



Wrap Parallel Concurrency

CSE 332 Sp25
Lecture 22

Announcements

	Monday	Tuesday	Wed	Thursday	Friday
This Week	Ex 9 (reductions, gs) due Ex 11 (parallel prog) out		TODAY		Ex 10 (F-J prog) due Ex 12 (concurrency, GS) out
Next Week	Veteran's Day (no class)	Ex 11 due			Ex 12 due

Optional readings ([Grossman](#)) covers next few weeks of parallelism and concurrency

Amdahl's Law: Moving Forward

Unparallelized code becomes a bottleneck quickly.

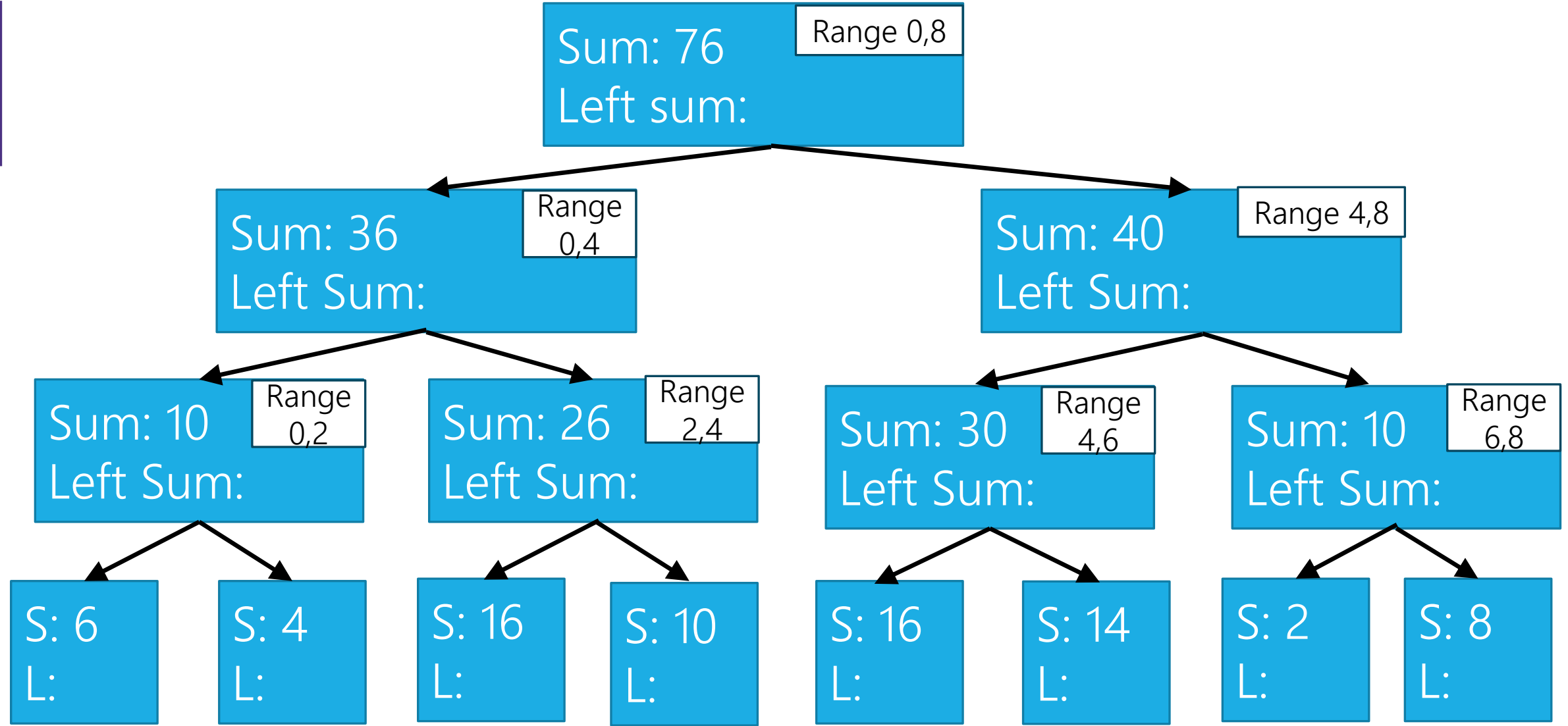
What do we do? Design smarter algorithms!

