CSE 332 Data Abstractions:
Parallel Sorting
&
Introduction to Concurrency

Kate Deibel
Summer 2012

Like last week was so like last week ago… like like like…

A QUICK REVIEW

Reductions

Such computations of this simple form are common enough to have a name: reductions (or reduces?)

Produce single answer from collection via an associative operator

- Examples: max, count, leftmost, rightmost, sum, …
- Non-example: median

Recursive results don't have to be single numbers or strings and can be arrays or objects with fields

- Example: Histogram of test results

But some things are inherently sequential

- How we process `arr[i]` may depend entirely on the result of processing `arr[i-1]`

Maps and Data Parallelism

A map operates on each element of a collection independently to create a new collection of the same size

- No combining results
- For arrays, this is so trivial some hardware has direct support (often in graphics cards)

Canonical example: Vector addition

```
int[] vector_add(int[] arr1, int[] arr2){
    assert (arr1.length == arr2.length);
    result = new int[arr1.length];
    FORALL(i=0; i < arr1.length; i++) {
        result[i] = arr1[i] + arr2[i];
    }
    return result;
}
```

Down Pass Example

| input | 6 | 4 | 16 | 10 | 16 | 14 | 2 | 8 |
| output | 6 | 10 | 26 | 36 | 52 | 66 | 68 | 76 |
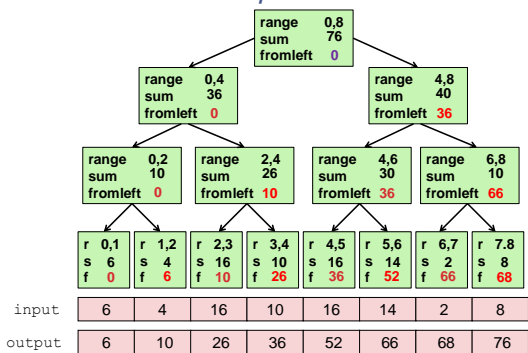
Pack (Think Filtering)

Given an array `input` and boolean function `f(e)` produce an array `output` containing only elements `e` such that `f(e)` is `true`

Example:

```
input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]
f(e): is e > 10?
output [17, 11, 13, 19, 24]
```

Is this parallelizable? Of course!

- Finding elements for the output is easy
- But getting them in the right place seems hard

Parallel Map + Parallel Prefix + Parallel Map

1. Use a parallel map to compute a bit-vector for true elements

```
input  [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]
bits   [ 1, 0, 0, 0,  1, 0,  1,  1, 0,  1]
```

2. Parallel-prefix sum on the bit-vector

```
bitsum [ 1, 1, 1, 1,  2, 2,  3,  4, 4,  5]
```

3. Parallel map to produce the output

```
output [17, 11, 13, 19, 24]
```

```
output = new array of size bitsum[n-1]
FORALL(i=0; i < input.length; i++){
  if(bits[i]==1)
    output[bitsum[i]-1] = input[i];
}
```

After this… perpendicular sorting…

PARALLEL SORTING

Quicksort Review

Recall that quicksort is sequential, in-place, and has expected time O(n $\log$ n)

	best/expected case
1. Pick a pivot element	O(1)
2. Partition all the data into:	O(n)
A. Elements less than the pivot	
B. The pivot	
C. Elements greater than the pivot	
3. Recursively sort A and C	2T(n/2)

Parallelize Quicksort?

How would we parallelize this?

1. Pick a pivot element
2. Partition all the data into: <pivot, pivot >pivot
3. Recursively sort A and C

Easy: Do the two recursive calls in parallel
- Work: unchanged $O(n \log n)$
- Span: T(n) = $O(n)$+T(n/2)
             = $O(n)$ + $O(n/2)$ + T(n/4)
             = $O(n)$
- So parallelism is $O(\log n)$ (i.e., work / span)

Doing Better

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

A Google search strongly suggests quicksort cannot do better as the partitioning cannot be parallelized
- The Internet has been known to be wrong!!
- But we will need auxiliary storage (will no longer in place)
- In practice, constant factors may make it not worth it, but remember Amdahl's Law and the long-term situation

Moreover, we already have everything we need to parallelize the partition step

Parallel Partition with Auxiliary Storage

Partition all the data into:
A. The elements less than the pivot
B. The pivot
C. The elements greater than the pivot

This is just two packs
- We know a pack is O(n) work, O(log n) span
- Pack elements < pivot into left side of aux array
- Pack elements >pivot into right side of aux array
- Put pivot between them and recursively sort
- With a little more cleverness, we can do both packs at once with NO effect on asymptotic complexity
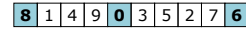
Analysis

With $O(\log n)$ span for partition, the total span for quicksort is:

$$
\begin{aligned}
T(n) &= O(\log n) + T(n/2) \\
&= O(\log n) + O(\log n/2) + T(n/4) \\
&= O(\log n) + O(\log n/2) + O(\log n/4) + T(n/8) \\
&\vdots \\
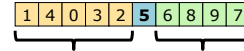&= O(\log^2 n)
\end{aligned}
$$

So parallelism (work / span) is $O(n / \log n)$

August 6, 2012   CSE 332 Data Abstractions, Summer 2012   13

Example

- Step 1: pick pivot as median of three

  | **8** | 1 | 4 | 9 | **0** | 3 | 5 | 2 | 7 | **6** |

- Steps 2a and 2c (combinable):
  Pack < and pack > into a second array
  - Fancy parallel prefix to pull this off not shown

  | 1 | 4 | 0 | 3 | 2 | **5** | 6 | 8 | 9 | 7 |

- Step 3: Two recursive sorts in parallel
  - Can sort back into original array (swapping back and forth like we did in sequential mergesort)

August 6, 2012   CSE 332 Data Abstractions, Summer 2012   14

Parallelize Mergesort?

