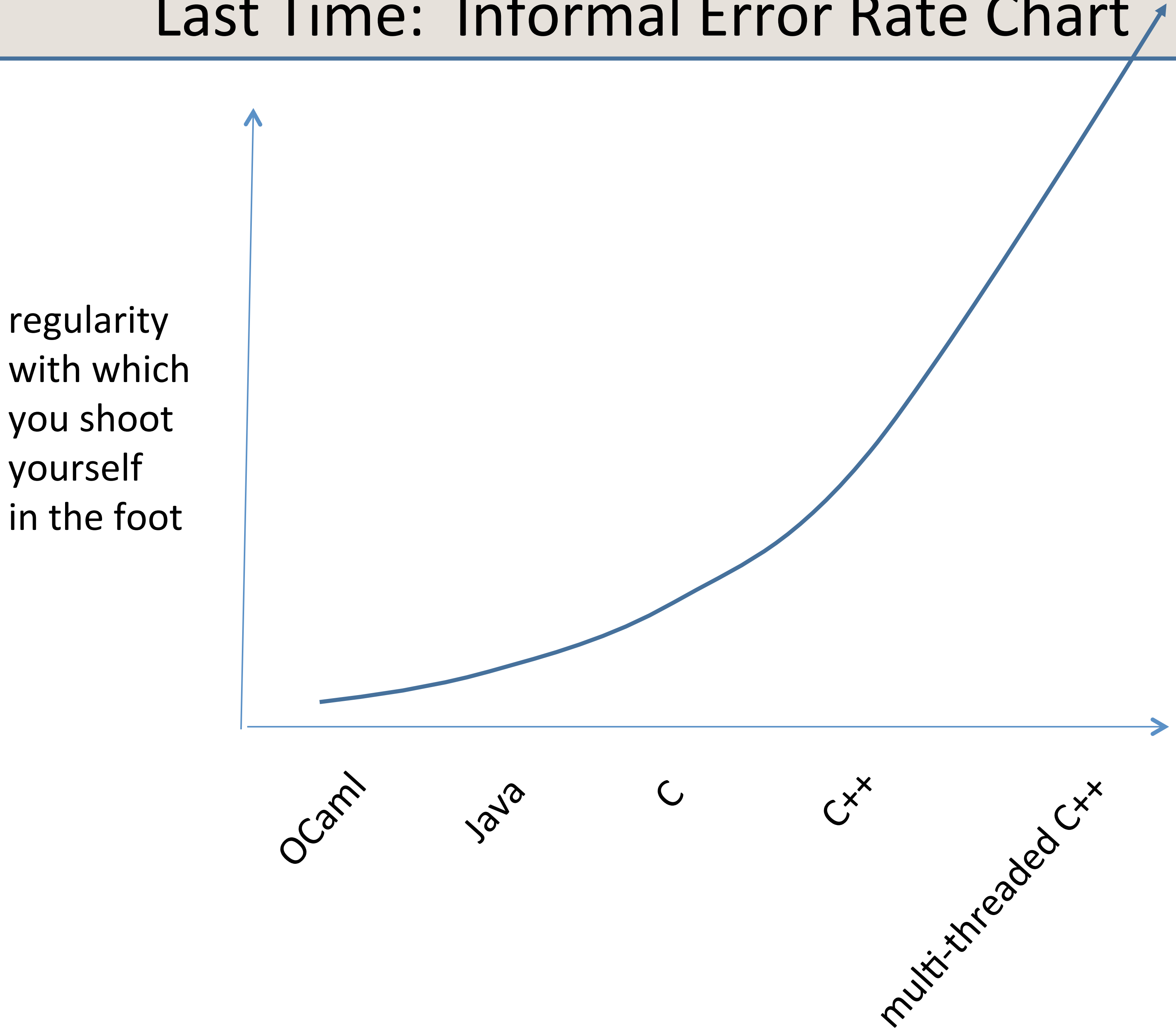
Threads, Futures
and some nastier things too

COS 326

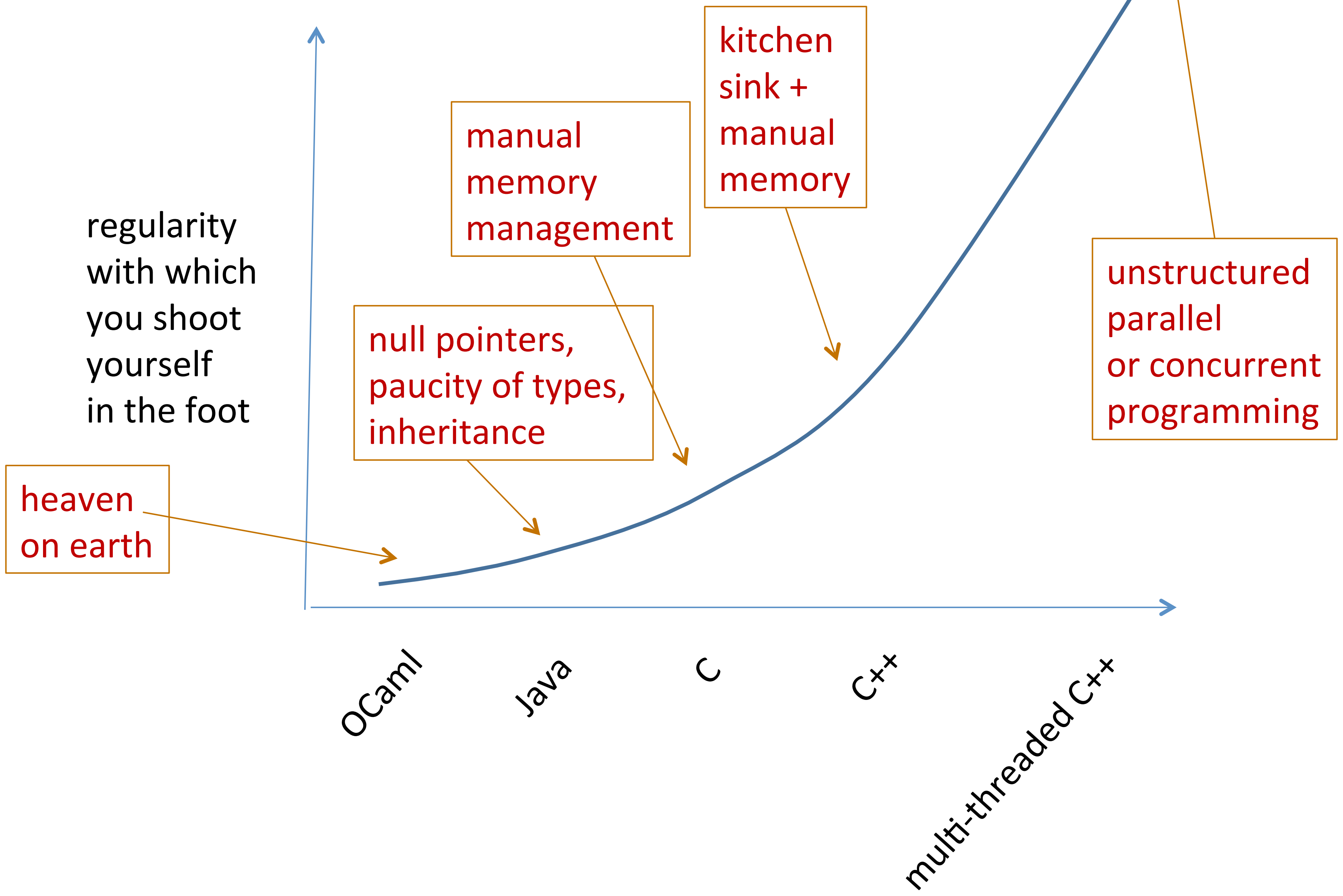
David Walker

Princeton University

Last Time: Informal Error Rate Chart



Last Time: Informal Error Rate Chart



Threads: A Warning

- *Concurrent Threads with Locks: the classic shoot-yourself-in-the-foot concurrent programming model*
  - all the classic error modes

- Why Threads?
  - almost all programming languages will have a threads library
    - OCaml in particular!
  - you need to know where the pitfalls are
  - the assembly language of concurrent programming paradigms
    - we'll use threads to build several higher-level programming models

Threads

- Threads: an abstraction of a processor.
  - programmer (or compiler) decides that some work can be done in parallel with some other work, e.g.:

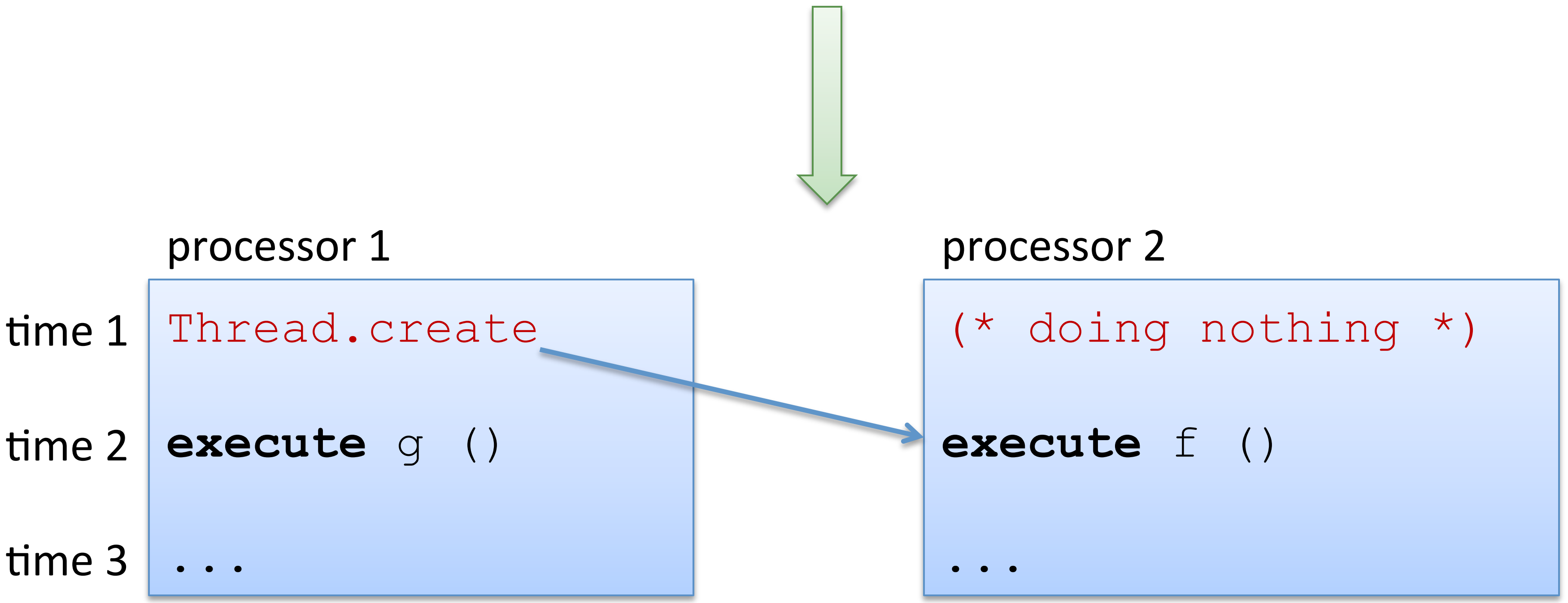
```
let _ = compute_big_thing() in
let y = compute_other_big_thing() in
...
```

  - we *fork* a thread to run the computation in parallel, e.g.:

```
let t = Thread.create compute_big_thing () in
let y = compute_other_big_thing () in
 ...
```

Intuition in Pictures

```
let t = Thread.create f () in
let y = g () in
 ...
```



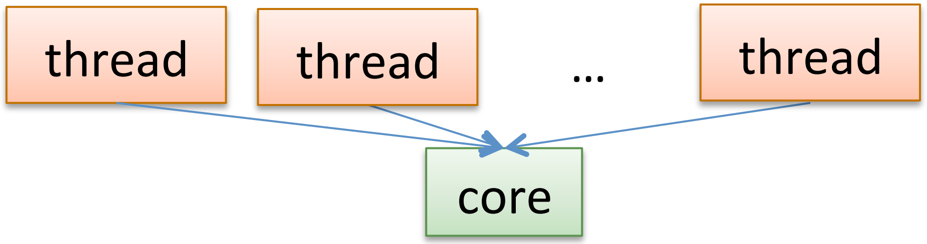
Of Course…

Suppose you have 2 available cores and you fork 4 threads. In a typical multi-threaded system,

- the operating system provides *the illusion* that there are an infinite number of processors.
  - not really: each thread consumes space, so if you fork too many threads the process will die.
- it *time-multiplexes* the threads across the available processors.
  - about every 10 msec, it stops the current thread on a processor, and switches to another thread.
  - so a thread is really a *virtual processor*.

