CSE 332
Data Structures & Parallelism

Shared-Memory Concurrency & Mutual Exclusion

Melissa Winstanley

Spring 2024

Course updates

- P3 CP1 is due on Thursday
- Reading is helpful!

Toward sharing resources (memory)

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

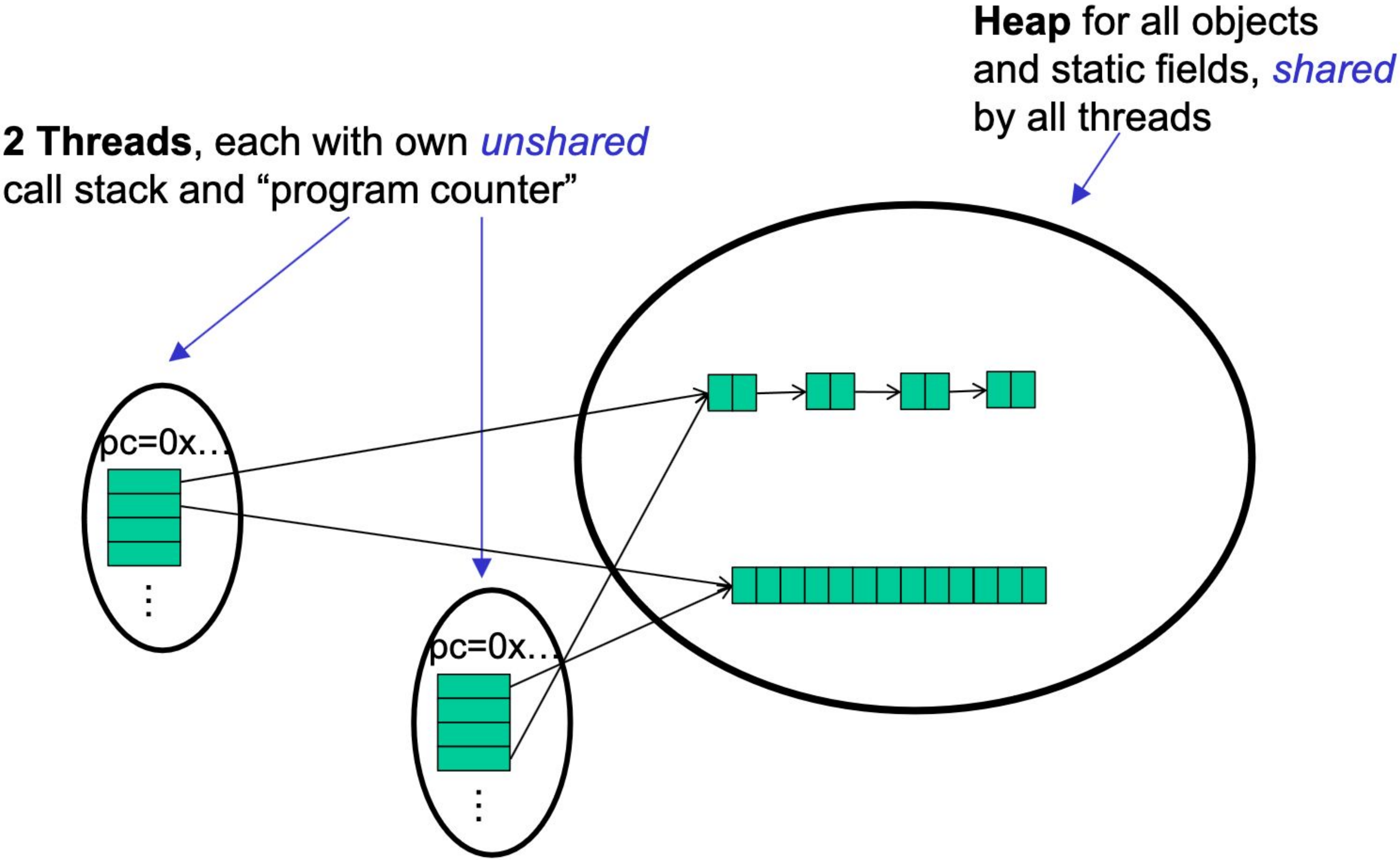
- Reduce span via parallel tasks

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

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)



Heap for all objects and static fields, *shared* by all threads
2 Threads, each with own *unshared* call stack and "program counter"
pc=0x…
pc=0x…

Sharing a Queue….

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

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

Concurrent Programming

Concurrency: Correctly and efficiently managing access to shared resources from multiple possibly-simultaneous clients

Requires *coordination*, particularly synchronization to avoid incorrect simultaneous access: make somebody block (wait) until the resource is free

- `join` is not what we want
- Want to 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

Why threads?

Unlike 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 do not want an exception in one to stop the other

Canonical example

Correct code in a single-threaded world

```
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.
}
```

Activity: What is the balance at the end?

