Concurrency and Mutual Exclusion

CSE 332 Spring 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Annie Mao	Howard Xiao	Lucy Jiang
Chris Choi	Jade Watkins	Nathan Lipiarski
Ethan Knutson	Khushi Chaudhari	Richard Jiang

Warm-up

- Write your answers on a piece of scratch paper:
  - Even after we executed its recursive calls in parallel, Parallel-QuickSort and Parallel-MergeSort's span was still O(n). What sequential operation(s) prevented them from an even faster span?
  - Define "non-determinism"

Announcements

- Quizzes are intended to be 2h per person, not 9h!
  - You will not (and should not expect) to get 100% on them
- Projects:
  - P3 released, please fill out partner survey
  - Use a token to unlock p2 repo if you need an extra late day
- Administration:
  - Updated guidance for scheduling 1:1s (use email; send us background info; expect 24h response time)
  - Please please please update your Zoom client to 5.x 

Lecture questions: pollev.com/cse332

Learning Objectives

- Understand how parallel and concurrent programming differ
- Describe interleavings and mutual exclusion
- Identify critical sections in sample code

Lecture Outline

- **Sharing Resources**
- Concurrency: Managing Correct Access to Shared Resources
- Mutual Exclusion and Critical Sections

Lecture questions: pollev.com/cse332

Review: Parallelism and Sharing Resources

- We've studied **parallel algorithms** using the fork-join model and focused on reducing span via parallel tasks

- This model has a simple structure to avoid **race conditions**
  - Each thread had a part of memory the "only it accessed"
    - Example: each array sub-range accessed by only one thread
  - Result of forked executor not accessed until after join() called
  - Structure (mostly) ensures bad simultaneous access wouldn't occur

- Model won't work well when:
  - **Memory** accessed by executors **is overlapping** or unpredictable
  - Executors doing **independent tasks** need to **access the same resources**
    - (rather than implementing the same algorithm)

Parallelism's Pitfall

- Fork-join model doesn't work well when:
  - Executors implementing the same algorithm access **overlapping memory**
  - Executors implementing different algorithms access **the same resources**

