IT UNIVERSITY OF COPENHAGEN

Functional Programming

Patrick Bahr

Asynchronous & Parallel Programming

What you learned in this course

- Core principles of functional programming:
  - (Recursive) Algebraic data types
  - Higher-order and recursive functions
  - Pattern matching
  - Polymorphic types and functions

What you learned in this course

- Core principles of functional programming:
  - (Recursive) Algebraic data types
  - Higher-order and recursive functions
  - Pattern matching
  - Polymorphic types and functions
- Runtime behaviour of functional programs:
  - Tail recursion (using accumulators & continuations)
  - Lazy evaluation (e.g. in sequences)

What you learned in this course

- Advanced methods for structuring code:
  - Monads (to propagate errors & state)
  - Parser combinators

What you learned in this course

- Advanced methods for structuring code:
  - Monads (to propagate errors & state)
  - Parser combinators
- F#-specific features:
  - Module system and interface files
  - Imperative features (loops, references)
  - Computation expressions (→ monads, sequences, ...)

This Week

Asynchronous & Parallel Programming

Overview

1. Background & Terminology
2. Asynchronous programming
3. Parallel programming
4. Message-based Synchronisation

Part I

Background & Terminology

Sequential vs. Parallel Computing

- A **sequential computation** executes a single program step by step.

[5; 8; 2]

Sequential vs. Parallel Computing

- A **sequential computation** executes a single program step by step.

$$[5; 8; 2] \xrightarrow{+1} [6; 8; 2] \xrightarrow{+1} [6; 9; 2] \xrightarrow{+1} [6; 9; 3]$$

Sequential vs. Parallel Computing

- A **sequential computation** executes a single program step by step.

$$[5;\ 8;\ 2] \xrightarrow{+1} [6;\ 8;\ 2] \xrightarrow{+1} [6;\ 9;\ 2] \xrightarrow{+1} [6;\ 9;\ 3]$$

- A **parallel computation** executes several programs or instructions at the same time:

$$\begin{array}{ccccccc} [5 & ; & 8 & ; & 2] \\ \downarrow{\scriptstyle +1} & & \downarrow{\scriptstyle +1} & & \downarrow{\scriptstyle +1} \\ [6 & ; & 9 & ; & 3] \end{array}$$

Why parallel programming?

- Until 2004, CPUs became faster every year

⇒ Sequential software became faster every year

- Today, CPU clock speed is ca. 3 - 4 GHz as in 2004
  - So sequential software has not become much faster*

Why parallel programming?

- Until 2004, CPUs became faster every year

  ⇒ Sequential software became faster every year
- Today, CPU clock speed is ca. 3 - 4 GHz as in 2004
  - So sequential software has not become much faster*

* massive simplification, but speed improvements are much harder to obtain

Why parallel programming?

- Until 2004, CPUs became faster every year

⇒ Sequential software became faster every year

- Today, CPU clock speed is ca. 3 - 4 GHz as in 2004
  - So sequential software has not become much faster*
- Instead, we get
  - **Multicore**: 2, 4, 8, ... CPUs on a chip
  - **Vector instructions** (SIMD) built into CPUs
  - Super-parallel **Graphics Processing Units** (GPU)

* massive simplification, but speed improvements are much harder to obtain

Moore’s Law

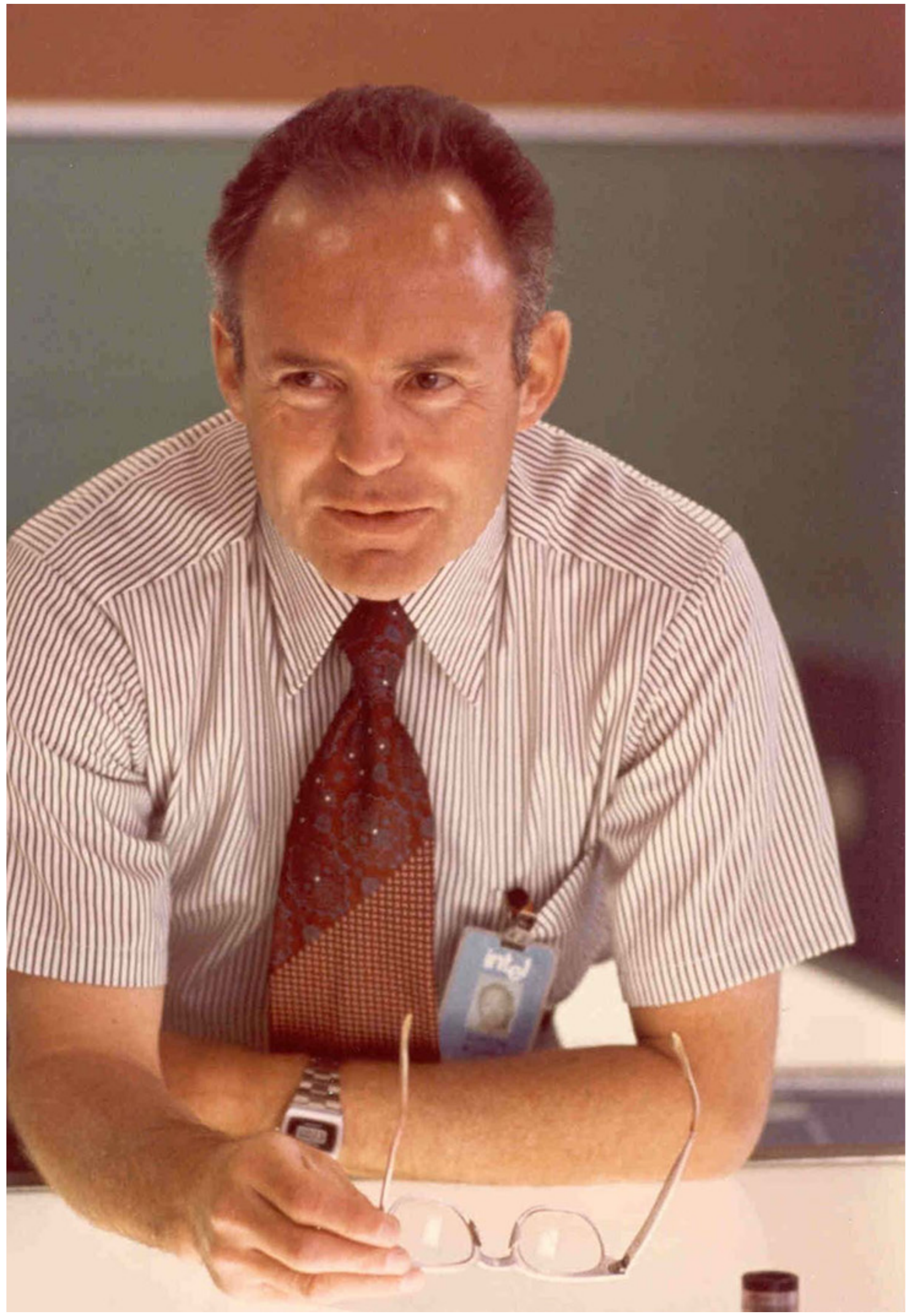
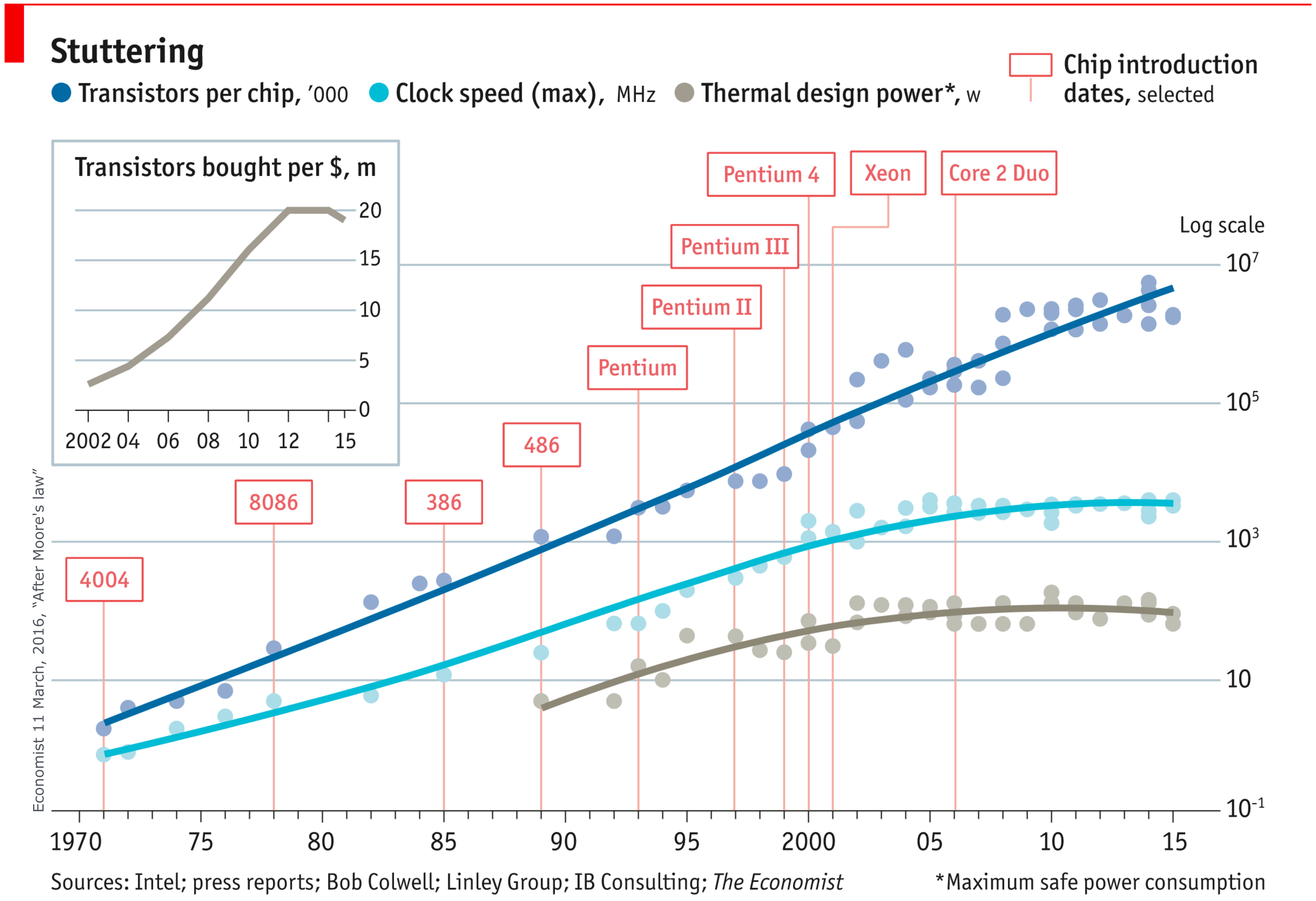


Photo credit: Intel Corporation

Gordon Moore (1929 - 2023)

IT UNIVERSITY OF COPENHAGEN

Moore’s Law



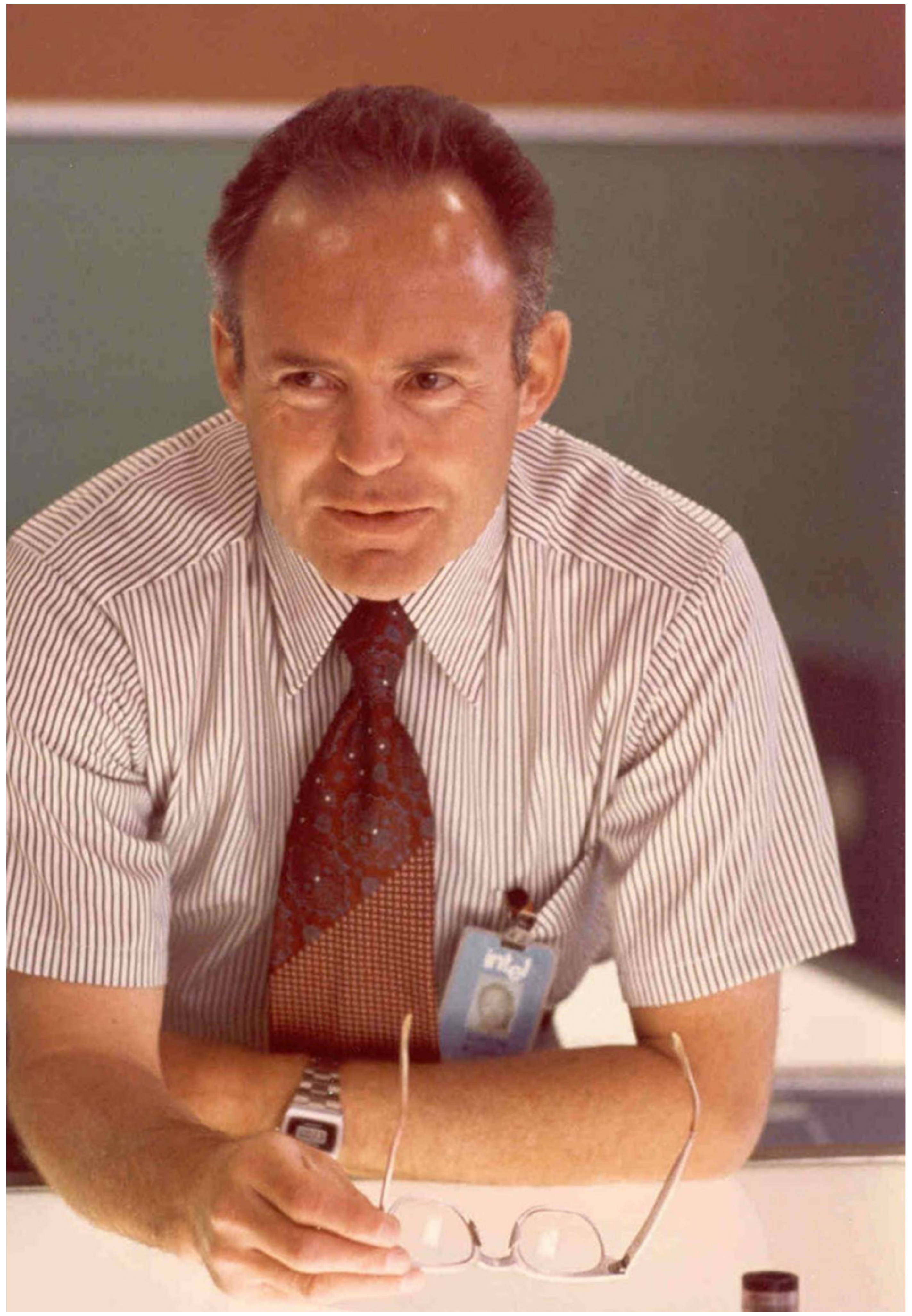
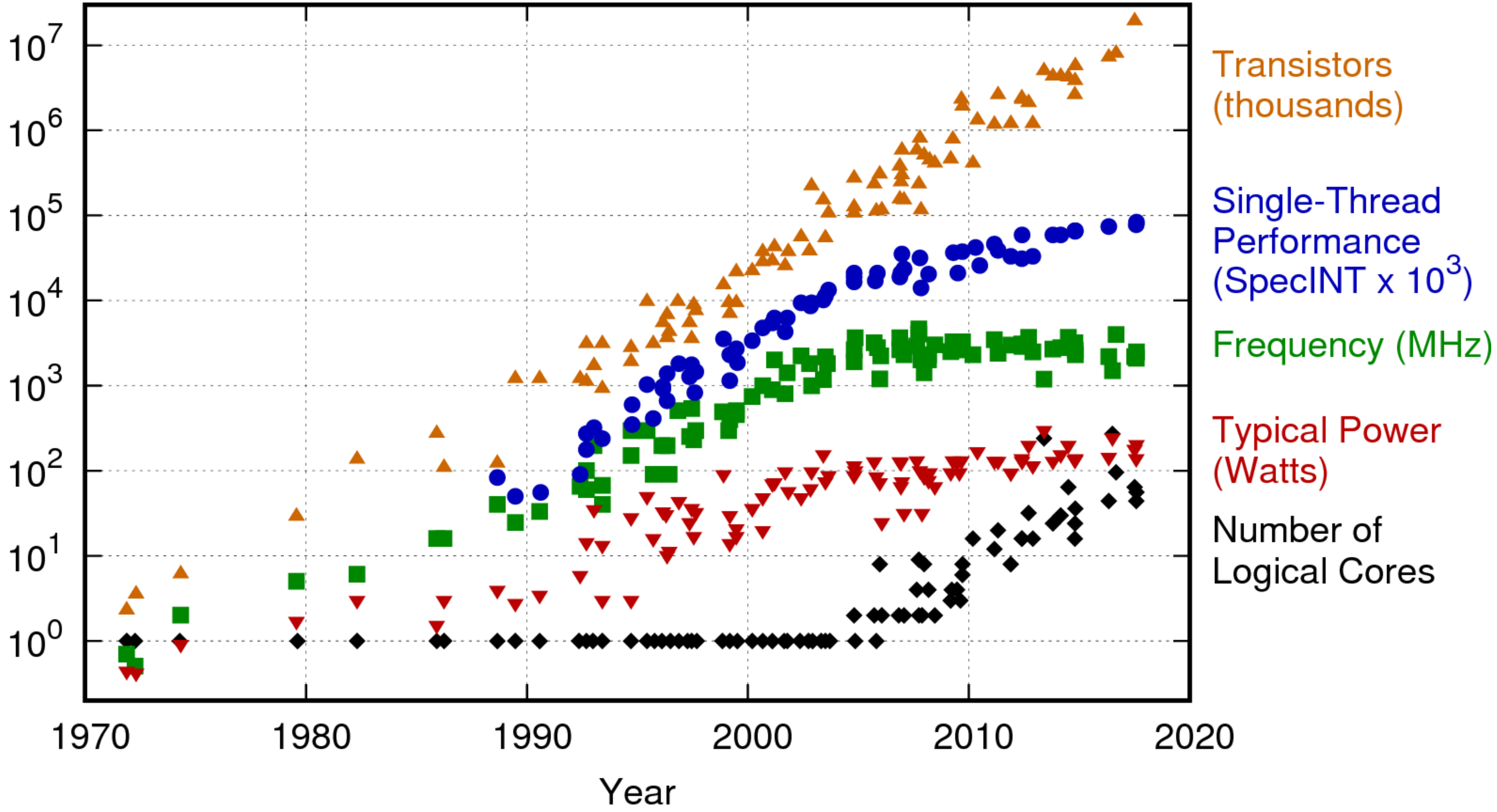


