:

Putting `synchronized` before a method declaration means the entire method body is surrounded by `synchronized(this) {...}`

Therefore, version #3 (next slide) means exactly the same thing as version #2 but is more concise

**Java version #3 (final version)**

```java
class BankAccount {
    synchronized int getBalance() {
        return balance;
    }
    synchronized void setBalance(int x) {
        balance = x;
    }
    synchronized void withdraw(int amount) {
        int b = getBalance();
        if(amount > b) throw ...
        setBalance(b - amount);
    }
    // deposit would also use synchronized
}
```
Addendum: More Java notes

- Class `java.util.concurrent.ReentrantLock` works much more like our pseudocode
  - Often use `try { ... } finally { ... }` to avoid forgetting to release the lock if there's an exception

- Also library and/or language support for readers/writer locks and condition variables (upcoming lectures)

- Lots of features and details (you are not responsible for) in Chapter 14 of CoreJava, Volume 1
  - For an entire book on advanced topics see "Java Concurrency in Practice" [Goetz et al]