Recall mergesort: sequential, not-in-place, worst-case $O(n \log n)$

		best/expected case
1.	Sort left half of array	T(n/2)
2.	Sort right half of array	T(n/2)
3.	Merge results	O(n)

Just like quicksort, doing the two recursive sorts in parallel changes the recurrence for the span to T(n) = O(n) + 1T(n/2) = O(n)
- Again, parallelism is O(log n)
- To do better, need to parallelize the merge
- The trick this time will not use parallel prefix

August 6, 2012   CSE 332 Data Abstractions, Summer 2012   15

Parallelizing the Merge

Need to merge two sorted subarrays (may not have the same size)

| 0 | 1 | 4 | 8 | 9 | | 2 | 3 | 5 | 6 | 7 |

Idea: Suppose the larger subarray has n elements. Then, in parallel:
- merge the first n/2 elements of the larger half with the "appropriate" elements of the smaller half
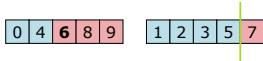- merge the second n/2 elements of the larger half with the remainder of the smaller half

August 6, 2012   CSE 332 Data Abstractions, Summer 2012   16

Example: Parallelizing the Merge

| 0 | 4 | 6 | 8 | 9 | | 1 | 2 | 3 | 5 | 7 |

August 6, 2012   CSE 332 Data Abstractions, Summer 2012   17

Example: Parallelizing the Merge

| 0 | 4 | **6** | 8 | 9 | | 1 | 2 | 3 | 5 | 7 |

1. Get median of bigger half: $O(1)$ to compute middle index

August 6, 2012   CSE 332 Data Abstractions, Summer 2012   18

Example: Parallelizing the Merge

1. Get median of bigger half: $O(1)$ to compute middle index
2. Find how to split the smaller half at the same value:
   O(`log` n) to do binary search on the sorted small half
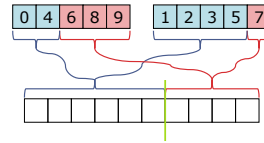
Example: Parallelizing the Merge

1. Get median of bigger half: $O(1)$ to compute middle index
2. Find how to split the smaller half at the same value:
   O(`log` n) to do binary search on the sorted small half
3. Two sub-merges conceptually splits output array: $O(1)$

Example: Parallelizing the Merge

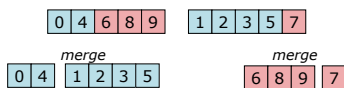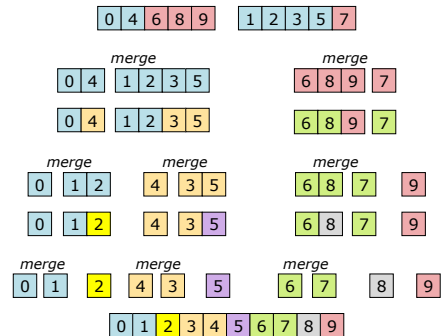
1. Get median of bigger half: $O(1)$ to compute middle index
2. Find how to split the smaller half at the same value:
   O(`log` n) to do binary search on the sorted small half
3. Two sub-merges conceptually splits output array: $O(1)$
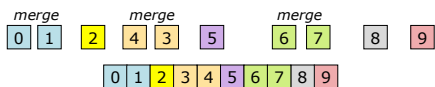4. Do two submerges in parallel

Example: Parallelizing the Merge

Example: Parallelizing the Merge
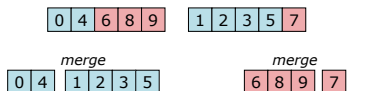
When we do each merge in parallel:
- we split the bigger array in half
- use binary search to split the smaller array
- And in base case we do the copy

Parallel Merge Pseudocode

Merge(arr[], $left_1$, $left_2$, $right_1$, $right_2$, out[], out_1, out_2)

   int leftSize = $left_2 - left_1$

   int rightSize = $right_2 - right_1$

   // Assert: $out_2 - out_1$ = 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 = $(left_2 - left_1)$/2

   binarySearch arr[right1..right2] to find j such that

     arr[j] ≤ arr[mid] ≤ arr[j+1]

   Merge(arr[], $left_1$, mid, $right_1$, j, out[], out_1, out_1+mid+j)

   Merge(arr[], mid+1, $left_2$, j+1, $right_2$, out[], out_1+mid+j+1, out_2)

Analysis

Sequential recurrence for mergesort:
$$T(n) = 2T(n/2) + O(n) \text{ which is } O(n \text{ log } n)$$

Parallel splitting but sequential merge:
work: same as sequential
span: $T(n)=1T(n/2)+O(n)$ which is $O(n)$

Parallel merge makes work and span harder to compute
- Each merge step does an extra $O(\text{log } n)$ binary search to find how to split the smaller subarray
- To merge n elements total, we must do two smaller merges of possibly different sizes
- But worst-case split is (1/4)n and (3/4)n
  - Larger array always splits in half
  - Smaller array can go all or none

Analysis

For just a parallel merge of n elements:
- Span is $T(n) = T(3n/4) + O(\text{log } n)$, which is $O(\text{log}^2 n)$
- Work is $T(n) = T(3n/4) + T(n/4) + O(\text{log } n)$ which is $O(n)$
- Neither bound is immediately obvious, but "trust us"

So for mergesort with parallel merge overall:
- Span is $T(n) = 1T(n/2) + O(\text{log}^2 n)$, which is $O(\text{log}^3 n)$
- Work is $T(n) = 2T(n/2) + O(n)$, which is $O(n \text{ log } n)$

So parallelism (work / span) is $O(n / \text{log}^2 n)$
- Not quite as good as quicksort's $O(n / \text{log } n)$
- But this is a worst-case guarantee and as always is just the asymptotic result

Articles of economic exchange within prison systems?