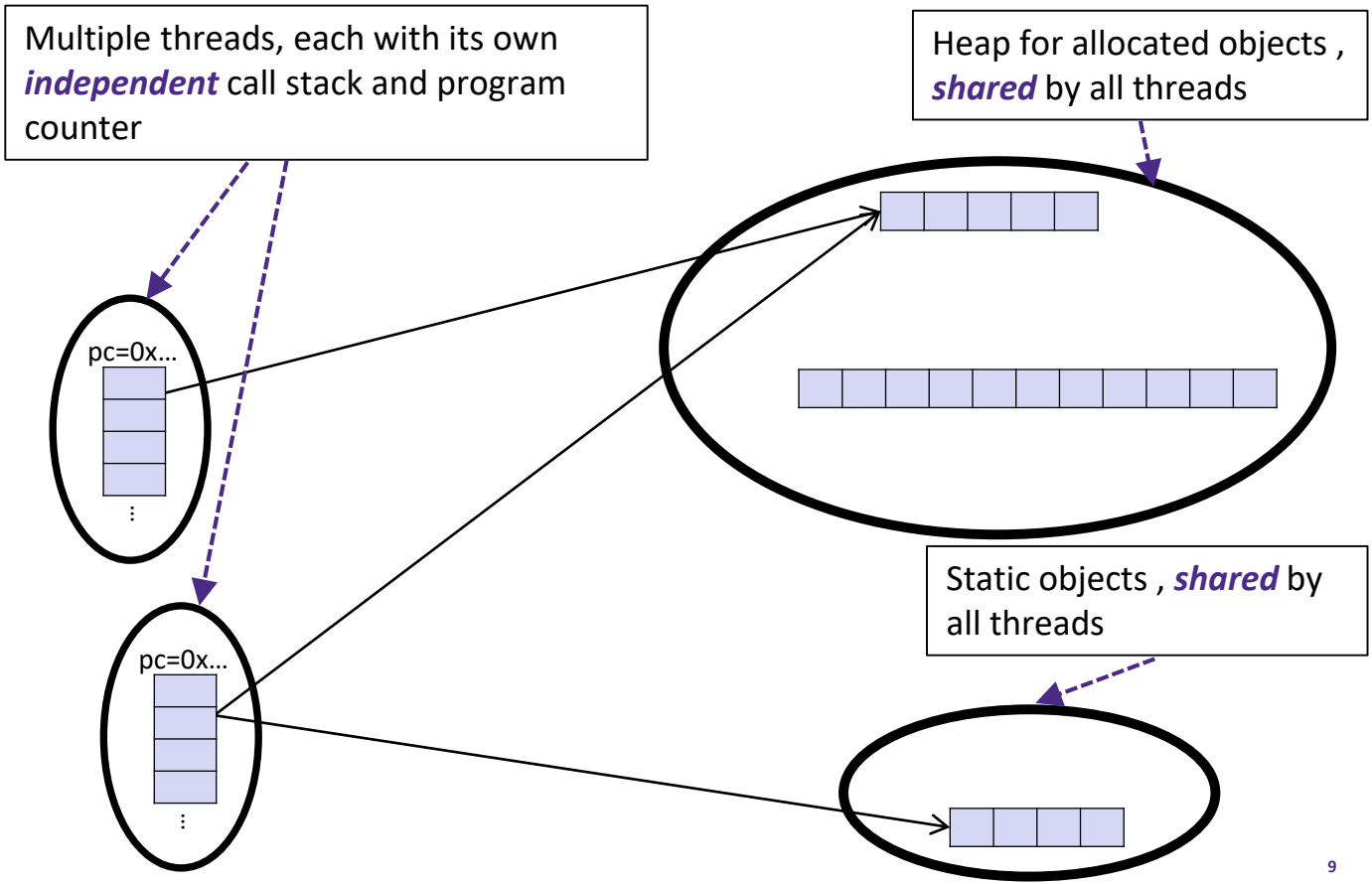
Parallelism: Non-overlapping Sharing

```
class SumTask extends RecursiveTask<Integer> {
  int lo; int hi; int[] arr;     // just the "input" arguments!

  protected Integer compute() {  // override: implement "main"
    if(hi - lo < SEQUENTIAL_CUTOFF) {
      // Just do the calculation in this thread
      int ans = 0;   // local variable instead of a member field
      for (int i=lo; i < hi; i++)
        ans += arr[i];
      return ans;    // direct return of answer
    } else {
      // Create ONE new thread to calculate the left sum
      SumTask left  = new SumTask(arr, lo, (hi+lo)/2);
      SumTask right = new SumTask(arr, (hi+lo)/2, hi);
      left.fork();   // create a thread and call its compute()
      int rightAns = right.compute(); // call compute() directly

      // Combine results
      int leftAns  = left.join();
      return leftAns + rightAns;
    }
  }
}
```

Can Overlapped Sharing Happen?



Overlapped Sharing (1 of 2)

- Threads are not just useful for parallelism
  - i.e., not always about implementing algorithms faster

- Threads are useful for:
  - Responsiveness
    - Respond to events in one thread while another is performing computation
  - Processor utilization (hide I/O latency)
    - If 1 thread "goes to disk," process still has something else to do
  - Failure isolation
    - Prevent an exception in one task from stopping conceptually-parallel tasks

Overlapped Sharing (2 of 2)

