Parallelism and Concurrency (Part II)

COS 326
David Walker
Princeton University

Pure Functions

A function (or expression) is *pure* if it has no *effects*.

- Valuable expressions should not have effects either

Recall that a function has an *effect* if its behavior cannot be completely explained by a *deterministic* relation between its inputs and its outputs

Expressions have effects when they:

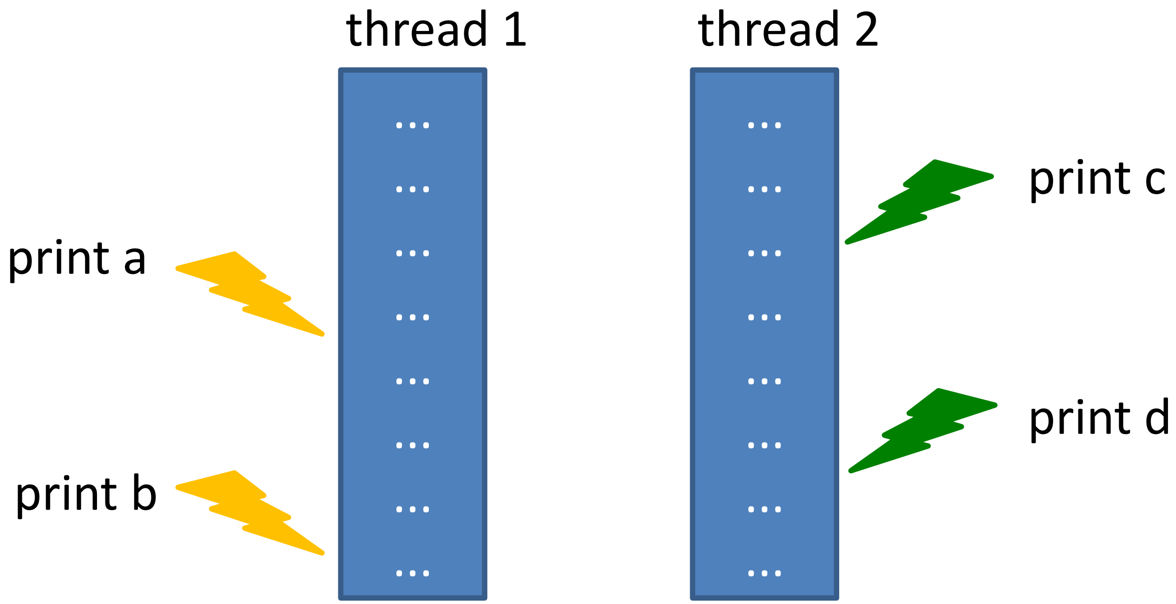
- don't terminate
- raise exceptions
- read from stdin/print to stdout
- read or write to a shared mutable data structure

(↓ increasingly difficult to deal with)

