Utrecht University

B3CC: Concurrency

04:Threads (2)

Tom Smeding

Last time...

- Processes, threads, threads and threads
- Mutual exclusion
  - Controlling access to shared resources
  - Only one process/thread is allowed in the *critical section* at once

Blocking algorithms

Locks

- A *lock* or *mutex* is a mechanism that enforces limits on access to a resource (mutual exclusion)
  - Conceptually simple!
- Programming languages with support for threads have some form of lock or barrier:
  - Haskell: locking and thread coordination via a mutable data structure called `MVar` (more later…)
  - C/C++: the `std::mutex<T>` class, the POSIX threads library, etc.
  - C#: the `lock` keyword, etc.
  - Rust: the `Mutex<T>` struct
  - etc…

Locks: historically

- No real hardware support
- e.g. Peterson's algorithm!
- Nowadays: nice hardware instructions!

- Thread A:

```
flag[0] = true;
turn    = 1;

while (flag[1]
    && turn == 1)
  /* do nothing */ ;

<critical section>

flag[0] = false;
```

- Thread B:

```
flag[1] = true;
turn    = 0;

while (flag[0]
    && turn == 0)
  /* do nothing */ ;

<critical section>

flag[1] = false;
```

Implementing locks

- The *compare-and-swap* (CAS) ("atomic compare-exchange") operation is an atomic instruction which allows mutual exclusion for any number of threads using a single bit of memory.

```
struct Result {
  bool success;
  int original;
}
Result atomic_compare_exchange(int *variable, int expected, int replacement);
```

- In **hardware**, *atomically* (as a single operation):

1. Compares the contents at a given memory location to the given value
2. If they are the same, writes a new value to that location
3. Returns:
   - whether the new value was written
   - the old value at the memory location

https://www.felixcloutier.com/x86/cmpxchg

Implementing locks

```
struct Result {
  bool success;
  int original;
}
Result atomic_compare_exchange(int *variable, int expected, int replacement);
```

- The *spin lock*:
  - Use a bit (here 'lock') where 0 represents unlocked and 1 represents locked

```
while (atomic_compare_exchange(&lock, 0, 1).original == 1)
  /* do nothing */ ;

<critical section>

lock = 0;
```

- In Haskell: can use compare-and-swap via `casIORef` (from `atomic-primops`)

The traditional mutex API

- mutex.acquireLock()

```
while (atomic_compare_exchange(&lock, 0, 1).original == 1)
  /* do nothing */ ;

<critical section>
```

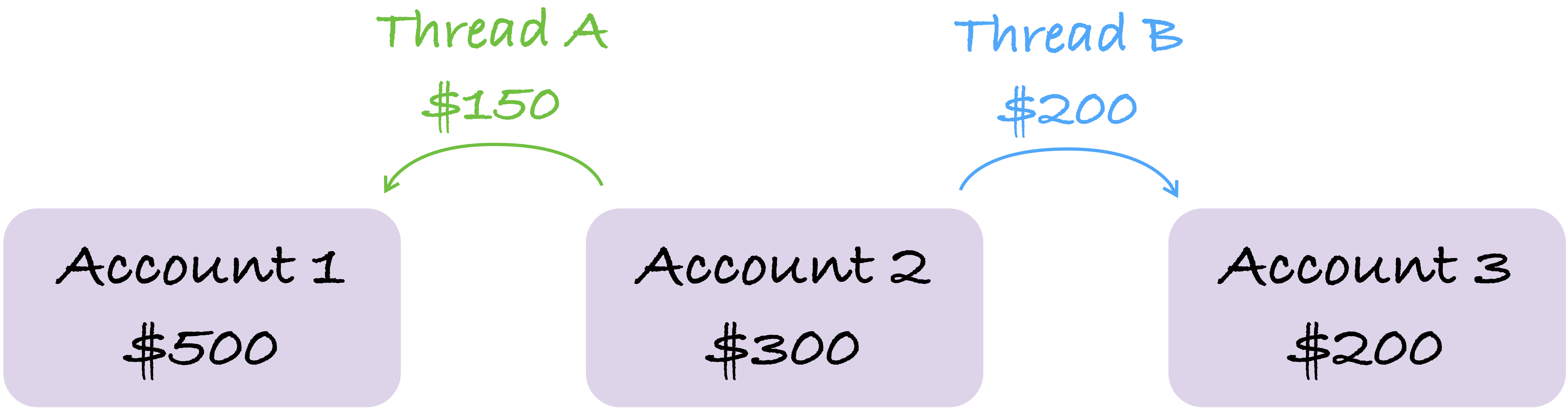
- mutex.releaseLock()

```
lock = 0;
```

