Parallel Sorts + Concurrency

CSE 332 Summer 2021

Instructor: Kristofer Wong

Teaching Assistants:

Alena Dickmann Arya GJ Finn Johnson

Joon Chong Kimi Locke Peyton Rapo

Rahul Misal Winston Jodjana

Announcements

- Exercise 9 reflection, Exercise 10 due tonight!
- Internships!
  - Want some extra eyes on your resume? Email staff (main mailing list)
  - Start applying **now**
- Get going on P3 early!!
  - Parallelism is great, but a PAIN to debug…

Lecture Outline

- Parallel Sorts (Divide & Conquer)
  - **Parallel QuickSort**
  - Parallel Mergesort
- Sharing Resources
- Intro to Concurrency
  - Managing Correct Access to Shared Resources
  - Mutual Exclusion and Critical Sections

Sequential QuickSort Review

Step	*Runtime Expression*
Pick the pivot value(s) • Hopefully these value(s) approximate the median	
Partition all the values into: A. The values less than the pivot(s) B. The pivot(s) C. The values greater than the pivot(s)	
Recursively QuickSort(A) and QuickSort(C)	

- Recurrence (assuming a good-enough pivot):
  - T(0) = T(1) = c1
  - T(n) = _____
  - Closed-form T(n) = _____

Copied from L7: Algorithm Analysis III

Really Common Recurrences

Recurrence Relation	*Closed Form*	*Name*	*Example*
T(n) = O(1) + T(n/2)	O(log n)	Logarithmic	Binary Search
T(n) = O(1) + T(n-1)	O(n)	Linear	Sum (v1: "Recursive Sum")
T(n) = O(1) + 2T(n/2)	O(n)	Linear	Sum (v2: "Recursive Binary Sum")
T(n) = O(n) + T(n/2)	O(n)	Linear	
T(n) = O(n) + 2T(n/2)	O(n log n)	Loglinear	MergeSort
T(n) = O(n) + T(n-1)	$O(n^2)$	Quadratic	
T(n) = O(1) + 2T(n-1)	$O(2^n)$	Exponential	Fibonacci

Parallelizing QuickSort: Attempt #1

Step	*Runtime Expression*
Pick the pivot value(s) • Hopefully these value(s) approximate the median	c_1
Partition all the values into: A. The values less than the pivot(s) B. The pivot(s) C. The values greater than the pivot(s)	c_2N
Recursively QuickSort(A) and QuickSort(C)	

- Let's parallelize the two recursive calls!
  - Work (unchanged): _____
  - T(n) = _____
  - Span: _____

Parallel QuickSort: Doing Better

- *O*(`log` *n*) speed-up with an infinite number of processors is okay, but a bit underwhelming
  - Sort 10^9 elements 30 times faster

- Google searches strongly suggest quicksort cannot do better because the partition cannot be parallelized
  - The Internet has been known to be wrong ☺
  - But we need auxiliary storage (no longer in place)
  - In practice, constant factors may make it not worth it, but remember Amdahl's Law…(exposing parallelism is important!)

- Already have everything we need to parallelize the partition…

Parallel Partition (not in place)

Step	*Span*
In parallel, partition all the values into: A. The values less than the pivot(s) B. The pivot(s) C. The values greater than the pivot(s)	

- Parallel partition is just two packs!
  - We know a pack is $O(n)$ work, $O(\log n)$ span
    1. Pack elements less than pivot into left side of **`aux`** array
    2. Pack elements greater than pivot into right size of **`aux`** array
  - Put pivot between them and recursively sort
  - With a little more cleverness, can do both packs at once but no effect on asymptotic complexity
- Parallel Partition Span: _____

Parallel QuickSort, Attempt #2: Example

1. Pick pivot (we'll use median-of-3)

8	1	4	9	0	3	5	2	7	6

2. Pack less-than, then pack greater-than
   - Must be sequential, since second pack needs a starting index

1	4	0	3	5	2

1	4	0	3	5	2	6	8	9	7

3. Recursively sort, in parallel
   - Can sort back into original array (like in mergesort)

0	1	2	3	4	5	6	7	8	9

- Parallel QuickSort:
  - T(n) = _____
  - Span: _____

Lecture Outline

- Parallel Sorts (Divide & Conquer)
  - Parallel QuickSort
  - **Parallel Mergesort**
- Sharing Resources
- Intro to Concurrency
  - Managing Correct Access to Shared Resources
  - Mutual Exclusion and Critical Sections