Consider the following problem:

Given an array of numbers, return an array with the "running sum"

3	7	6	2	4
---	---	---	---	---

3	10	16	18	22
---	----	----	----	----



6	4	16	10	16	14	2	8
---	---	----	----	----	----	---	---

Your left child gets your left sum.

Sum: 76
Left sum: 0

Range 0,8

Your right child has a left sum of:
Your left sum + its sibling's sum.

Sum: 36
Left Sum: 0

Range
0,4

Sum: 40
Left Sum: $0 + 36 = 36$

Range 4,8

Sum: 10
Left Sum: 0

Range
0,2

Sum: 26
Left Sum: 10

Range
2,4

Sum: 30
Left Sum: 36

Range
4,6

Sum: 10
Left Sum: 66

Range
6,8

S: 6
L: 0

S: 4
L: 6

S: 16
L: 10

S: 10
L: 26

S: 16
L: 36

S: 14
L: 52

S: 2
L: 66

S: 8
L: 68

6

4

16

10

16

14

2

8

Second Pass

Once we've finished calculating the sums, we'll start on the left sums.
Can we do that step in parallel?

YES!

Why are we doing two separate passes?
Those sum values have to be stored and ready.

Second pass has:

Work:

Span:

Second Pass

Once we've finished calculating the sums, we'll start on the left sums.
Can we do that step in parallel?

YES!

Why are we doing two separate passes?
Those sum values have to be stored and ready.

Second pass has:

Work: $O(n)$

Span: $O(\log n)$

Third Pass

What's our final answer?