Not an effect: reading from immutable data structures

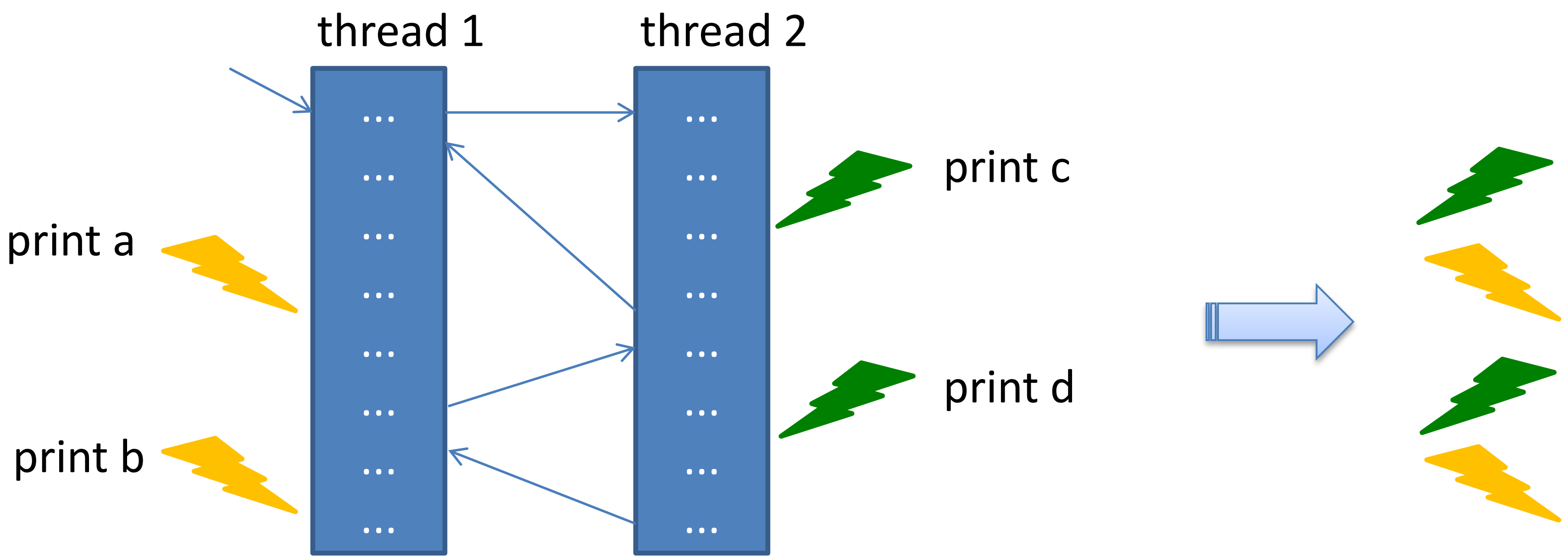
Effects and Parallelism

The combination of effects and parallelism is difficult to reason about: The run-time system is responsible for scheduling the instructions in each thread. Depending on the schedule, the effects happen in a different order



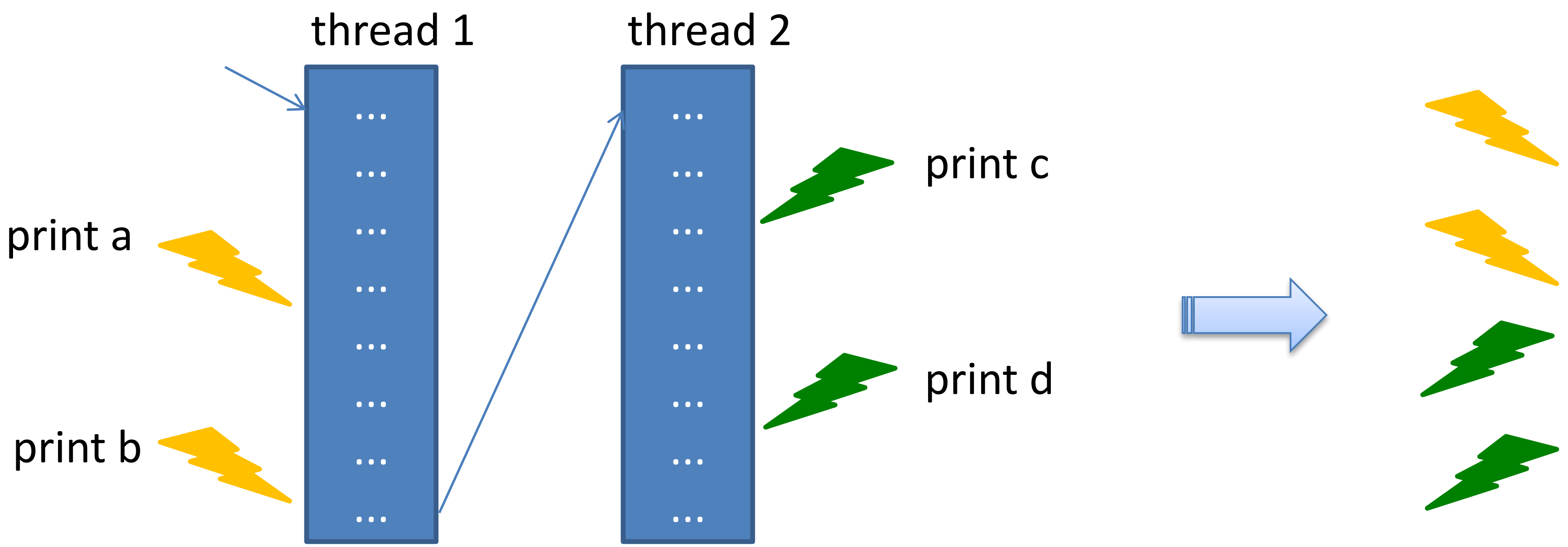
Effects and Parallelism

The combination of effects and parallelism is difficult to reason about: The run-time system is responsible for scheduling the instructions in each thread. Depending on the schedule, the effects happen in a different order