Two threads both trying to **withdraw**() from the same account:

- Assume **initial balance 150**

```
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(100);
```

Thread 2

```
x.withdraw(75);
```

Activity: A bad “interleaving”

```
balance = 150
```

Interleaved **withdraw**() calls on the same account

- Assume initial balance == 150
- This ***should*** cause a **WithdrawTooLarge** exception

Thread 1: `withdraw(100)`

```
1 int b = getBalance();

5 if (amount > b)
     throw new …;
6 setBalance(b – amount);
```

Thread 2: `withdraw(75)`

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

time

Activity: Other possible interleavings

How else could two threads interleave?

```
void withdraw(int amount) {
  int b = getBalance();
  if (amount > b)
    throw new WithdrawTooLargeException();
  setBalance(b - amount);
}
```

Thread 1

```
x.withdraw(100);
```

Thread 2

```
x.withdraw(75);
```

Activity: A “good” execution is also possible

Interleaved **withdraw**() calls on the same account

- Assume initial balance == 150
- This ***should*** cause a **WithdrawTooLarge** exception

Thread 1: `withdraw(100)`

Thread 2: `withdraw(75)`

time ↓

Thread 2:

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

Thread 1:

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

A bad fix, Another bad interleaving

```
balance = 150
```

Interleaved **withdraw**() calls on the same account

- Assume initial balance == 150
- This ***should*** cause a **WithdrawTooLarge** exception

Thread 1: `withdraw(100)`

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

Thread 2: `withdraw(75)`

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

time

Incorrect “fix”

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

```
void withdraw(int amount) {
  if (amount > getBalance())
    throw new WithdrawTooLargeException();
  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?

What we want: **Mutual exclusion**

The fix: Allow at most one thread to withdraw from account **A** at a time

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

Called **mutual exclusion**:

- One thread using a resource (here: a bank account) means another thread must wait
- We call the area of code that we want to have mutual exclusion (only one thread can be there at a time) a critical section.

Programmer (you!) must implement **critical sections**:

- “The compiler” has no idea what interleavings should or should not be allowed in your program
- But you need language primitives to do it!

Why is this wrong?

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

```
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;
  }
  … // deposit would spin on the same boolean
}
```

- Say we tried to coordinate it ourselves using a boolean variable – “busy”
- It’s technically possible under certain assumptions, but won’t work in real languages anyway

What we need

There are many ways out of this conundrum, but we need help from the programming language…

One solution: **Mutual-Exclusion Locks** (aka Mutex, or just Lock)

- Still on a conceptual level at the moment, 'Lock' is not a Java class (though Java's approach is similar)

We will define Lock as an ADT with operations:

- **new**: make a new lock, initially "*not held*"
- **acquire**: blocks if this lock is already currently "*held*"
  - Once "*not held*", makes lock "*held*" [all at once!]
  - 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

Note: ‘Lock’ is not an actual Java class

Almost-correct pseudocode

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

Questions about the previous slide

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? (one lock for withdraw and one lock for deposit)
    c) One lock for the <u>bank</u> (containing multiple bank accounts)?
3. There is a bug in withdraw(), can you find it?
4. Do we need locks for:
    a) getBalance?
    b) setBalance?

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 are public, which may be reasonable
- If they do *not* 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

```
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();
}
```

Have two versions of **`setBalance`**!

- **`withdraw`** calls **`setBalance1`** (since it already has the lock)
- Outside world calls **`setBalance2`**
- Could work (if adhered to), but not good style; also not very convenient

- Alternately, we can modify the meaning of the Lock ADT to support ***re-entrant locks***
  - Java does this
  - Then just always use **`setBalance2`**

Re-entrant lock idea

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

- "Remembers"
  - the thread (if any) that currently holds it
  - a *count*
- When the lock goes from *not-held* to *held*, the count is set to 0
- If (code running in) the current holder calls **acquire** :
  - it does not block
  - it increments the count
- On **release** :
  - if the count is > 0, the count is decremented
  - if the count is 0, the lock becomes *not-held*

Re-entrant locks work

```
int setBalance(int x) {
  lk.acquire();
  balance = x;
  lk.release();
}

void withdraw(int amount) {
  lk.acquire();
  ...
  setBalance1(b - amount);
  lk.release();
}
```

This simple code works fine provided **`lk`** is a reentrant lock

- Okay to call **`setBalance`** directly
- Okay to call **`withdraw`** (won’t block forever)

Java's Re-entrant Lock

`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:

```
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`**

```
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 version #1 (correct but can be improved)

```
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 new WithdrawTooLargeException();
      setBalance(b - amount);
    }
  }
}
```

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 new WithdrawTooLargeException();
      setBalance(b – amount);
    }
  }
}
```

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

```
class BankAccount {
  private int balance = 0;
  synchronized int getBalance()
    { return balance; }
  synchronized void setBalance(int x)
    { balance = x; }
  synchronized void withdraw(int amount) {
    int b = getBalance();
    if (amount > b)
      throw new WithdrawTooLargeException();
    setBalance(b – amount);
  }
}
```