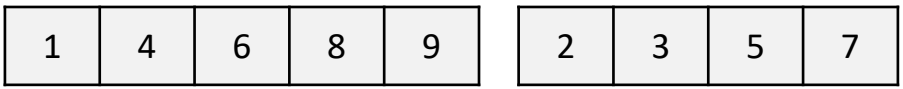
Parallelizing MergeSort

Step	*Runtime Expression*
Recursively MergeSort(A) and MergeSort(B)	
Merge(A, B)	

- Just like QuickSort, do the two recursive sorts in parallel:
  - Span is $T(n) = c_1 n + 1T(n/2) = O(n)$
  - Work is $O(n \log n)$
  - Parallelism = work/span = $O(\log n)$
  - To do better, need to parallelize the merge
    - The trick won't use parallel prefix this time…

Parallelizing the Merge (1 of 2)

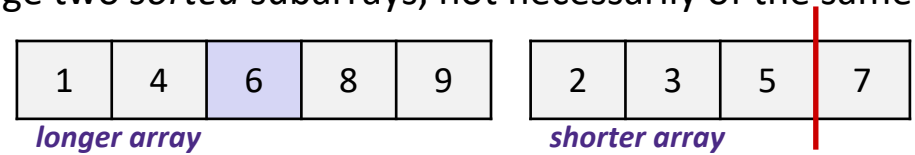
- Problem statement:
  - Merge two *sorted* subarrays, not necessarily of the same size



- Intuition:
  - Suppose the longer subarray has *m* elements. In parallel:
    - Merge the first *m*/2 elements of the longer half with the "appropriate" elements of the shorter half
    - Merge the second *m*/2 elements of the longer half with the rest of the shorter half

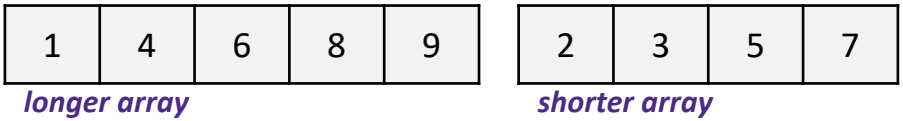
Parallelizing the Merge (2 of 2)

- Problem statement:
  - Merge two *sorted* subarrays, not necessarily of the same size

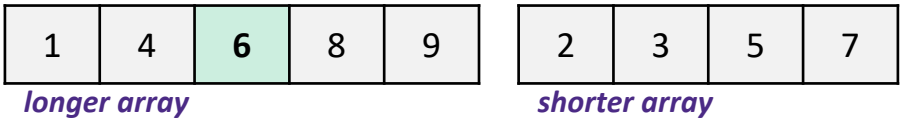


- Step #1:
  - Pick the median of the *longer array* in constant time
  - Binary search the *shorter array* to find the first element >median
- Step #2 (in parallel):
  - Merge the lower part of the *longer array* (<=median) with the lower part of the *shorter array*
  - Merge upper part of the *longer array* (>median onward) with the upper part of the *shorter array*

Parallelizing the Merge: Example (1 of 7)

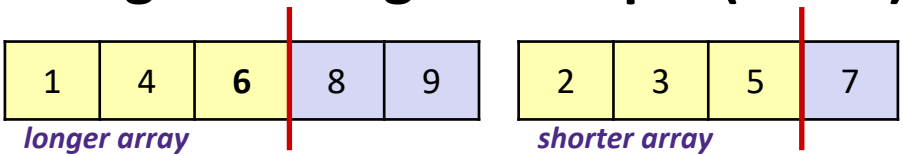


Parallelizing the Merge: Example (2 of 7)



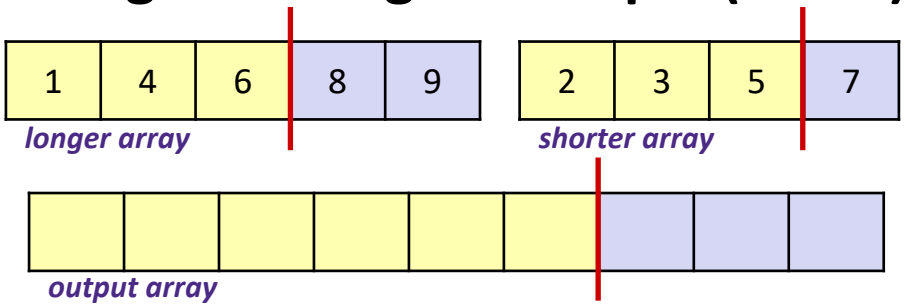
- Pick the median of the *longer array*: O(1) to compute index