CONCURRENCY

Toward Sharing Resources

We have studied parallel algorithms using fork-join with the goal of lowering span via parallel tasks

All of the algorithms so far have had a very simple *structure* to avoid race conditions
- Each thread has memory only it can access: Example: array sub-range
- Or we used **fork** and **join** as a contract for who could access certain memory at each moment:
  *On **fork**, "loan" some memory to "forkee" and do not access that memory again until after **join** on the "forkee"*

This is far too limiting

What if memory accessed by threads is overlapping or unpredictable?

What if threads doing independent tasks need access to the same resources (as opposed to implementing the same algorithm)?

When we started talking about parallelism, we mentioned a topic we would talk about later
- Now is the time to talk about concurrency

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
- Blocking via **join** is not what we want
- We want to block until another thread is "done with what we need" and not the more extreme "until completely done executing"

Even correct concurrent applications are usually highly non-deterministic:
- how threads are scheduled affects what each thread sees in its different operations
- non-repeatability complicates testing and debugging

Examples Involving Multiple Threads

Processing different bank-account operations
- What if 2 threads change the same account at the same time?

Using a shared cache (hashtable) of recent files
- What if 2 threads insert the same file at the same time?

Creating a pipeline with a queue for handing work to next thread in sequence (a virtual assembly line)?
- What if enqueuer and dequeuer adjust a circular array queue at the same time?

Why Threads?

Unlike parallelism, this is not about implementing algorithms faster

But threads still have other uses:
- *Code structure for responsiveness*
  - Respond to GUI events in one thread
  - Perform an expensive computation in another
- *Processor utilization (mask I/O latency)*
  - If 1 thread "goes to disk," do something else
- *Failure isolation*
  - Convenient structure if want to interleave tasks not 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
  - We must be much more disciplined when designing/implementing a concurrent program
  - We will discuss common idioms known to work

Canonical example: Bank Account

The following is 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.
}
```

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, as a thread can be pre-empted for time-slicing
  (e.g., T1 runs 50 ms, T2 runs 50ms, T1 resumes)

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…

Bad Interleaving

Interleaved `withdraw(100)` calls on same account
Assume initial `balance == 150`
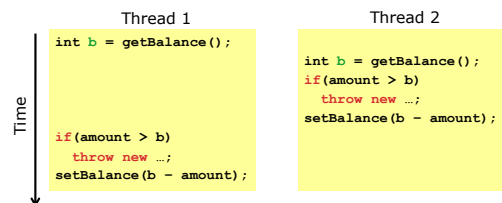
Thread 1
```
int b = getBalance();


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

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

Time

6

Incorrect Attempt to "Fix"

Interleaved `withdraw(100)` calls on same account
Assume initial `balance == 150`

Thread 1	Thread 2

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

Time ↓

> This interleaving would work and throw an exception

Incorrect Attempt to "Fix"

Interleaved `withdraw(100)` calls on same account
Assume initial `balance == 150`

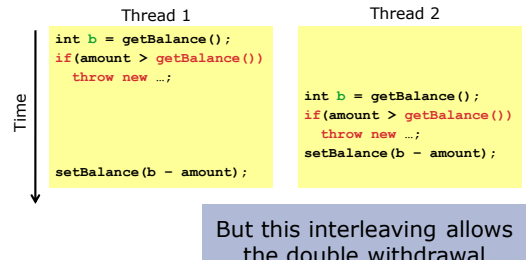| Thread 1	Thread 2

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

Time ↓

> But this interleaving allows the double withdrawal
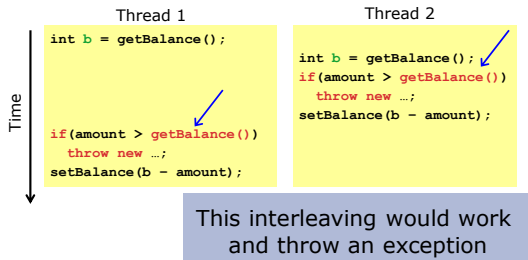
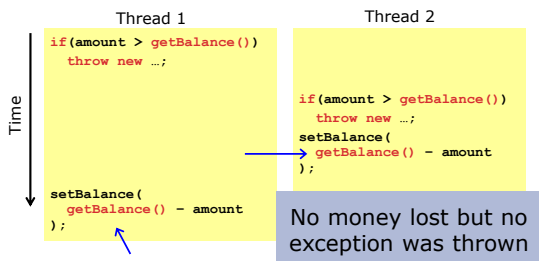## Another Incorrect Attempt to "Fix"

Interleaved `withdraw(100)` calls on same account
Assume initial `balance == 150`

Thread 1	Thread 2

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

Time ↓

> No money lost but no exception was thrown

Incorrect Attempt to "Fix"

It can be tempting, but is generally wrong, to attempt to "fix" a bad interleaving by rearranging or repeating operations