OCaml, Concurrency and Parallelism

Unfortunately, even if your computer has 2, 4, 6, 8 cores, OCaml cannot exploit them. It multiplexes all threads over a single core



Hence, OCaml provides concurrency, but not parallelism. *Why?* Because OCaml (like Python) has no parallel "runtime system" or garbage collector. Other functional languages (Haskell, F#, ...) do.

Fortunately, when thinking about *program correctness*, it doesn't matter that OCaml is not parallel -- I will often pretend that it is.

You can hide I/O latency, do multiprocess programming or distribute tasks amongst multiple computers in OCaml.

Coordination

```
Thread.create : ('a -> 'b) -> 'a -> Thread.t

let t = Thread.create f () in
let y = g () in
 ...
```

How do we get back the result that `t` is computing?

First Attempt

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
let y = g() in
  match !r with
    | Some v -> (* compute with v and y *)
    | None -> failwith "impossible"
```

What’s wrong with this?

Second Attempt

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
let y = g() in
let rec wait() =
  match !r with
    | Some v -> v
    | None -> wait()
in
let v = wait() in
  (* compute with v and y *)
```

Two Problems

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
let y = g() in
let rec wait() =
  match !r with
    | Some v -> v
    | None -> wait()
in
let v = wait() in
  (* compute with v and y *)
```

First, we are *busy-waiting*.

- consuming CPU without doing something useful.
- CPU could either be running a useful thread/program or power down.

Two Problems

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
let y = g() in
let rec wait() =
  match !r with
    | Some v -> v
    | None -> wait()
in
let v = wait() in
  (* compute with v and y *)
```

Second, an operation like r := Some v may not be *atomic.*

- r := Some v requires us to copy the bytes of Some v into the ref r
- we might see part of the bytes (corresponding to Some) before we've written in the other parts (e.g., v).
- So the waiter might see the wrong value.

Atomicity

Consider the following:

```
let inc(r:int ref) = r := (!r) + 1
```

and suppose two threads are incrementing the same ref r:

Thread 1	Thread 2
`inc(r);`	`inc(r);`
`!r`	`!r`

If r initially holds 0, then what will Thread 1 see when it reads r?

Atomicity

The problem is that we can’t see exactly what instructions the compiler might produce to execute the code.

It might look like this:

<u>Thread 1</u>

```
EAX := load(r);
EAX := EAX + 1;
store EAX into r
EAX := load(r)
```

<u>Thread 2</u>

```
EAX := load(r);
EAX := EAX + 1;
store EAX into r
EAX := load(r)
```

Atomicity

But a clever compiler might optimize this to:

Thread 1

```
EAX := load(r);
EAX := EAX + 1;
store EAX into r
~~EAX := load(r)~~
```

Thread 2

```
EAX := load(r);
EAX := EAX + 1;
store EAX into r
~~EAX := load(r)~~
```

Atomicity

Furthermore, we don't know when the OS might interrupt one thread and run the other.

```
Thread 1                  Thread 2
EAX := load(r);           EAX := load(r);
EAX := EAX + 1;           EAX := EAX + 1;
store EAX into r          store EAX into r
EAX := load(r)            EAX := load(r)
```

(The situation is similar, but not quite the same on multi-processor systems.)

The Happens Before Relation

We don't know exactly when each instruction will execute, but there are some constraints: the *Happens Before* relation

Rule 1: Given two expressions (or instructions) in sequence, e1; e2 we know that e1 happens before e2.

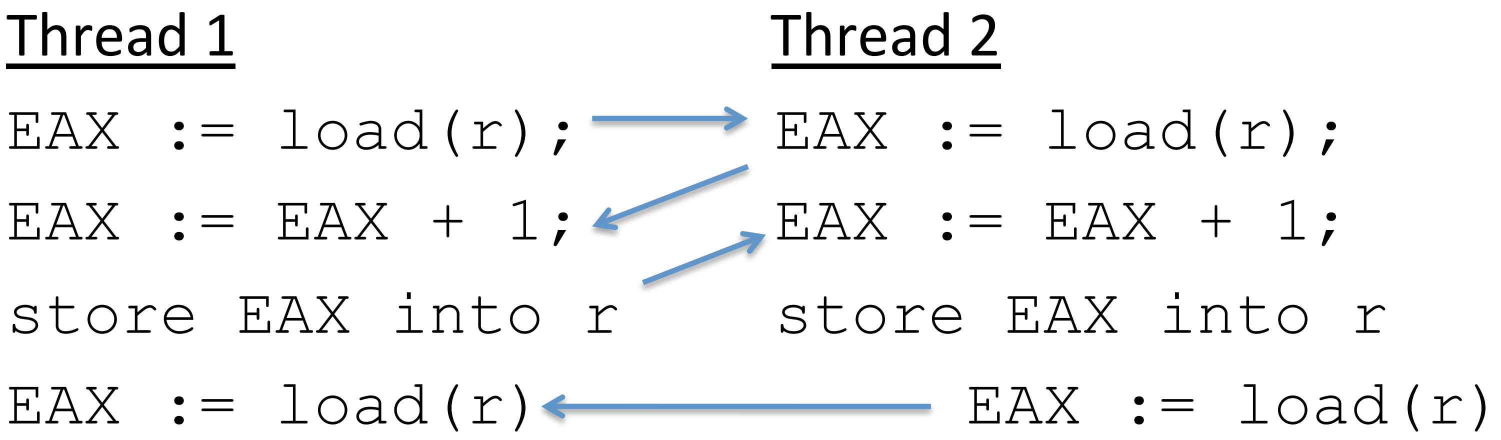
Rule 2: Given a program:

```
let t = Thread.create f x in
....
Thread.join t;
e
```

we know that (f x) happens before e.

Atomicity

One possible interleaving of the instructions:



```
Thread 1                  Thread 2
EAX := load(r);           EAX := load(r);
EAX := EAX + 1;           EAX := EAX + 1;
store EAX into r          store EAX into r
EAX := load(r)                  EAX := load(r)
```

What answer do we get if r points to 0 to start?

Atomicity

Another possible interleaving:

```
Thread 1                  Thread 2
EAX := load(r);           EAX := load(r);
EAX := EAX + 1;           EAX := EAX + 1;
store EAX into r          store EAX into r
EAX := load(r)                  EAX := load(r)
```

What answer do we get this time?

Atomicity

Another possible interleaving:

<u>Thread 1</u>   <u>Thread 2</u>

```
EAX := load(r);          EAX := load(r);
EAX := EAX + 1;          EAX := EAX + 1;
store EAX into r         store EAX into r
EAX := load(r)             EAX := load(r)
```

What answer do we get this time?

Moral: The system is responsible for *scheduling* execution of instructions.

Moral: This can lead to an enormous degree of *nondeterminism*.

Atomicity

In fact, today's multicore processors don't treat memory in a *sequentially consistent* fashion.

<u>Thread 1</u>

```
EAX := load(r);
EAX := EAX + 1;
store EAX into r
EAX := load(r)
```

<u>Thread 2</u>

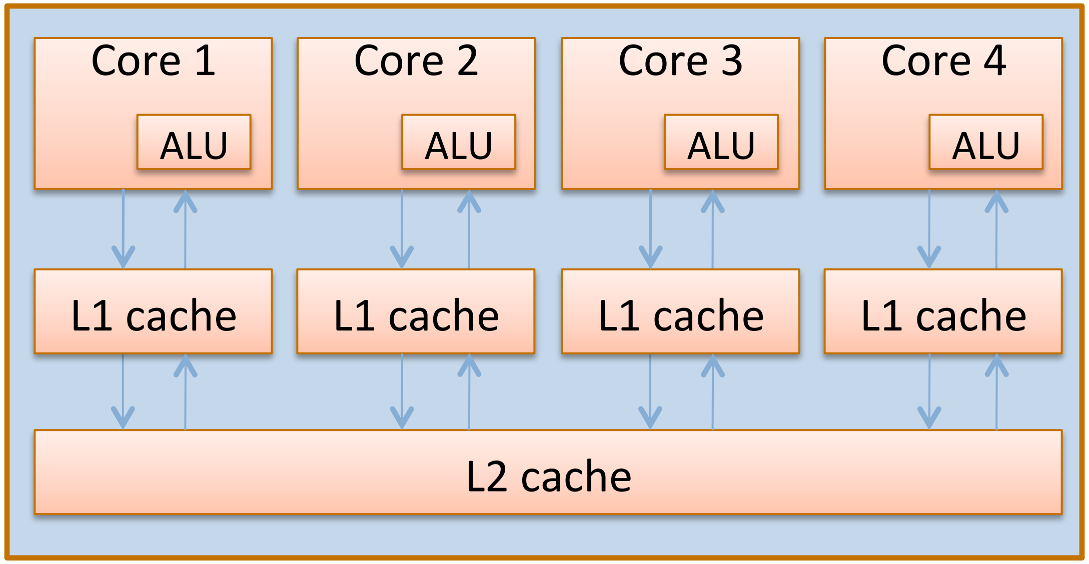
```
EAX := load(r);
EAX := EAX + 1;
store EAX into r
         EAX := load(r)
```

That means that *we can't even assume that what we will see corresponds to some interleaving of the threads' instructions!*

Beyond the scope of this class! But the take-away is this: It's not a good idea to use ordinary loads/stores to synchronize threads; you should use explicit synchronization primitives so the hardware and optimizing compiler don't optimize them away.

Atomicity

In fact, today's multicore processors don't treat memory in a *sequentially consistent* fashion. That means that *we can't even assume that what we will see corresponds to some interleaving of the threads' instructions!*



When Core1 stores to "memory", it *lazily* propagates to Core2's L1 cache. The load at Core2 might not see it, unless there is an explicit synchronization.

Beyond the scope of this class! But the take-away is this: It's not a good idea to use ordinary loads/stores to synchronize threads; you should use explicit synchronization primitives so the hardware and optimizing compiler don't optimize them away.

Summary: Interleaving & Race Conditions

Calculate possible outcomes for a program by considering all of the possible interleavings of the *atomic* actions performed by each thread.