Our sequential code said element i of the new array should be

`arr[i] + output[i-1]`

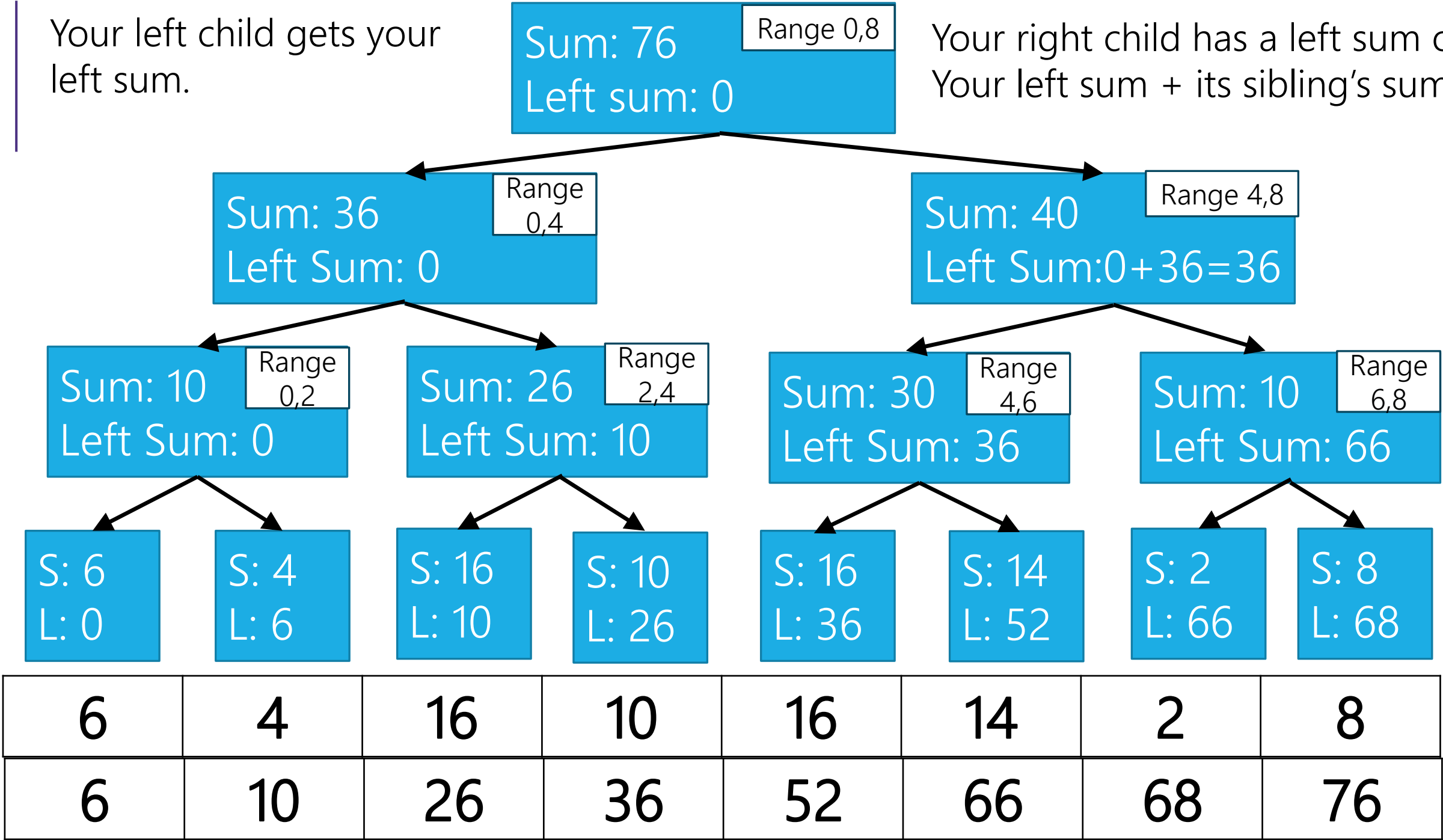
Or equivalently

`arr[i] + left_sum[i]`

Just need one more map using the data structure.

Your left child gets your left sum.

Your right child has a left sum of:
Your left sum + its sibling's sum.



Analyzing Parallel Prefix

What's the

Work?

Span?

First pass was a slightly modified version of our sum reduce code.

Second pass had a similar structure

Third pass was a map

Analyzing Parallel Prefix

What's the

Work $O(n)$

Span $O(\log n)$

First pass was a slightly modified version of our sum reduce code.

Second pass had a similar structure.

Third pass was a map.

Our Patterns So Far

1. Map

- Apply a function to every element of an array

2. Reduce

- Create a single object to summarize an array (e.g., sum of all elements)

3. Prefix

- Compute $\text{answer}[i] = f(\text{arr}[i], \text{answer}[i-1])$

Parallel Pack (aka Filter)

You want to find all the elements in an array meeting some property.
And move ONLY those into a new array.

Input:

6	4	16	10	16	14	2	8
---	---	----	----	----	----	---	---

Want every element ≥ 10

Output:

16	10	16	14
----	----	----	----

Parallel Pack

Easy – do a map to find the right elements...

Hard – How do you copy them over?

Parallel Pack

Easy – do a map to find the right elements...

Hard – How do you copy them over?

I need to know what array location to store in,

i.e. how many elements to my left will go in the new array.

Parallel Pack

Easy – do a map to find the right elements...

Hard – How do you copy them over?

I need to know what array location to store in,

i.e. how many elements to my left will go in the new array.

–Use Parallel Prefix!

Parallel Pack

Step 1: Parallel Map – produce bit vector of elements meeting property

6	4	16	10	16	2	14	8
0	0	1	1	1	0	1	0

Step 2: Parallel prefix sum on the bit vector

0	0	1	2	3	3	4	4
---	---	---	---	---	---	---	---

Step 3: Parallel map for output.

16	10	16	14
----	----	----	----

Step 3

How do we do step 3?

i.e. what's the map?

```
if (bits[i] == 1)
    output[ bitsum[i] - 1] = input[i];
```

Parallel Pack

We did 3 phases:

A map

A prefix

And another map.

Work:

Span:

Remark: You could fit this into 2 phases instead of 3. Won't change $O()$.

Parallel Pack

We did 3 phases:

A map

A prefix

And another map.

Work: $O(n)$

Span: $O(\log n)$

Remark: You could fit this into 2 phases instead of 3. Won't change $O()$.

Four Patterns

We've now seen four common patterns in parallel code

1. Map
2. Reduce
3. Prefix
4. Pack (a.k.a. Filter)

Making other code faster

Sometimes making parallel algorithms is just “can I turn my existing code into maps/reduces/prefixes/packs.