Parallelizing the Merge: Example (3 of 7)



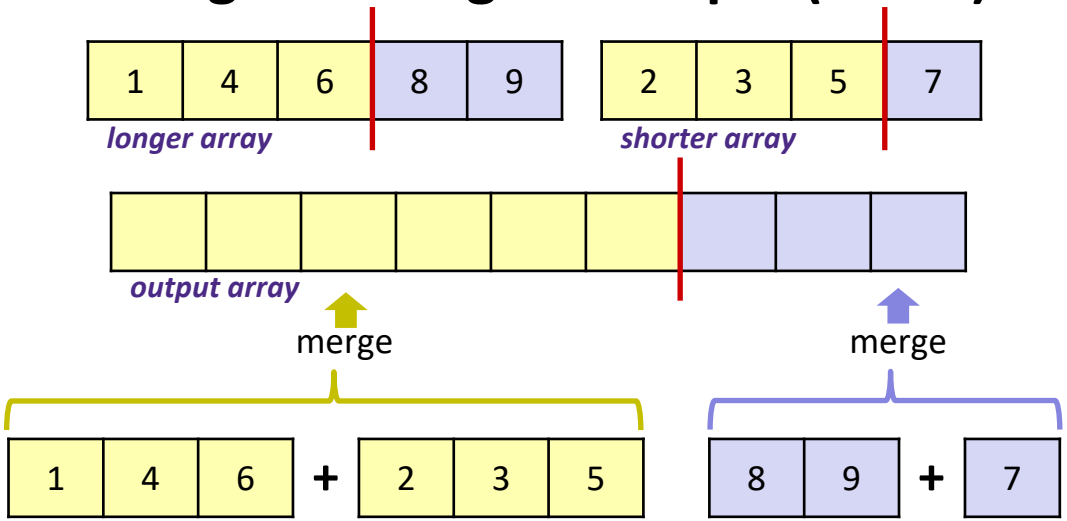
- Pick the median of the *longer array*: O(1) to compute index
- Split the *shorter array* at the same value: O(log n) for binary search

Parallelizing the Merge: Example (4 of 7)



- Pick the median of the *longer array*: O(1) to compute index
- Split the *shorter array* at the same value: O(log n) for binary search
- Calculate where to split the output array: O(1)