- C/C++: `pthread_mutex_*`, `std::mutex<T>`; C#: `lock`; Rust: `Mutex<T>`; …

Example: bank accounts

- Model bank accounts and operations like withdrawing, depositing, and transferring money between accounts
  - It should not be possible to observe a state where, during a transfer, money has been withdrawn from one account but yet to be deposited into the target account



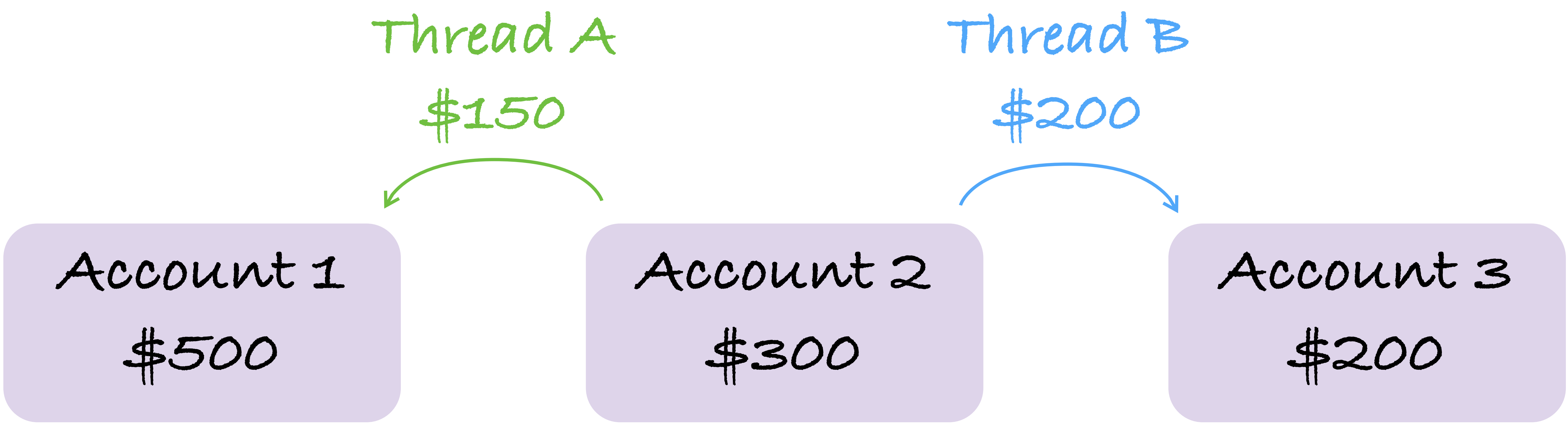
Attempt #1

- The basic idea:

```
struct Account {
  int balance;
};

void deposit(int amount, Account *acc) {
  int previous = acc->balance;
  acc->balance = previous + amount;
}

void withdraw(int amount, Account *acc) {
  deposit(-amount, acc);
}

void transfer(int amount, Account *from, Account *to) {
  withdraw(amount, from);
  deposit (amount, to);
}
```