- Subject to the *happens-before* relation.
  - can't have a child thread's actions happening before a parent forks it.
  - can't have later instructions execute earlier *in the same thread.*
- Here, *atomic* means indivisible actions.
  - For example, on most machines reading or writing a 32-bit word is atomic.
  - But, writing a multi-word object is usually *not* atomic.
  - Most operations like "b := b - w" are implemented in terms of a series of simpler operations such as
    - r1 = read(b); r2 = read(w); r3 = r1 – r2; write(b, r3)

Reasoning about all interleavings is ~~*hard*~~. just about impossible for people

- Number of interleavings grows exponentially with number of statements.
- It's hard for us to tell what is and isn't atomic in a high-level language.
- YOU ARE DOOMED TO FAIL IF YOU HAVE TO WORRY ABOUT THIS STUFF!

Summary: Interleaving & Race Conditions

Calculate possible outcomes for a program by considering all of the possible interleavings of the *atomic* actions performed by each thread.

- Subject to the *happen*
  - can't have a child th before a parent forks it.
  -
- He
  - omic.
  -
  - s of

WARNING

If you see people talk about interleavings, BEWARE!
It probably means they're assuming
"sequential consistency,"
which is an oversimplified, naïve model of what the
parallel computer really does.
It's actually more complicated than that.

Reasoning about all interleavings is ~~*hard*~~. just about impossible for people

- Number of interleavings grows exponentially with number of statements.
- It's hard for us to tell what is and isn't atomic in a high-level language.
- YOU ARE DOOMED TO FAIL IF YOU HAVE TO WORRY ABOUT THIS STUFF!

A conventional solution for shared-memory parallelism

```
let inc(r:int ref) = r := (!r) + 1
```

Thread 1	Thread 2
`lock(mutex);`	`lock(mutex);`
`inc(r);`	`inc(r);`
`!r`	`!r`
`unlock(mutex);`	`unlock(mutex);`

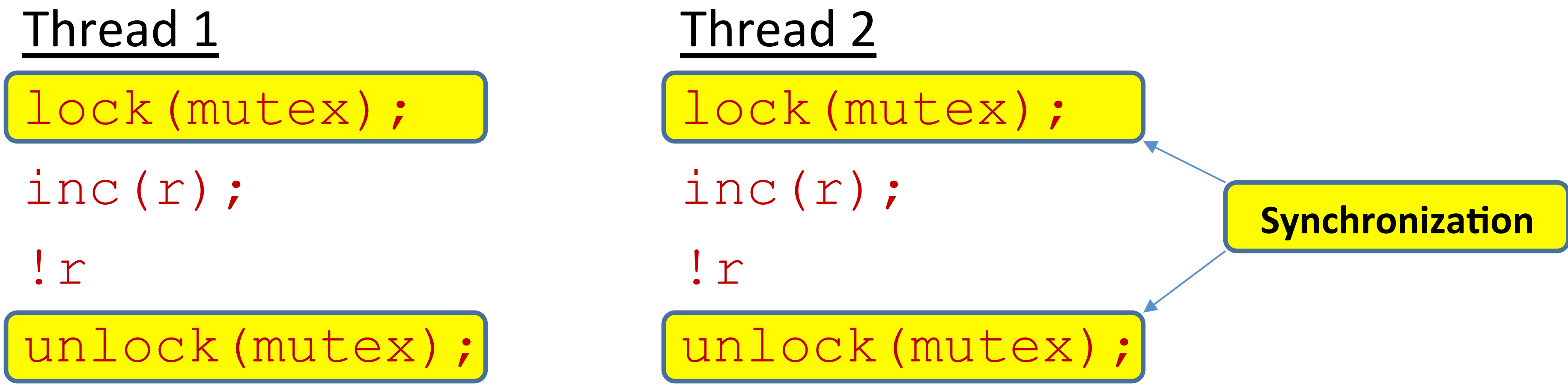
Guarantees *mutual exclusion* of these critical sections.

This solution works (even for real machines that are not sequentially consistent), **but…**

Complex to program, subject to *deadlock*, prone to bugs, not fault-tolerant, hard to reason about.

A conventional solution for shared-memory parallelism

```
let inc(r:int ref) = r := (!r) + 1
```



```
Thread 1                 Thread 2
lock(mutex);             lock(mutex);
inc(r);                  inc(r);
!r                       !r
unlock(mutex);           unlock(mutex);
```

Guarantees *mutual exclusion* of these critical sections.

This solution works (even for real machines that are not sequentially consistent), **but…**

Complex to program, subject to *deadlock*, prone to bugs, not fault-tolerant, hard to reason about.

Another approach to the coordination Problem

```
Thread.create : ('a -> 'b) -> 'a -> Thread.t

let t = Thread.create f () in
let y = g () in
 ...
```

How do we get back the result that `t` is computing?

One Solution (using join)

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
 let y = g() in
    Thread.join t ;
    match !r with
    | Some v -> (* compute with v and y *)
    | None -> failwith “impossible”
```

One Solution (using join)

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
 let y = g() in
   Thread.join t ;
   match !r with
   | Some v -> (* compute    ith v and y *)
   | None -> failwith "impos     e"
```

`Thread.join t` causes the current thread to *wait* until the thread `t` terminates.

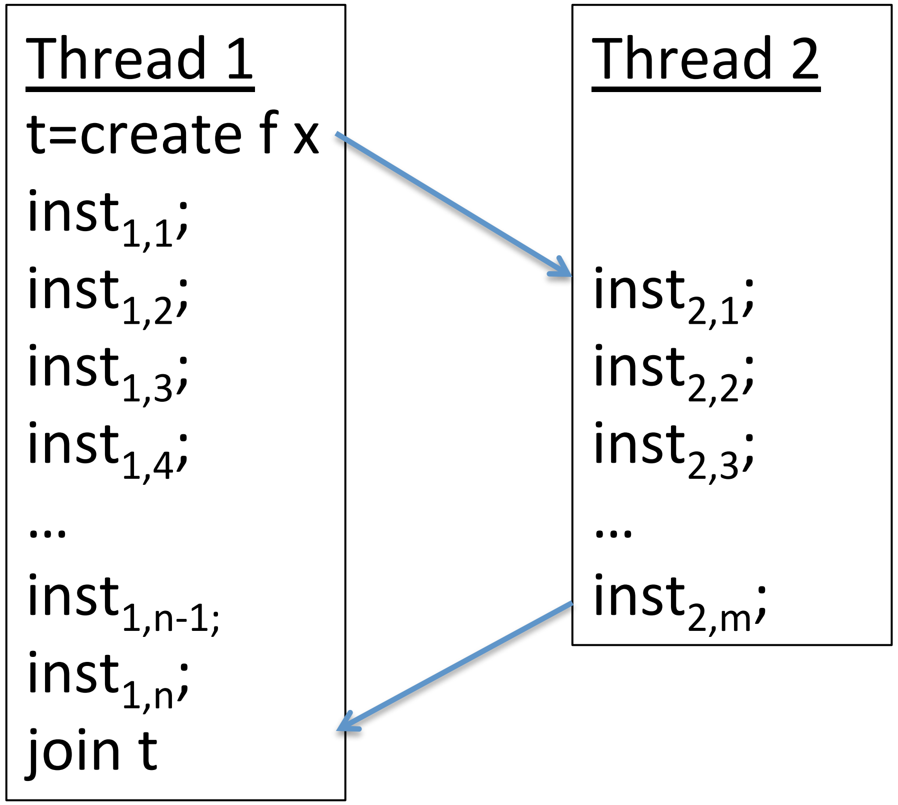
One Solution (using join)