Photo credit: Intel Corporation

Gordon Moore (1929 - 2023)

More GHz → More Cores



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Parallel programming is hard

- **Thus:** More speed requires **parallel programming**
- **But:** parallel programming is **difficult and error-prone**
  - especially when working with **low-level abstractions** (threads, explicit synchronisation, processes etc.)
  - **Instead:** work with **high-level abstractions** that hide the details, which are hard to get right (not unlike what we did with monads...)
- The right abstractions are important: see Mozilla's Rust

Terminology

- **Processes:** computations that the OS may run in parallel.

  Typically, one per application

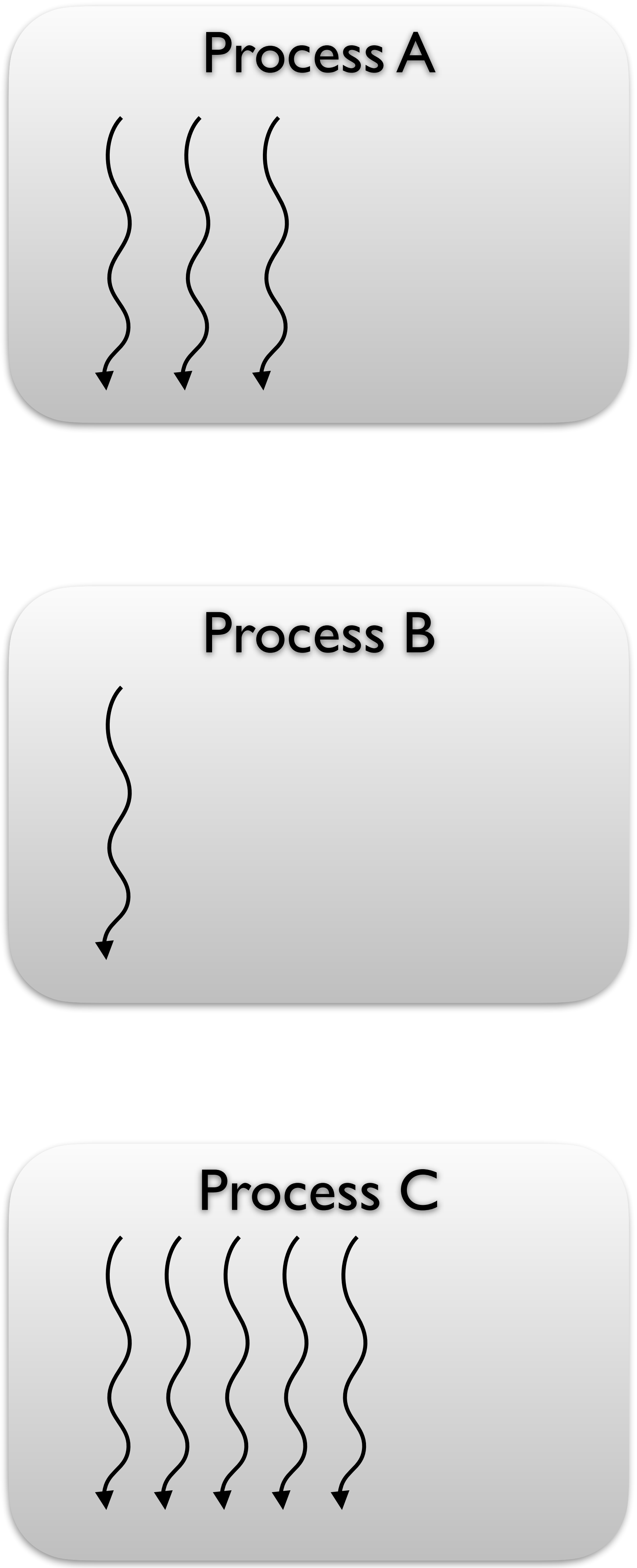
Terminology

- **Processes:** computations that the OS may run in parallel.

  Typically, one per application

- **Threads:** computations that can run in parallel inside a process

  Each thread has its **own stack,** but **shares the heap** with other threads in the process



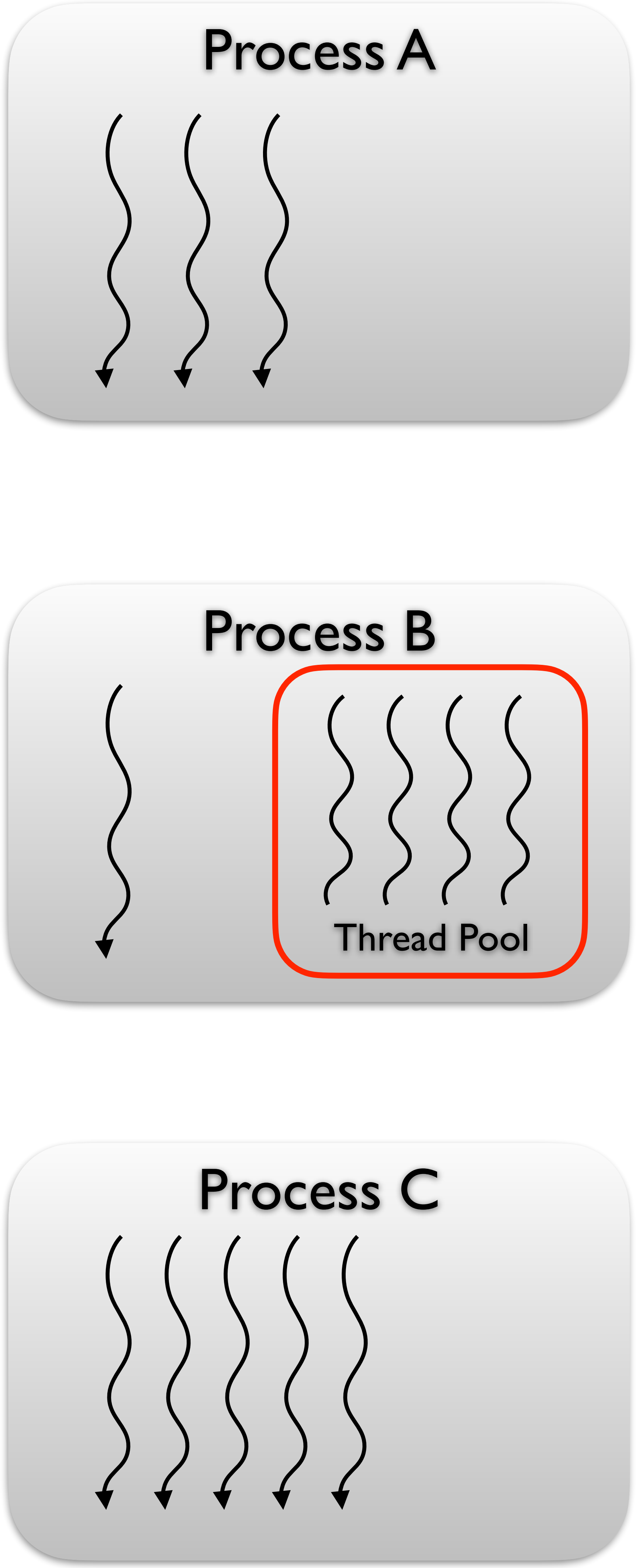
Terminology

- **Processes:** computations that the OS may run in parallel.

  Typically, one per application
- **Threads:** computations that can run in parallel inside a process

  Each thread has its **own stack,** but **shares the heap** with other threads in the process
- **Thread pool:** collection of threads that can be given **tasks** to execute

  After task is finished, its thread is returned to thread pool



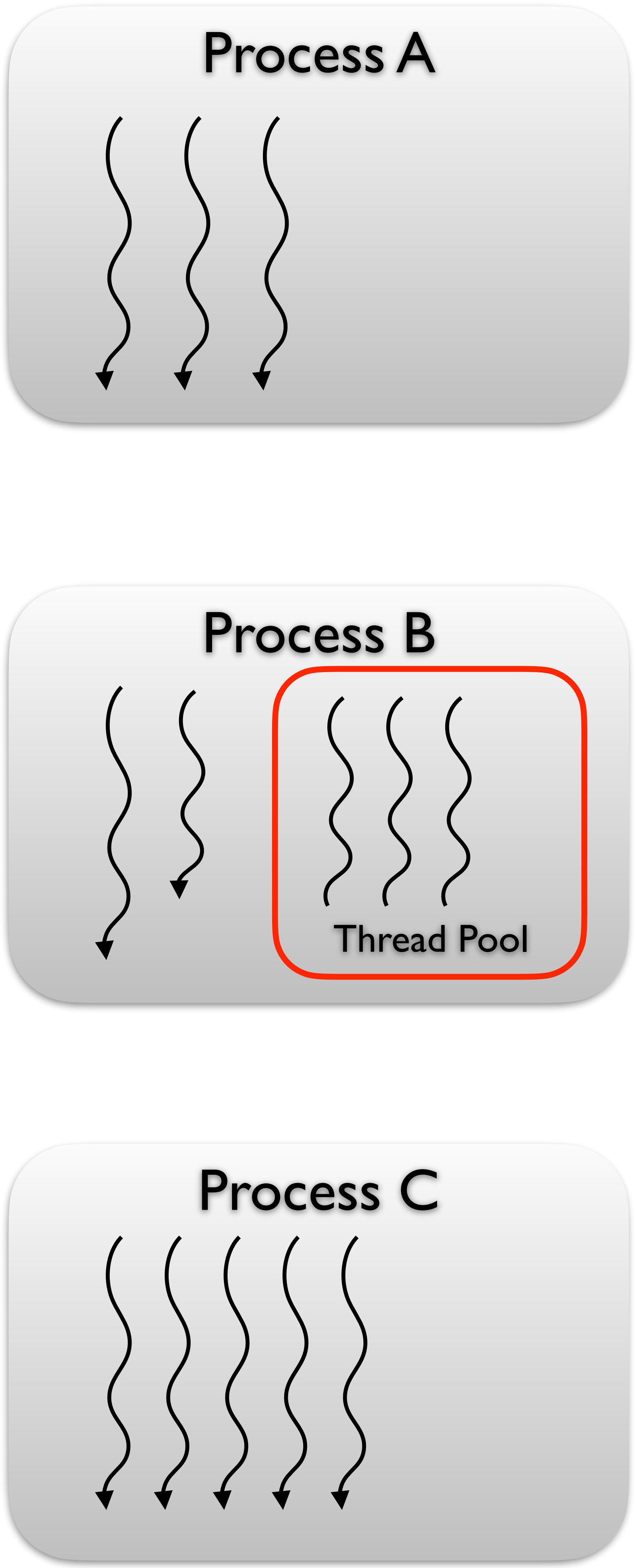
Terminology

- **Processes:** computations that the OS may run in parallel.

  Typically, one per application
- **Threads:** computations that can run in parallel inside a process

  Each thread has its **own stack,** but **shares the heap** with other threads in the process
- **Thread pool:** collection of threads that can be given **tasks** to execute

  After task is finished, its thread is returned to thread pool



Terminology

- **Processes:** computations that the OS may run in parallel.

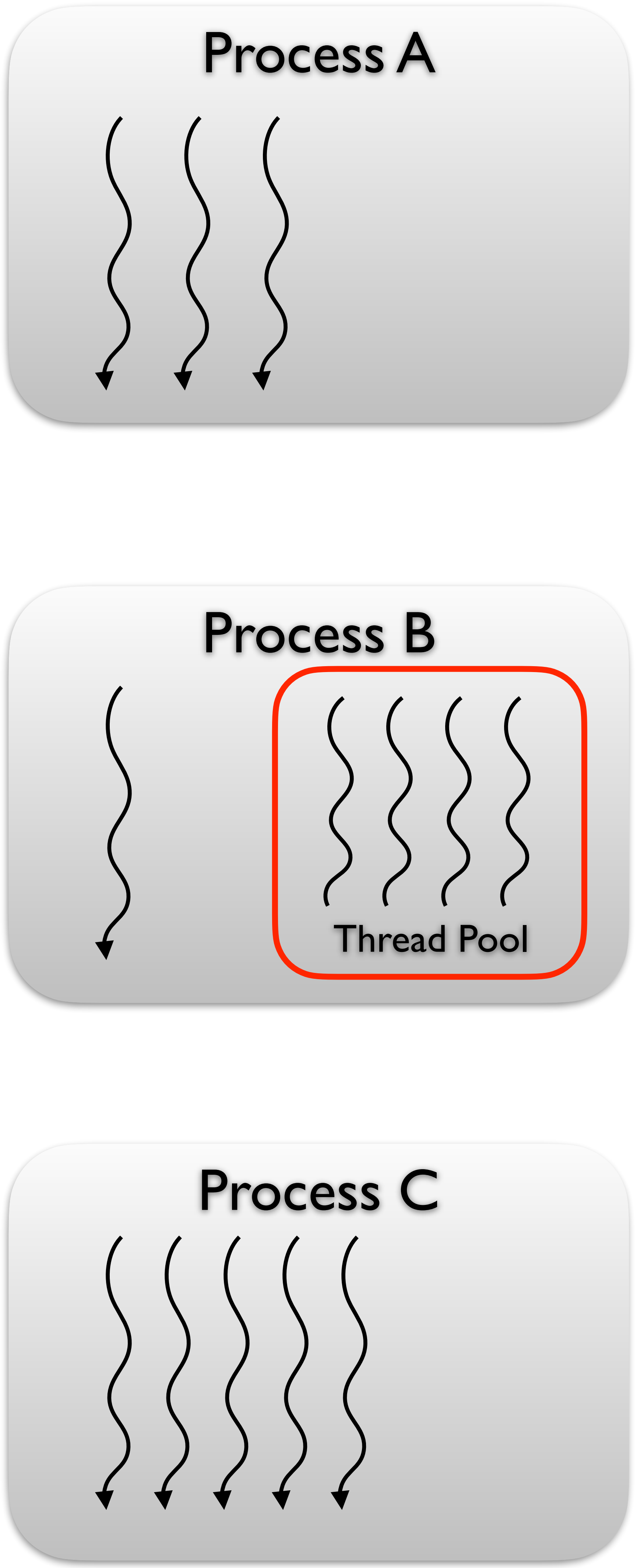
  Typically, one per application

- **Threads:** computations that can run in parallel inside a process

  Each thread has its **own stack,** but **shares the heap** with other threads in the process

- **Thread pool:** collection of threads that can be given **tasks** to execute

  After task is finished, its thread is returned to thread pool



Synchronous vs. Asynchronous

This distinction is typically made for **I/O operations**, that have to **wait** for input to arrive.