Example: bank accounts

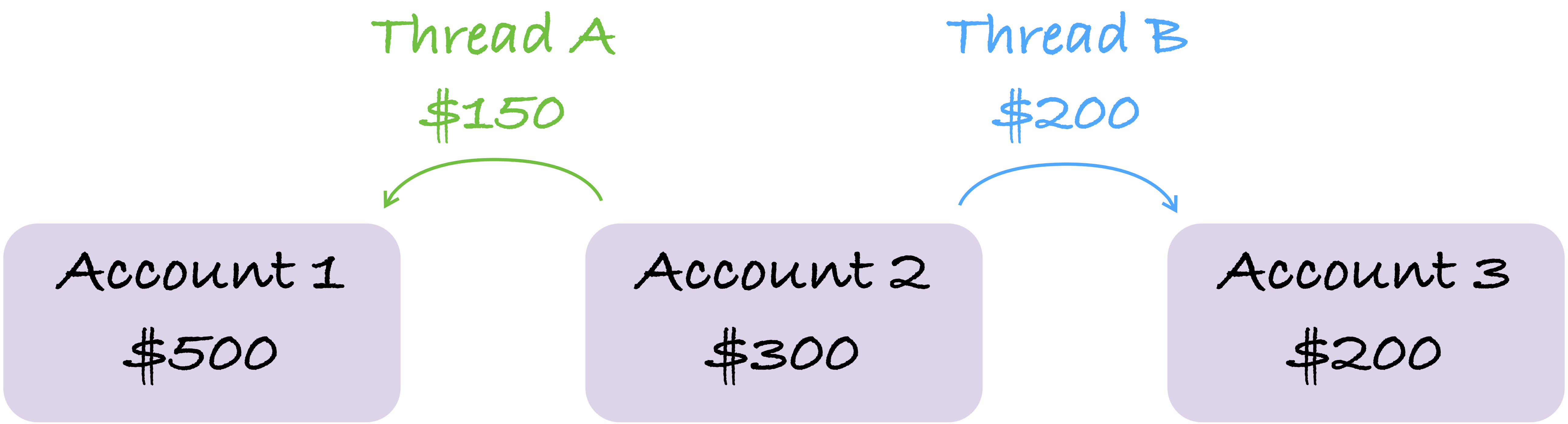
- Example: bank accounts



- Thread A:
- Thread B:

Example: bank accounts

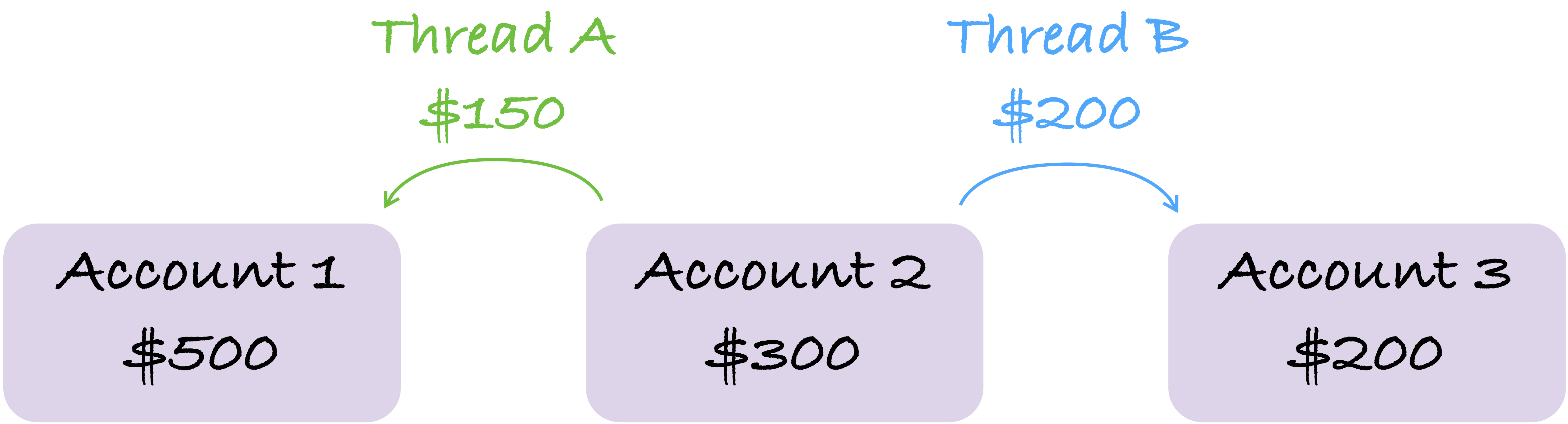
- Example: bank accounts



- Thread A:
  - Read balance of account 2: $300
- Thread B:

Example: bank accounts

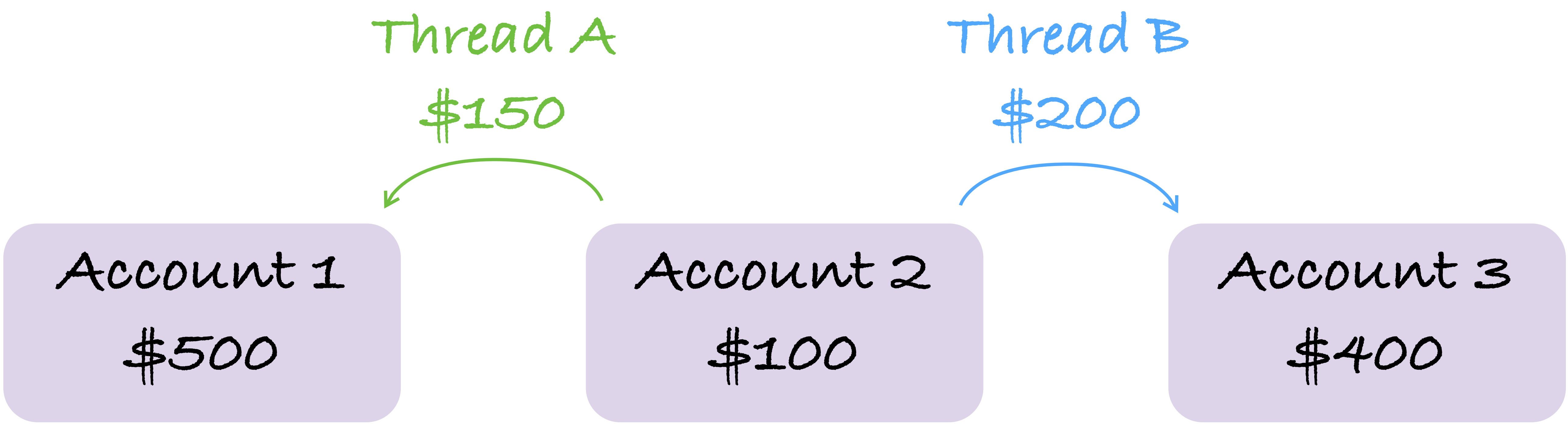
- Example: bank accounts



- Thread A:
  - Read balance of account 2: $300
- Thread B:
  - Read balance of account 2: $300