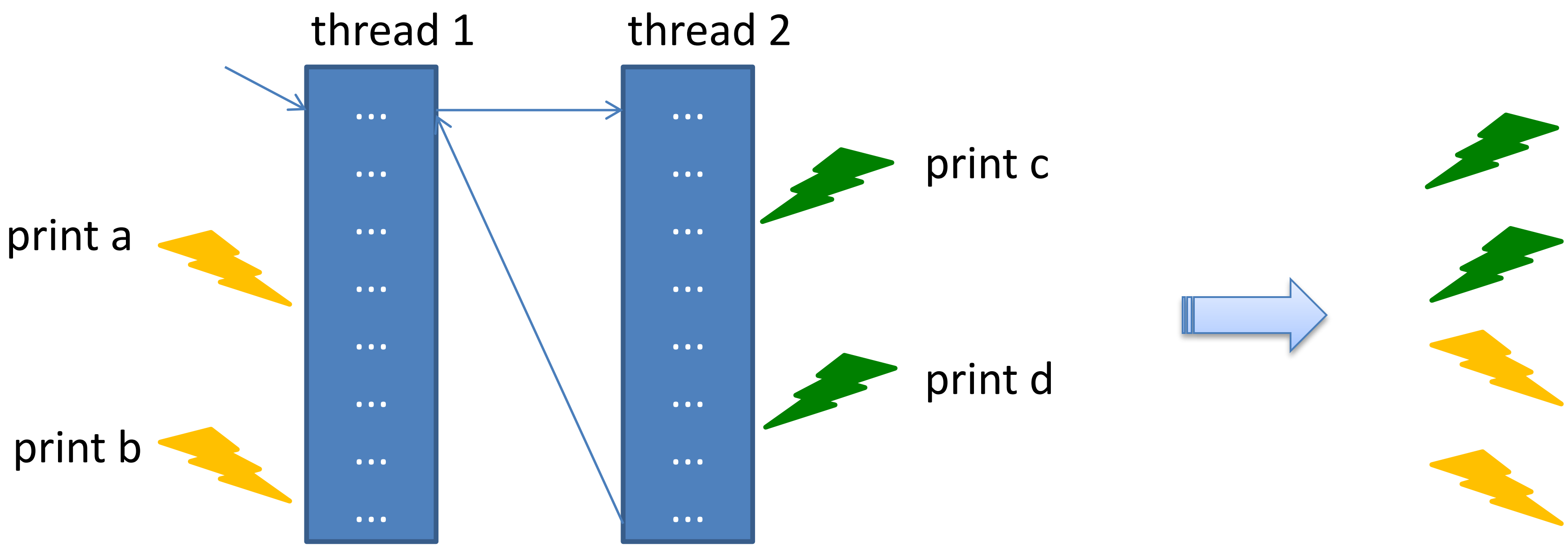
Effects and Parallelism

The combination of effects and parallelism is difficult to reason about: The run-time system is responsible for scheduling the instructions in each thread. Depending on the schedule, the effects happen in a different order