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
 let y = g() in
    Thread.join t ;
    match !r with
    | So  e v -> (* compute with v and y *)
    |     e -> failwith "impossible"
```

Synchronization

So after the join, we know that any of the operations of t have *completed*.

In Pictures



<u>Thread 1</u>

t=create f x

$inst_{1,1}$;

$inst_{1,2}$;

$inst_{1,3}$;

$inst_{1,4}$;

…

$inst_{1,n-1;}$

$inst_{1,n}$;

join t

<u>Thread 2</u>

$inst_{2,1}$;

$inst_{2,2}$;

$inst_{2,3}$;

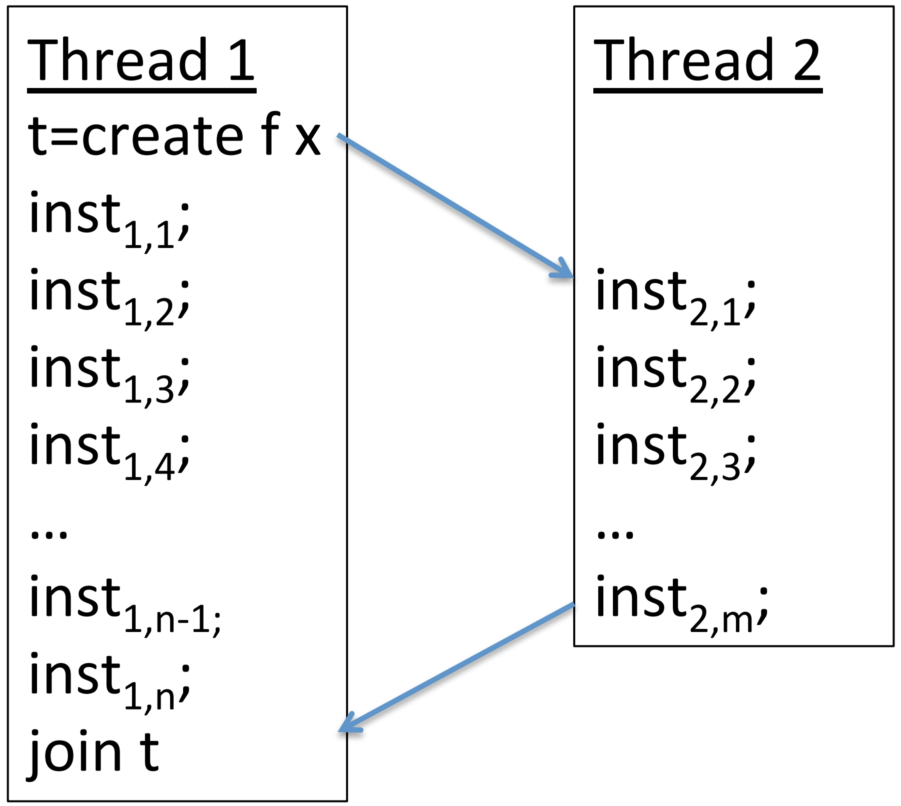
…

$inst_{2,m}$;

We know that for each thread the previous instructions must happen before the later instructions.

So for instance, $inst_{1,1}$ must happen before $inst_{1,2}$.

In Pictures



<u>Thread 1</u>

t=create f x

$\text{inst}_{1,1}$;

$\text{inst}_{1,2}$;

$\text{inst}_{1,3}$;

$\text{inst}_{1,4}$;

…

$\text{inst}_{1,n-1;}$

$\text{inst}_{1,n}$;

join t

<u>Thread 2</u>

$\text{inst}_{2,1}$;

$\text{inst}_{2,2}$;

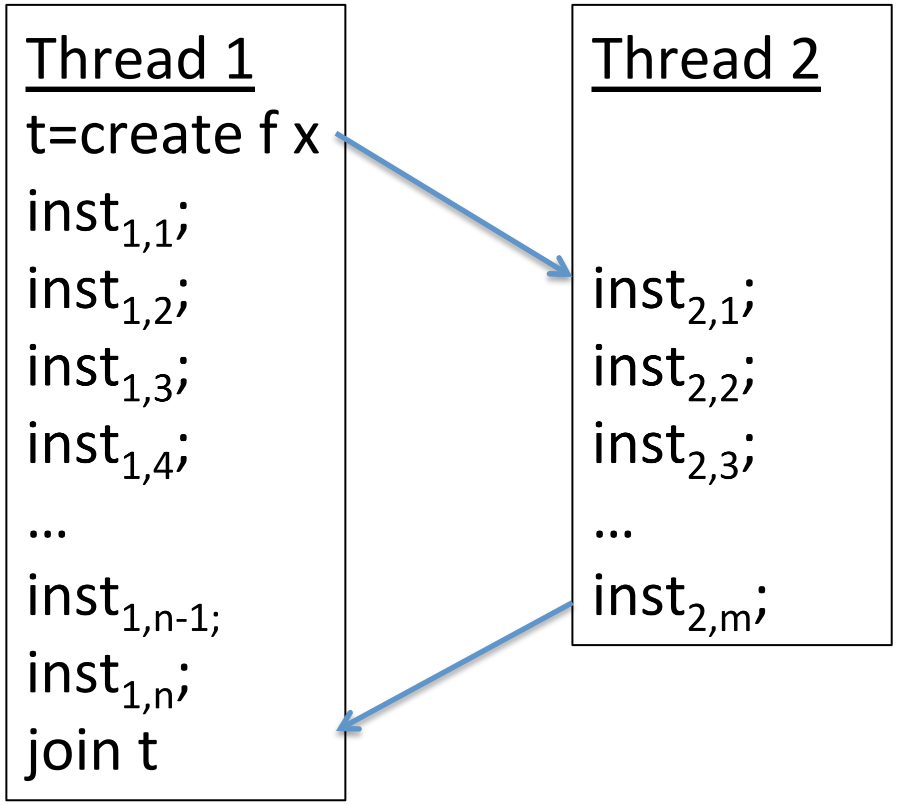
$\text{inst}_{2,3}$;

…

$\text{inst}_{2,m}$;

We also know that the fork must happen before the first instruction of the second thread.

In Pictures



Thread 1

t=create f x

$\text{inst}_{1,1}$;

$\text{inst}_{1,2}$;

$\text{inst}_{1,3}$;

$\text{inst}_{1,4}$;

…

$\text{inst}_{1,n-1;}$

$\text{inst}_{1,n}$;

join t

Thread 2

$\text{inst}_{2,1}$;

$\text{inst}_{2,2}$;

$\text{inst}_{2,3}$;

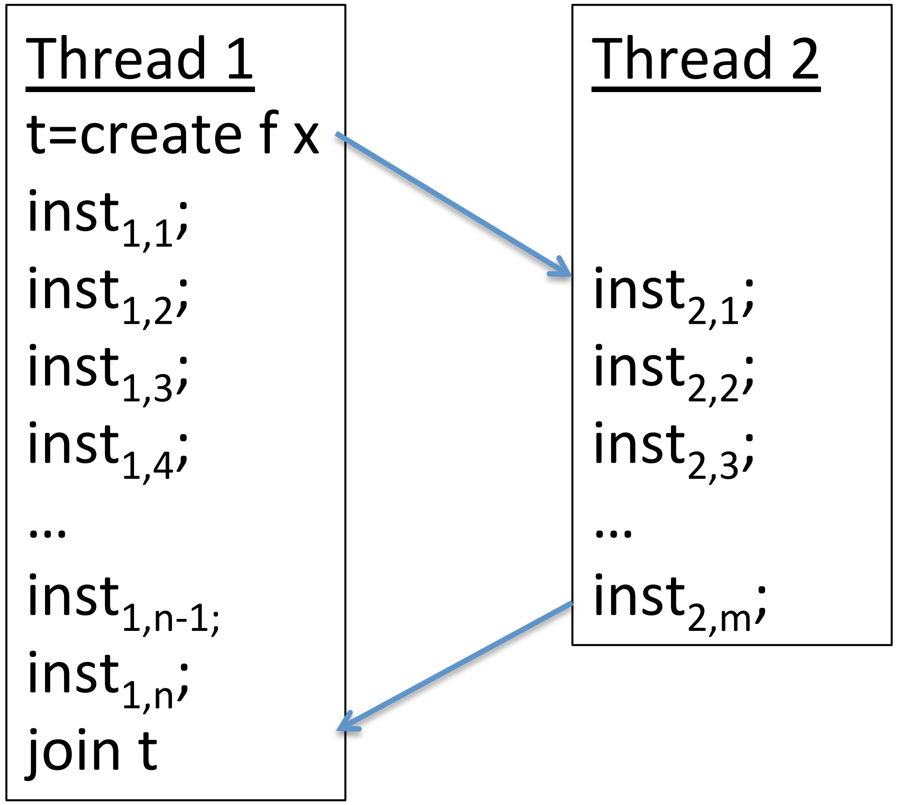
…

$\text{inst}_{2,m}$;

We also know that the fork must happen before the first instruction of the second thread.

And thanks to the join, we know that all of the instructions of the second thread must be completed before the join finishes.

In Pictures



However, in general, we do not know whether $inst_{1,i}$ executes before or after $inst_{2,j}$.

In general, *synchronization instructions like fork and join reduce the number of possible interleavings.*

Synchronization cuts down nondeterminism.

In the absence of synchronization we don't know anything…

Fork-join synchronization

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
 let y = g() in
    Thread.join t ;
    match !r with
    | Some v -> (* compute with v and y *)
    | None -> failwith “impossible”
```

Synchronization

So after the join, we know that any of the operations of t have *completed*.

FUTURES: A PARALLEL PROGRAMMING ABSTRACTION

Futures

The fork-join pattern we just saw is so common, we'll create an abstraction for it:

```
module type FUTURE =
sig
  type 'a future

   (* future f x forks a thread to run f(x)
      and stores the result in a future when complete *)
  val future : ('a->'b) -> 'a -> 'b future

   (* force f causes us to wait until the
      thread computing the future value is done
      and then returns its value. *)
  val force : 'a future -> 'a
end
```

Does that interface looks familiar .... ?

Future Implementation

```
module Future : FUTURE =
struct
  type 'a future = {tid   : Thread.t      ;
                    value : 'a option ref }










end
```

Future Implementation

```
module Future : FUTURE =
struct
  type 'a future = {tid   : Thread.t       ;
                    value : 'a option ref }

  let future(f:'a->'b)(x:'a) : 'b future =
    let r = ref None in
    let t = Thread.create (fun () -> r := Some(f x)) ()
    in
    {tid=t ; value=r}






end
```

Future Implementation

```
module Future : FUTURE =
struct
  type 'a future = {tid   : Thread.t       ;
                    value : 'a option ref }

  let future(f:'a->'b)(x:'a) : 'b future =
    let r = ref None in
    let t = Thread.create (fun () -> r := Some(f x)) ()
    in
    {tid=t ; value=r}

  let force (f:'a future) : 'a =
    Thread.join f.tid ;
    match !(f.value) with
    | Some v -> v
    | None -> failwith "impossible!"
end
```

Now using Futures

```
let x = future f () in
let y = g () in
let v = force x in
(* compute with v and y *)
```

Back to the Futures

```
module type FUTURE =
sig
  type 'a future

  val future : ('a->'b) -> 'a -> 'b future
  val force :'a future -> 'a
end
```

```
val f : unit -> int
val g : unit -> int
```

with futures library:

```
let x = future f () in
let y = g () in
let v = force x in
y + v
```

without futures library:

```
let r = ref None
let t = Thread.create
          (fun _ -> r := Some(f ()))
          ()
in
let y = g() in
Thread.join t ;
match !r with
    Some v -> y + v
  | None -> failwith "impossible"
```

Back to the Futures

```
module type FUTURE =
sig
  type 'a future

  val future : ('a->'b) -> 'a -> 'b future
  val force :'a future -> 'a
end
```

```
val f : unit -> int
val g : unit -> int
```

with futures library:

```
let x = future f () in
let y = g () in
let v = force x in
y + v
```

without futures library:

```
let r = ref None
let t = Thread.create
           (fun _ -> r := Some(f ()))
           ()
in
let y = g() in
Thread.join t ;
match !r with
    Some v -> y + v
  | None -> failwith "impossible"
```

what happens if we delete these lines?

Back to the Futures

```
module type FUTURE =
sig
  type 'a future

  val future : ('a->'b) -> 'a -> 'b future
  val force :'a future -> 'a
end
```

```
val f : unit -> int
val g : unit -> int
```

with futures library:

```
let x = future f () in
let y = g () in
let v = force x in
y + x
```

without futures library:

```
let r = ref None
let t = Thread.create
           (fun _ -> r := Some(f ()))
           ()
in
let y = g() in
Thread.join t ;
match !r with
    Some v -> y + v
  | None -> failwith "impossible"
```

what happens if we use x and forget to force?

Back to the Futures

```
module type FUTURE =
sig
  type 'a future

  val future : ('a->'b) -> 'a -> 'b future
  val force :'a future -> 'a
end
```

```
val f : unit -> int
val g : unit -> int
```

with futures library:

```
let x = future f () in
let y = g () in
let v = force x in
y + x
```

without futures library:

```
let r = ref None
let t = Thread.create
           (fun _ -> r := Some(f ()))
           ()
in
let y = g() in
Thread.join t ;
match !r with
    Some v -> y + v
  | None -> failwith "impossible"
```

Moral: Futures + typing ensure entire categories of errors can't happen -- you protect yourself from your own stupidity

Back to the Futures

```
module type FUTURE =
sig
  type 'a future

  val future : ('a->'b) -> 'a -> 'b future
  val force :'a future -> 'a
end
```

```
val f : unit -> int
val g : unit -> int
```

with futures library:

```
let x = future f () in
let v = force x in
let y = g () in
y + x
```

without futures library:

```
let r = ref None
let t = Thread.create
            (fun _ -> r := Some(f ()))
            ()
in
Thread.join t ;
let y = g() in
match !r with
    Some v -> y + v
  | None -> failwith "impossible"
```

what happens if you relocate force, join?

Back to the Futures

```
module type FUTURE =
sig
  type 'a future

  val future : ('a->'b) -> 'a -> 'b future
  val force :'a future -> 'a
end
```

```
val f : unit -> int
val g : unit -> int
```

with futures library:

```
let x = future f () in
let v = force x in
let y = g () in
y + x
```

without futures library:

```
let r = ref None
let t = Thread.create
           (fun _ -> r := Some(f ()))
           ()
in
Thread.join t ;
let y = g() in
match !r with
    Some v -> y + v
  | None -> failwith "impossible"
```

Moral: Futures are not a universal savior

An Example: Mergesort on Arrays

```
let mergesort (cmp:'a->'a->int)
              (arr : 'a array) : 'a array =
  let rec msort (start:int) (len:int) : 'a array =
    match len with
      | 0 -> Array.of_list []
      | 1 -> Array.make 1 arr.(start)
      | _ -> let half = len / 2 in
             let a1 = msort start half in
             let a2 = msort (start + half)
                            (len - half) in
               merge a1 a2

  and merge (a1:'a array) (a2:'a array) : 'a array =
      ...
```

An Example: Mergesort on Arrays

```ocaml
let mergesort (cmp:'a->'a->int) (arr : 'a array) : 'a array =
  let rec msort (start:int) (len:int) : 'a array =
    match len with
      | 0 -> Array.of_list []
      | 1 -> Array.make 1 arr.(start)
      | _ -> let half = len / 2 in
             let a1 = msort start half in
             let a2 = msort (start + half) (len - half) in
               merge a1 a2
  and merge (a1:'a array) (a2:'a array) : 'a array =
    let a = Array.make (Array.length a1 + Array.length a2) a1.(0) in
    let rec loop i j k =
      match i < Array.length a1, j < Array.length a2 with
        | true, true -> if cmp a1.(i) a2.(j) <= 0 then
                          (a.(k) <- a1.(i) ; loop (i+1) j (k+1))
                        else (a.(k) <- a2.(j) ; loop i (j+1) (k+1))
        | true, false -> a.(k) <- a1.(i) ; loop (i+1) j (k+1)
        | false, true -> a.(k) <- a2.(j) ; loop i (j+1) (k+1)
        | false, false -> ()
    in
      loop 0 0 0 ; a
  in
    msort 0 (Array.length arr)
```

An Example: Mergesort on Arrays

```
let mergesort (cmp:'a->'a->int)
              (arr : 'a array) : 'a array =
  let rec msort (start:int) (len:int) :
    match len with
      | 0 -> Array.of_list []
      | 1 -> Array.make 1 arr.(start)
      | _ -> let half = len / 2 in
             let a1 = msort start half in
             let a2 = msort (start + half)
                            (len - half) in
               merge a1 a2

  and merge (a1:'a array) (a2:'a array) : 'a array =
      ...
```

Opportunity for parallelization

Making Mergesort Parallel

```
let mergesort (cmp:'a->'a->int)
              (arr : 'a array) : 'a array =
  let rec msort (start:int) (len:int) : 'a array =
    match len with
      | 0 -> Array.of_list []
      | 1 -> Array.make 1 arr.(start)
      | _ -> let half = len / 2 in
             let a1_f =
               Future.future (msort start) half in
             let a2 =
               msort (start + half)(len - half) in
             merge (Future.force a1_f) a2

  and merge (a1:'a array) (a2:'a array) : 'a array =
```

Divide-and-Conquer

This is an instance of a basic *divide-and-conquer* pattern in parallel programming

- take the problem to be solved and divide it in half
- fork a thread to solve the first half
- simultaneously solve the second half
- synchronize with the thread we forked to get its results
- combine the two solution halves into a solution for the whole problem.