Example: Reading the contents of a file.

Synchronous vs. Asynchronous

This distinction is typically made for **I/O operations**, that have to **wait** for input to arrive.

Example: Reading the contents of a file.

Synchronous operations

halt execution and only continue after the input/ output has arrived

Synchronous vs. Asynchronous

This distinction is typically made for **I/O operations**, that have to **wait** for input to arrive.

Example: Reading the contents of a file.

Synchronous operations

halt execution and only continue after the input/ output has arrived

Asynchronous operations

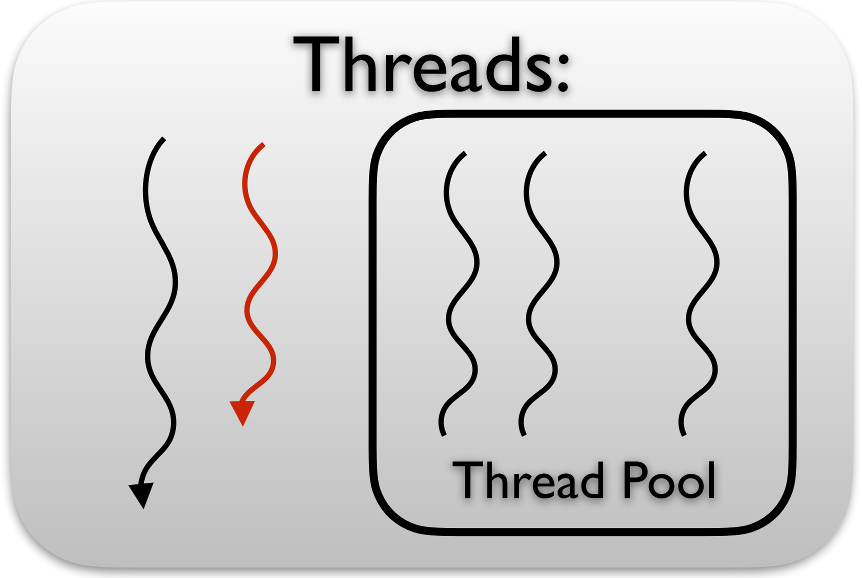
provide the opportunity for other tasks to be executed while waiting for I/O

Asynchronous Operations

How do asynchronous operations work?

Instead of blocking the CPU until operation complete:

1. The executing thread is returned to thread pool. (Another thread may then take its place.)

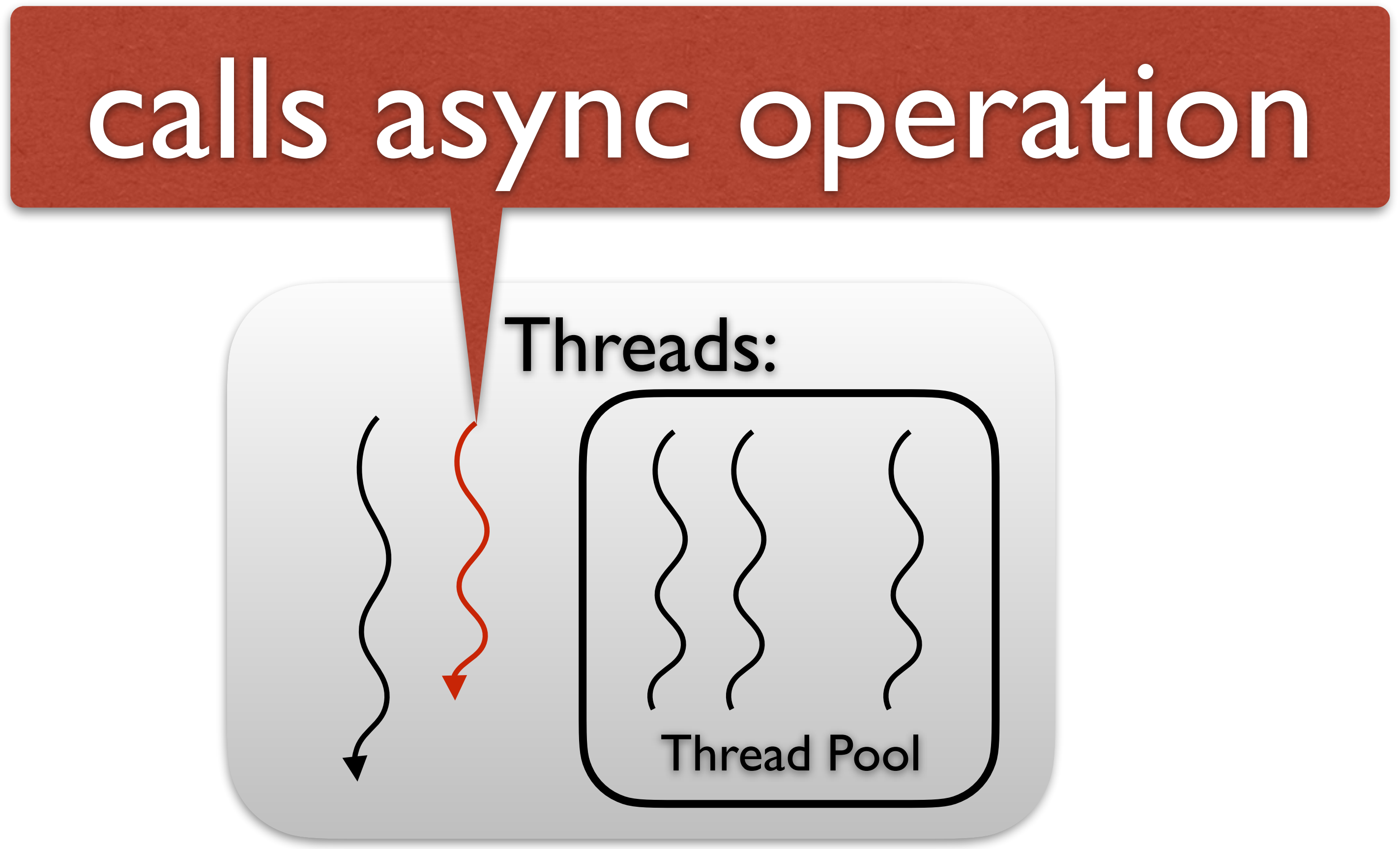


Asynchronous Operations

How do asynchronous operations work?

Instead of blocking the CPU until operation complete:

1. The executing thread is returned to thread pool. (Another thread may then take its place.)

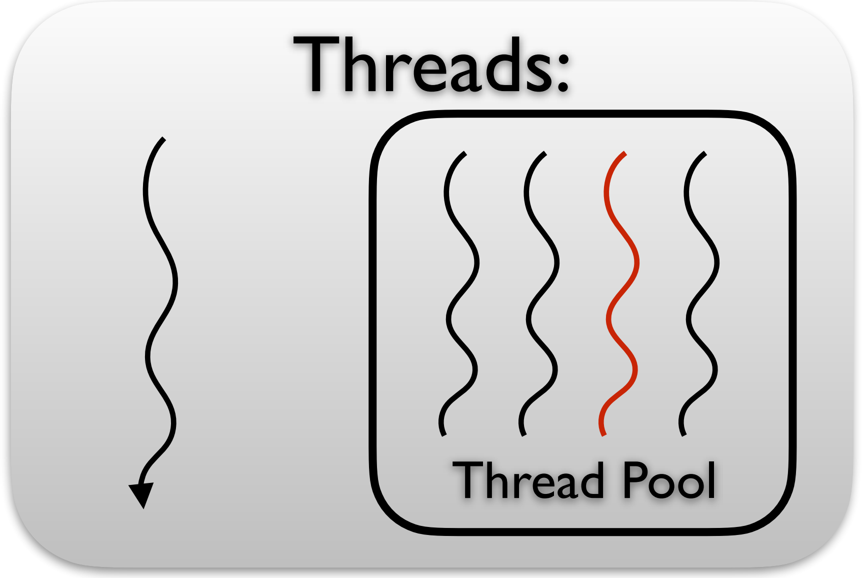


Asynchronous Operations

How do asynchronous operations work?

Instead of blocking the CPU until operation complete:

1. The executing thread is returned to thread pool. (Another thread may then take its place.)

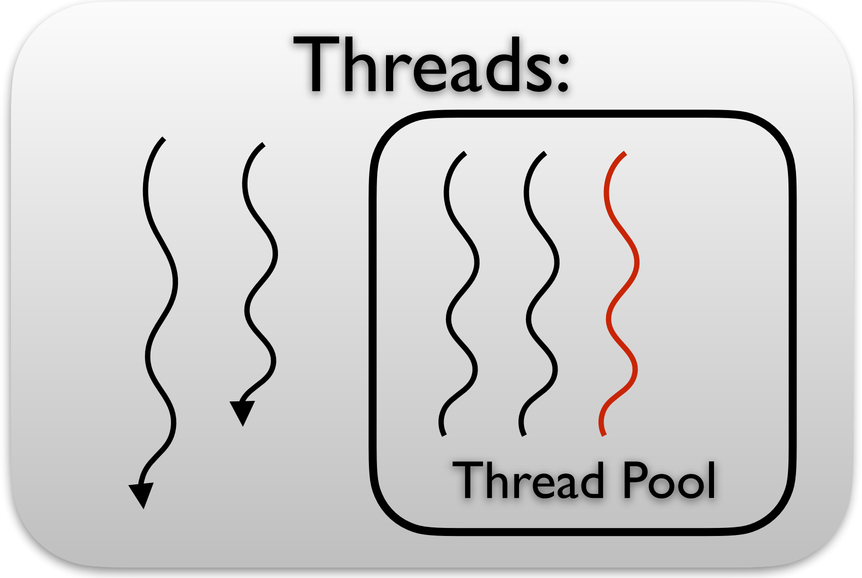


Asynchronous Operations

How do asynchronous operations work?

Instead of blocking the CPU until operation complete:

1. The executing thread is returned to thread pool. (Another thread may then take its place.)

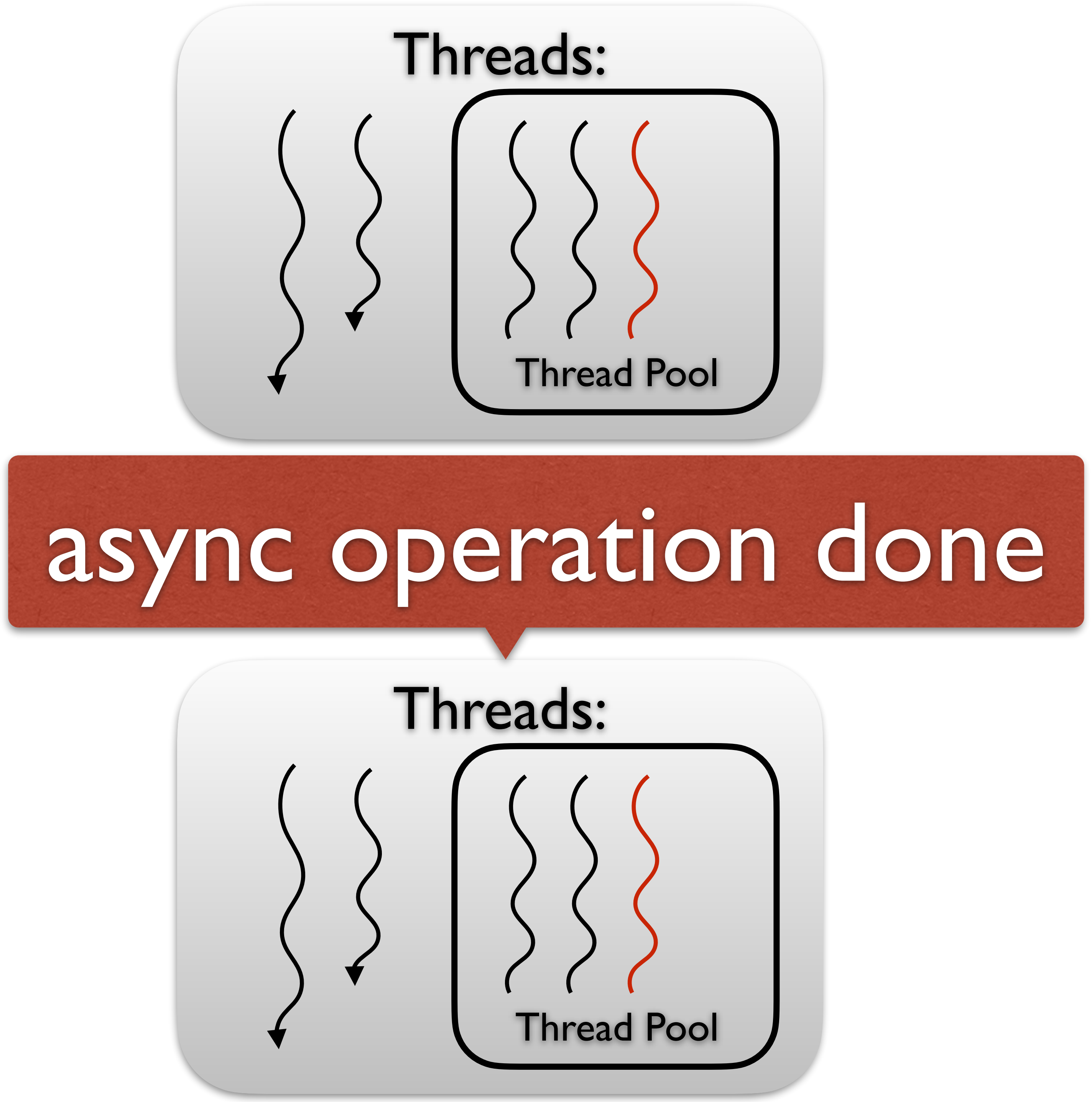


Asynchronous Operations

How do asynchronous operations work?

Instead of blocking the CPU until operation complete:

1. The executing thread is returned to thread pool. (Another thread may then take its place.)
2. When operation is completed, the task can be resumed.