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

Only narrows the problem by one statement

- Imagine a withdrawal is interleaved after computing the value of the parameter `getBalance()-amount` but before invocation of the function `setBalance`

Compiler optimizations may even remove the second call to `getBalance()` since you did not tell it you need to synchronize

Mutual Exclusion

The simplest fix is to allow only one thread at a time to withdraw from the account

- Also exclude other simultaneous account operations that could potentially result in bad interleavings (e.g., deposits)

Mutual exclusion: One thread doing something with a resource means that any other thread must wait until the resource is available

- Define critical sections of code that are mutually exclusive

Programmer must implement critical sections

- "The compiler" has no idea what interleavings should or should not be allowed in your program
- But you will need language primitives to do this

Incorrect Attempt to "Do it Ourselves"

```
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 InsufficientFundsException();
    setBalance(b – amount);
    busy = false;
  }
  // deposit would spin on same boolean
}
```

7

This Just Moves the Problem

Thread 1	Thread 2

Time

```
while(busy) { }

busy = true;

int b = getBalance();




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

```
while(busy) { }

busy = true;

int b = getBalance();
if(amount > b)
  throw new …;
setBalance(b - amount);
```
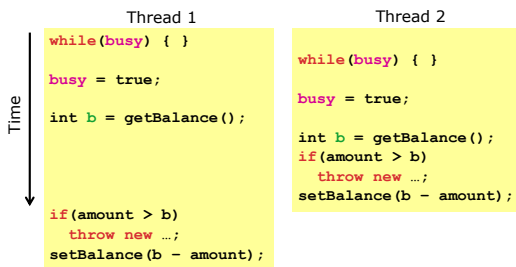
Need Help from the Language

There are many ways out of this conundrum

One basic solution: Locks
- Still on a conceptual, '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 atomically, cannot be interrupted (requires hardware and system support)
- **release**: makes this lock *"not held"*
  - if ≥1 threads are blocked, exactly 1 will acquire it

Still Incorrect Pseudocode

```
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 InsufficientFundsException();
    setBalance(b - amount);
    lk.release();
  }
  // deposit would also acquire/release lk
}
```

Some Mistakes

A lock is a very primitive mechanism but must be used correctly to implement critical sections

Incorrect: Forget to release lock, other threads blocked forever
- Previous slide is wrong because of the exception possibility

```
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
- Balance is the shared resource that is being protected

Poor performance: Use same lock for every bank account
- No simultaneous withdrawals from *different* accounts

Other Operations

- If **withdraw** and **deposit** use same lock, then simultaneous method calls are synchronized
- But what about **getBalance** and **setBalance**
  - Assume they are **public** (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

```
…
    lk.acquire();
    int b = getBalance();
…
```

One Bad Option

```
int setBalanceUnsafe(int x) {
  balance = x;
}

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

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

Two versions of setBalance
- Safe and unsafe versions
- You use one or the other, depending on whether you already have the lock

Technically could work
- Hard to always remember
- And definitely poor style

Better to modify meaning of the Lock ADT to support *re-entrant locks*

Re-Entrant Locking

A re-entrant lock is also known as a recursive lock
- "Remembers" the thread that currently holds it
- Stores a *count* of "how many" times it is held

When lock goes from ***not-held*** to ***held***, the count is 0

If 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***

`withdraw` can acquire the lock, and then call `setBalance`

Java's Re-Entrant Lock

`java.util.concurrent.locks.ReentrantLock`
- Has methods `lock()` and `unlock()`

Be sure to guarantee that the lock is always released

```
myLock.lock();
try {
        // method body
} finally {
        myLock.unlock();
}
```

Regardless of what happens in the 'try', the finally code will execute and release the lock

A Java Convenience: `Synchronized`

You can use the `synchronized` statement as an alternative to declaring a `ReentrantLock`

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

1. Evaluates expression to an object
   - Every *object* "is a lock" in Java (not *primitives*)
2. Acquires the lock, blocking if necessary
   - "If you get past the {, you have the lock"
3. Releases the lock "at the matching }"
   - Release occurs even if control leaves due to a throw, return, or whatever
   - So it is impossible to forget to release the lock

Version 1: Correct but not "Java Style"

```
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

Example motivations with our bank record?
- Plenty!

It is more common to synchronize on `this`
- It is also convenient
- No need to declare an extra object

Version 2: Something Tastes Bitter

```
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

Java provides a concise and standard way to say the same thing:

Applying the `synchronized` keyword to a method declaration means the entire method body is surrounded by

```
synchronized(this){
    …
}
```

Next version means exactly the same thing, but is more concise and more the "style of Java"

Version 3: Final Version

```
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 …
    setBalance(b – amount);
  }
  // deposit would also use synchronized
}
```

Some horses like wet tracks or dry tracks or muddy tracks…

MORE ON RACE CONDITIONS

Races

A race condition occurs when the computation result depends on scheduling (how threads are interleaved on ≥1 processors)
- Only occurs if T1 and T2 are scheduled in a particular way
- As programmers, we cannot control the scheduling of threads
- Program correctness must be independent of scheduling

Race conditions are bugs that exist only due to concurrency
- No interleaved scheduling with 1 thread

Typically, the problem is some *intermediate state* that "messes up" a concurrent thread that "sees" that state

We will distinguish between data races and bad interleavings, both of which are types of race condition bugs

Data Races

A data race is a type of *race condition* that can happen in two ways:
- Two threads **potentially** write a variable at the same time
- One thread **potentially** write a variable while another reads

Not a race: simultaneous reads provide no errors

Potentially is important
- We claim that code itself has a data race independent of any particular actual execution

Data races are bad, but they are not the only form of race conditions
- We can have a race, and bad behavior, without any data race

Stack Example