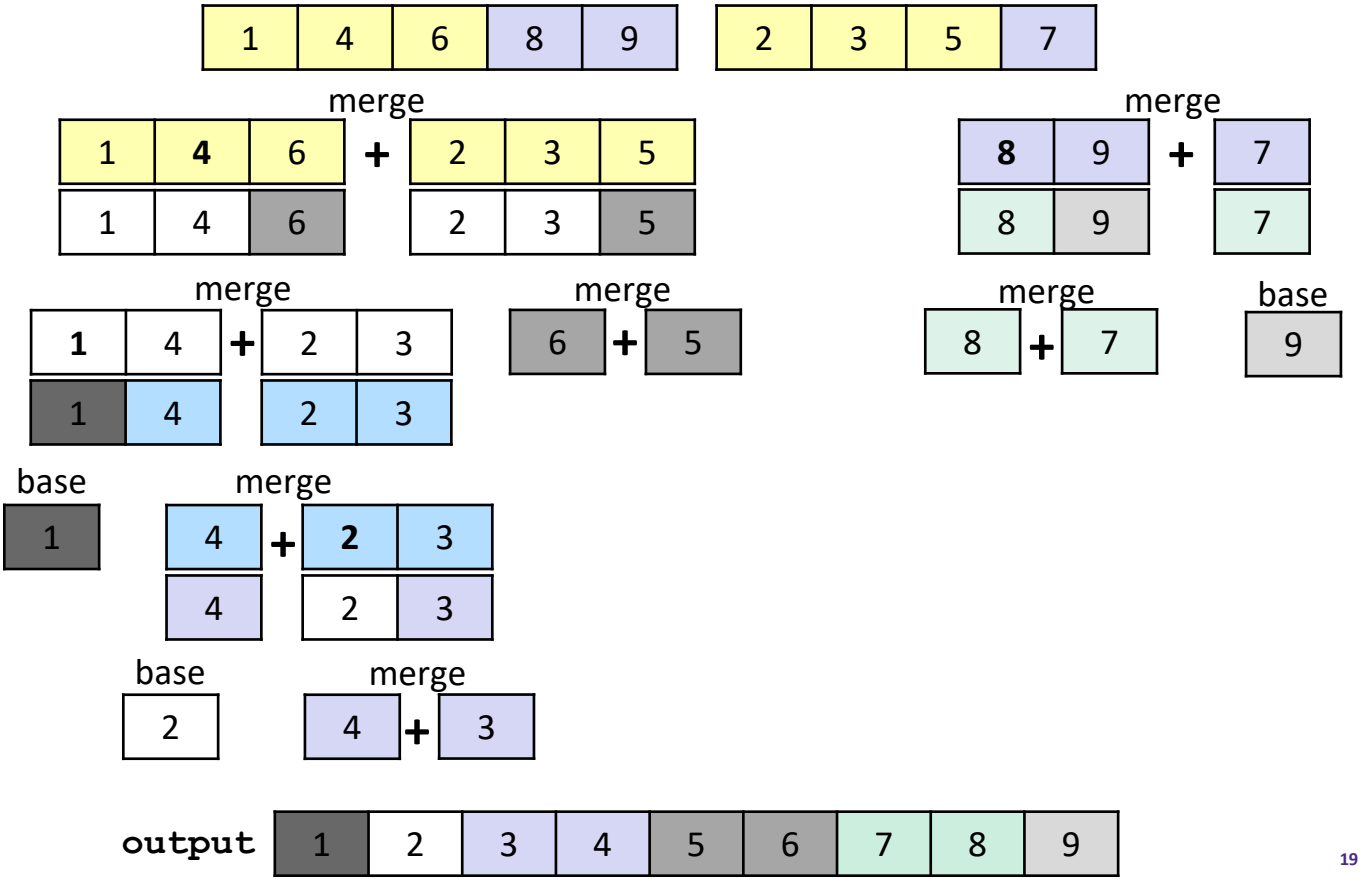
Parallelizing the Merge: Example (5 of 7)