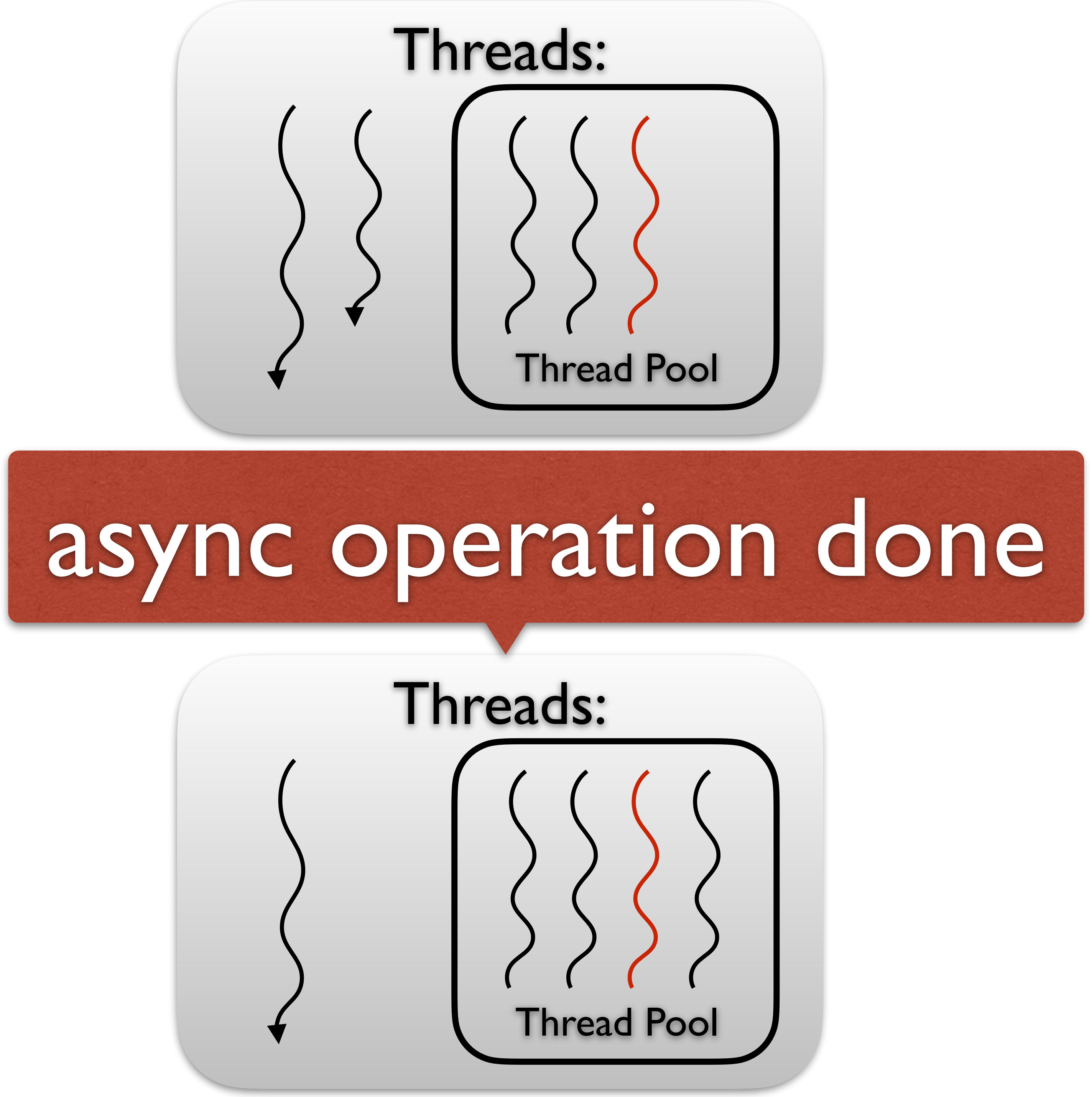


Asynchronous Operations

How do asynchronous operations work?

Instead of blocking the CPU until operation complete:

1. The executing thread is returned to thread pool. (Another thread may then take its place.)
2. When operation is completed, the task can be resumed.

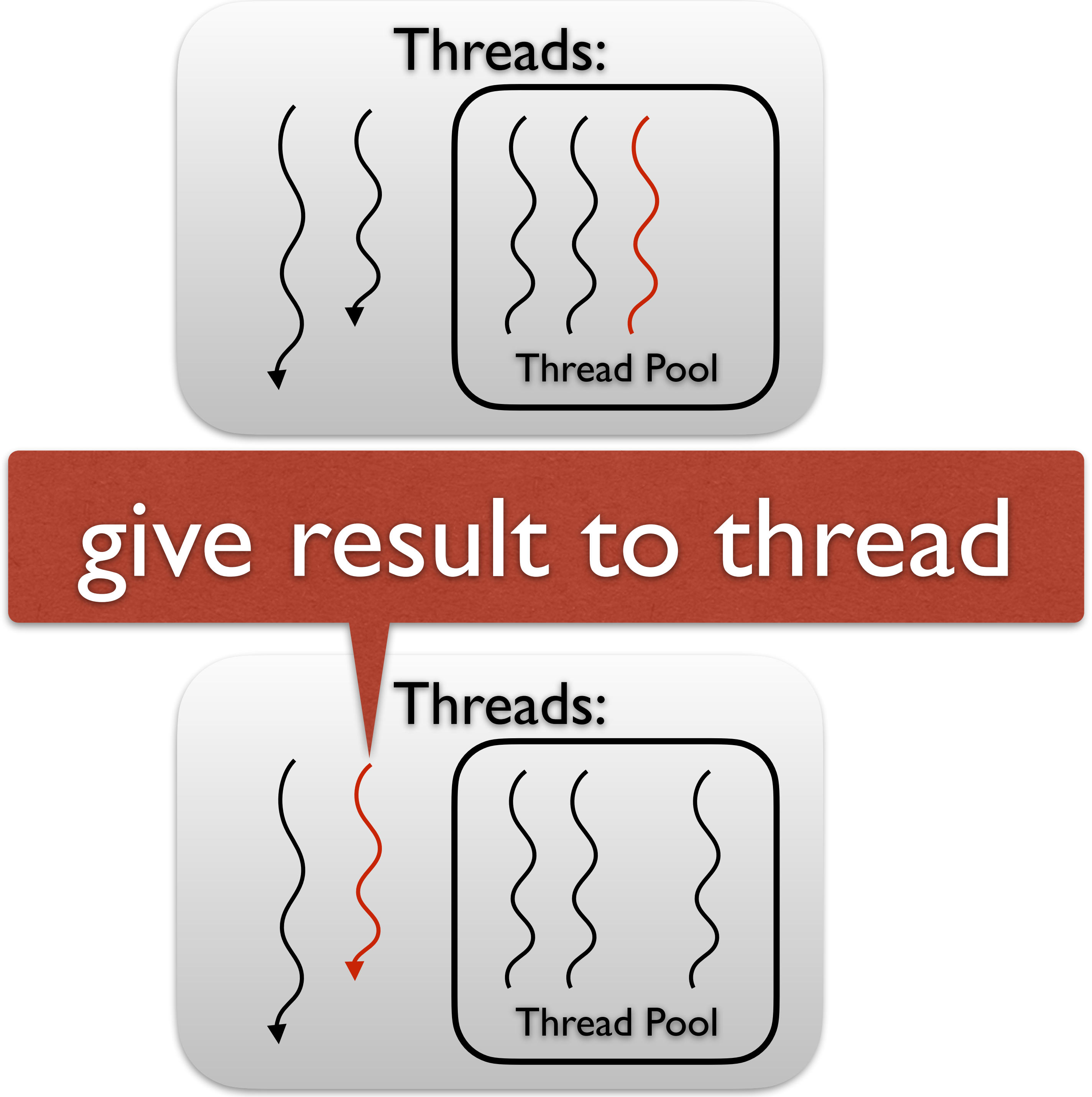


Asynchronous Operations

How do asynchronous operations work?

Instead of blocking the CPU until operation complete:

1. The executing thread is returned to thread pool. (Another thread may then take its place.)
2. When operation is completed, the task can be resumed.



Benefits of asynchronous operations

Idea of asynchronous operations:

1. The executing thread is returned to thread pool.
2. When operation is completed, the task is resumed.

Benefits of asynchronous operations

Idea of asynchronous operations:

1. The executing thread is returned to thread pool.
2. When operation is completed, the task is resumed.

Benefits:

⇒ Other tasks can be performed by the CPU while waiting

⇒ Uses very little overhead vs. synchronous operations

IT UNIVERSITY OF COPENHAGEN

Summary

- **parallel vs. sequential** execution

Summary

- **parallel vs. sequential** execution
- We **need parallel programming** to make use of multicore CPUs

Summary

- **parallel vs. sequential** execution
- We **need parallel programming** to make use of multicore CPUs
- **synchronous vs. asynchronous** operations

Summary

- **parallel vs. sequential** execution
- We **need parallel programming** to make use of multicore CPUs
- **synchronous vs. asynchronous** operations
- terminology of parallelism: **process**, **thread**, **thread pool**, **task**

Part II

Asynchronous Programming

Asynchronous Computations

`Async<T>` — Type of **asynchronous computations** that return a value of type T

`async { … }` — Computation expression to program asynchronous computations of type `Async<T>`

Asynchronous Computations

`Async<T>` — Type of **asynchronous computations** that return a value of type T

`async { … }` — Computation expression to program asynchronous computations of type `Async<T>`

Example

```
let doesItHaveFacebook url : Async<bool> =
    let webCl = new System.Net.WebClient()
    async { let! html = webCl.AsyncDownloadString(Uri url)
            let hasFB = html.IndexOf("facebook.com") >= 0
            return hasFB }
```

Asynchronous Computations

<u>Recall:</u> a value `c : unit -> T` is a computation that when executed produces a value of type `T`.

we execute the computation by writing: `c ()`

Asynchronous Computations

Recall: a value `c : unit -> T` is a computation that when executed produces a value of type T.

we execute the computation by writing: `c ()`

Similarly, a value `a : Async<T>` is a computation that when executed produces a value of type T.

Asynchronous Computations

Recall: a value `c : unit -> T` is a computation that when executed produces a value of type T.

we execute the computation by writing: `c ()`

Similarly, a value `a : Async<T>` is a computation that when executed produces a value of type T.

we execute the computation by writing:

```
Async.RunSynchronously a
```

Asynchronous Computations

IT UNIVERSITY OF COPENHAGEN

Recall: a value … tation that when executed produ…

we execu… ng: `c ()`

More Importantly

`Async` provides support for

1. asynchronous operations
2. parallel computations

Similarly, a value `a : Async<T>` is a computation that when executed produces a value of type T.

we execute the computation by writing:

`Async.RunSynchronously a`

Asynchronous Operations with Async

```
webCl.DownloadString : Uri -> string
```

```
webCl.AsyncDownloadString : Uri -> Async<string>
```

```
let doesItHaveFacebook url : Async<bool> =
let webCl = new System.Net.WebClient()
   async { let! html = webCl.AsyncDownloadString(Uri url)
           let hasFB = html.IndexOf("facebook.com") >= 0
           return hasFB }
```

Asynchronous Operations with Async


```
webCl.DownloadString : Uri -> string
```

blocks the current thread until download complete

```
webCl.AsyncDownloadString : Uri -> Async<string>
```

```
let doesItHaveFacebook url : Async<bool> =
let webCl = new System.Net.WebClient()
   async { let! html = webCl.AsyncDownloadString(Uri url)
           let hasFB = html.IndexOf("facebook.com") >= 0
           return hasFB }
```

IT UNIVERSITY OF COPENHAGEN

Asynchronous Operations with Async

```
webCl.DownloadString : Uri -> string
```

blocks the current thread until download complete

```
webCl.AsyncDownloadString : Uri -> Async<string>
```

does not block: while waiting for the download to complete, the thread is returned to thread pool

```
let doesItHaveFacebook url : Async<bool> =
let webCl = new System.Net.WebClient()
   async { let! html = webCl.AsyncDownloadString(Uri url)
           let hasFB = html.IndexOf("facebook.com") >= 0
           return hasFB }
```

Asynchronous Operations with Async


```
webCl.DownloadString : Uri -> string
```

blocks the current thread until download complete

```
webCl.AsyncDownloadString : Uri -> Async<string>
```

does not block: while waiting for the download to complete, the thread is returned to thread pool

Question: Why is there no '!' here?

```
let doesItHaveFacebook url : Async<bool> =
let webCl = new System.Net.WebClient()
   async { let! html = webCl.AsyncDownloadString(Uri url)
           let hasFB = html.IndexOf("facebook.com") >= 0
           return hasFB }
```

Asynchronous Operations with Async

```
webCl.DownloadString : Uri -> string
```

blocks the current thread until download complete

```
webCl.AsyncDownloadString : Uri -> Async<string>
```

does not block: while waiting for the download to complete, the thread is returned to thread pool

```
let doesItHaveFacebook url : Async<bool> =
let webCl = new System.Net.WebClient()
   async { let! html = webCl.AsyncDownloadString(Uri url)
           let hasFB = html.IndexOf("facebook.com") >= 0
           return hasFB }
```

Some asynchronous operations

`stream.AsyncRead: int -> Async<string>`
read first n chars from `stream`

`stream.AsyncWrite: string -> Async<unit>`
write string to `stream`

`Async.Sleep: int -> Async<unit>`
wait for n milliseconds

Parallel Computations with Async

Let’s check several websites:

```
let doTheyHaveFacebook1 : Async<bool list> = async {
    let! itu = doesItHaveFacebook "https://itu.dk"
    let! dr  = doesItHaveFacebook "https://dr.dk"
    let! pol = doesItHaveFacebook "https://politiken.dk"
    return [itu; dr; pol] }
```

Parallel Computations with Async

Let’s check several websites:

```
let doTheyHaveFacebook1 : Async<bool list> = async {
    let! itu = doesItHaveFacebook "https://itu.dk"
    let! dr  = doesItHaveFacebook "https://dr.dk"
    let! pol = doesItHaveFacebook "https://politiken.dk"
    return [itu; dr; pol] }
```

If we run this computation, it will compute **sequentially**:

```
> Async.RunSynchronously doTheyHaveFacebook1;;
val it : bool list = [true; false; true]
```

Parallel Computations with Async

Let’s check several websites:

```
let doTheyHaveFacebook2 : Async<bool []> =
    let itu = doesItHaveFacebook "https://itu.dk"
    let dr  = doesItHaveFacebook "https://dr.dk"
    let pol = doesItHaveFacebook "https://politiken.dk"
    Async.Parallel [itu; dr; pol]
```

Parallel Computations with Async

Let's check several websites:

```
let doTheyHaveFacebook2 : Async<bool []> =
    let itu = doesItHaveFacebook "https://itu.dk"
    let dr  = doesItHaveFacebook "https://dr.dk"
    let pol = doesItHaveFacebook "https://politiken.dk"
    Async.Parallel [itu; dr; pol]
```

If we run this computation, it will compute **in parallel**:

```
> Async.RunSynchronously doTheyHaveFacebook2;;
val it : bool [] = [|true; false; true|]
```

Parallel Computations with Async

```
let doTheyHaveFacebook2 : Async<bool []> =
    let itu = doesItHaveFacebook "https://itu.dk"
    let dr  = doesItHaveFacebook "https://dr.dk"
    let pol = doesItHaveFacebook "https://politiken.dk"
    Async.Parallel [itu; dr; pol]
```

```
Async.Parallel : seq<Async<'a>> -> Async<'a []>
```

Parallel Computations with Async

```
let doTheyHaveFacebook2 : Async<bool []> =
    let itu = doesItHaveFacebook "https://itu.dk"
    let dr  = doesItHaveFacebook "https://dr.dk"
    let pol = doesItHaveFacebook "https://politiken.dk"
    Async.Parallel [itu; dr; pol]
```

```
Async.Parallel : seq<Async<'a>> -> Async<'a []>
```

Or more compactly using map function:

```
let doTheyHaveFacebook3 : Async<bool []> =
    let urls = [“https://itu.dk"; "https://dr.dk";
                "https://politiken.dk"]
    urls |> List.map doesItHaveFacebook |> Async.Parallel
```

Parallel Computations with Async

```
> Async.RunSynchronously (doTheyHaveFacebook1);;
 Real: 00:00:01.655, CPU: 00:00:00.551, GC gen0: 4, gen1: 2
 val it : bool list = [true; false; true]

> Async.RunSynchronously (doTheyHaveFacebook2);;
 Real: 00:00:00.826, CPU: 00:00:00.590, GC gen0: 3, gen1: 2
 val it : bool [] = [|true; false; true|]
```

Start Async tasks asynchronously

```
let hasFacebook url : Async<unit> =
    async { let! fb = doesItHaveFacebook url
            printfn "%s: %b" url fb }
```

Start Async tasks asynchronously

```
let hasFacebook url : Async<unit> =
    async { let! fb = doesItHaveFacebook url
            printfn "%s: %b" url fb }
```

```
Async.Start : Async<unit> -> unit
```

returns immediately; does not wait for computation to complete

Start Async tasks asynchronously

```
let hasFacebook url : Async<unit> =
    async { let! fb = doesItHaveFacebook url
            printfn "%s: %b" url fb }
```

```
Async.Start : Async<unit> -> unit
```

returns immediately; does not wait for computation to complete

Example

```
Async.Start(hasFacebook "https://itu.dk")
Async.Start(hasFacebook "https://dr.dk")
Async.Start(hasFacebook "https://politiken.dk")
```

Start Async tasks asynchronously

```
let hasFacebook url : Async<unit> =
    async { let! fb = doesItHaveFacebook url
            printfn "%s: %b" url fb }
```

```
Async.Start : Async<unit> -> unit
```

returns immediately; does not wait for computation to complete

These will run in parallel.