Other times parallel code with optimal span often requires changing to a different algorithm that parallelizes better.

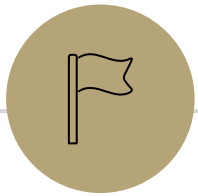
- These strategies often increase the work (slightly).

Two more optional examples: merge sort and quicksort, in parallel.

Details of the algorithms might change

- E.g., merge step in mergesort altered to run quicker in parallel.

Not responsible for them, but if you’re curious, see lecture 21 slides (or the Grossman text).



Amdahl's Law

Amdahl's Law

Now it's time for some bad news.

In practice, your program won't **just** sum all the elements in an array.

You will have a program with

Some parts that parallelize well

- Can turn them into a map or a reduce.

Some parts that won't parallelize at all

- Operations on a linked list. (**data structures matter!!!**)
- Reading a text file.
- A computation where each step needs the result of the previous steps.

Amdahl's Law

Let the work be 1 unit of time.

Let S be the portion of the code that is unparallelizable ("sequential").

$$T_1 = S + (1 - S) = 1.$$

At best we can get perfect linear speedup on the parallel portion

$$T_P \geq S + \frac{1-S}{P}$$

So overall speedup with P processors

$$\frac{T_1}{T_P} \leq \frac{1}{S + (1-S)/P}$$

$$\text{Therefore Parallelism: } \frac{T_1}{T_\infty} \leq \frac{1}{S}$$

Amdahl's Law

Suppose our program takes 100 seconds.
And S is 1/3 (i.e. 33 seconds).

What is the running time with

3 processors

6 processors

22 processors

67 processors

1,000,000 processors (approximately).

Amdahl's Law

$$\frac{T_1}{T_P} \leq \frac{1}{S + \frac{1-S}{P}}$$

Amdahl's Law

Suppose our program takes 100 seconds.
And S is 1/3 (i.e. 33 seconds).

What is the running time with

3 processors: $33 + 67/3 \approx 55$ seconds

6 processors: $33 + 67/6 \approx 44$ seconds

22 processors: $33 + 67/22 \approx 36$ seconds

67 processors $33 + 67/67 \approx 34$ seconds

1,000,000 processors (approximately). ≈ 33 seconds

Amdahl's Law

$$\frac{T_1}{T_P} \leq \frac{1}{S + \frac{1-S}{P}}$$

Amdahl's Law

This is BAD NEWS

If $\frac{1}{3}$ of our program can't be parallelized, we can't get a speedup better than 3.

No matter how many processors we throw at our problem.

And while the first few processors make a huge difference, the benefit diminishes quickly.

Amdahl's Law and Moore's Law

In the Moore's Law days, 12 years was long enough to get 100x speedup.

Suppose in 12 years, the clock speed is the same, but you have 256 processors.

What portion of your program can you hope to leave unparallelized?

$$100 \leq \frac{1}{S + \frac{1-S}{256}}$$

[wolframalpha says] $S \leq 0.0061$.

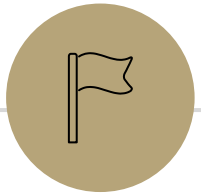
Amdahl's Law and Moore's Law

Moore's Law was "a business decision"

- How much effort/money/employees are dedicated to improving processors so computers got faster.

Amdahl's Law is a theorem

- You can prove it formally.



Concurrency

Sharing Resources

So far we've been writing parallel algorithms that don't share resources.