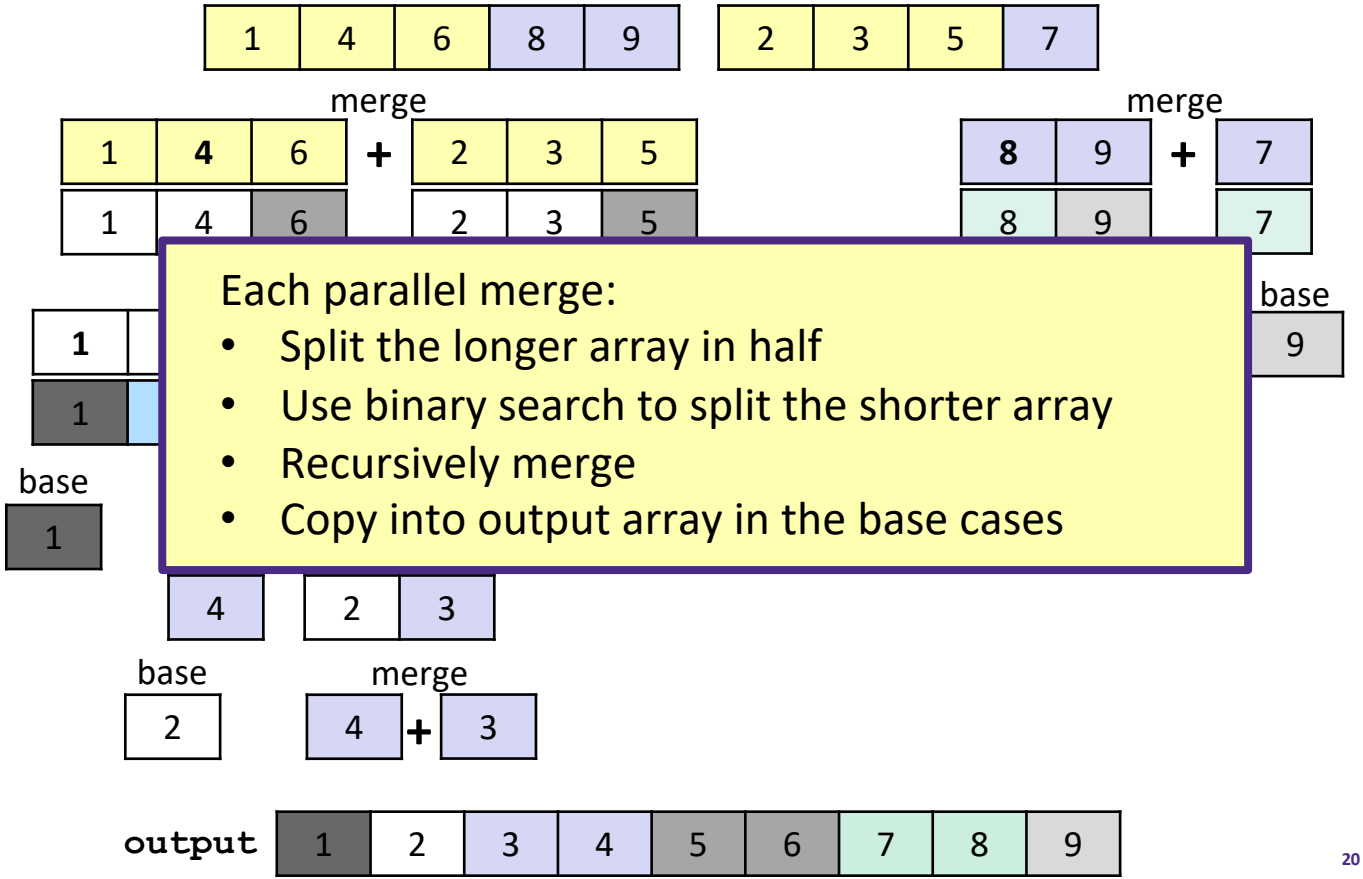
- Pick the median of the *longer array*: O(1) to compute index
- Split the *shorter array* at the same value: O(log n) for binary search
- Calculate where to split the output array: O(1)
- Do the sub-merges in parallel

🤔 *how do we sub-merge?* 🤔

Parallelizing the Merge: Example (6 of 7)



Parallelizing the Merge: Example (7 of 7)



Parallel Merge: Pseudocode

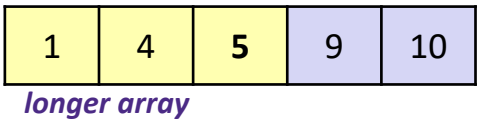
```
Merge(arr[], left1, left2, right1, right2, out[], out1, out2 )
   int leftSize = left2 – left1
   int rightSize = right2 – right1

   // Assert: out2 – out1 = leftSize + rightSize
   // We will assume leftSize > rightSize without loss of generality
   if (leftSize + rightSize < CUTOFF)
      sequential merge and copy into out[out1..out2]

   int mid = (left2 – left1)/2
   binarySearch arr[right1..right2] to find j such that
      arr[j] ≤ arr[mid] ≤ arr[j+1]

   Merge(arr[], left1, mid, right1, j, out[], out1, out1+mid+j)
   Merge(arr[], mid+1, left2, j+1, right2, out[], out1+mid+j+1, out2)
```

Parallel-MergeSort: Analysis (1 of 3)

- *Sequential* MergeSort:

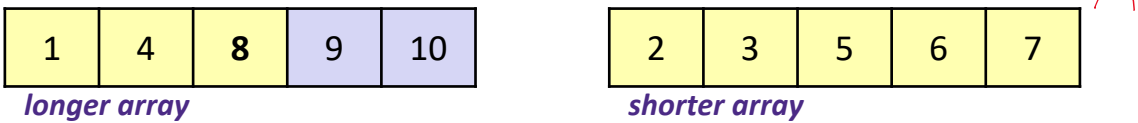
  $T(n) = 2T(n/2) + c_2 \quad \in \quad O(n \log n)$

- MergeSort with *parallel recursive calls* and a *sequential merge*:
  - **Work**: $O(n \log n)$
  - **Span**: $T(n) = \mathbf{1}T(n/2) + c_2 \quad \in \quad O(n)$



Parallel-MergeSort: Analysis (2 of 3)