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

Sharing a Queue

- Imagine 2 threads
  - Running at the same time
  - Accessing 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;
  }
}
```

Overlapped Sharing Needs Concurrency

- **Concurrency**: *Correctly* and *efficiently* managing access to shared resources from multiple possibly-simultaneous clients
  - Requires *coordination*, particularly **synchronization**, to avoid incorrect simultaneous access
    - Make thread *block* (wait) until the resource is free
    - `join` is not what we want
    - Want other thread to be "done using *what we need*", not "completely done executing"
- Correct concurrent applications are usually highly **non-deterministic**
  - How threads are scheduled affects order of operations
  - Non-repeatability complicates testing and debugging

Attributes of Concurrent Programs

- In concurrent programs, it is common that:
  - Threads access the same resources in an *unpredictable order*
  - Threads access the same resources at *(approx.) the same time*
  - Correctness requires that simultaneous access be prevented
  - Simultaneous access is rare
    - Makes testing and debugging difficult
    - Rare != Impossible; need to be disciplined when designing / implementing

- In other words: concurrent programs are non-deterministic

Lecture Outline

- Sharing Resources
- **Concurrency: Managing Correct Access to Shared Resources**
- Mutual Exclusion and Critical Sections

Lecture questions: pollev.com/cse332

Concurrency: Canonical Example

- In a single-threaded world, this code is correct!

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

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

  public 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 they **interleave**
  - e.g. T1 runs for 50 ms, pauses somewhere, T2 picks up for 50ms
  - Can happen with one processor; if **pre-empted** due to time-slicing

- 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…

Activity: What is the Balance at the End?

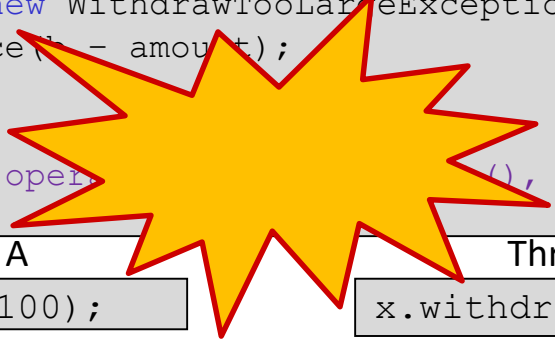
- Two threads both withdraw() from the same account:
  - Assume initial balance == 150

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

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

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

  // … other operations, etc.
}
```



Thread A

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

Thread B

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

Activity: A Bad Interleaving

- Interleaved withdraw() calls on the same account
  - Assume initial balance == 150
  - This *should* cause a WithdrawTooLarge exception (but doesn't)

Thread A: `withdraw(100)`

```
int b = getBalance();




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

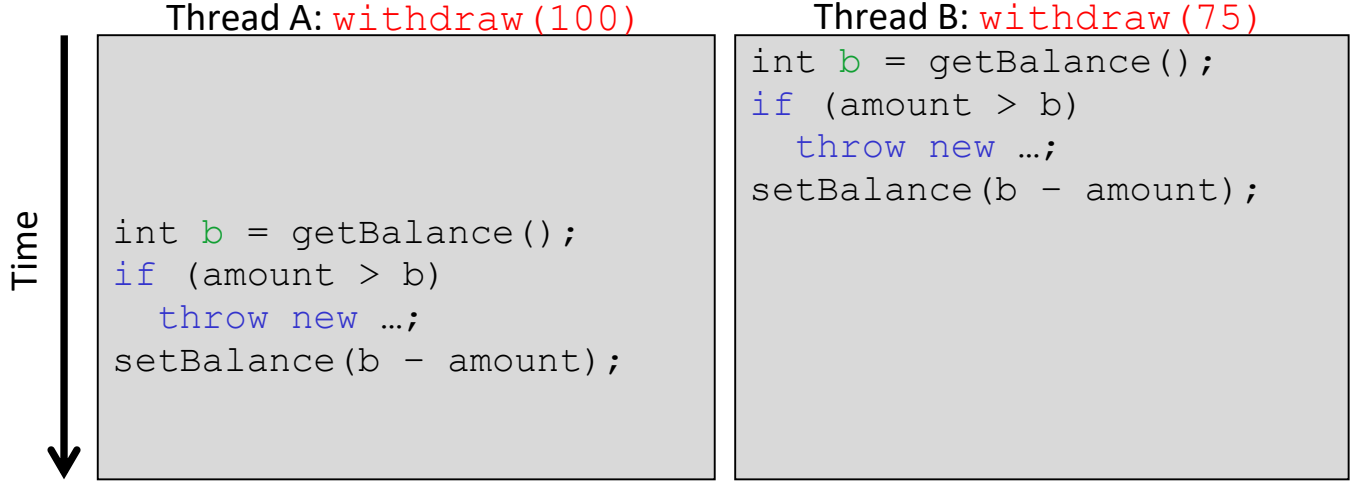
Thread B: `withdraw(75)`

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

Time

A Good Interleaving is Also Possible

- Interleaved withdraw() calls on the same account
  - Assume initial balance == 150
  - This *does* cause a WithdrawTooLarge exception



A Bad Fix: Another Bad Interleaving

- Interleaved withdraw() calls on the same account
  - Assume initial balance == 150
  - This *should* cause a WithdrawTooLarge exception (but doesn't)

Time

Thread A: `withdraw(100)`