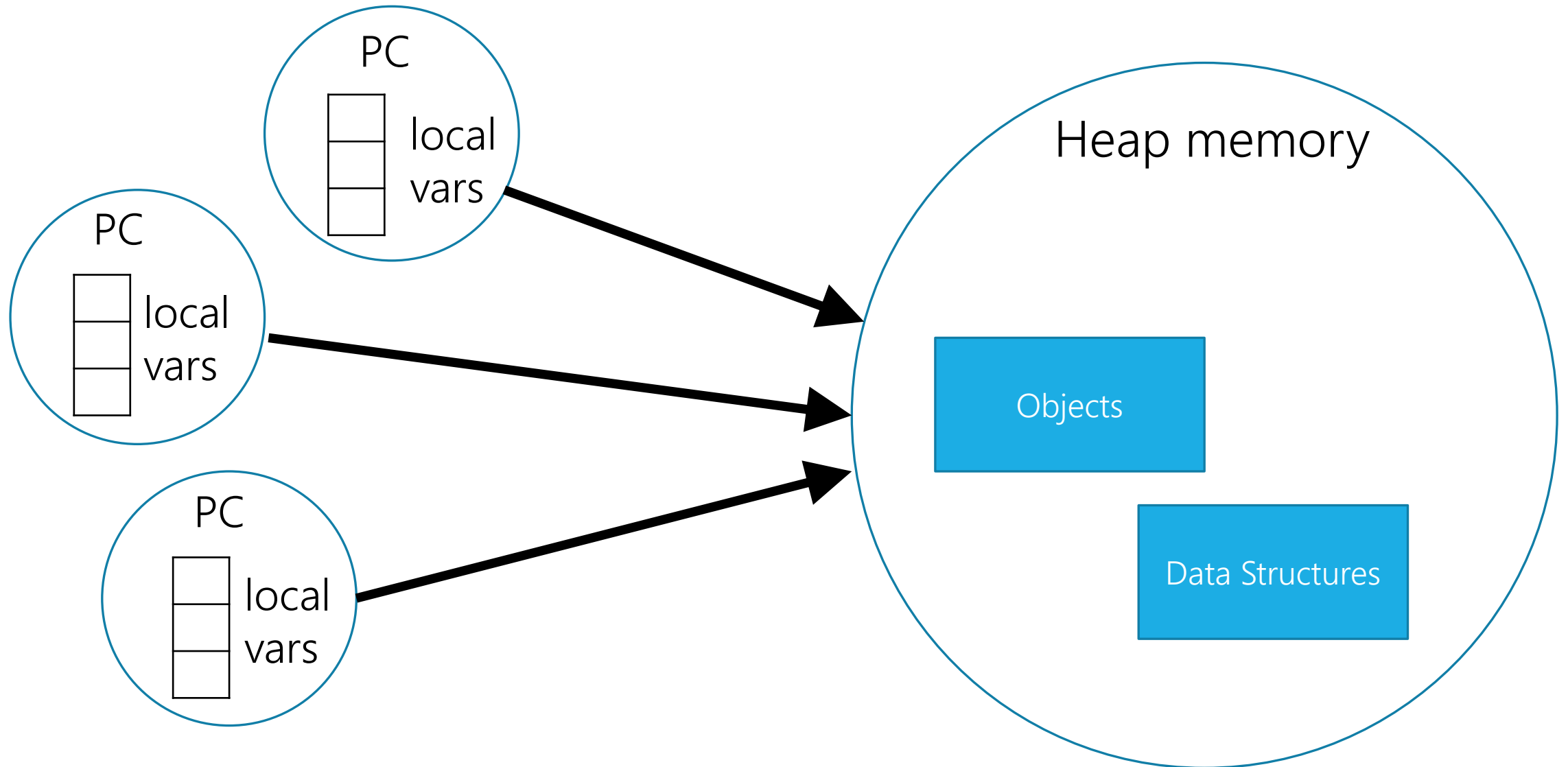
Fork-join algorithms all had a simple structure

- Each thread had memory only it accesses.
- Results of one thread not accessed until joined.
- The **structure** of the code ensured sharing didn't go wrong.

Can't use the same strategy when memory overlaps

Thread doing independent tasks on same resources.

Parallel Code



Why Concurrency?

If we're not using them to solve the same big problem, why threads?

Code responsiveness

- One thread responds to GUI, another does big computations

Processor utilization

- If a thread needs to go to disk, can throw another thread on while it waits.

Failure isolation

- Don't want one exception to crash the whole program.

Concurrency

Different threads might access the same resources
In unpredictable orders or even simultaneously

Simultaneous access is rare

- Makes testing very difficult
- Instead, we'll be disciplined when writing the code.

In this class, we'll focus on code idioms that are known to work.

Only some discussion of Java specifics – there are more details in the Grossman notes.

Sharing a Queue

Two threads both want to insert into a queue.

Each has its own program counter, they can each be running different parts of the code simultaneously.

They can arbitrarily “interrupt” each other.

What can go wrong?