Example: bank accounts

- Example: bank accounts

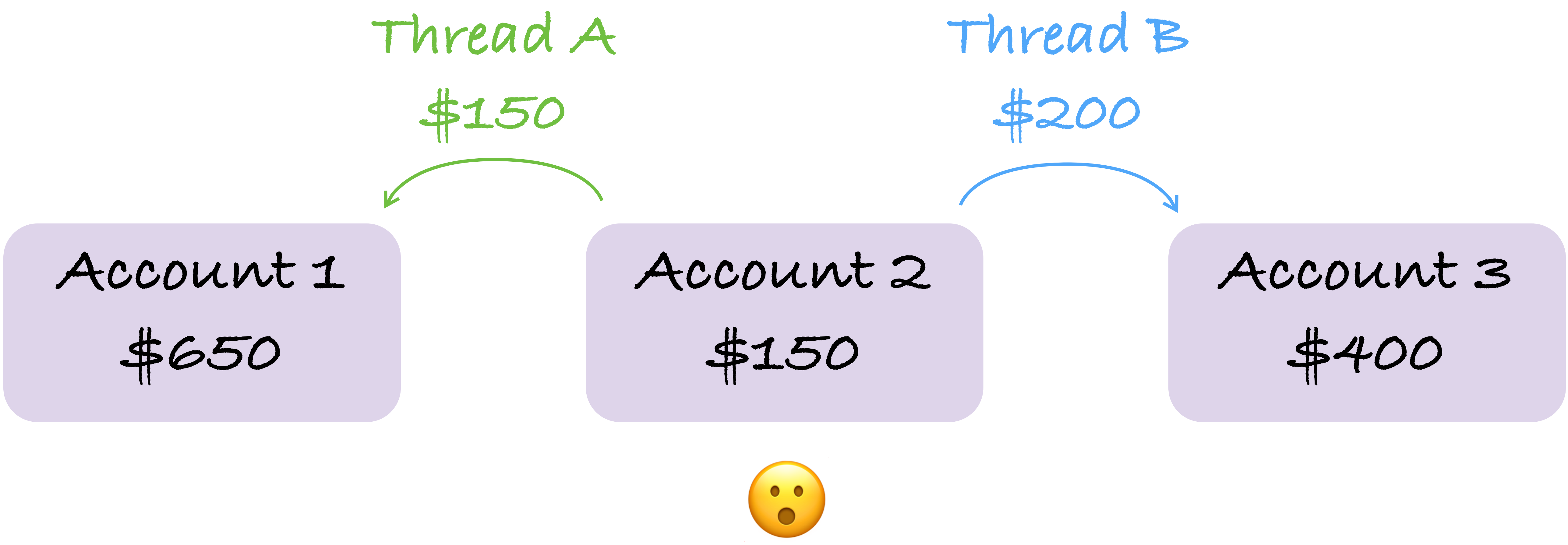


- Thread A:
  - Read balance of account 2: $300

- Thread B:
  - Read balance of account 2: $300
  - Update balance of account 2

Example: bank accounts

- Example: bank accounts



- Thread A:
  - Read balance of account 2: $300
  - Update balance of account 2
- Thread B:
  - Read balance of account 2: $300
  - Update balance of account 2

Attempt #2

- Use locks so that updates are atomic:

Let's include a lock this time

```
struct Account {
  int balance;
  Mutex lock;
};

void deposit(int amount, Account *acc) {
  acc->lock.acquireLock();
  acc->balance = acc->balance + amount;
  acc->lock.releaseLock();
}

void transfer(int amount, Account *from, Account *to) {
  withdraw(amount, from);
  deposit (amount, to);
}
```

Put balance update in a critical section

Oh no, inconsistent state!

Attempt #3

- We need to implement `transfer` differently

```
struct Account {
  int balance;
  Mutex lock;
};
```

```
void transfer(int amount, Account *from, Account *to) {
  from->lock.acquireLock();
  to->lock.acquireLock();
  from->balance = from->balance - amount;
  to->balance   = to->balance   + amount;
  to->lock.releaseLock();
  from->lock.releaseLock();
}
```

- Thread A:
  - transfer(100, acc1, acc2)
- Thread B:
  - transfer(200, acc2, acc1)

Attempt #4

- Take locks in an a fixed (but arbitrary) order; release in the opposite order

```
struct Account {
  int balance;
  Mutex lock;
};
```

```
void transfer(int amount, Account *from, Account *to) {
  if (from->accountNumber < to->accountNumber) {
    from->lock.acquireLock();
    to->lock.acquireLock();
    ...
    to->lock.releaseLock();
    from->lock.releaseLock();
  } else {
    to->lock.acquireLock();
    from->lock.acquireLock();
    ...
    from->lock.releaseLock();
    to->lock.releaseLock();
  }
}
```

Extending the example

- What happens if we want to…
  - Block (wait) until the 'from' account has sufficient funds?
  - Withdraw from a second account if the first does not have sufficient funds?
    - Suppose I hold locks #3 and #5…
    - And now need to acquire lock #2, or #4, or…

Advantages and disadvantages

- Difficulties / problems (among others):
  - Taking locks in the wrong order
  - Too few locks (*lock contention* decreases the amount of available concurrency)
  - Too many locks (increases overhead and subtle lock dependencies that can increase the change of *deadlock*)
  - Error recovery (exceptions -> data integrity)
  - No modular programming! (`transfer`; lock order)
- Advantages:
  - Easy critical sections if you have a single lock
  - Mutual exclusion!

User space

Threads in Haskell

Threads

- The fundamental action in concurrency: create a new thread of control

```
forkIO :: IO () -> IO ThreadId
```

  - Takes a computation of type `IO ()` as its argument
  - This `IO` action executes in a new thread concurrently with other threads
  - No specified order in which threads execute
  - Haskell user space threads are very cheap: ~1.5 KB / thread, easily run thousands of threads

Example

- Interleaving of two threads

```
import Control.Concurrent
import Control.Monad

main :: IO ()
main = do
  let n = 100

  forkIO $ replicateM_ n (putChar 'A')
  forkIO $ replicateM_ n (putChar 'x')

  putStrLn "done"
```

- Interleaving of two threads
  - The program exits when `main` returns, even if there are other threads still running!
    - How to check whether the child thread has completed?
  - The term `n :: Int` is shared between both threads (captured); this is safe because it is *immutable*