```
class Stack<E> {
  private E[] array = (E[])new Object[SIZE];
  int index = -1;
  synchronized boolean isEmpty() {
    return index==-1;
  }
  synchronized void push(E val) {
    array[++index] = val;
  }
  synchronized E pop() {
    if(isEmpty())
      throw new StackEmptyException();
    return array[index--];
  }
}
```

A Race Condition: But Not a Data Race

```
class Stack<E> {
  …
  synchronized boolean isEmpty() {…}
  synchronized void push(E val) {…}
  synchronized E pop(E val) {…}

E peek() {
  E ans = pop();
  push(ans);
  return ans;
}
```

In a sequential world, this code is of iffy, ugly, and questionable *style*, but *correct*

The "algorithm" is the only way to write a **peek** helper method if this interface is all you have to work with

Note that peek() throws the StackEmpty exception via its call to pop()

peek in a Concurrent Context

peek has no *overall* effect on the shared data
- It is a "reader" not a "writer"
- State should be the same after it executes as before

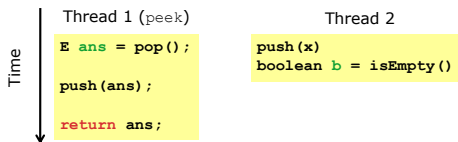This implementation creates an inconsistent *intermediate state*
- Calls to **push** and **pop** are synchronized, so there are no *data races* on the underlying array
- But there is still a **race condition**
- This intermediate state should not be exposed
  - Leads to several *bad interleavings*

```
E peek() {
    E ans = pop();
    push(ans);
    return ans;
}
```

Example 1: peek and isEmpty

Property we want:
If there has been a **push** (and no **pop)**, then **isEmpty** should return **false**
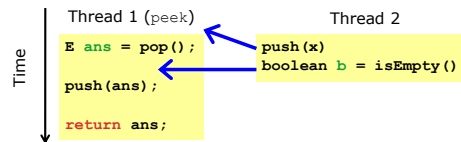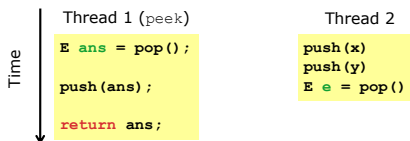
With **peek** as written, property can be violated – how?

Thread 1 (peek)	Thread 2
E ans = pop();	push(x)
	boolean b = isEmpty()
push(ans);	
return ans;	

Time

Example 1: peek and isEmpty

Property we want:
If there has been a **push** (and no **pop)**, then **isEmpty** should return **false**

Race causes error with:
T2: push(x)
T1: pop()
T2: isEmpty()

With **peek** as written, pr violated – how?

Thread 1 (peek)	Thread 2
E ans = pop();	push(x)
	boolean b = isEmpty()
push(ans);	
return ans;	

Time

Example 2: peek and push

Property we want:
Values are returned from pop in LIFO order

With **peek** as written, property can be violated – how?
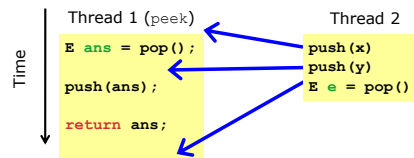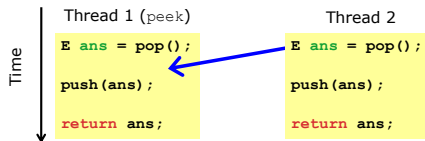
Thread 1 (peek)	Thread 2
E ans = pop();	push(x)
	push(y)
push(ans);	E e = pop()
return ans;	

Time

Example 2: peek and push

Property we want:
Values are returned from pop in LIFO order

Race causes error with:
T2: push(x)
T1: pop()
T2: push(x)
T1: push(x)

With **peek** as written, pr violated – how?

Thread 1 (peek)	Thread 2
E ans = pop();	push(x)
	push(y)
push(ans);	E e = pop()
return ans;	

Time

11

Example 3: peek and peek

Property we want:
peek does not throw an exception unless the stack is empty

With **peek** as written, property can be violated – how?

	Thread 1 (peek)	Thread 2
Time	`E ans = pop();`	`E ans = pop();`
	`push(ans);`	`push(ans);`
	`return ans;`	`return ans;`

The Fix

peek needs synchronization to disallow interleavings
- The key is to make a *larger critical section*
- This protects the intermediate state of peek
- Use re-entrant locks; will allow calls to **push** and **pop**
- Can be done in stack (left) or an external class (right)

```
class Stack<E> {
  …
  synchronized E peek(){
    E ans = pop();
    push(ans);
    return ans;
  }
}
```

```
class C {
  <E> E myPeek(Stack<E> s){
    synchronized (s) {
      E ans = s.pop();
      s.push(ans);
      return ans;
    }
  }
}
```

An Incorrect "Fix"

So far we have focused on problems created when **peek** performs writes that lead to an incorrect intermediate state

A tempting but incorrect perspective
- If an implementation of **peek** does not write anything, then maybe we can skip the synchronization?

Does not work due to *data races* with **push** and **pop**
- Same issue applies with other readers, such as **isEmpty**

Another Incorrect Example