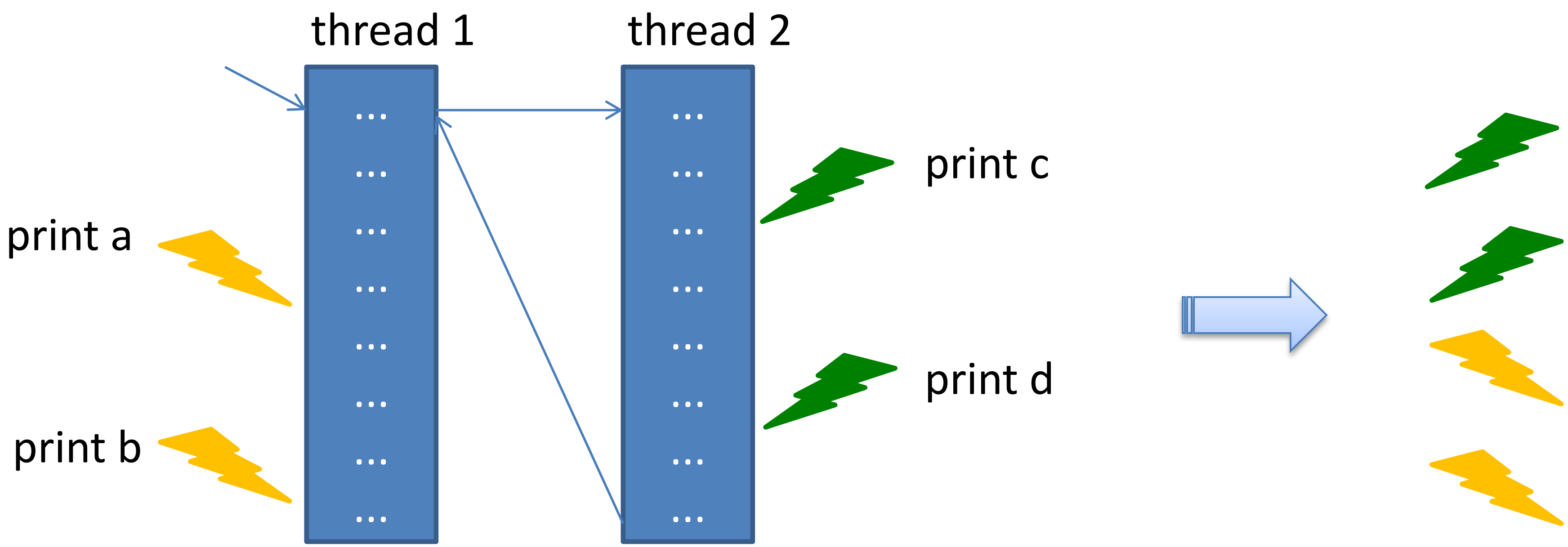
Effects and Parallelism

The combination of effects and parallelism is difficult to reason about: The run-time system is responsible for scheduling the instructions in each thread. Depending on the schedule, the effects happen in a different order



Effects and Parallelism

The combination of effects and parallelism is difficult to reason about: The run-time system is responsible for scheduling the instructions in each thread. Depending on the schedule, the effects happen in a different order



Understanding the output requires consideration of *all interleavings* of instructions. So many combinations! So much *non-determinism*!

Benign Effects & Futures

Not all uses of effects create non-determinism. Eg: Futures

```
sig
  type 'a future
  val future : (unit -> 'a) -> 'a future
  val force : 'a future -> 'a
end
```

```
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
```

Benign Effects & Futures

Provided your code contains no other effects, futures do not introduce non-determinism!

Consequence: when it comes to reasoning about the correctness of your programs, *pure functional code + parallel futures is no harder than pure functional sequential code*!

Equational reasoning laws:

```
if e1 is valuable then:
```

```
let x = e1 in        ==      let x = future (fun _ -> e1) in
e2                           e2[force x/x]
```

Moreover

Benign Effects & Futures

```
if e1 is valuable then:
```

```
let x = e1 in          ==      let x = future (fun _ -> e1) in
e2                             e2[force x/x]
```

```
type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree

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

Benign Effects & Futures

if e1 is valuable then:

```
let x = e1 in          ==     let x = future (fun _ -> e1) in
e2                            e2[force x/x]
```

```
type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree

let rec fold (f:'a -> 'b -> 'b -> 'b) (u:'b) (t:'a tree) : 'b =
  match t with
   | Leaf -> u
   | Node (n,left,right) ->
      let left' = future (fun _ -> fold f u left) in
      let right' = fold f u right in
      f n (force left') right'
```

Benign Effects & Futures