Sharing state

Sharing state

- `IORef`: mutable reference to some value
  - i.e. a regular variable in C#
  - Compare-and-swap behaviour using `casIORef`
  - Not designed for concurrency: need to protect critical sections yourself
- `MVar`: *synchronised* mutable references
  - Like `IORefs`, but with a *lock* attached for safe concurrent access
  - …plus some very useful semantics

IORefs

- In most languages variables are mutable by default
- In Haskell, mutable variables must be handled explicitly
  - Notice that whether a variable is mutable is now reflected in its type!

```
import Data.IORef

newIORef   :: a -> IO (IORef a)
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

IORefs: Example

- Shared state concurrency using IORef

```haskell
import Control.Concurrent
import Data.IORef

main :: IO ()
main = do
  ref <- newIORef 0

  forkIO $ writeIORef ref 1
  forkIO $ writeIORef ref 2

  result <- readIORef ref   -- ¯\_(ツ)_/¯
  print result
```

MVars

- Synchronising variables for communication between concurrent threads
  - An `MVar` is a box that is either *empty* or *full*
  - `takeMVar` removes the value from the box; blocks if it is currently empty
  - `putMVar` puts a value in the box; blocks if it is currently full
  - readMVar reads the current value without removing it (and blocks if empty)

```
import Control.Concurrent

newMVar      :: a -> IO (MVar a)
newEmptyMVar :: IO (MVar a)
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()
readMVar     :: MVar a -> IO a
```

MVars

- Synchronising variables for communication between concurrent threads

```haskell
import Control.Concurrent

main :: IO ()
main = do
  m <- newEmptyMVar
  forkIO $ do
    putMVar m "hello"
    putMVar m "world"

  x <- takeMVar m
  putStr x
  putStr ", "
  y <- takeMVar m
  putStr y
```

MVars

- The runtime system can (sometimes) detect when a group of threads are deadlocked
  - Only a conservative approximation to the future behaviour of the program
  - Can be useful for debugging (but don't rely on it)

```haskell
main :: IO ()
main = do
  m <- newEmptyMVar
  takeMVar m
```

```
$ ./Test
Test: thread blocked indefinitely in an MVar operation
```

MVars

- If multiple threads are blocked in `takeMVar` or `putMVar`, a *single thread* is woken up in FIFO order: fairness
- If `readMVar` blocks, it will receive the *next* put value
- Other useful operations
  - `withMVar` can be used to protect critical sections (read the docs!)

```
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()
readMVar     :: MVar a -> IO a

withMVar     :: MVar a -> (a -> IO b) -> IO b
```

An MVar is…

- A lock
  - `MVar ()` behaves as a lock: full is unlocked, empty is locked
  - Can be used as a mutex to protect some shared state or critical section
- A one-place channel
  - For passing messages between threads
  - An asynchronous channel with a buffer size of one
- A container for shared mutable data
- A *building block* for constructing larger concurrent data structures

MVars as a building block (I)

Asynchronous computations

Asynchronous computations

- The goal:
  - Want a way to run computations asynchronously and wait for their results
  - Cancel running computations
  - Basic interface:

```
data Async a

runAsync :: IO a -> IO (Async a)
wait     :: Async a -> IO a
poll     :: Async a -> IO (Maybe a)
cancel   :: Async a -> IO ()
```

runAsync

- Perform an action *asynchronously*, and later wait for the results

```
data Async a = Async ThreadId (MVar a)

runAsync :: IO a -> IO (Async a)
runAsync action = do
  var <- newEmptyMVar
  tid <- forkIO $ do
    res <- action
    putMVar var res

  return (Async tid var)
```

wait, cancel and poll

- Wait for the computation to complete or cancel it

```haskell
wait :: Async a -> IO a
wait (Async _ var) = readMVar var

cancel :: Async a -> IO ()
cancel (Async tid _) = killThread tid

poll :: Async a -> IO (Maybe a)
poll (Async _ var) = tryReadMVar var
```

Next time

- Non-blocking algorithms
- IORefs and MVars as building blocks

- You can now implement the first two parts of IBAN: count and list

- Next week Monday: no lectures because of the protests