```
class Stack<E> {
  private E[] array = (E[])new Object[SIZE];
  int index = -1;
  boolean isEmpty() { // unsynchronized: wrong?!
    return index==-1;
  }
  synchronized void push(E val) {
    array[++index] = val;
  }
  synchronized E pop() {
    return array[index--];
  }
  E peek() { // unsynchronized: wrong!
    return array[index];
  }
}
```

Why Wrong?

It *looks like* **isEmpty** and **peek** can "get away with this" because **push** and **pop** adjust the stacl's state using "just one tiny step"

But this code is still *wrong* and depends on language-implementation details you cannot assume
- Even "tiny steps" may require multiple steps in implementation: `array[++index] = val` probably takes at least two steps
- Code has a data race, allowing very strange behavior

Do not introduce a data race, even if every interleaving you can think of is correct

Getting It Right

Avoiding race conditions on shared resources is difficult
- Decades of bugs have led to some conventional wisdom and general techniques known to work

We will discuss some key ideas and trade-offs
- More available in the suggested additional readings
- None of this is *specific* to Java or a particular book
  - May be hard to appreciate in beginning
  - Come back to these guidelines over the years
  - Do not try to be fancy

Three Choices for Memory

For every memory location in your program (e.g., object field), you must obey at least one of the following:

1. Thread-local: Do not use the location in >1 thread
2. Immutable: Never write to the memory location
3. Synchronized: Control access via synchronization

needs synchronization

all memory — thread-local memory — immutable memory

Yale University is the best place to study locks…

GOING FURTHER WITH EXCLUSION AND LOCKING

Thread-Local

Whenever possible, do not share resources!

- Easier for each thread to have its own thread-local copy of a resource instead of one with shared updates
- Correct only if threads do not communicate through resource
  - In other words, multiple copies are correct approach
  - Example: `Random` objects
- Note: Because each call-stack is thread-local, never need to synchronize on local variables

In typical concurrent programs, the vast majority of objects should be thread-local and shared-memory usage should be minimized

---

Immutable

Whenever possible, do not update objects

- Make new objects instead

One of the key tenets of *functional programming* (see CSE 341 Programming Languages)

- Generally helpful to avoid *side-effects*
- Much more helpful in a concurrent setting

If a location is only ever read, never written, no synchronization needed

- Simultaneous reads are *not* races (not a problem!)

In practice, programmers usually over-use mutation so you should do your best to minimize it

---

Everything Else: Keep it Synchronized

After minimizing the amount of memory that is both (1) thread-shared and (2) mutable, we need to follow guidelines for using locks to keep that data consistent

Guideline #0: No data races

- Never allow two threads to read/write or write/write the same location at the same time

Necessary:
   In Java or C, a program with a data race is almost always wrong

But Not Sufficient:
   Our `peek` example had no data races

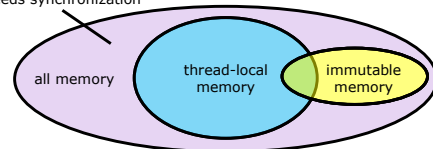## Consistent Locking

Guideline #1: Consistent Locking

For each location that requires synchronization, we should have a lock that is always held when reading or writing the location

- We say the lock guards the location
- The same lock can guard multiple locations (and often should)
- Clearly document the guard for each location
- In Java, the guard is often the object containing the location
  - `this` inside object methods
  - Also common to guard a larger structure with one lock to ensure mutual exclusion on the structure

Consistent Locking

The mapping from locations to guarding locks is *conceptual,* and must be enforced by you as the programmer

- It partitions the shared-&-mutable locations into "which lock"

Consistent locking is:

Not Sufficient:
It prevents all data races, but still allows bad interleavings

- Our peek example used consistent locking, but had exposed intermediate states and bad interleavings

Not Necessary:

- Can dynamically change the locking protocol

Beyond Consistent Locking

Consistent locking is an excellent guideline

- A "default assumption" about program design
- You will save yourself many a headache using this guideline

But it is not required for correctness:
Different *program phases* can use different locking techniques

- Provided all threads coordinate moving to the next phase

Example from Project 3 Version 5:

- A shared grid being updated, so use a lock for each entry
- But after the grid is filled out, all threads except 1 terminate thus making synchronization no longer necessary (i.e., now only thread local)
- And later the grid is only read in response to queries thereby making synchronization doubly unnecessary (i.e., immutable)

Whole-grain locks are better than overly processed locks...

LOCK GRANULARITY

Lock Granularity

Coarse-Grained: Fewer locks (more objects per lock)

- Example: One lock for entire data structure (e.g., array)
- Example: One lock for all bank accounts
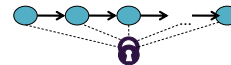
Fine-Grained: More locks (fewer objects per lock)

- Example: One lock per data element (e.g., array index)
- Example: One lock per bank account

"Coarse-grained vs. fine-grained" is really a continuum

Trade-Offs

Coarse-grained advantages

- Simpler to implement
- Faster/easier to implement operations that access multiple locations (because all guarded by the same lock)
- Easier to implement modifications of data-structure shape

Fine-grained advantages

- More simultaneous access (improves performance when coarse-grained would lead to unnecessary blocking)

Guideline #2: Lock Granularity
Start with coarse-grained (simpler), move to fine-grained (performance) only if *contention* on coarse locks is an issue. Alas, often leads to bugs.

Example: Separate Chaining Hashtable

Coarse-grained: One lock for entire hashtable
Fine-grained: One lock for each bucket

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

Maintaining a `numElements` field will destroy the potential benefits of using separate locks for each bucket, why?
Updating each insert without a coarse lock would be a data race

Critical-Section Granularity

A second, orthogonal granularity issue is the size of critical-sections
- How much work should we do while holding lock(s)

If critical sections run for too long:
- Performance loss as other threads are blocked

If critical sections are too short:
- Bugs likely as you broke up something where other threads shouldn't be able to see intermediate state

Guideline #3: Granularity
> Do not do expensive computations or I/O in critical sections, but also do not introduce race conditions

Example: Critical-Section Granularity

Suppose we want to change the value for a key in a hashtable without removing it from the table
- Assume `lock` guards the whole table

Papa Bear's critical section was too long

Table is locked during the expensive call