Example

```
Async.Start(hasFacebook "https://itu.dk")
Async.Start(hasFacebook "https://dr.dk")
Async.Start(hasFacebook "https://politiken.dk")
```

Exceptions & Cancellations

asynchronous computations may terminate prematurely because

1. An **exception** was raised
2. The computation was **cancelled**

Exceptions & Cancellations

asynchronous computations may terminate prematurely because

1. An **exception** was raised
2. The computation was **cancelled**

```
Async.Start:
  Async<unit> * ?CancellationToken -> unit
```

optional argument: pass along a token we can use to cancel the computation

Example

```
let ts1 = new CancellationTokenSource()
let ts2 = new CancellationTokenSource()

Async.Start(hasFacebook "https://itu.dk", ts1.Token)
Async.Start(hasFacebook "https://dr.dk", ts2.Token)
Async.Start(hasFacebook "https://politiken.dk",ts2.Token)

ts2.Cancel()
```

IT UNIVERSITY OF COPENHAGEN

Example

```
let ts1 = new CancellationTokenSource()
let ts2 = new CancellationTokenSource()

Async.Start(hasFacebook "https://itu.dk", ts1.Token)
Async.Start(hasFacebook "https://dr.dk", ts2.Token)
Async.Start(hasFacebook "https://politiken.dk",ts2.Token)

ts2.Cancel()
```

produces the output: `https://itu.dk: true`

Reacting to Cancellations & Exceptions

You can provide **continuations** that are executed in case of cancellations/exceptions.

```
Async.StartWithContinuations :
   Async<'T>
   * ('T -> unit)
   * (exn -> unit)
   * (OperationCanceledException -> unit)
   * ?CancellationToken ->
   unit
```

Reacting to Cancellations & Exceptions

You can provide **continuations** that are executed in case of cancellations/exceptions.

```
Async.StartWithContinuations :
    Async<'T>
    * ('T -> unit)
    * (exn -> unit)
    * (OperationCanceledException -> unit)
    * ?CancellationToken ->
    unit
```

success continuation

exception continuation

cancellation continuation

Example

```
let ts1 = new CancellationTokenSource()
let ts2 = new CancellationTokenSource()

let ok (b: bool) = printf "%b" b
let ex  e        = printf "Oh no! %A" e
let can e        = printf "(╯°□°) ╯︵ ┻━┻ %A" e

Async.StartWithContinuations(doesItHaveFacebook "https://itu.dk",
                             ok, ex, can, ts1.Token)
Async.StartWithContinuations(doesItHaveFacebook "https://dr.dk",
                             ok, ex, can, ts2.Token)
Async.StartWithContinuations(doesItHaveFacebook "not-a-url",
                             ok, ex, can, ts2.Token)
ts2.Cancel();
```

Example

```
le
le

le
le
le
```

Output (not necessarily in this order)

```
Oh no! System.UriFormatException: …
(╯°□°) ╯︵ ┻━┻ System.OperationCanceledException: …
true
```

```
Async.StartWithContinuations(doesItHaveFacebook "https://itu.dk",
                             ok, ex, can, ts1.Token)
Async.StartWithContinuations(doesItHaveFacebook "https://dr.dk",
                             ok, ex, can, ts2.Token)
Async.StartWithContinuations(doesItHaveFacebook "not-a-url",
                             ok, ex, can, ts2.Token)
ts2.Cancel();
```

Summary

- **`Async<T>`** type for asynchronous computations

Summary

- **`Async<T>`** type for asynchronous computations
- **asynchronous operations** have return type `Async<T>`

Summary

- **`Async<T>`** type for asynchronous computations
- **asynchronous operations** have return type `Async<T>`
- **parallel execution** of `Async<T>` computations (`Async.Parallel`)

Summary

- **`Async<T>`** type for asynchronous computations
- **asynchronous operations** have return type `Async<T>`
- **parallel execution** of `Async<T>` computations (`Async.Parallel`)
- **asynchronous execution** of `Async<T>` computations (`Async.Start`)

Summary

- **`Async<T>`** type for asynchronous computations
- **asynchronous operations** have return type `Async<T>`
- **parallel execution** of `Async<T>` computations (`Async.Parallel`)
- **asynchronous execution** of `Async<T>` computations (`Async.Start`)
- cancellation of computations using **cancellation tokens**

Part III

Parallel Programming

IT UNIVERSITY OF COPENHAGEN

Parallelism

Data parallelism vs. **Task parallelism**

Data parallelism
same computation, but on different data

Task parallelism
different computations performed in parallel

Parallelism

Data parallelism
<u>same computation</u>,
but on different data

vs.

Task parallelism
<u>different computations</u>
performed in parallel

[5 ; 8 ; 2]

+1 +1 +1

[6 ; 9 ; 3]

same computation '+1'
applied to each number

Parallelism

Data parallelism vs. **Task parallelism**

Data parallelism
same computation, but on different data

[5 ; 8 ; 2]

+1 +1 +1

[6 ; 9 ; 3]

same computation '+1' applied to each number

Task parallelism
different computations performed in parallel

[5 ; 2]

+1 *2

[6 ; 4]

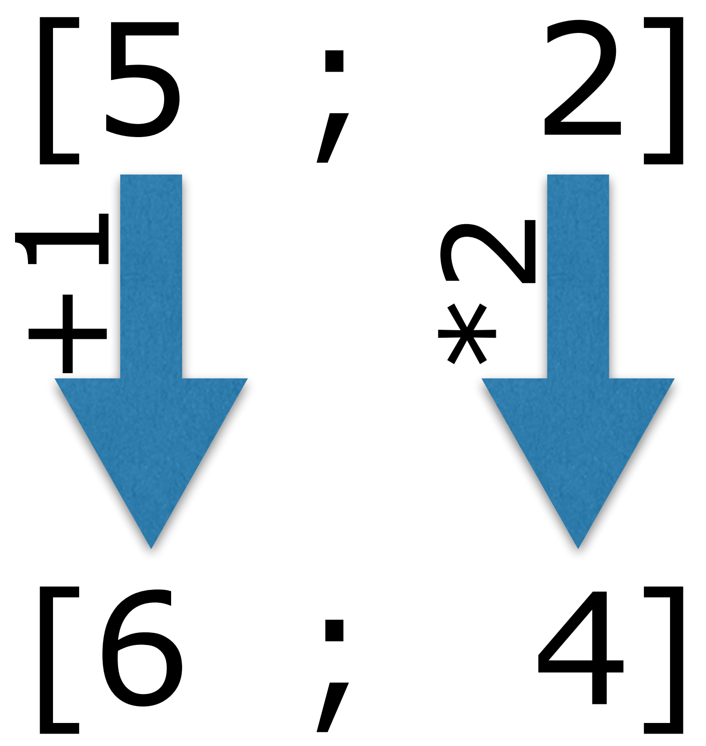
different computations are performed in parallel

Task parallelism

Async can be used for task parallelism:

```
let taskPar n m =
  Async.Parallel [async { return n + 1 } ;
                  async { return m * 2 } ]
    |> Async.RunSynchronously
```

```
> taskPar 5 2;;
 val it : int [] = [|6; 4|]
```

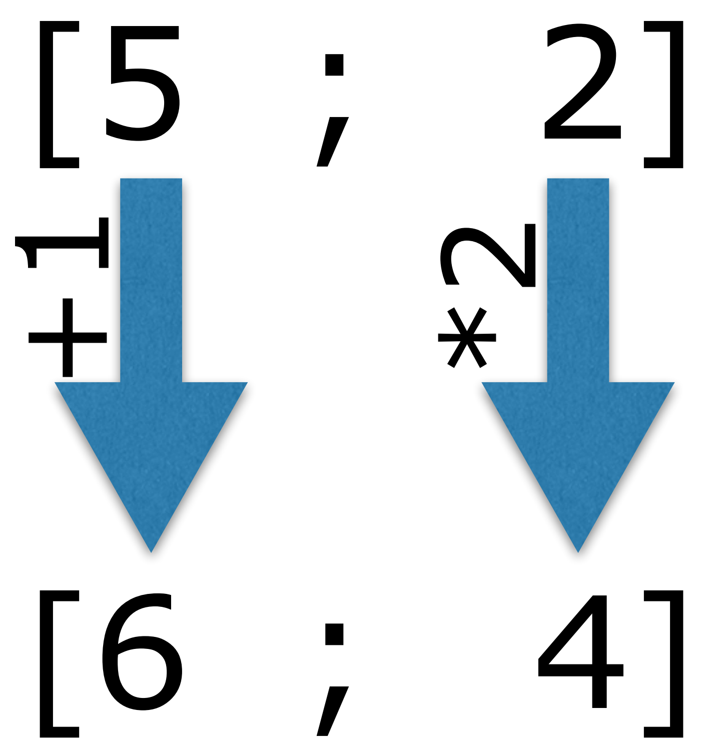


Task parallelism

Alternatively, we can use the Task module:

```
open System.Threading.Tasks

let taskPar2 n m : int * int =
   let r1 = Task.Factory.StartNew(fun () -> n + 1)
   let r2 = Task.Factory.StartNew(fun () -> m * 2)
   (r1.Result, r2.Result)
```

```
> taskPar2 5 2;;
  val it : int * int = (6, 4)
```



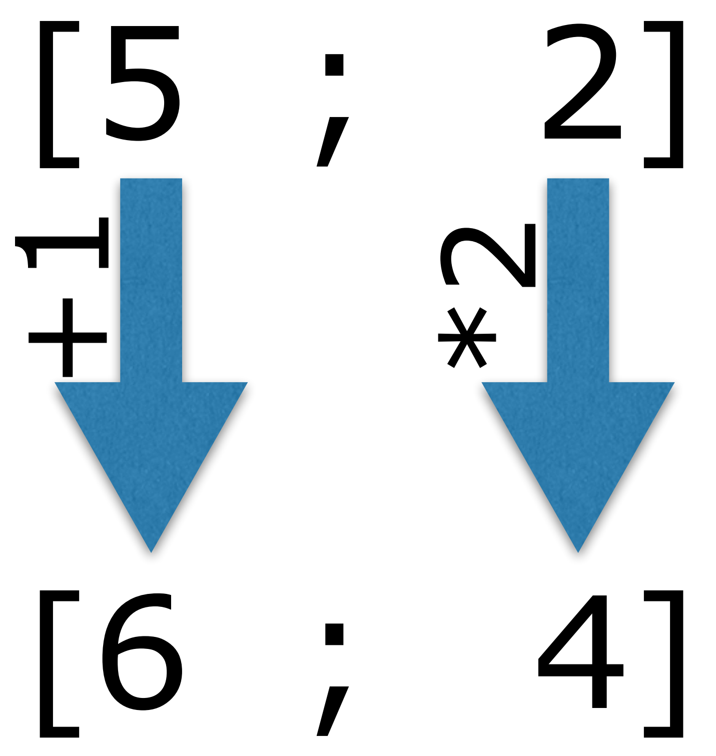
Task parallelism

Alternati e Task module:

r1 : Task<int>
r2 : Task<int>

```
open Sy          sks

let taskPar2 n m : int * int =
  let r1 = Task.Factory.StartNew(fun () -> n + 1)
  let r2 = Task.Factory.StartNew(fun () -> m * 2)
  (r1.Result, r2.Result)
```

```
> taskPar2 5 2;;
val it : int * int = (6, 4)
```



Task parallelism

Alternati … e Task module:

```
open Sy … sks

let taskPar2 n m : int * int =
  let r1 = Task.Factory.StartNew(fun () -> n + 1)
  let r2 = Task.Factory.StartNew(fun () -> m * 2)
  (r1.Result, r2.Result)
```

r1 : Task<int>
r2 : Task<int>

r2.Result : int
blocks until task done

```
> taskPar2 5 2;;
  val it : int * int = (6, 4)
```

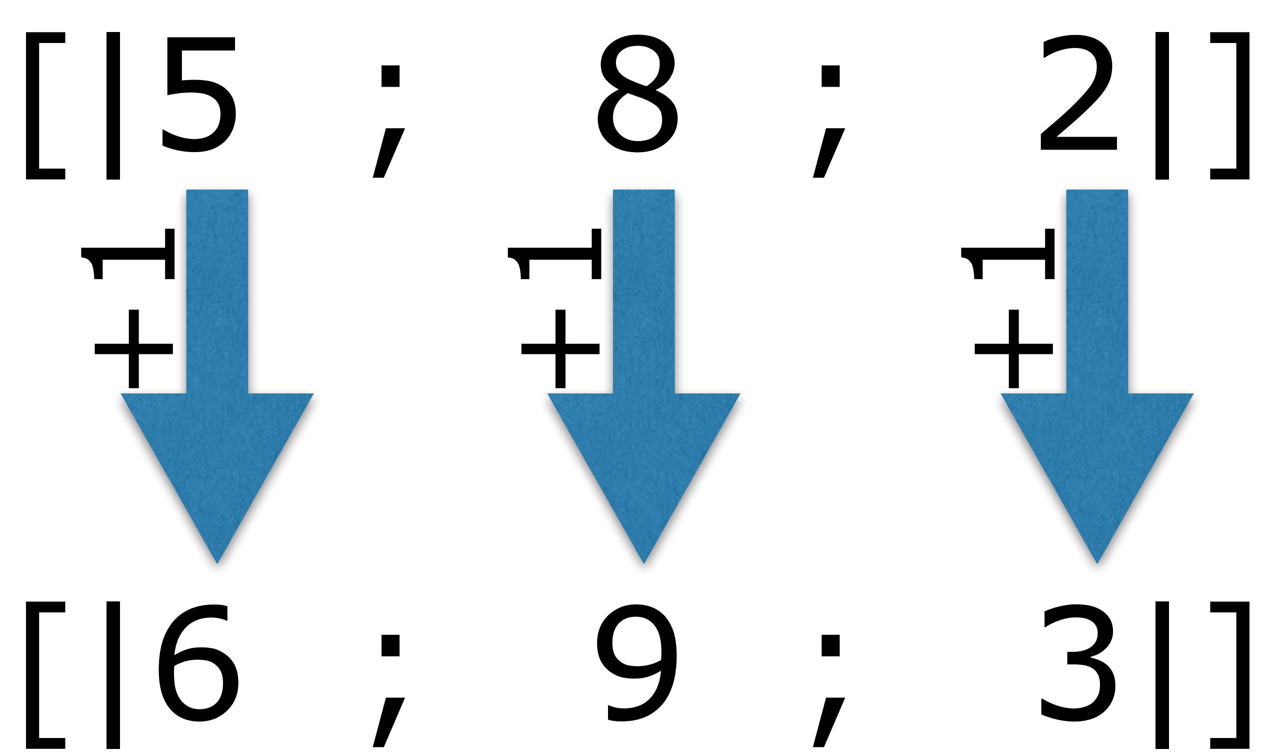
[5 ; 2]

+1 *2

[6 ; 4]

Data Parallelism

`Array.Parallel` module provides functions for data parallelism, e.g. `map`




```
map : ('T -> 'U) -> 'T [] -> 'U []
```

Example

```
> Array.Parallel.map (fun n -> n + 1) [|5; 8; 2|];;
   val it : int [] = [|6; 9; 3|]
```

Data Parallelism

Many more functions in `Array.Parallel`

```
mapi : (int -> 'T -> 'U) -> 'T [] -> 'U []
choose : ('T -> 'U option) -> 'T [] -> 'U []
collect : ('T -> 'U []) -> 'T [] -> 'U []
init : int -> (int -> 'T) -> 'T []
iter : ('T -> unit) -> 'T [] -> unit
iteri : (int -> 'T -> unit) -> 'T [] -> unit
partition : ('T -> bool) -> 'T [] -> 'T [] * 'T []
```

IT UNIVERSITY OF COPENHAGEN

Part IV

Message-based Synchronisation

Problems with Parallelism

1. Race conditions

2. Deadlock

Race conditions

- If parallel computations access a shared resource
  - e.g. mutable variable, file, network connection
- behaviour becomes unpredictable (depends on timing)

```
let mutable n = 0
seq [1..10000]
|> Seq.map (fun _ -> async{ n <- n+1 })
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
printfn "%d" n
```

Race conditions

- If parallel computations access a shared resource
  - e.g. mutable variable, file, network connection
- behaviour becomes unpredictable (depends on timing)

```
let mutable n = 0
seq [1..10000]
|> Seq.map (fun _ -> async{ n <- n+1 })
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
printfn "%d" n
```

One would expect the output to be 10000.