- What about *just* the *parallel merge* of two subarrays?
  - Let the total length of the two subarrays be n
  - $O(\log n)$ binary search to split the shorter subarray
  - Worst-case split is $(3/4)n$ and $(1/4)n$
    - Happens when the two subarrays are of the same length (n/2) and the shorter subarray splits into two pieces of the most uneven sizes possible: one of size n/2, one of size 0
  - **Work** is $T(n) = T(3n/4) + T(n/4) + c_1 \log n \in O(n)$
  - **Span** is $T(n) = T(3n/4) + c_2 \log n \in O(\log^2 n)$
    - (neither bound is immediately obvious, but "trust me")



Parallel-MergeSort: Analysis (2 of 3)

- MergeSort with *parallel recursive calls* and a *parallel merge*:
  - **Work** is $T(n) = 2T(n/2) + c_1 n \in O(n \log n)$
  - **Span** is $T(n) = 1T(n/2) + c_2 \log^2 n \in O(\log^3 n)$
  - So, **parallelism** (work / span) is $O(n / \log^2 n)$
    - Not quite as good as QuickSort's $O(n / \log n)$ parallelism
    - But, unlike Quicksort, this is a worst-case guarantee
    - And, as always, this is just the asymptotic result

Summary

- Parallel-prefix sum can be used on a bit-vector to generate indices for a packed array
  - Which can then be used for parallel-pack

- Parallel-QuickSort starts with parallelizing its recursive calls
  - But its runtime is lower-bounded by the O(n) pivot operation
  - Parallel pivot is two

Lecture Outline

- Parallel Sorts (Divide & Conquer)
  - Parallel QuickSort
  - Parallel Mergesort
- **Sharing Resources**
- Intro to Concurrency
  - Managing Correct Access to Shared Resources
  - Mutual Exclusion and Critical Sections

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)