```
synchronized(lock) {
  v1 = table.lookup(k);
  v2 = expensive(v1);
  table.remove(k);
  table.insert(k,v2);
}
```

Example: Critical-Section Granularity

Suppose we want to change the value for a key in a hashtable without removing it from the table
- Assume `lock` guards the whole table

Mama Bear's critical section was too short

If another thread updated the entry, we will lose the intervening update

```
synchronized(lock) {
  v1 = table.lookup(k);
}
v2 = expensive(v1);
synchronized(lock) {
  table.remove(k);
  table.insert(k,v2);
}
```

Example: Critical-Section Granularity

Suppose we want to change the value for a key in a hashtable without removing it from the table
- Assume `lock` guards the whole table

Baby Bear's critical section was just right

if another update occurred, we will try our update again

```
done = false;
while(!done) {
  synchronized(lock) {
    v1 = table.lookup(k);
  }
  v2 = expensive(v1);
  synchronized(lock) {
    if(table.lookup(k)==v1) {
      done = true;
      table.remove(k);
      table.insert(k,v2);
}}}
```

Atomicity

An operation is *atomic* if no other thread can see it partly executed
- Atomic as in "appears indivisible"
- We typically want ADT operations atomic, even to other threads running operations on the same ADT

Guideline #4: Atomicity
- Think in terms of what operations need to be *atomic*
- Make critical sections just long enough to preserve atomicity
- *Then* design locking protocol to implement critical sections

In other words:
> *Think about atomicity first and locks second*

Do Not Roll Your Own

In real life, you rarely write your own data structures
- Excellent implementations provided in standard libraries
- Point of CSE 332 is to understand the key trade-offs, abstractions, and analysis of such implementations

Especially true for concurrent data structures
- Far too difficult to provide fine-grained synchronization without race conditions
- Standard thread-safe libraries like `ConcurrentHashMap` are written by world experts and been extensively vetted

Guideline #5: Libraries
Use built-in libraries whenever they meet your needs

Motivating Memory-Model Issues

Tricky and *surprisingly wrong* unsynchronized concurrent code

```
class C {
  private int x = 0;
  private int y = 0;

  void f() {
    x = 1;
    y = 1;
  }

  void g() {
    int a = y;
    int b = x;
    assert(b >= a);
  }
}
```

First understand why it looks like the assertion cannot fail:

Easy case:
A call to g ends before any call to f starts

Easy case:
At least one call to f completes before call to g starts

If calls to f and g interleave…

Interleavings Are Not Enough

There is no interleaving of `f` and `g` such that the assertion fails

Proof #1:
Exhaustively consider all possible orderings of access to shared memory (there are 6)

Interleavings are Not Enough

Proof #2:
Exhaustively consider all possible orderings of access to shared memory (there are 6)

If `!(b>=a)`, then `a==1` and `b==0`.
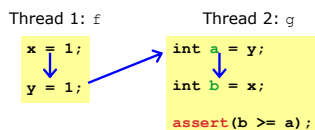But if `a==1`, then `y=1` happened before `a=y`.
Because programs execute in order:
    `a=y` happened before `b=x`
    and `x=1` happened before `y=1`
So by transitivity, `b==1`.
Contradiction.

Thread 1: f   Thread 2: g
```
x = 1;               int a = y;
y = 1;               int b = x;
                     assert(b >= a);
```

Wrong

However, the code has a *data race*
- Unsynchronized read/write or write/write of the memory same location

If code has data races, you cannot reason about it with interleavings
- This is simply the rules of Java (and C, C++, C#, other languages)
- Otherwise we would slow down all programs just to "help" those with data races, and that would not be a good engineering trade-off
- So the assertion can fail

Why

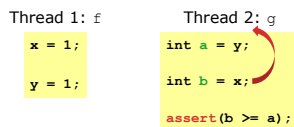For performance reasons, the compiler and the hardware will often reorder memory operations
- Take a compiler or computer architecture course to learn more as to why this is good thing

Thread 1: f   Thread 2: g
```
x = 1;               int a = y;
y = 1;               int b = x;
                     assert(b >= a);
```

Of course, compilers cannot just reorder anything they want without careful consideration
- Each thread computes things by executing code in order
- Consider: `x=17; y=x;`

The Grand Compromise

The compiler/hardware will NEVER:
- Perform a memory reordering that affects the result of a single-threaded program
- Perform a memory reordering that affects the result of a data-race-free multi-threaded program

So: If no interleaving of your program has a data race, then you can *forget about all this reordering nonsense:* the result will be equivalent to some interleaving

The Big Picture:
- Your job is to avoid data races
- The compiler/hardware's job is to give illusion of interleaving *if you do your job right*

Fixing Our Example

Naturally, we can use synchronization to avoid data races and then, indeed, the assertion cannot fail