I got the output 9971.

I ran it again and got the output 9979.

Why?

```
let mutable n = 0
seq [1..10000]
|> Seq.map (fun _ -> async{ n <- n+1 })
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
printfn "%d" n
```

1. Store n in temp
2. Increment temp
3. Store temp in n

Why?

```
let mutable n = 0
seq [1..10000]
|> Seq.map (fun _ -> async{ n <- n+1 })
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
printfn "%d" n
```

Thread 1

1. Store n in temp
2. Increment temp
3. Store temp in n

Thread 2

1. Store n in temp2
2. Increment temp2
3. Store temp2 in n

Why?

```
let mutable n = 0
seq [1..10000]
|> Seq.map (fun _ -> async{ n <- n+1 })
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
printfn "%d" n
```

Thread 1	temp	Thread 2	temp2
1. Store n in temp		1. Store n in temp2	
2. Increment temp		2. Increment temp2	
3. Store temp in n		3. Store temp2 in n	

Why?

```
let mutable n = 0
seq [1..10000]
|> Seq.map (fun _ -> async{ n <- n+1 })
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
printfn "%d" n
```

Thread 1	temp	Thread 2	temp2
1. Store n in temp	0	1. Store n in temp2	
2. Increment temp		2. Increment temp2	
3. Store temp in n		3. Store temp2 in n	

Why?

```
let mutable n = 0
seq [1..10000]
|> Seq.map (fun _ -> async{ n <- n+1 })
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
printfn "%d" n
```

Thread 1	temp	Thread 2	temp2
1. Store n in temp	0	1. Store n in temp2	
2. Increment temp	1	2. Increment temp2	
3. Store temp in n		3. Store temp2 in n	

Why?

```
let mutable n = 0
seq [1..10000]
|> Seq.map (fun _ -> async{ n <- n+1 })
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
printfn "%d" n
```

Thread 1	temp	Thread 2	temp2
1. Store n in temp	0	1. Store n in temp2	
2. Increment temp	1	2. Increment temp2	
3. Store temp in n	1	3. Store temp2 in n	

Why?

```
let mutable n = 0
seq [1..10000]
|> Seq.map (fun _ -> async{ n <- n+1 })
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
printfn "%d" n
```

Thread 1	temp	Thread 2	temp2
1. Store n in temp	0	1. Store n in temp2	1
2. Increment temp	1	2. Increment temp2	
3. Store temp in n	1	3. Store temp2 in n	

Why?

```
let mutable n = 0
seq [1..10000]
|> Seq.map (fun _ -> async{ n <- n+1 })
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
printfn "%d" n
```

Thread 1	temp	Thread 2	temp2
1. Store n in temp	0	1. Store n in temp2	1
2. Increment temp	1	2. Increment temp2	2
3. Store temp in n	1	3. Store temp2 in n	

Why?

```
let mutable n = 0
seq [1..10000]
|> Seq.map (fun _ -> async{ n <- n+1 })
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
printfn "%d" n
```

Thread 1	temp	Thread 2	temp2
1. Store n in temp	0	1. Store n in temp2	1
2. Increment temp	1	2. Increment temp2	2
3. Store temp in n	1	3. Store temp2 in n	2

Why?

```
let mutable n = 0
seq [1..10000]
|> Seq.map (fun _ -> async{ n <- n+1 })
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
printfn "%d" n
```

Thread 1	temp	Thread 2	temp2
1. Store n in temp		1. Store n in temp2	
2. Increment temp		2. Increment temp2	
3. Store temp in n		3. Store temp2 in n	

Why?

```
let mutable n = 0
seq [1..10000]
|> Seq.map (fun _ -> async{ n <- n+1 })
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
printfn "%d" n
```

Thread 1	temp	Thread 2	temp2
1. Store n in temp	0	1. Store n in temp2	
2. Increment temp		2. Increment temp2	
3. Store temp in n		3. Store temp2 in n	

Why?

```
let mutable n = 0
seq [1..10000]
|> Seq.map (fun _ -> async{ n <- n+1 })
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
printfn "%d" n
```

Thread 1	temp	Thread 2	temp2
1. Store n in temp	0	1. Store n in temp2	0
2. Increment temp		2. Increment temp2	
3. Store temp in n		3. Store temp2 in n	

Why?

```
let mutable n = 0
seq [1..10000]
|> Seq.map (fun _ -> async{ n <- n+1 })
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
printfn "%d" n
```

Thread 1	temp	Thread 2	temp2
1. Store n in temp	0	1. Store n in temp2	0
2. Increment temp	1	2. Increment temp2	
3. Store temp in n		3. Store temp2 in n	

Why?

```
let mutable n = 0
seq [1..10000]
|> Seq.map (fun _ -> async{ n <- n+1 })
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
printfn "%d" n
```

Thread 1	temp	Thread 2	temp2
1. Store n in temp	0	1. Store n in temp2	0
2. Increment temp	1	2. Increment temp2	1
3. Store temp in n		3. Store temp2 in n	

Why?

```
let mutable n = 0
seq [1..10000]
|> Seq.map (fun _ -> async{ n <- n+1 })
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
printfn "%d" n
```

Thread 1	temp	Thread 2	temp2
1. Store n in temp	0	1. Store n in temp2	0
2. Increment temp	1	2. Increment temp2	1
3. Store temp in n	1	3. Store temp2 in n	

Why?

```
let mutable n = 0
seq [1..10000]
|> Seq.map (fun _ -> async{ n <- n+1 })
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
printfn "%d" n
```

Thread 1	temp	Thread 2	temp2
1. Store n in temp	0	1. Store n in temp2	0
2. Increment temp	1	2. Increment temp2	1
3. Store temp in n	1	3. Store temp2 in n	1

Race conditions

Also printing to console accesses a shared resource!

```
seq [1 .. 10]
|> Seq.map (fun i -> async{ printfn "I'm no. %d" i})
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
```

Race conditions

Also printing to console accesses a shared resource!

```
seq [1 .. 10]
 |> Seq.map (fun i -> async{ printfn "I'm no. %d" i})
 |> Async.Parallel
 |> Async.Ignore
 |> Async.RunSynchronously
```

This is the output we obtain:

```
I'm no. I'm no. 3I'm no. 5I'm no. 4I'm no. 21





I'm no. 9
I'm no. I'm no. 10
8
I'm no. 7
I'm no. 6
```

Locking

- <u>The problem</u> is that access to shared resources is **interleaved**
- <u>Instead</u>, we want that access is **uninterrupted** (i.e. “atomic”)

```
let l = new System.Object()
seq [1 .. 10]
|> Seq.map (fun i ->
    async{ lock l (fun _ -> printfn "I'm no. %d" i)})
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
```

Locking

- The problem is that access to shared resources is **interleaved**
- Instead, we want that access is **uninterrupted** (i.e. “atomic”)

```
let l = new System.Object()
seq [1 .. 10]
|> Seq.map (fun i ->
    async{ lock l (fun _ -> printfn "I'm no. %d" i)})
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
```

Locking

- The problem is that access to shared resources is **interleaved**
- Instead, we want that access is **uninterrupted** (i.e. “atomic”)

```
let l = new System.Object()
seq [1 .. 10]
|> Seq.map (fun i ->
    async{ lock l (fun _ -> printfn "I'm no. %d" i)})
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
```

This is executed without interruption

Locking

- <u>The problem</u> is that access to shared resources is **interleaved**
- <u>Instead</u>, we want that access is **uninterrupted** (i.e. “atomic”)

```
let l = new System.Object()
seq [1 .. 10]
|> Seq.map (fun i ->
     async{ lock l (fun _ -> printfn "I'm no.
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
```

This is executed without interruption

output:

```
I'm no. 4
I'm no. 8
I'm no. 6
I'm no. 1
I'm no. 5
I'm no. 2
I'm no. 7
I'm no. 3
I'm no. 10
I'm no. 9
```

Problem with Locking

- Only one thread can acquire the lock

```
lock l (fun _ -> printfn "I'm no. %d" i)
```

  - If access to the shared resource is frequent, threads spend much of their time waiting to acquire the lock

- Even worse: Possibility of a **deadlock**
  - E.g. program freezes if we run into this situation:
    - thread1 has lock1 and wants to acquire lock2
    - thread2 has lock2 and wants to acquire lock1

Message-based Synchronisation

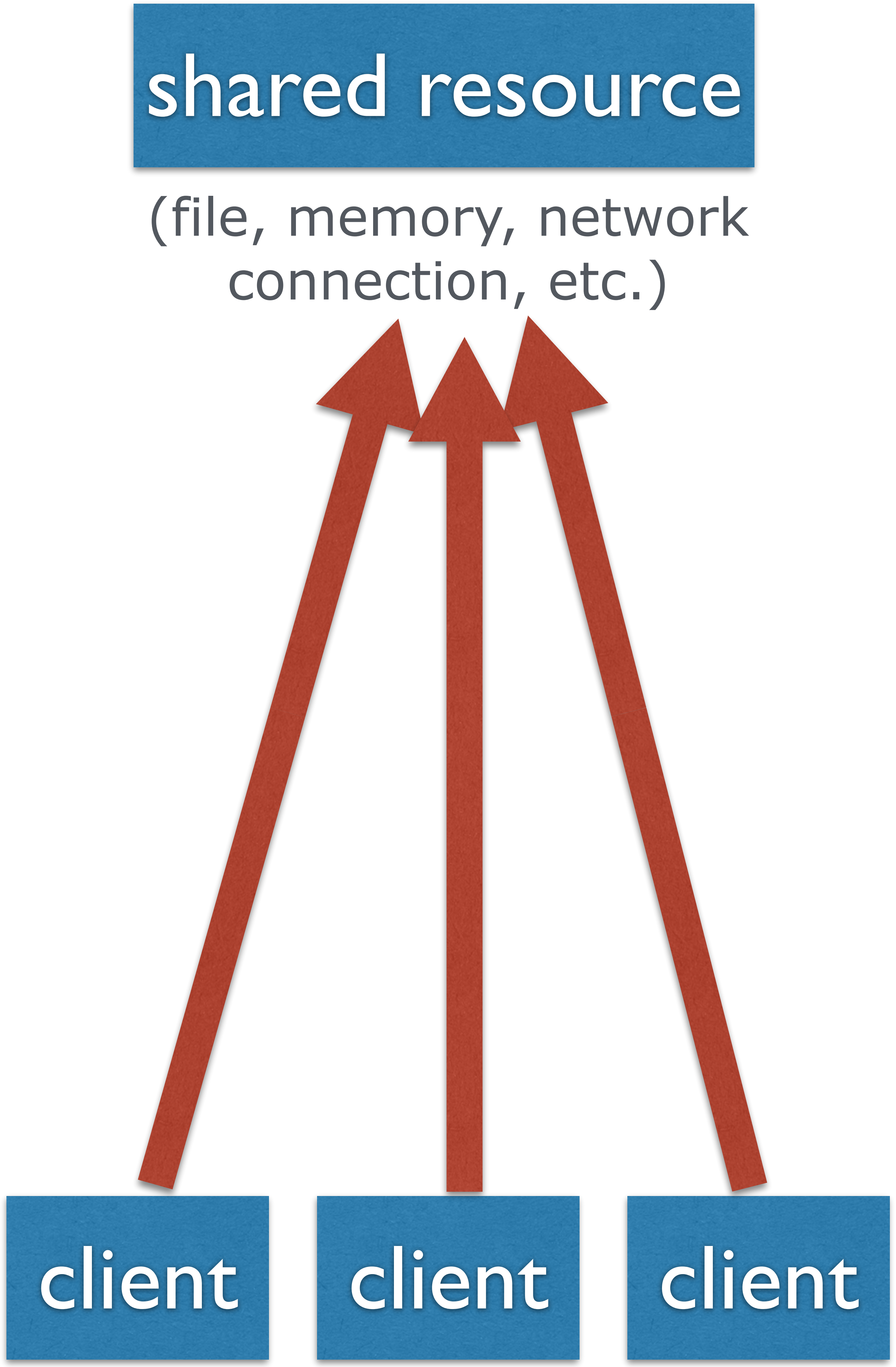
We recommend using a **message-based** approach instead of locking.

- share a single resource with several threads
- avoid deadlocks
- reduce waiting times to near 0

Message-based Synchronisation

We recommend using a **message-based** approach instead of locking.