Warning: the fact that we only had to rewrite 2 lines of code for mergesort made the parallelization transformation look deceptively easy

- we also had to verify that any two threads did not touch overlapping portions of the array -- if they did we would have to again worry about scheduling nondeterminism

Caveats

There is some overhead for creating a thread.

- On uniprocessor, parallel code *slower* than sequential code.

Even on a multiprocessor, we do *not always* want to fork.

- when the subarray is small, faster to sort it sequentially than to fork
  - similar to using insertion sort when arrays are small vs. quicksort
- this is known as a *granularity problem*
  - more parallelism than we can effectively take advantage of.

Caveats

In a good implementation of futures, a compiler and run-time system might look to see whether the cost of doing the fork is justified by the amount of work that will be done. Today, it's up to you to figure this out… ☹

- typically, use parallel divide-and-conquer until:
  - (a) we have generated *at least* as many threads as there are processors
    - often *more threads* than processors because different jobs take different amounts of time to complete and we would like to keep all processors busy
  - (b) the sub-arrays have gotten small enough that it's not worth forking.

We're not going to worry about these performance-tuning details but rather focus on the distinctions between *parallel* and *sequential algorithms*.

Another Example

```
type 'a tree = Leaf | Node of 'a node
and 'a node = {left  : 'a tree ;
               value : 'a      ;
               right : 'a tree }

let rec fold (f:'a -> 'b -> 'b -> 'b) (u:'b)
             (t:'a tree) : 'b =
  match t with
  | Leaf -> u
  | Node n ->
     f n.value (fold f u n.left) (fold f u n.right)

let sum (t:int tree) = fold (+) 0 t
```

Another Example

```
type 'a tree = Leaf | Node of 'a node
and 'a node = {left  : 'a tree ;
               value : 'a      ;
               right : 'a tree }

let rec pfold (f:'a -> 'b -> 'b -> 'b) (u:'b)
              (t:'a tree) : 'b =
  match t with
  | Leaf -> u
  | Node n ->
     let l_f = Future.future (pfold f u) n.left in
     let r = pfold f u n.right in
     f n.value (Future.force l_f) r

let sum (t:int tree) = pfold (+) 0 t
```

Note

- If the tree is unbalanced, then we're not going to get the same speedup as if it's balanced.
- Consider the degenerate case of a list.
  - The forked child will terminate without doing any useful work.
  - So the parent is going to have to do all that work.
  - Pure overhead… ☹
- In general, lists are a horrible data structure for parallelism.
  - *we can't cut the list in half in constant time*
  - for arrays and trees, we can do that (assuming the tree is balanced.)

Side Effects?

```
type 'a tree = Leaf | Node of 'a node
and 'a node = { left  : 'a tree ;
                value : 'a      ;
                right : 'a tree }

let rec pfold (f:'a -> 'b -> 'b -> 'b) (u:'b)
              (t:'a tree) : 'b =
  match t with
  | Leaf -> u
  | Node n ->
     let l_f = Future.future (pfold f u) n.left in
     let r = pfold f u n.right in
     f n.value (Future.force l_f) r

let print (t:int tree) =
  pfold (fun n _ _ -> Printf.print “%d\n” n) ()
```

Huge Point

If code is purely functional, then it never matters in what order it is run.

If f () and g () are pure then all of the following are equivalent:

```
let x = f() in
let y = g() in
e
```

```
let x_f = future f () in
let y   = g ()        in
let x   = force x_f   in
e
```

```
let y = g () in
let x = f () in
e
```

```
let y_g = future g () in
let x   = f ()        in
let y   = force y_g   in
e
```

As soon as we introduce *side-effects*, the order starts to matter.

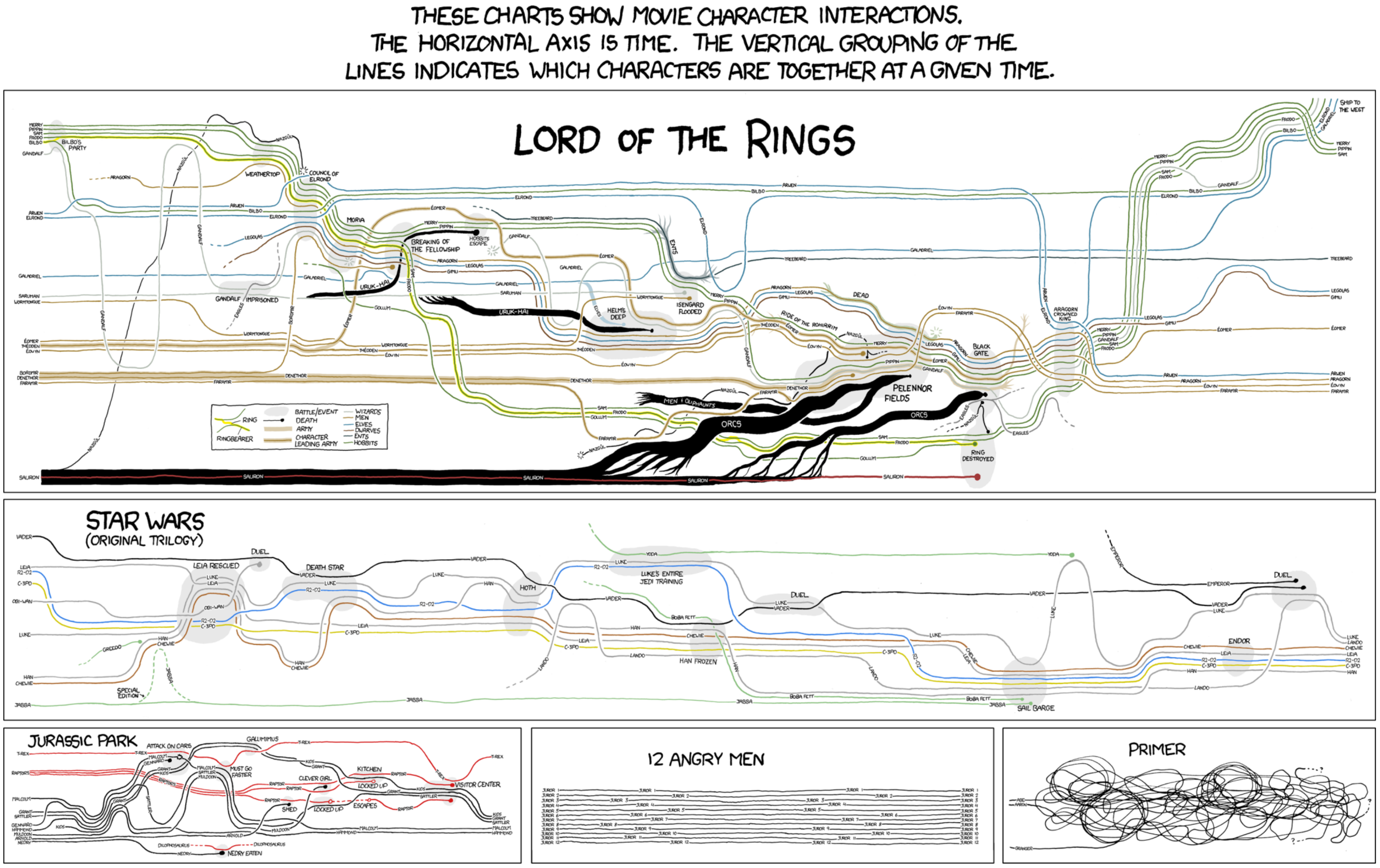
- This is why, IMHO, *imperative* languages where even the simplest of program phrases involves a side effect, are doomed.
- Of course, we've been saying this for 30 years!
- See J. Backus's Turing Award lecture, *"Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs."*

  http://www.cs.cmu.edu/~crary/819-f09/Backus78.pdf

Programming with locks

(trigger warning: this gets pretty nasty)

Perils of Parallelism



MANAGING MUTABLE DATA

Consider a Bank Account ADT

```
type account = { name : string; mutable bal : int }

let create (n:string) (b:int) : account =
  { name = n; bal = b }

let deposit (a:account) (amount:int) : unit =
  if a.bal + amount < max_balance then
    a.bal <- a.bal + amount

let withdraw (a:account) (amount:int) : int =
  if a.bal >= amount then (
    a.bal <- a.bal - amount;
    amount
  ) else 0
```

Simulating a Bank

```
val bank : account array

let rec atm (loc:string) =
  let id = getAccountNumber() in
  let w = getWithdrawAmount() in
  let d = withdraw (bank.(id)) w in
  dispenseDollars d ;
  atm loc

let world () =
  Thread.create atm "Princeton, Nassau" ;
  Thread.create atm "NYC, Penn Station" ;
  Thread.create atm "Boston, Lexington Square"
```

The ATM problem

- Suppose two ATMs, running in separate threads, try to perform a withdrawal from the same bank account around the same time.