```
class C {
  private int x = 0;
  private int y = 0;
  void f() {
    synchronized(this) { x = 1; }
    synchronized(this) { y = 1; }
  }
  void g() {
    int a, b;
    synchronized(this) { a = y; }
    synchronized(this) { b = x; }
    assert(b >= a);
  }
}
```

A Second Fix: Stay Away from This

Java has **volatile** fields: accesses do not count as data races
- But you cannot read-update-write

```
class C {
  private volatile int x = 0;
  private volatile int y = 0;
  void f() {
    x = 1; y = 1;
  }
  void g() {
    int a = y; int b = x;
    assert(b >= a);
  }
}
```

Implementation Details
- Slower than regular fields but faster than locks
- Really for experts: avoid them; use standard libraries instead
- And why do you need code like this anyway?

Code That is Wrong

Here is a more realistic example of code that is wrong
- No *guarantee* Thread 2 will *ever* stop (due to data race)
- But honestly it will "likely work in practice"

```
class C {
  boolean stop = false;

  void f() {
    while(!stop) {
      // draw a monster
    }
  }

  void g() {
    stop = didUserQuit();
  }
}
```

Thread 1: `f()`

Thread 2: `g()`

Not nearly as silly as Deathlok from Marvel comics...

DEADLOCK

Motivating Deadlock Issues

Consider the following method for transfering money between bank accounts

```
class BankAccount {
  …
  synchronized void withdraw(int amt) {…}
  synchronized void deposit(int amt) {…}
  synchronized void transferTo(int amt,
                      BankAccount a) {
    this.withdraw(amt);
    a.deposit(amt);
  }
}
```
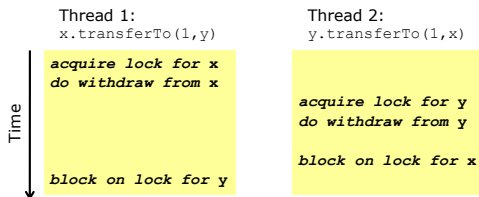
During call to a.deposit, the thread holds two locks
- Let's investigate when this may be a problem

The Deadlock

Suppose **x** and **y** are fields holding accounts

Thread 1:
`x.transferTo(1,y)`

acquire lock for x
do withdraw from x

block on lock for y

Thread 2:
`y.transferTo(1,x)`

acquire lock for y
do withdraw from y

block on lock for x

Time

The Dining Philosophers

Five philosophers go out to dinner together at an Italian restaurant

They sit at a round table; one fork per plate setting

For etiquette reasons, the philosophers need two forks to eat spaghetti properly

When the spaghetti comes, each philosopher proceeds to grab their right fork, then their left fork

'Locking' for each fork results in a ***deadlock***

Deadlock

A deadlock occurs when there are threads $T_1, …, T_n$ such that:

- For i=1 to n-1, T_i is waiting for at least one resource held by T_{i+1}
- T_n is waiting for a resource held by T_1

In other words, there is a *cycle* of waiting
- More formally, a graph of dependencies is cyclic

Deadlock avoidance in programming amounts to techniques to ensure a cycle can never arise

Back to Our Example

Options for deadlock-proof transfer:

1. Make a smaller critical section:
   `transferTo` not synchronized
   - Exposes intermediate state after `withdraw` before `deposit`
   - May be okay, but exposes wrong total amount to bank

2. Coarsen lock granularity:
   One lock for all accounts allowing transfers between them
   - Works, but sacrifices concurrent deposits/withdrawals

3. Give every bank-account a unique number and always acquire locks in the same order
   - *Entire program* should obey this order to avoid cycles
   - Code acquiring only one lock can ignore the order

Ordering Locks

```
class BankAccount {
  …
  private int acctNumber; // must be unique
  void transferTo(int amt, BankAccount a) {
    if(this.acctNumber < a.acctNumber)
      synchronized(this) {
      synchronized(a) {
        this.withdraw(amt);
        a.deposit(amt);
      }}
    else
      synchronized(a) {
      synchronized(this) {
        this.withdraw(amt);
        a.deposit(amt);
      }}
  }
}
```

StringBuffer Example

From the Java standard library

```
class StringBuffer {
  private int count;
  private char[] value;
  …
  synchronized append(StringBuffer sb) {
    int len = sb.length();
    if(this.count + len > this.value.length)
      this.expand(…);
    sb.getChars(0,len,this.value,this.count);
    …
  }
  synchronized getChars(int x, int, y,
                        char[] a, int z) {
    "copy this.value[x..y] into a starting at z"
  }
}
```

Two Problems

Problem #1:
Lock for `sb` not held between calls to `sb.length` and `sb.getChars`
- So `sb` could get longer
- Would cause `append` to throw an `ArrayBoundsException`

Problem #2:
Deadlock potential if two threads try to `append` in opposite directions, identical to the bank-account first example

Not easy to fix both problems without extra copying:
- Do not want unique ids on every `StringBuffer`
- Do not want one lock for all `StringBuffer` objects

Actual Java library:
Fixed neither (left code as is; changed documentation)
- Up to clients to avoid such situations with their own protocols

Perspective

Code like account-transfer and string-buffer append are difficult to deal with for deadlock

Easier case: different types of objects
- Can establish and document a fixed order among types
- Example: "When moving an item from the hashtable to the work queue, never try to acquire the queue lock while holding the hashtable lock"

Easier case: objects are in an acyclic structure
- Can use the data structure to determine a fixed order
- Example: "If holding a tree node's lock, do not acquire other tree nodes' locks unless they are children"