```
int b = getBalance();




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

Thread B: `withdraw(75)`

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




setBalance(b – amount);
```

ANOTHER Bad Fix: Another Bad Interleaving

- Interleaved withdraw() calls on the same account
  - Assume initial balance == 150
  - This *should* cause a WithdrawTooLarge exception (but doesn't)

Thread A: `withdraw(100)`

```
int b = getBalance();



if (amount > getBalance())
  throw new …;


setBalance(getBalance()
  – amount);
```

Thread B: `withdraw(75)`

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


setBalance(getBalance()
  – amount);
```

Time

In all 3 of these "bad" examples, instead of an exception we had a "lost withdrawl"

Incorrect "Fixes"

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

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

  // Maybe the balance was changed
  setBalance(getBalance() – amount);
}
```

- This fixes nothing!
  - Potentially narrows the problem by one statement
  - And that's not even guaranteed!
    - The compiler could optimize it into the old version, because you didn't indicate a need to synchronize

Lecture Outline

- Sharing Resources
- Concurrency: Managing Correct Access to Shared Resources
- **Mutual Exclusion and Critical Sections**

Lecture questions: pollev.com/cse332

The Correct Fix: Mutual Exclusion

- Want at most one thread at a time to withdraw from account A
  - Exclude other simultaneous operations on A (e.g., deposit)
- More generally, we want **mutual exclusion**:
  - One thread using a resource means another thread must wait
- The area of code needing mutual exclusion is a **critical section**
- Programmer (you!) must identify and protect critical sections:
  - Compiler doesn't know which interleavings are allowed/disallowed
  - But you still need system-level primitives to do it!

Why Do We Need System-level Primitives?

- Why can't we implement our own mutual-exclusion protocol?
  - Can we coordinate it ourselves using a boolean variable "**busy**"?
  - Possible under certain assumptions, but won't work in real languages

```
class BankAccount {
  private int balance = 0;
  private boolean busy = false;

  public 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 same boolean
}
```

the critical section

Because We Just Moved the Problem!

- Initially, **busy** = false

Thread A: `withdraw(100)`

```
while (busy) { }

busy = true;

int b = getBalance();