- More specifically:
  - suppose bank.(0) is an account that starts with $100
  - thread 1 tries to withdraw $50 and thread 2 tries to withdraw $75 at roughly the same time

Simplifying the situation…

```
b = ref 100
```

```
let w = 50 in
if !b > w then
   (b <- !b - w ;
    w)
else
   0
```

```
let w = 75 in
if !b > w then
   (b <- !b - w ;
    w)
else
   0
```

Simplifying the situation…

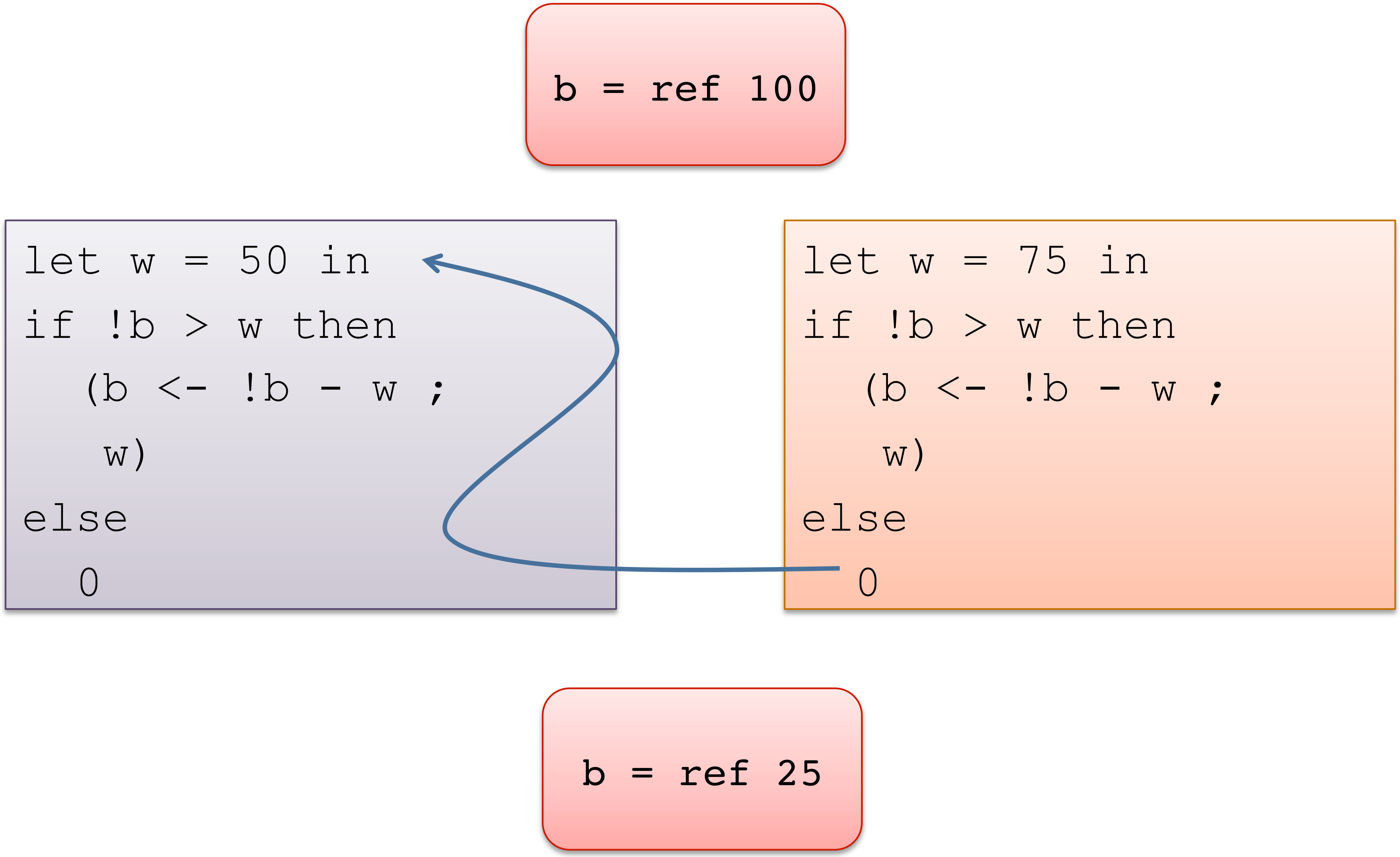
```
b = ref 100
```

```
let w = 50 in
if !b > w then
   (b <- !b - w ;
    w)
else
   0
```

```
let w = 75 in
if !b > w then
   (b <- !b - w ;
    w)
else
   0
```

```
b = ref 50
```

Simplifying the situation…



```
b = ref 100
```

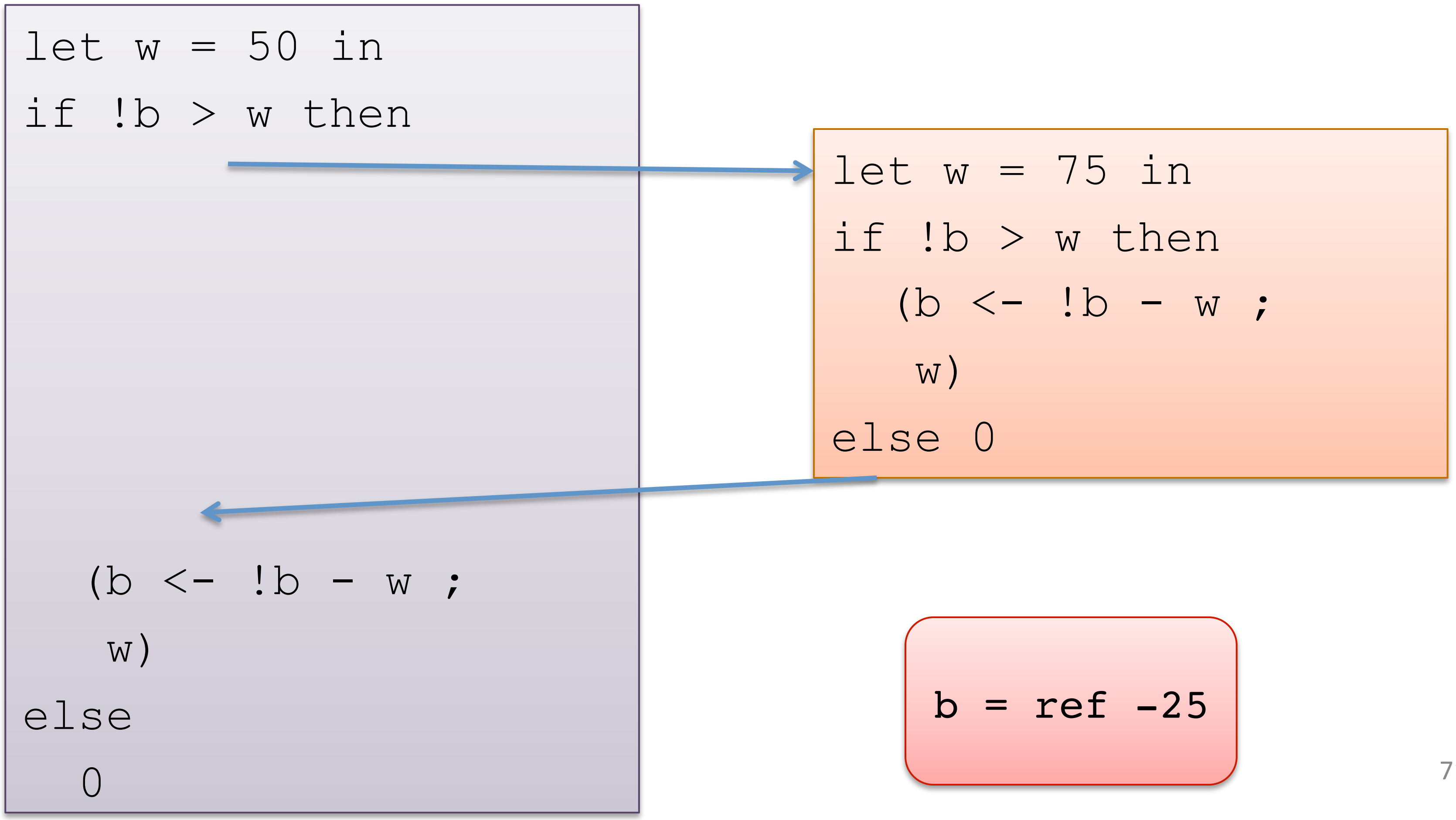
```
let w = 50 in
if !b > w then
   (b <- !b - w ;
    w)
else
   0
```

```
let w = 75 in
if !b > w then
   (b <- !b - w ;
    w)
else
   0
```

```
b = ref 25
```

Another schedule ...

```
b = ref 100
```



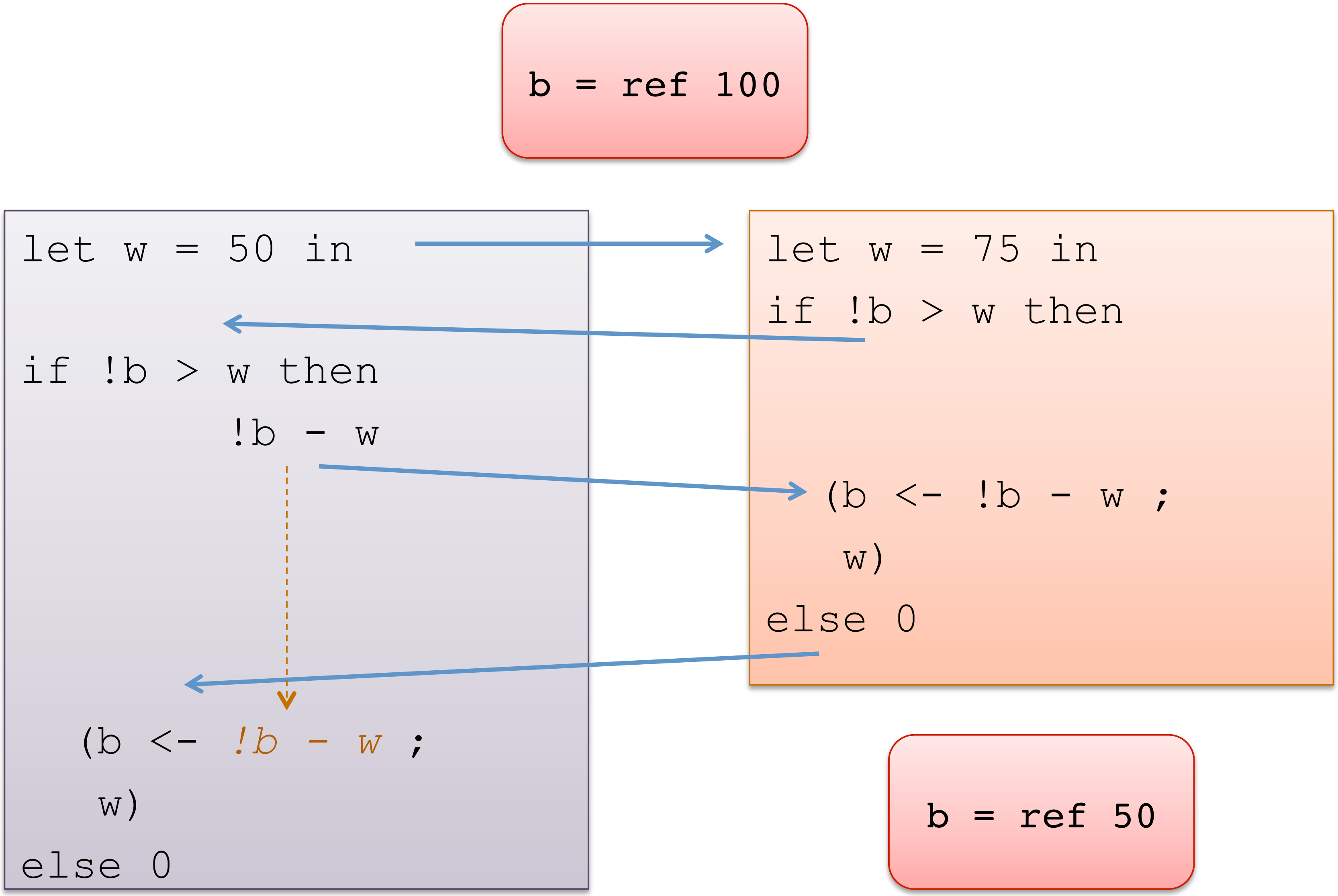
```
let w = 50 in
if !b > w then
```

```
let w = 75 in
if !b > w then
   (b <- !b - w ;
    w)
else 0
```

```
   (b <- !b - w ;
    w)
else
  0
```

```
b = ref -25
```

Good for you ... (less so for the bank)