Bad Interleaving

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

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

Bad Interleaving

```
Enqueue (x) {
```

```
1 if (back==null) {
```

```
3   back=new Node (x) ;
```

```
5   front=back;
```

```
}
```

```
else{
```

```
    back.next=new Node (x) ;
```

```
    back=back.next;
```

```
}
```

```
Enqueue (x) {
```

```
    if (back==null) {
```

```
        back=new Node (x) ;
```

```
        front=back;
```

```
}
```

```
else{
```

```
    back.next=new Node (x) ;
```

```
    back=back.next;
```

```
}
```

Bad Interleaving

```
if (back==null) {
```

```
    back=new Node(10);
```

```
    front=back;
```

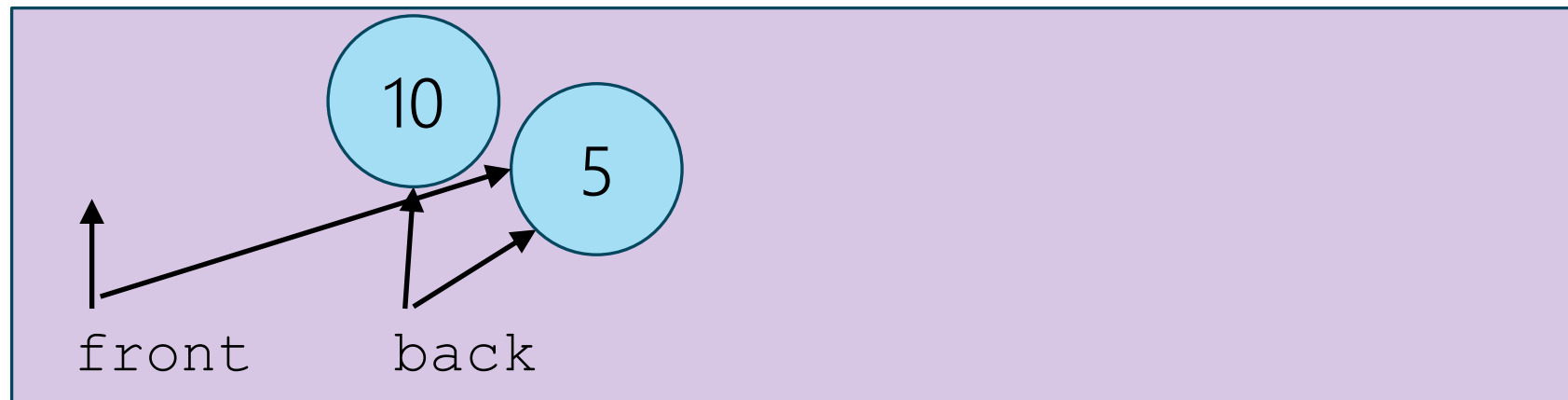
```
}
```

```
if (back==null) {
```

```
    back=new Node(5);
```

```
    front=back;
```

```
}
```



One Example

```
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);
    }
    ...
}
```


Bad Interleavings

Suppose the account has `balance` of 150.

Two threads run: one withdrawing 100, another withdrawing 75.

Find a bad interleaving – what can go wrong?

Bad Interleaving

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

Bad Interleaving

<code>void withdraw(int amount) {</code>	<code>void withdraw(int amount) {</code>
1 <code>int b = getBalance();</code>	<code>int b = getBalance();</code> 3
2 <code>if (amount > b)</code>	<code>if (amount > b)</code> 4
<code>throw new ...;</code>	<code>throw new ...;</code>
6 <code>setBalance(b-amount);</code>	<code>setBalance(b-amount);</code> 5
<code>}</code>	<code>}</code>

Bad Interleavings

What's the problem?

We stored the result of `balance` locally, but another thread overwrote it after we stored it.

The value became stale.