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

Thread B: `withdraw(75)`

```
while (busy) { }

busy = true;

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

Time

critical but interrupted

critical and uninterrupted

Unhappy bank; we have a "lost withdrawal"

- Problem: time elapses between *checking and setting* **busy**
  - System can interrupt a thread then, letting another thread "sneak in"

What We Actually Need: Lock ADT

- All ways out of this conundrum require system-level support

- One solution: **Mutual-Exclusion Locks** (aka **Mutex**, or just **Lock**)
  - For now, still discussing concepts; `Lock` is not a Java class

- We will define **Lock** as an ADT with operations:
  - **`new`**: make a new lock, initially "not held"
  - **`acquire`**: blocks current thread if this lock is "held"
    - Once "not held", makes lock "held"
    - Checking & setting the "held" boolean is a single uninterruptible operation
    - Fixes problem we saw before!!
  - **`release`**: makes this lock "not held"
    - If >= 1 threads are blocked on it, another thread – but only one! – can now acquire

Why a System-level Lock Works

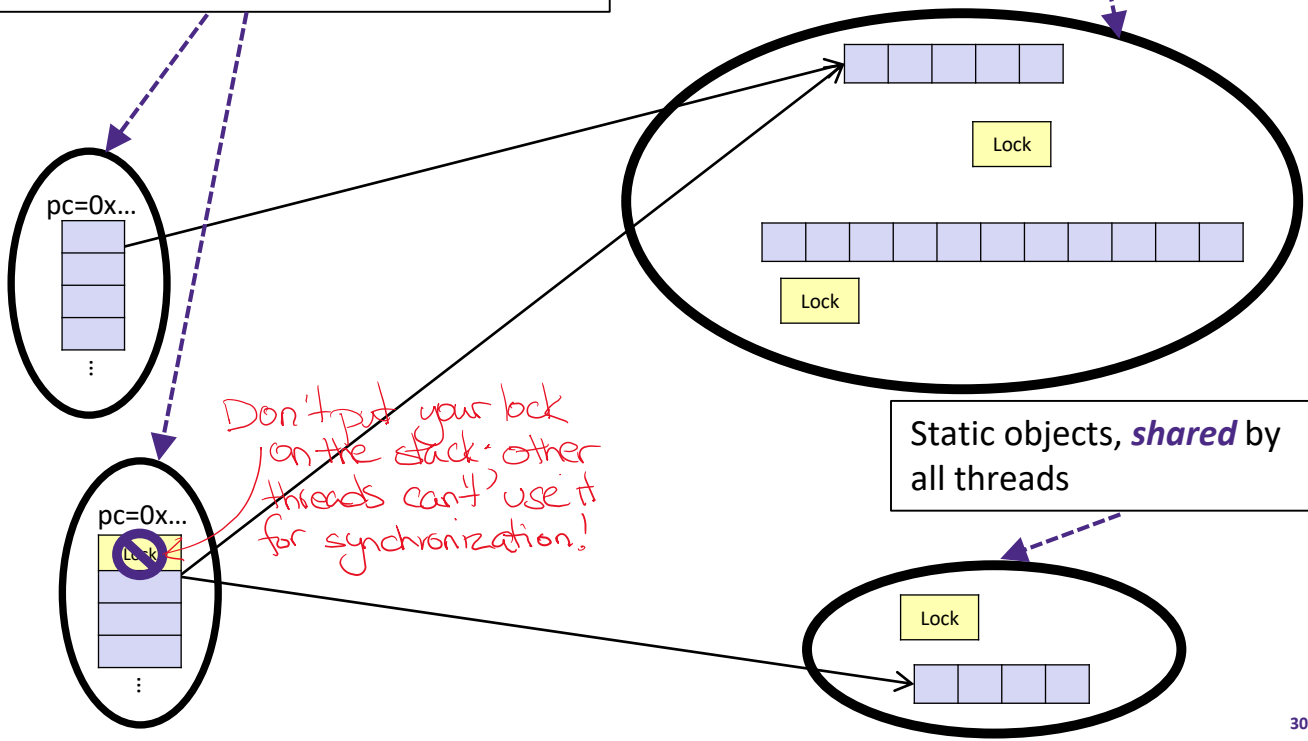
- Lock must ensure that, given simultaneous acquires/releases, "the correct thing" will happen
  - E.g.: if we have two acquires: one will "win" and one will block

- How can this be implemented?
  - The key is that the "check if held; if not, make held" operation must happen "all-at-once". It cannot be interrupted!
  - Thus, requires and uses hardware and O/S support
    - See computer-architecture or operating-systems course
  - In CSE 332, we'll assume a lock is a primitive and just use it

Locks Must Be Accessible By Multiple Threads!

Multiple threads, each with its own ***independent*** call stack and program counter

Heap for allocated objects, ***shared*** by all threads



Static objects, ***shared*** by all threads

Almost-Correct Pseudocode

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

  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

Activity: 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?
      - 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?

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

Answers: Some Common Locking Mistakes

- 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
- **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!

Summary

- Threads are useful beyond just fork-join-style parallelism
  - But general use-cases require **concurrency** to ensure correctness when dealing with overlapped sharing

- Overlapped sharing introduces **non-determinism** because the system controls the scheduling of threads
  - Therefore, the system must also provide **locks** to ensure **mutual exclusion** in **critical sections** of code
  - Mutual exclusion is the technique we employ to prevent **bad interleavings**