```
if e1 is valuable then:
```

```
let x = e1 in        ==     let x = future (fun _ -> e1) in
e2                          e2[force x/x]
```

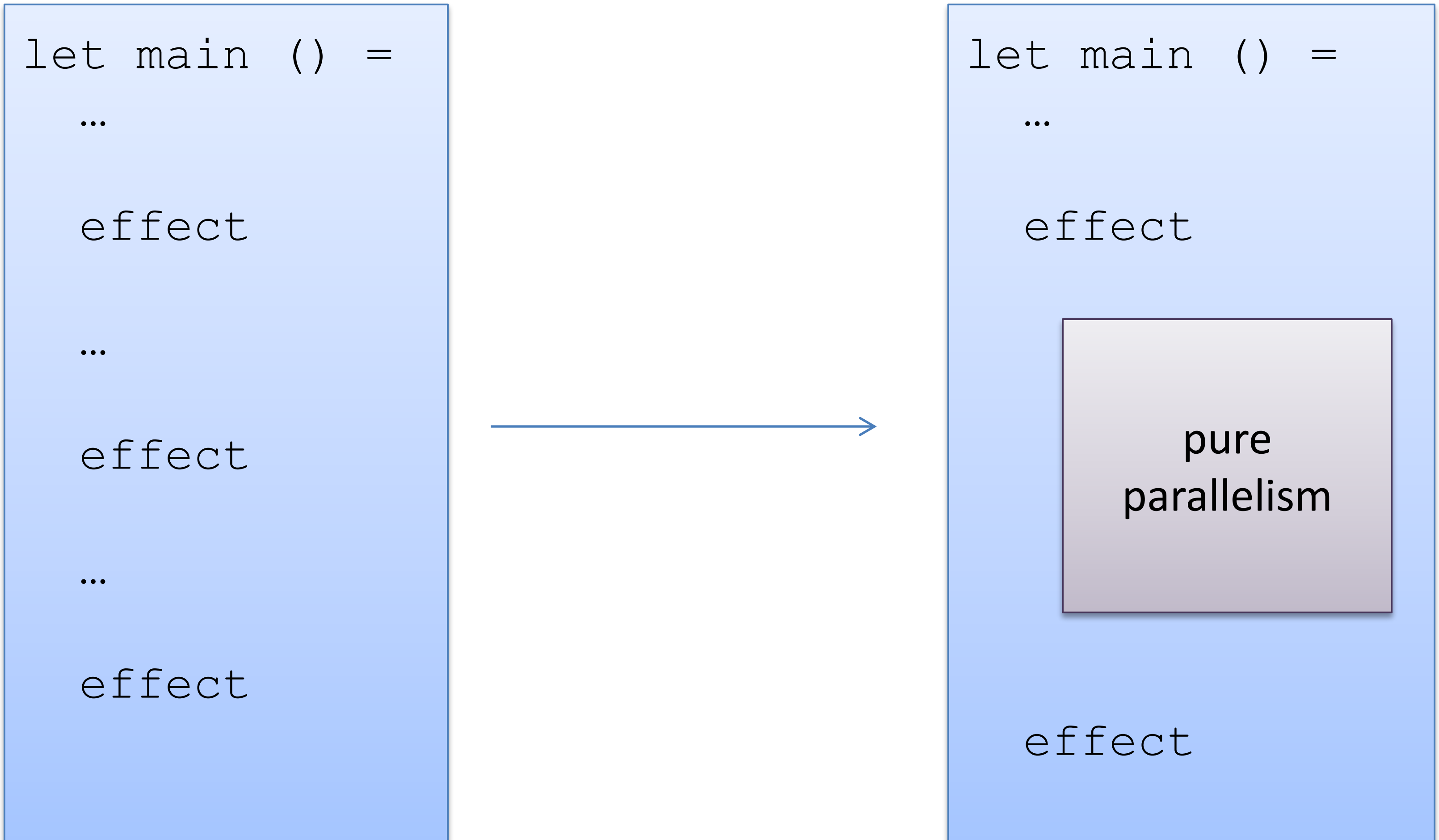
```
type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree

let rec fold (f:'a -> 'b -> 'b -> 'b) (u:'b) (t:'a tree) : 'b =
  match t with
   | Leaf -> u
   | Node (n,left,right) ->
       let left' = future (fun _ -> fold f u left) in
       let right' = fold f u right in
       f n (force left') right'
```

Moral: It is *vastly easier* to introduce parallelism in to *a pure functional program* using futures than using naked references, locks, join

Benign Effects & Futures

- What if your program has effects? (Most useful programs do!)
- Try to push the effects to the *edges* of your program and put parallelism in the middle. *Especially* limit mutable data.



LOCKS AND MUTABLE DATA

What happens here?

```
val bank : account array

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

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

Consider a Bank Acount 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
```

Synchronization: Locks

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

- The ATMs shouldn't ever terminate!
- Yet join only allows us to wait until one thread terminates.

Instead, we're going to us a *mutex lock* to synchronize threads.

- mutex is short for "mutual exclusion"
- locks will give us a way to introduce some controlled access to resources – in this case, the bank accounts.
- 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
```

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:in
  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
```

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)
;;
```

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 }

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 non-empty, 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 * '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 h)

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

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?

```
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))
```

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_                              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)
```

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.

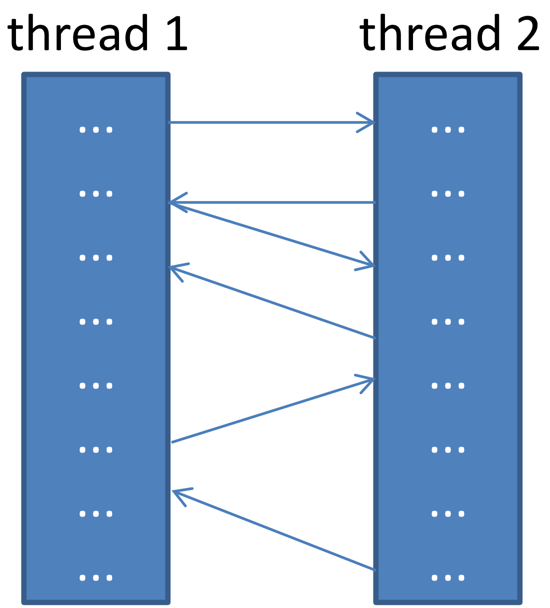
SUMMARY

Programming with mutation, threads and locks

Reasoning about pure parallel programs that include futures is easy -- no harder than ordinary, sequential programs

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
– non-modular: 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



END