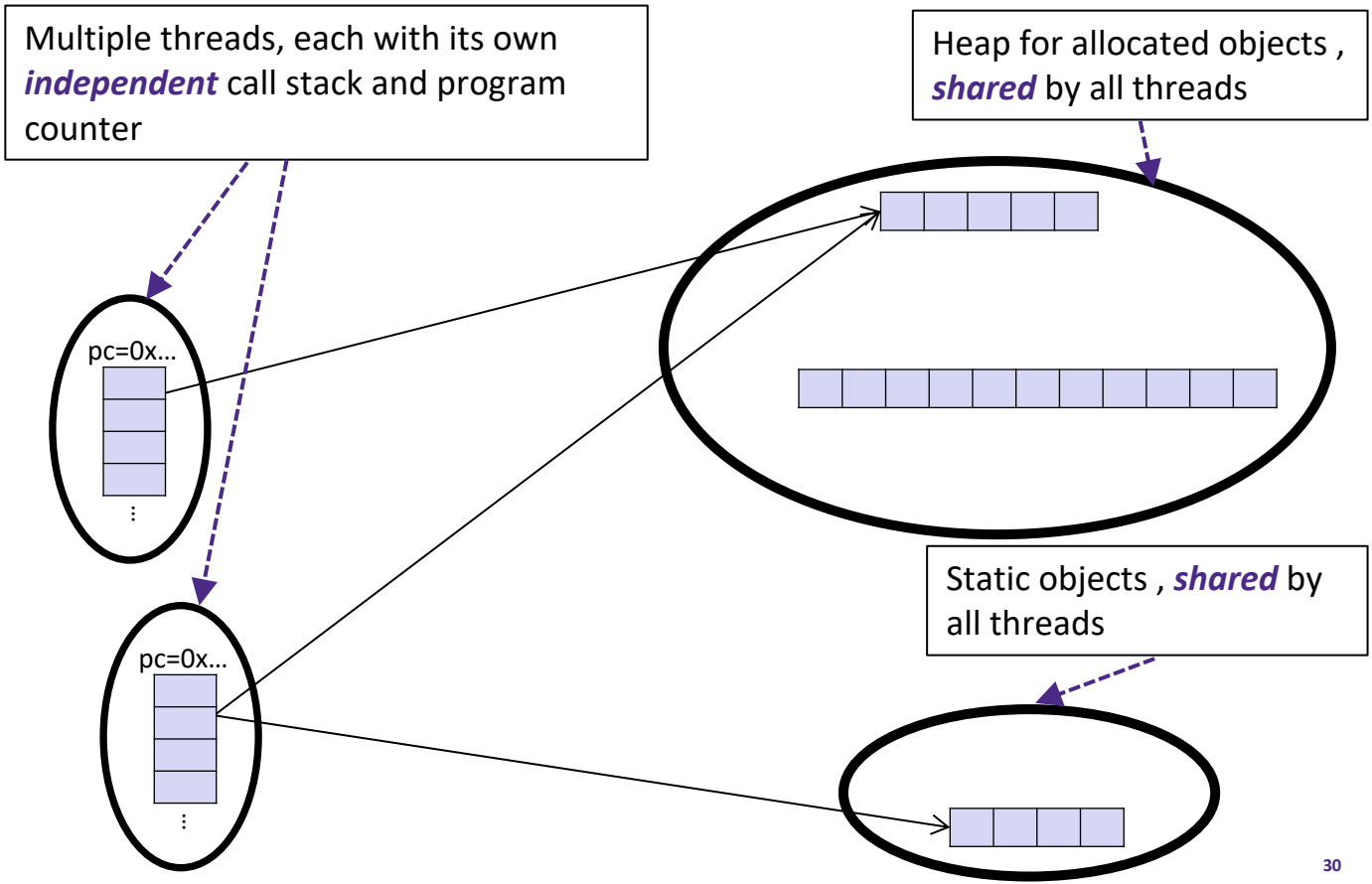
Parallelism's Pitfall

- 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

- Parallel Sorts (Divide & Conquer)
  - Parallel QuickSort
  - Parallel Mergesort
- Sharing Resources
- Intro to Concurrency
  - **Managing Correct Access to Shared Resources**
  - Mutual Exclusion and Critical Sections

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)`

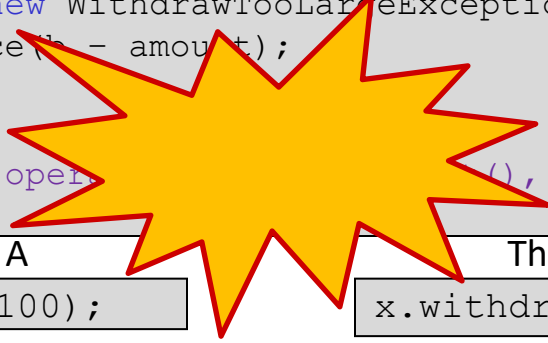
- If second call starts before first finishes, we say they **interleave**
  - e.g. T1 runs for 50 ms, pauses somewhere, T2 picks up for 50ms
  - Can happen with one processor; if **pre-empted** due to time-slicing

- 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

Thread A: `withdraw(100)`

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

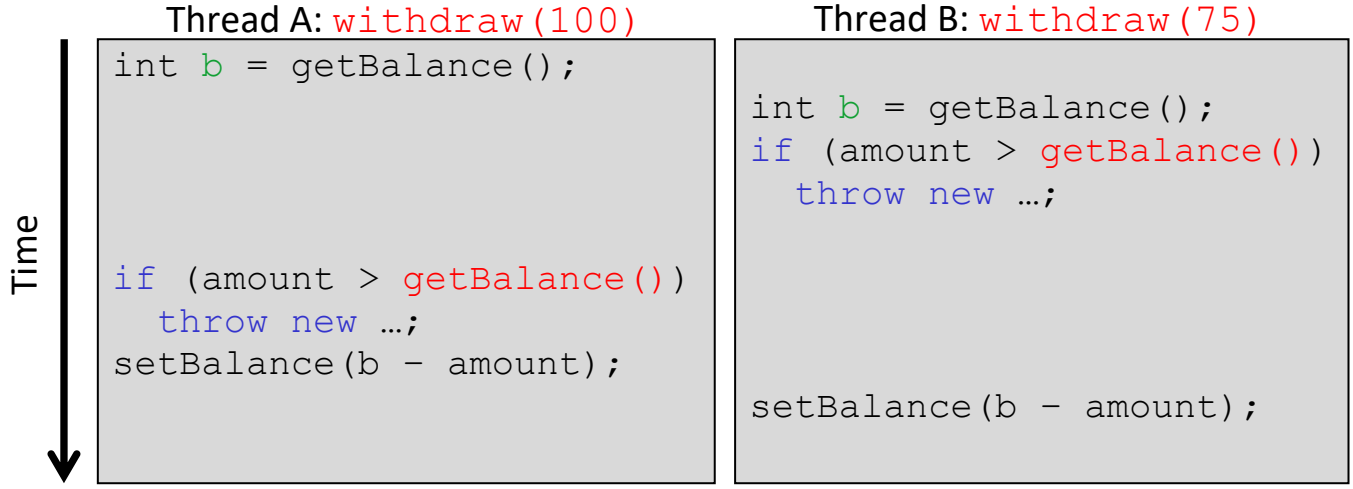
Thread B: `withdraw(75)`

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

Time

A 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)