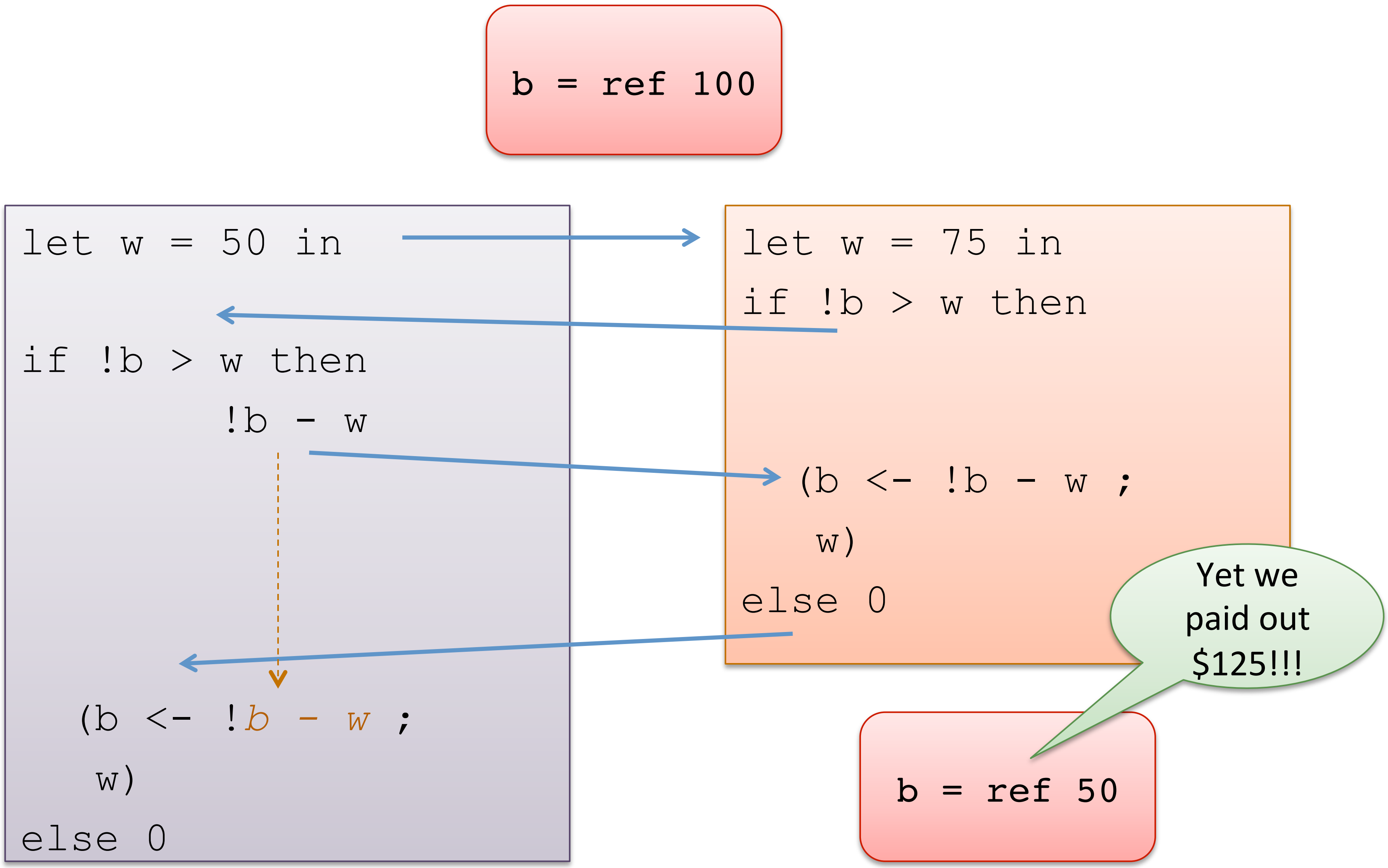
```
b = ref 100
```

```
let w = 50 in

if !b > w then
        !b - w

   (b <- !b - w ;
    w)
else 0
```

```
let w = 75 in
if !b > w then

   (b <- !b - w ;
    w)
else 0
```

```
b = ref 50
```

Good for you ... (less so for the bank)



```
b = ref 100
```

```
let w = 50 in

if !b > w then
         !b - w



   (b <- !b - w ;
    w)
else 0
```

```
let w = 75 in
if !b > w then


  (b <- !b - w ;
   w)
else 0
```

```
b = ref 50
```

More Synchronization: Locks

This is not a problem we can fix with fork/join/futures

- Thread.join waits until one thread terminates
- But the ATMs shouldn't ever terminate:

```
let rec atm (loc:string) =
  let id = getAccountNumber() in
  let w = getWithdrawAmount() in
  let d = withdraw (bank.(id)) w in
  dispenseDollars d ;
  atm loc
```

- *Fundamental problem*: atms are long-running computations that compete over a shared resource (the bank)

More Synchronization: Locks

This is not a problem we can fix with fork/join/futures

- Thread.join waits until one thread terminates
- But the ATMs shouldn't ever terminate:

```
let rec atm (loc:string) =
  let id = getAccountNumber() in
  let w = getWithdrawAmount() in
  let d = withdraw (bank.(id)) w in
  dispenseDollars d ;
  atm loc
```

- *Fundamental problem*: atms are long-running computations that compete over a shared resource (the bank)

Solution: use a *mutex lock* to synchronize threads.

- mutex is short for "mutual exclusion"
- locks give control over resource access
- controlled access to a shared resource is a *concurrency problem*, not a *parallelization problem*

Mutex Locks in OCaml

```
module type Mutex =
  sig
    type t  (* type of mutex locks *)

    val create : unit -> t (* create a fresh lock *)

    (* try to acquire the lock - makes
       the thread go to sleep until the lock
       is free.  So at most one thread "owns" the lock. *)
    val lock : t -> unit

    (* releases the lock so other threads can
       wake up and try to acquire the lock. *)
    val unlock : t -> unit

    (* similar to lock, but never blocks.  Instead, if
       the lock is already locked, it returns "false". *)
    val try_lock : t -> bool
  end
```

Adding a Lock

```
type account = { name : string; mutable bal : int; lock : Mutex.t }

let create (n:string) (b:int) : account =
  { name = n; bal = b; lock = Mutex.create() }

let deposit (a:account) (amount:int) : unit =
  Mutex.lock a.lock;
    if a.bal + amount < max_balance then
      a.bal <- a.bal + amount;
  Mutex.unlock a.lock

let withdraw (a:account) (amount:int) : int =
  Mutex.lock a.lock;
    let result =
      if a.bal >= amount then (
        a.bal <- a.bal - amount;
        amount ) else 0
    in
  Mutex.unlock a.lock;
  result
```

Adding a Lock

```
type account = { name : string; mutable bal : int; lock : Mutex.t }

let create (n:string) (b:int) : account =
  { name = n; bal = b; lock = Mutex.create() }

let deposit (a:account) (amount:int) : unit =
  Mutex.lock a.lock;
    if a.bal + amount < max_balance then
      a.bal <- a.bal + amount;
  ~~Mutex.unlock a.lock~~

let withdraw (a:account) (amount:int) : int =
  Mutex.lock a.lock;
    let result =
      if a.bal >= amount then (
        a.bal <- a.bal - amount;
        amount ) else 0
    in
  Mutex.unlock a.lock;
  result
```

pretty easy to forget to unlock your lock

Better

```
type account = { name : string; mutable bal : int; lock : Mutex.t }

let create (n:string) (b:int) : account =
  { name = n; bal = b; lock = Mutex.create() }

let deposit (a:account) (amount:int) : unit =
  with_lock a.lock (fun () ->
    if a.bal + amount < max_balance then
      a.bal <- a.bal + amount))

let withdraw (a:account) (amount:int) : int =
  with_lock a.lock (fun () ->
      if a.bal >= amount then (
        a.bal <- a.bal - amount;
        amount ) else 0
  )
```

```
let with_lock (l:Mutex.t)
              (f:unit->'b) : 'b =
  Mutex.lock l;
  let res = f () in
  Mutex.unlock l;
  res
```

General Design Pattern

Associate any shared, mutable thing with a lock.

- Java takes care of this for you (but only for one simple case.)
- In OCaml, C, C++, etc. it's up to you to create & manage locks.

In every thread, before reading or writing the object, acquire the lock.

- This prevents other threads from interleaving their operations on the object with yours.
- *Easy error: forget to acquire or release the lock.*

When done operating on the mutable value, release the lock.

- It's important to minimize the time spent holding the lock.
- That's because you are blocking all the other threads.
- *Easy error: raise an exception and forget to release a lock…*
- *Hard error: lock at the wrong granularity (too much or too little)*

Better Still

```
type account = { name : string; mutable bal : int; lock : Mutex.t }

let create (n:string) (b:int) : account =
  { name = n; bal = b; lock = Mutex.create() }

let deposit (a:account) (amount:int) : unit =
  with_lock a.lock (fun () ->
    if a.bal + amount < max_balance then
      a.bal <- a.bal + amount))

let withdraw (a:account) (amount:ir
  with_lock a.lock (fun () ->
      if a.bal >= amount then (
        a.bal <- a.bal - amount;
        amount ) else 0
  )
```

```
let with_lock (l:Mutex.t)
              (f:unit->'b) : 'a =
  Mutex.lock l;
  let res =
    try f ()
    with exn -> (Mutex.unlock l;
                 raise exn)
  in
  Mutex.unlock l;
  res
```

Unfortunately…

This design pattern of associating a lock with each object, and using with_lock on each method works well when we need to make the method seem atomic.

- In fact, Java has a *synchronize* construct to cover this.

But it does *not* work when we need to do some set of actions on *multiple* objects.

MANAGING MULTIPLE MUTABLE DATA STRUCTURES

Another Example

```
type 'a stack = { mutable contents : 'a list;
                  lock : Mutex.t
                };;

let empty () = {contents=[]; lock=Mutex.create()};;

let push (s:'a stack) (x:'a) : unit =
    with_lock s.lock (fun _ ->
      s.contents <- x::s.contents)
;;

let pop (s:'a stack) : 'a option =
    with_lock s.lock (fun _ ->
      match s.contents with
      | [] -> None
      | h::t -> (s.contents <- t ; Some h))
;;
```

Another Example

```
type 'a stack = { mutable contents : 'a list;
                  lock : Mutex.t }

val empty : () -> 'a stack
val push  : 'a stack -> a -> unit
val pop   : 'a stack -> 'a option

let transfer_one (s1:'a stack) (s2: 'a stack) =
  with_lock s1.lock (fun _ ->
    match pop s1 with
   | None -> ()
   | Some x -> push s2 x)
```

Another Example

```
type 'a stack = { mutable contents : 'a list;
                  lock : Mutex.t }


val empty : () -> 'a stack
val push  : 'a stack -> a -> unit
val pop   : 'a stack -> 'a option


let transfer_one (s1:'a stack) (s2: 'a stack) =
  with_lock s1.lock (fun _ ->
    match pop s1 with
   | None -> ()
   | Some x -> push s2 x)
```

Unfortunately, we already hold `s1.lock` when we invoke `pop s1` which tries to acquire the lock.

Another Example

```
type 'a stack = { mutable contents : 'a list;
                  lock : Mutex.t }

val empty : () -> 'a stack
val push  : 'a stack -> a -> unit
val pop   : 'a stack -> 'a option

let transfer_one (s1:'a stack) (s2: 'a stack) =
  with_lock s1.lock (fun _ ->
    match pop s1 with
   | None -> ()
   | Some x -> push s2 x)
```

Unfortunately, we already hold `s1.lock` when we invoke `pop s1` which tries to acquire the lock.