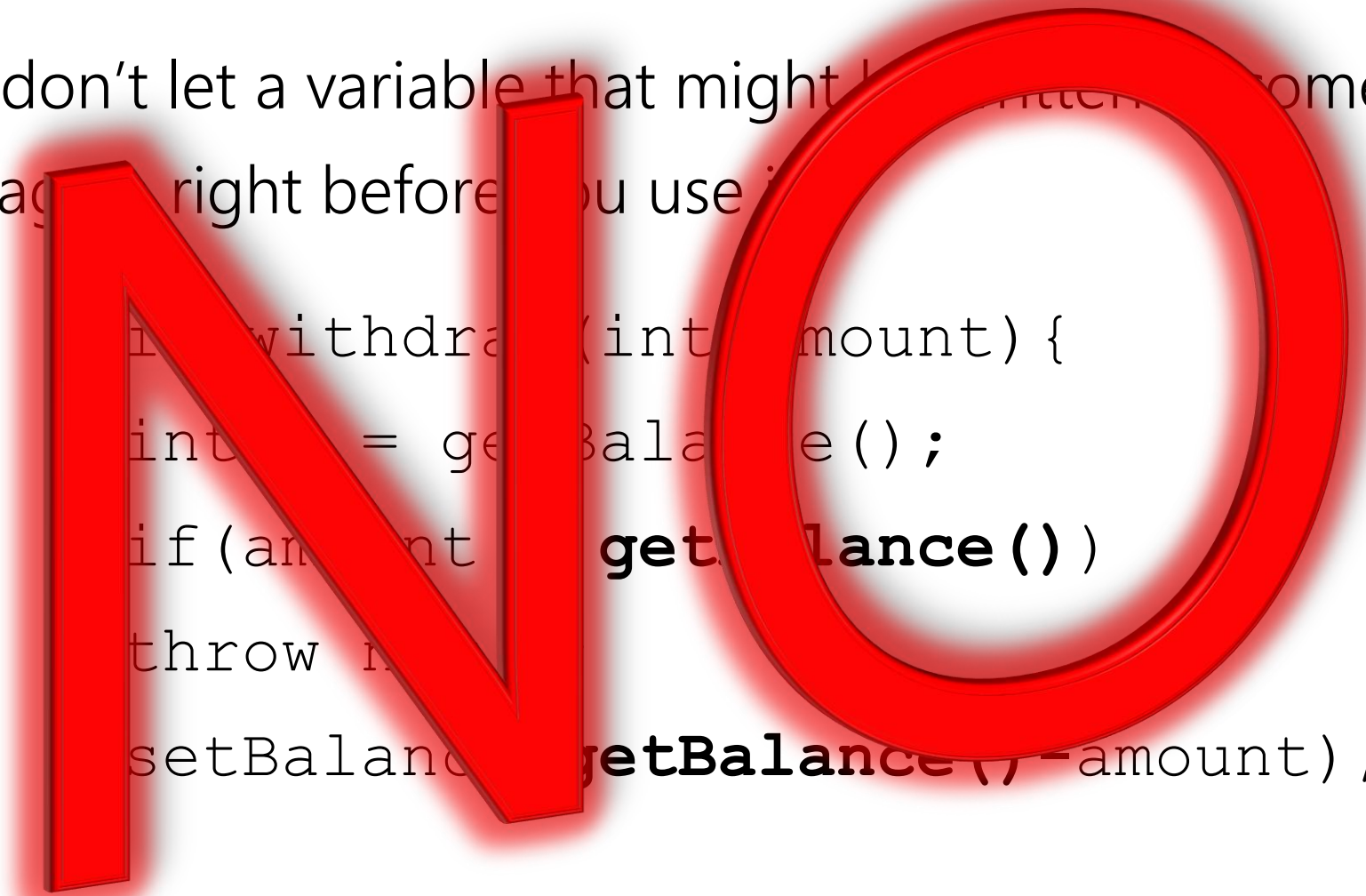
A Principle

Principle: don't let a variable that might be written become stale.
Ask for it again right before you use it

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

A Principle

Principle: don't let a variable that might be updated become stable.
Ask for it again right before you use it.



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

That's not a real concurrency principle. It doesn't solve anything.

Bad Interleaving

There's still a bad interleaving. Find one.

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

Bad Interleaving

There's still a bad interleaving. Find one.

<pre>void withdraw(int amount) { 1 int b = getBalance(); 2 if (amount > getBalance()) throw new ...; 7 setBalance(getBalance() - amount); 5 }</pre>	<pre>void withdraw(int amount) { int b = getBalance(); 3 if (amount > getBalance()) 4 throw new ...; setBalance(8 getBalance() - amount); 6 }</pre>
---	---

Bad Interleaving

There's still a bad interleaving. Find one.