- share a single resource with several threads
- avoid deadlocks
- reduce waiting times to near 0

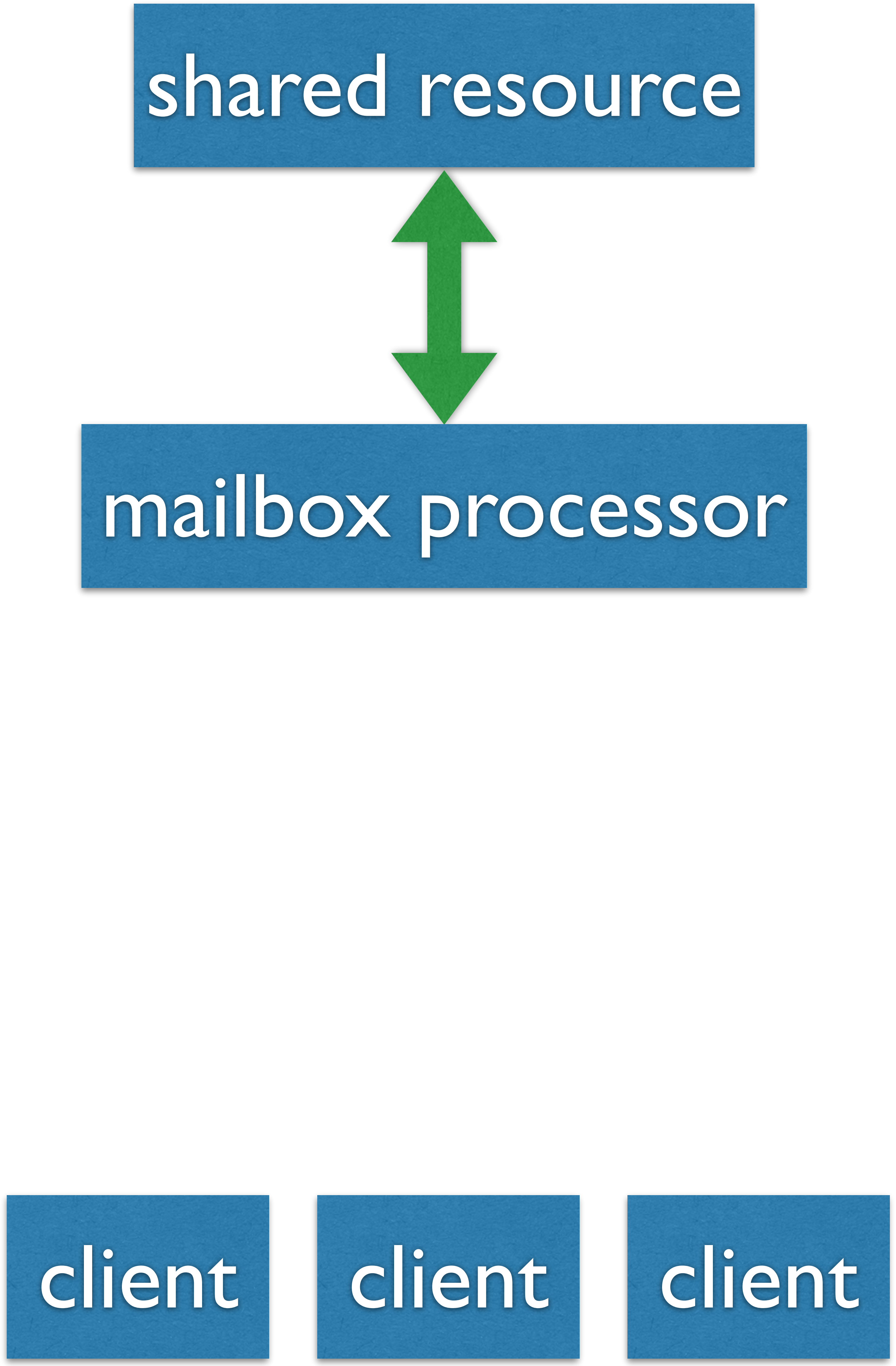


Message-based Synchronisation

- Only one thread (the **mailbox processor**) has access to the shared resource
- Other threads (the **clients**) send messages to **mailbox** requesting operations on the shared resource
- The mailbox processor works through the messages in the mailbox **one by one** performing the requested operations

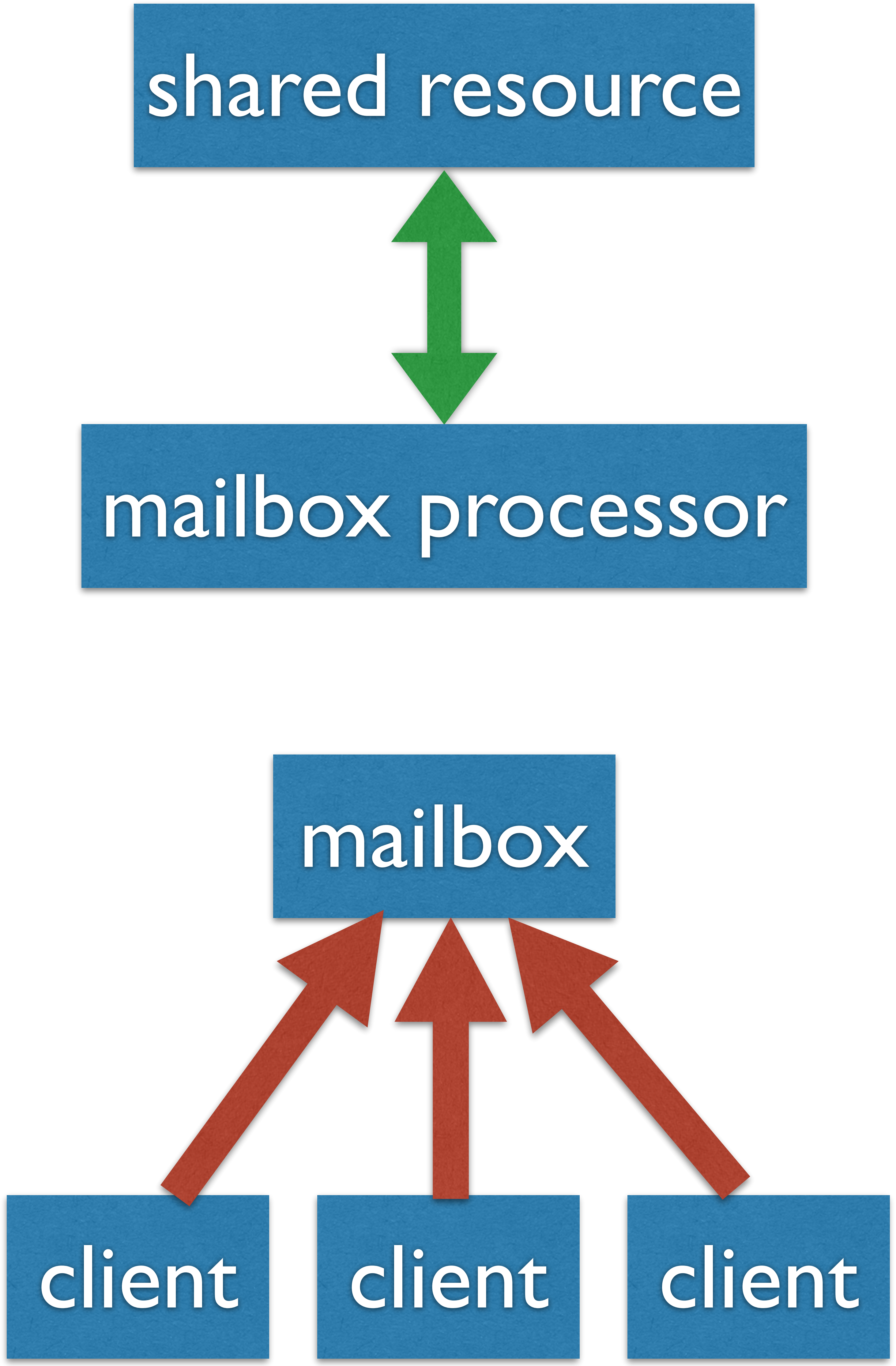
Message-based Synchronisation

- Only one thread (the **mailbox processor**) has access to the shared resource
- Other threads (the **clients**) send messages to **mailbox** requesting operations on the shared resource
- The mailbox processor works through the messages in the mailbox **one by one** performing the requested operations



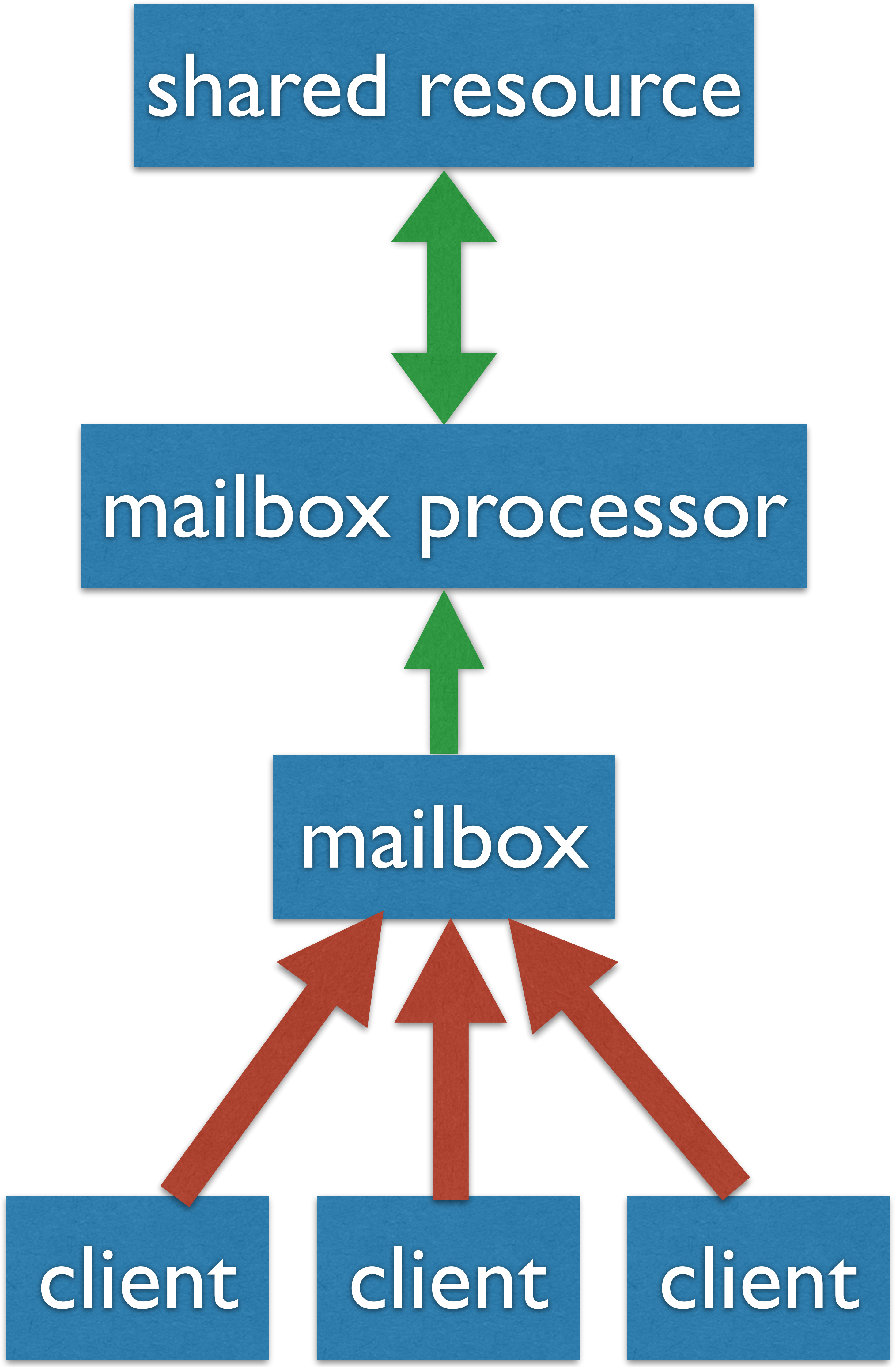
Message-based Synchronisation

- Only one thread (the **mailbox processor**) has access to the shared resource
- Other threads (the **clients**) send messages to **mailbox** requesting operations on the shared resource
- The mailbox processor works through the messages in the mailbox **one by one** performing the requested operations



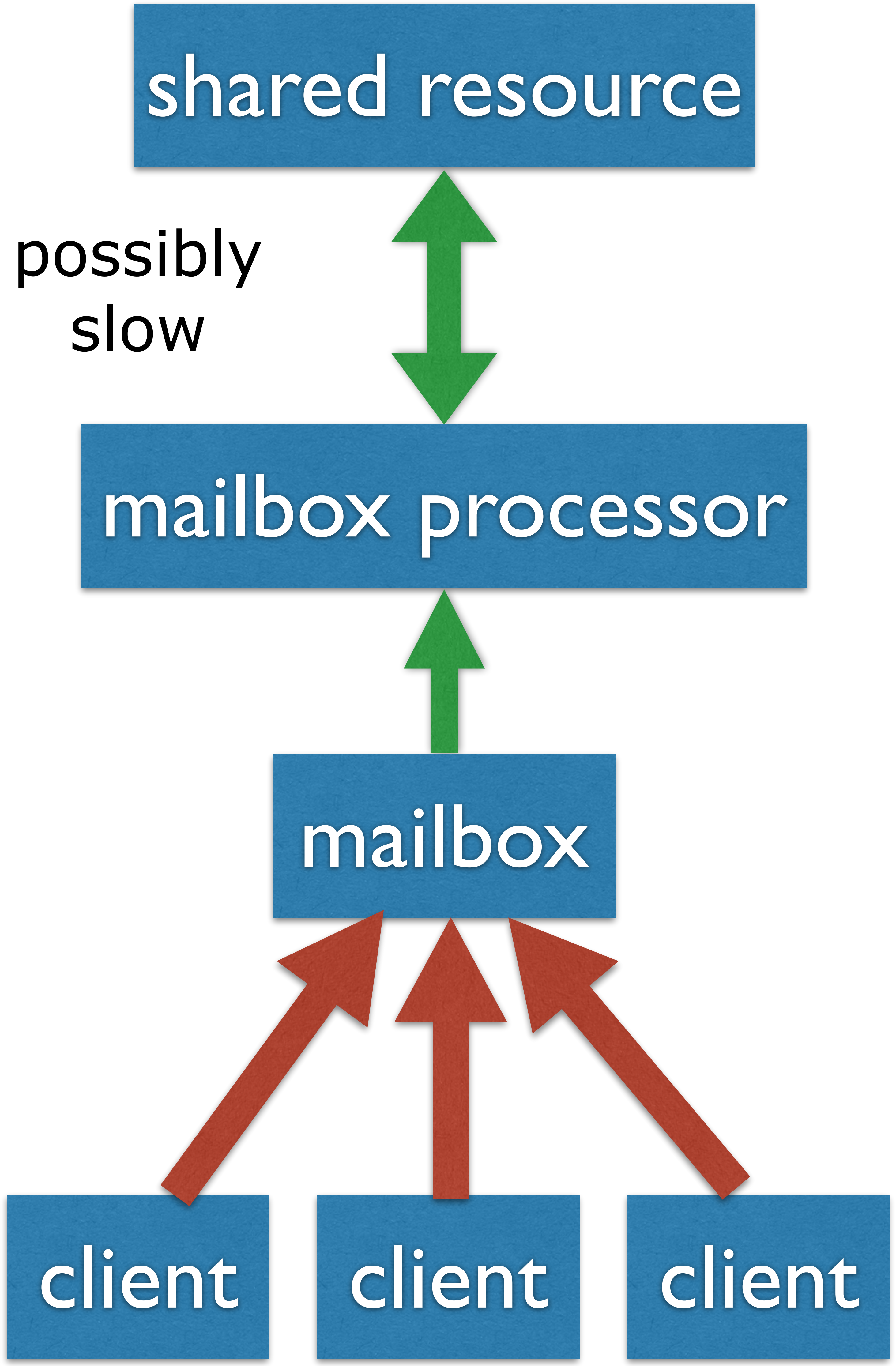
Message-based Synchronisation

- Only one thread (the **mailbox processor**) has access to the shared resource
- Other threads (the **clients**) send messages to **mailbox** requesting operations on the shared resource
- The mailbox processor works through the messages in the mailbox **one by one** performing the requested operations