So we end up *dead-locked.*

Another Example

```
type 'a stack = { mutable contents : 'a list;
                  lock : Mutex.t }

val empty : () -> 'a stack
val push  : 'a stack -> a -> unit
val pop   : 'a stack -> 'a option

let transfer_one (s1:'a stack) (s2: 'a stack) =
  with_lock s1.lock (fun _ ->
    match pop s1 with
   | None -> ()
   | Some x -> push s2 x)
```

Avoid deadlock by deleting the line that aquires s1.lock initially

A trickier problem

```
type 'a stack = { mutable contents : 'a list;
                  lock : Mutex.t }

val empty : () -> 'a stack
val push  : 'a stack -> a -> u
val pop   : 'a stack ->

let pop_two (s1:'a stack)
            (s2:'a stack) : ('a * 'a) option =
  match pop s1, pop s2 with
     | Some x, Some y -> Some (x,y)
     | Some x, None -> push s1 x ; None
     | None, Some y -> push s2 y ; None
```

Either:

(1) pop one from each if both nonempty, or

(2) have no effect at all

A trickier problem

```
type `a stack = { mutable contents : `a list;
                  lock : Mutex.t }

val empty : () -> `a stack
val push  : `a stack -> a -> unit
val pop   : `a stack -> `a option

let pop_two (s1:`a stack)
            (s2:`a stack) : (`      `a) option =
  match pop s1, pop s2 with
     | Some x, Some y -> Some (x,y)
     | Some x, None -> push s1 x ; None
     | None, Some y -> push s2 y ; None
```

But some other thread could sneak in here and try to perform an operation on our contents before we've managed to push the value back on.

Yet another broken solution

```
let no_lock_pop (s1:'a stack) : 'a option =
  match s1.contents with
  | [] -> None
  | h::t -> (s1.contents <- t ; Some h)

let no_lock_push (s1:'a stack) (x :'a) : unit =
  contents <- x::contents

let pop_two (s1:'a stack)
            (s2:'a stack) : ('a * 'a) option =
  with_lock s1.lock (fun _ ->
  with_lock s2.lock (fun _ ->
  match no_lock_pop s1, no_lock_pop s2 with
     | Some x, Some y -> Some (x,y)
     | Some x, None -> no_lock_push s1 x ; None
     | None, Some y -> no_lock_push s2 y ; None))
```

Yet another broken solution

```
let no_lock_pop (s1:‘a stack) : ‘a option =
  match s1.contents with
  | [] -> None
  | h::t -> (s1.contents <- t ; Some

let no_lock_push (s1:‘a stack)
  contents <- x::contents

let pop_two (s1:‘a stack)
            (s2:‘a stack) : (‘a * ‘a) option =
  with_lock s1.lock (fun _ ->
  with_lock s2.lock (fun _ ->
  match no_lock_pop s1, no_lock_pop s2 with
     | Some x, Some y -> Some (x,y)
     | Some x, None -> no_lock_push s1 x ; None
     | None, Some y -> no_lock_push s2 y ; None))
```

Problems?

Yet another broken solution

```
let no_lock_pop (s1:‘a stack) : ‘a option =
  match s1.contents with
  | [] -> None
  | h::t -> (s1.contents <- t ; Some

let no_lock_push (s1:‘a stack)
  contents <- x::contents

let pop_two (s1:‘a stack)
            (s2:‘a stack) : (‘a * ‘a) option =
  with_lock s1.lock (fun _ ->
  with_lock s2.lock (fun _ ->
  match no_lock_pop s1, no_lock_pop s2 with
     | Some x, Some y -> Some (x,y)
     | Some x, None -> no_lock_push s1 x ; None
     | None, Some y -> no_lock_push s2 y ; None))
```

What happens if we call pop_two x x?

Yet another broken solution

```
let no_lock_pop (s1:
  match s1.contents w
  | [] -> None
  | h::t -> (s1.contents <- t ;

let no_lock_push (s1:'a stack)
  contents <- x::contents


let pop_two (s1:'a stack)
            (s2:'a stack) : ('a * 'a) option =
  with_lock s1.lock (fun _ ->
  with_lock s2.lock (fun _ ->
  match no_lock_pop s1, no_lock_pop s2 with
     | Some x, Some y -> Some (x,y)
     | Some x, None -> no_lock_push s1 x ; None
     | None, Some y -> no_lock_push s2 y ; None))
```

In particular, consider:

```
Thread.create (fun _ -> pop_two x y)
Thread.create (fun _ -> pop_two y x)
```

What happens if two threads are trying to call pop_two at the same time?

Yet another broken solution

```
let no_lock_pop (s1:
  match s1.contents w
  | [] -> None
  | h::t -> (s1.contents <- t ; So     h)

let no_lock_push (s1:'a stack)
  contents <- x::contents

let pop_two (s1:'a stack)
            (s2:'a stack) : (
  with_lock
  with_lock
  match no_                                    ith
    | Some x, Some y -> Some (x,y)
    | Some x, None -> no_lock_push s1 x ; None
    | None, Some y -> no_lock_push s2 y ; None))
```

In particular, consider:

```
Thread.create (fun _ -> pop_two x y)
Thread.create (fun _ -> pop_two y x)
```

One possible interleaving:
T1 acquires x's lock.
T2 acquires y's lock.
T1 tries to acquire y's lock and blocks.
T2 tries to acquire x's lock and blocks.

DEADLOCK

A fix

```
type 'a stack = { mutable contents : 'a list; lock : Mutex.t; id : int }

let new_id : unit -> int =
  let c = ref 0 in (fun _ -> c := (!c) + 1 ; !c)

let empty () = {contents=[]; lock=Mutex.create(); id=new_id()};;

let no_lock_pop_two (s1:'a stack) (s2:'a stack) : ('a * 'a) option =
      match no_lock_pop s1, no_lock_pop s2 with
       | Some x, Some y -> Some (x,y)
       | Some x, None -> no_lock_push s1 x; None
       | None, Some y -> no_lock_push s2 y; None

let pop_two (s1:'a stack) (s2:'a stack) : ('a * 'a) option =
  if s1.id < s2.id then
    with_lock s1.lock (fun _ ->
    with_lock s2.lock (fun _ ->
      no_lock_pop_two s1 s2))
  else if s1.id > s2.id then
    with_lock s2.lock (fun _ ->
    with_lock s1.lock (fun _ ->
      no_lock_pop_two s1 s2))
  else with_lock s1.lock (fun _ -> no_lock_pop_two s1 s2)
```

sigh …

```
type 'a stack = { mutable contents : 'a list; lock : Mutex.t; id : int }

let new_id : unit -> int =
  let c = ref 0 in let l = Mutex.create() in
 (fun _ -> with_lock l (fun _ -> (c := (!c) + 1 ; !c)))

let empty () = {contents=[]; lock=Mutex.create(); id=new_id()};;

let no_lock_pop_two (s1:'a stack) (s2:'a stack) : ('a * 'a) option =
      match no_lock_pop s1, no_lock_pop s2 with
        | Some x, Some y -> Some (x,y)
        | Some x, None -> no_lock_push s1 x; None
        | None, Some y -> no_lock_push s2 y; None

let pop_two (s1:'a stack) (s2:'a stack) : ('a * 'a) option =
  …
;;
```

Refined Design Pattern

- *Associate a lock with each shared, mutable object.*
- *Choose some ordering on shared mutable objects.*
  - doesn't matter what the order is, as long as it is total.
  - in C/C++, often use the address of the object as a unique number.
  - Our solution: *add a unique ID number to each object*
- *To perform actions on a set of objects S atomically:*
  - acquire the locks for the objects in S *in order*.
  - perform the actions.
  - release the locks.

Refined Design Pattern

- *Associate a lock with each shared, mutable object.*
- *Choose some ordering on shared mutable objects.*
  - doesn't matter what the order
  - in C/C++, often use the address
    number.
  - Our solution: *add a unique ID*
- *To perform actions on a set of ob*
  - acquire the locks for the objects in S *in order*.
  - perform the actions.
  - release the locks.

Important!

Acquire all the locks you will need
BEFORE
performing any irreversible actions!

BUT: IN A BIG PROGRAM, IT IS REALLY HARD TO GET THIS RIGHT

A HUGE COMPONENT OF PL RESEARCH INVOLVES TRYING TO FIND THE MISTAKES PEOPLE MAKE WHEN DOING THIS. AVOID WHENEVER POSSIBLE! USE FUNCTIONAL ABSTRACTIONS!

SUMMARY

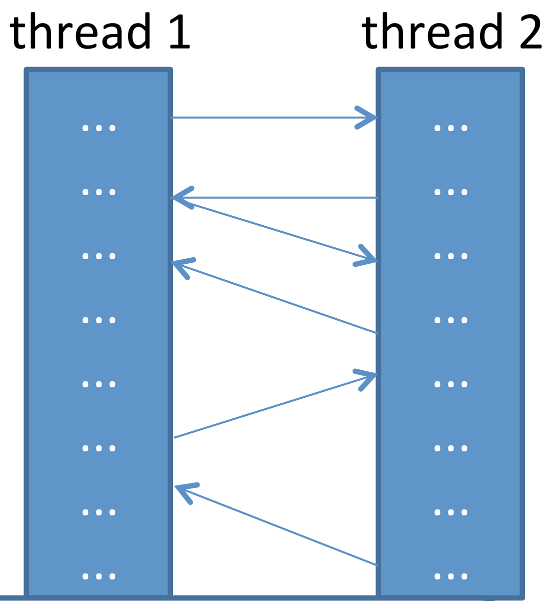
Programming with mutation, threads and locks

Reasoning about the correctness of pure parallel programs that include futures is easy -- no harder than ordinary, sequential programs. (Reasoning about their performance may be harder.)

Reasoning about shared variables and semaphores is *hard* in general, but *futures* are a *discipline* for getting it right.

Much of programming-language design is the art of finding good disciplines in which it's harder* to write bad programs.

Even aside from PL design, the same is true of software engineering with Abstract Data Types: engineer *disciplines* in your interfaces, harder for the user to get it wrong.



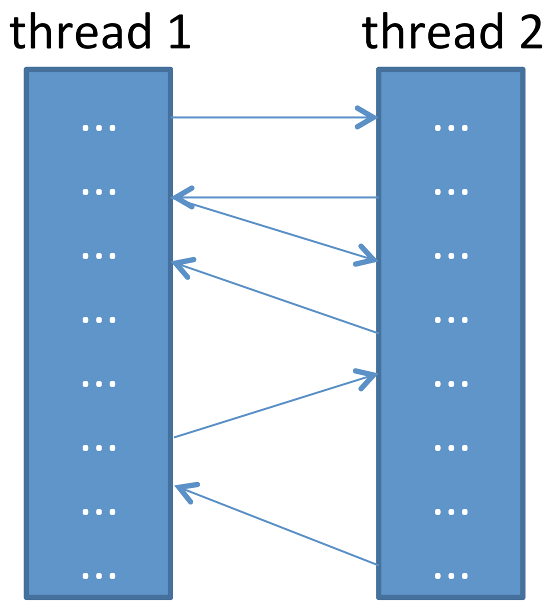
*but somebody will always find a way…

Programming with mutation, threads and locks

Reasoning about the correctness of pure parallel programs that include futures is easy -- no harder than ordinary, sequential programs. (Reasoning about their performance may be harder.)

Reasoning about concurrent programs with *effects* requires considering *all interleavings* of instructions of concurrently executing threads.*

– often too many interleavings for normal humans to keep track of
– nonmodular: you often have to look at the details of each thread to figure out what is going on
– locks cut down interleavings
– but knowing you have done it right still requires deep analysis



**and worse…*