<pre>void withdraw(int amount) { 1 int b = getBalance(); 2 if (amount > getBalance()) throw new ...; 6 setBalance(getBalance() - amount); 5 }</pre>	<pre>void withdraw(int amount) { int b = getBalance(); 3 if (amount > getBalance()) 4 throw new ...; setBalance(8 getBalance() - amount); 7 }</pre>
---	---

In this version, we can have negative balances without throwing the exception!

A Real Principle

Mutual Exclusion (aka Mutex, aka Locks)

Rewrite our methods so only one thread can use a resource at a time

- All other threads must wait.

We need to identify the critical section

- Portion of the code only a single thread can execute at once.

This MUST be done by the programmer.

BankAccount v.2

```
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;
    }
```

Does this code work?

```
...
}
```

Locks

We can still have a bad interleaving.

If two threads see `busy = false` and get past the loop simultaneously.

We need a single operation that

- Checks if `busy` is false
- AND sets it to true if it is
- Where no other thread can interrupt us.

An operation is **atomic** if no other threads can interrupt it/interleave with it.

Locks

There's no regular java command to do that.

We need a new library

`Lock` (not the real Java class, but will let us understand the principles)

`acquire()` – blocks if lock is unavailable. When lock becomes available, one thread only gets lock.

`release()` – allow another thread to acquire lock.

Need OS level support to implement.

Take an operating systems course to learn more.

Locks

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

Using Locks

Questions:

What is the critical section (i.e., the part of the code protected by the lock)?

How many locks should we have

- One per `BankAccount` object?
- Two per `BankAccount` object (one in `withdraw` and a different lock in `deposit`)?
- One (static) one for the entire class (shared by all `BankAccount` objects)?

Using Locks

More Questions:

There is a subtle bug in `withdraw()`, what is it?

Do we need locks for

- `getBalance()`?
- `setBalance()`?
- For the purposes of this question, assume those methods are public.

Using Locks

How many locks?

Different locks for withdraw and deposit will lead to bad interleavings.

- The shared resource is `balance` not the methods themselves.

One `lock` for the whole class isn't wrong...but it is a bad design decision.

Only one thread anywhere can do any withdraw/deposit operation; No matter how many bank accounts there are.

There's a tradeoff in how granular you make critical sections:

- Bigger: easier to rule out errors, but fewer threads can work at once.

Using Locks

Bug in `withdraw`:

- When you throw an exception, you still hold onto the lock!

You could release the `lock` before throwing the exception.

Or use `try{} finally{} blocks`

```
try{ critical section }  
finally{ lk.release() }
```

Re-entrant Locks

Do we need to lock `setBalance()` ?

If it's public, yes.

But now we have a problem:

`withdraw` will acquire the `lock`,

Then call `setBalance...`

Which needs the same `lock`

Re-entrant Locks

Our locks need to be **re-entrant**.

That is, the `lock` isn't held by a single method call

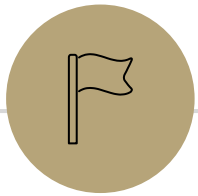
But rather by a thread.

- Execution can **re-enter** another critical section, while holding the same lock.

Lock needs to know which `release` call is the “real” release, and which one is just the end of an inner method call.

Intuition: have a counter. Increment it when you “re-acquire” the lock, decrement when you release. Until releasing on 0 then really release.

Take an operating systems course to learn more.



Some Java Notes

Real Java locks

A re-entrant lock is available in:

```
java.util.concurrent.locks.ReentrantLock
```

Methods are `lock()` and `unlock()`

synchronized

Java has built-in support for reentrant locks with the keyword `synchronized`

```
synchronized (expression) {  
    Critical section  
}
```

- Expression must evaluate to an object.
 - Every object "is a lock" in java
 - Lock is acquired at the opening brace and released at the matching closing brace.
 - Released even if control leaves due to throw/return/etc.

synchronized

If your whole method is a critical section

And the object you want for your lock is `this`

You can change the method header to include `synchronized`.

E.g. `private synchronized void getBalance()`

Equivalent of having

`synchronized(this) { }` around entire method body.