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

- Parallel Sorts (Divide & Conquer)
  - Parallel QuickSort
  - Parallel Mergesort
- Sharing Resources
- Intro to Concurrency
  - Managing Correct Access to Shared Resources
  - **Mutual Exclusion and Critical Sections**

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
}
```

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

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

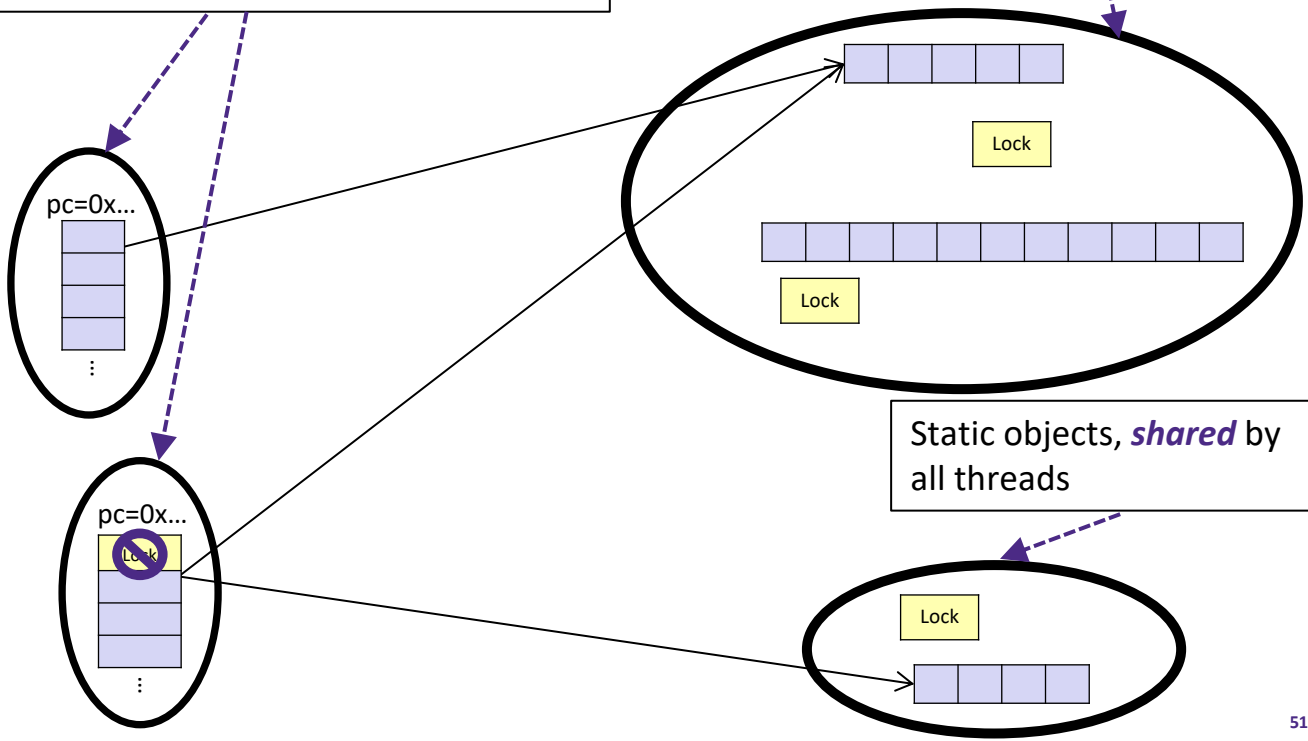
- Lock must ensure that, given simultaneous acquires/releases, "the correct thing" will happen
  - E.g.: if we have two acquires: one will "win" and one will block

- 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



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
}
```

Note: 'Lock' is not an actual Java class

gradescope

gradescope.com/courses/275833

Answer the questions based on 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**