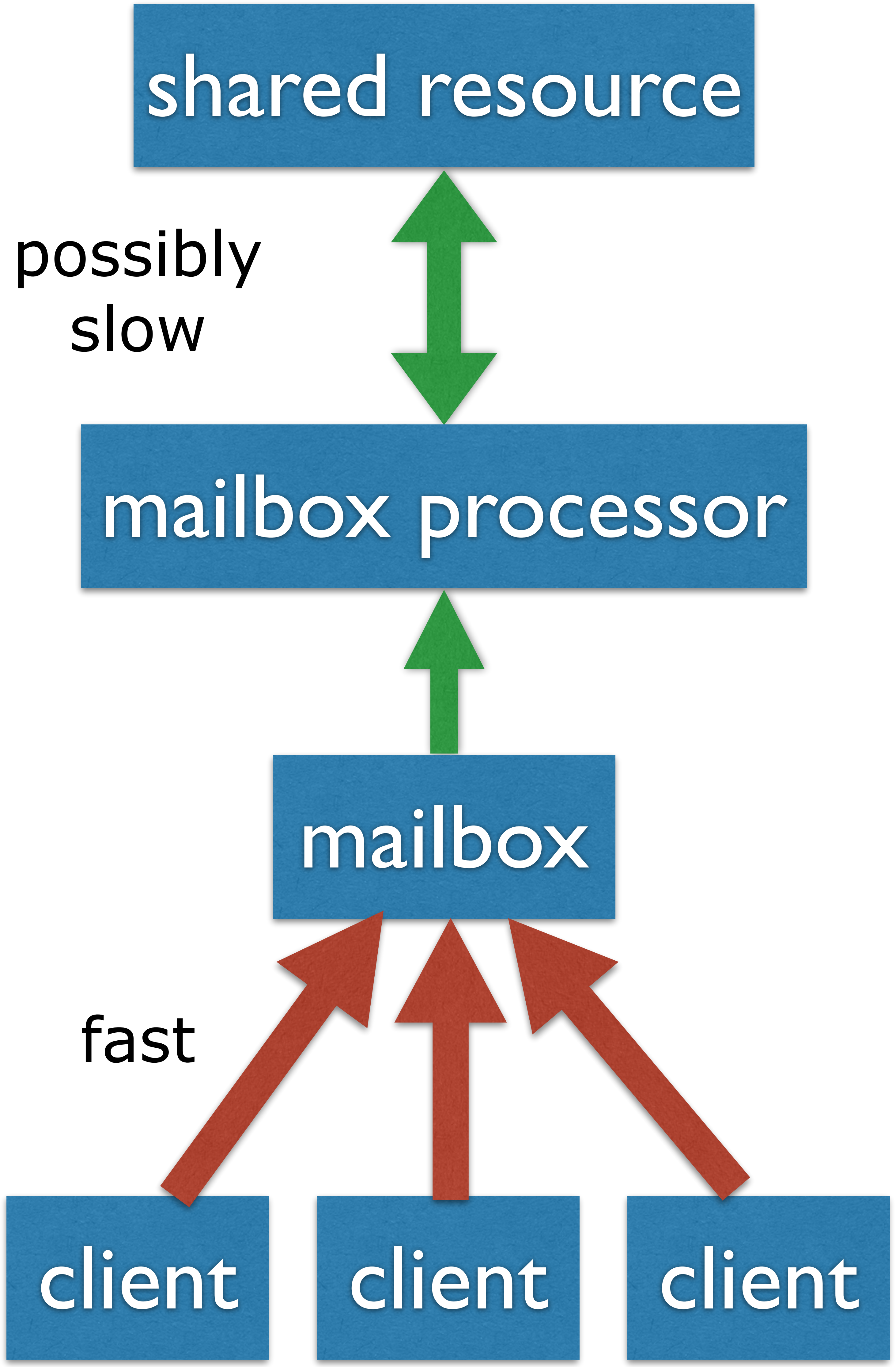
Message-based Synchronisation

- Only one thread (the **mailbox processor**) has access to the shared resource
- Other threads (the **clients**) send messages to **mailbox** requesting operations on the shared resource
- The mailbox processor works through the messages in the mailbox **one by one** performing the requested operations



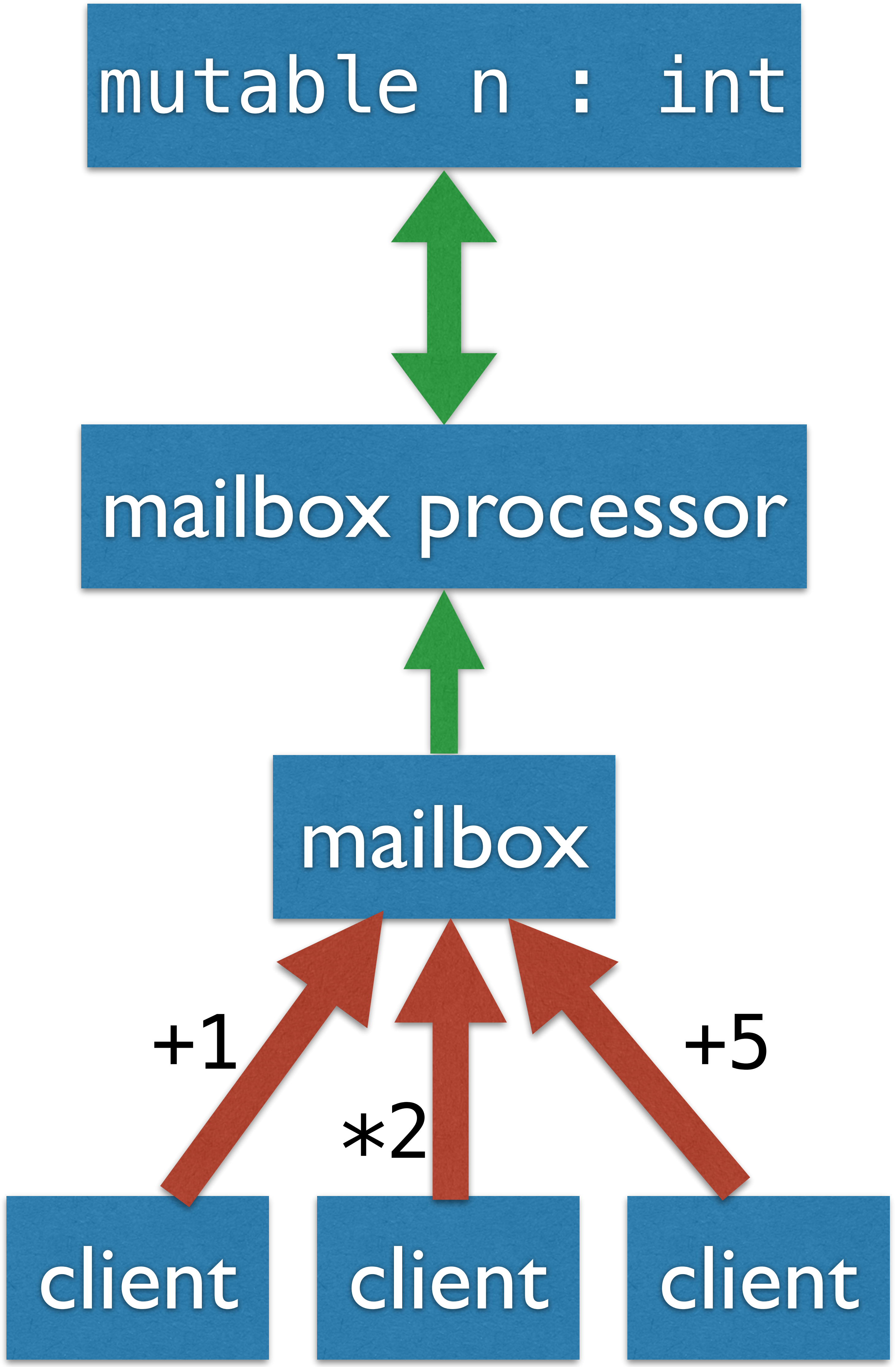
Message-based Synchronisation

- Only one thread (the **mailbox processor**) has access to the shared resource
- Other threads (the **clients**) send messages to **mailbox** requesting operations on the shared resource
- The mailbox processor works through the messages in the mailbox **one by one** performing the requested operations



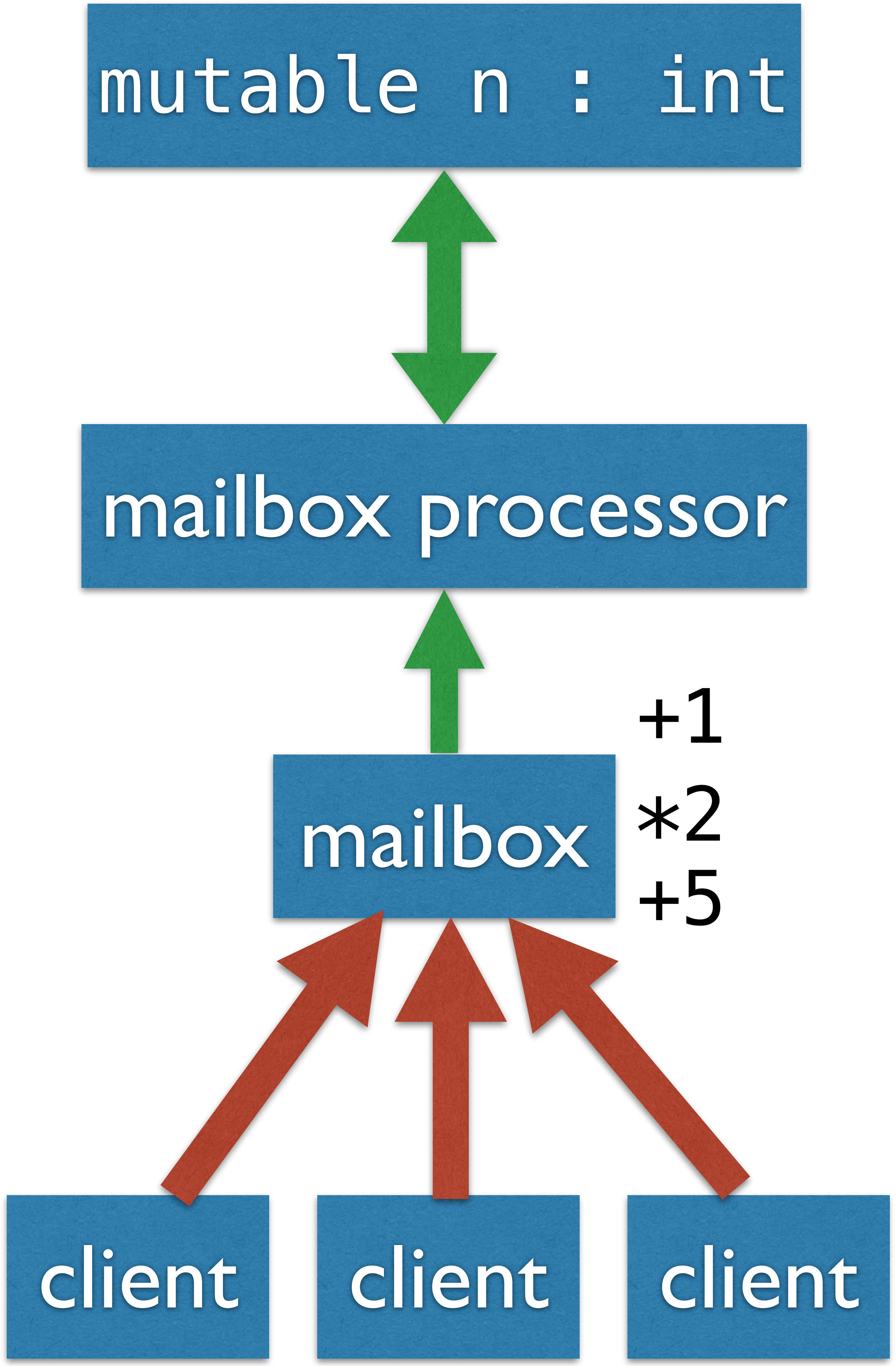
Message-based Synchronisation

- Only one thread (the **mailbox processor**) has access to the *mutable state*
- Other threads (the **clients**) send messages to **mailbox** requesting *changes to the mutable state*
- The mailbox processor works through the messages in the mailbox **one by one** performing the requested *changes*



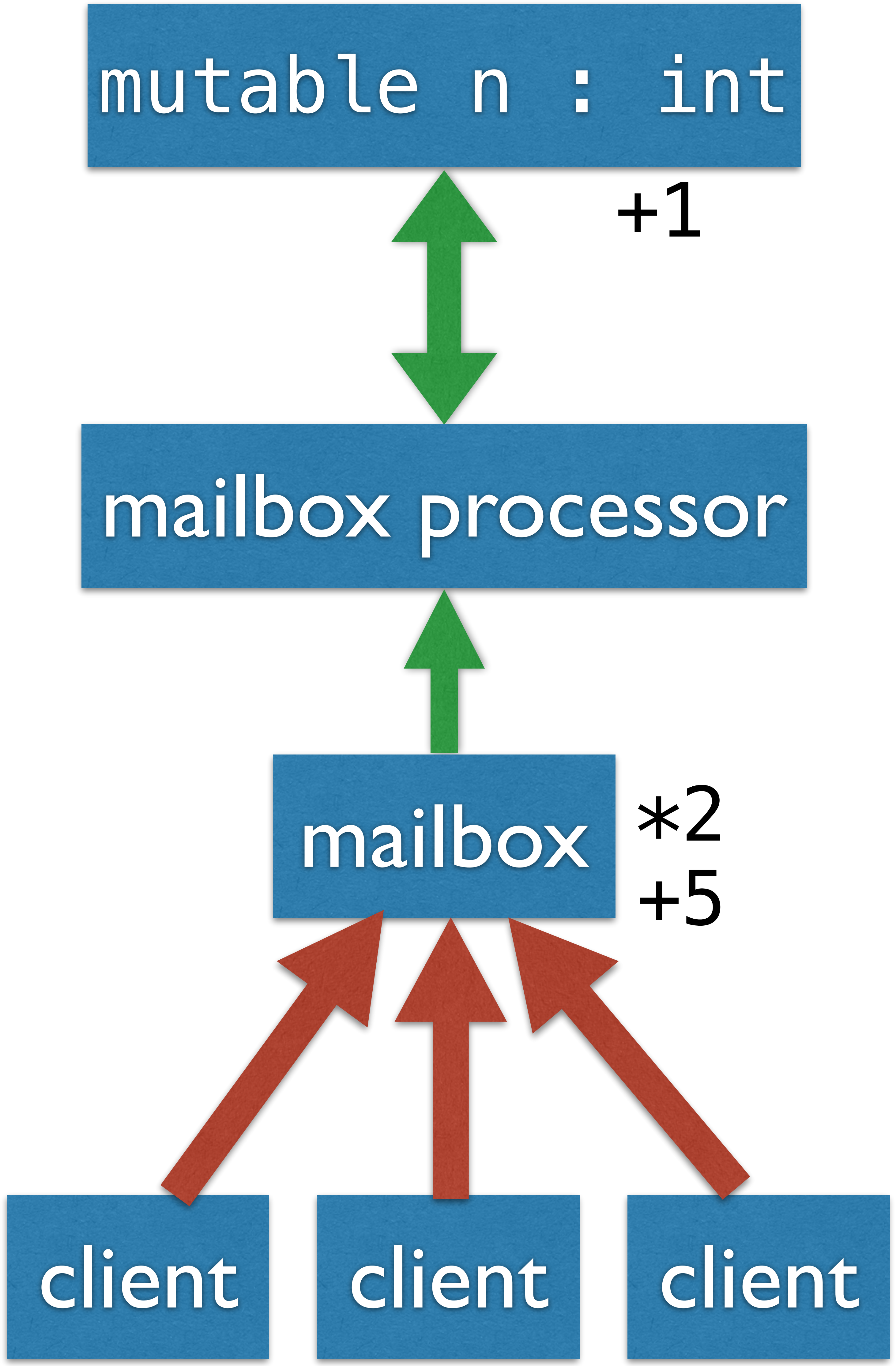
Message-based Synchronisation

- Only one thread (the **mailbox processor**) has access to the *mutable state*
- Other threads (the **clients**) send messages to **mailbox** requesting *changes to the mutable state*
- The mailbox processor works through the messages in the mailbox **one by one** performing the requested *changes*



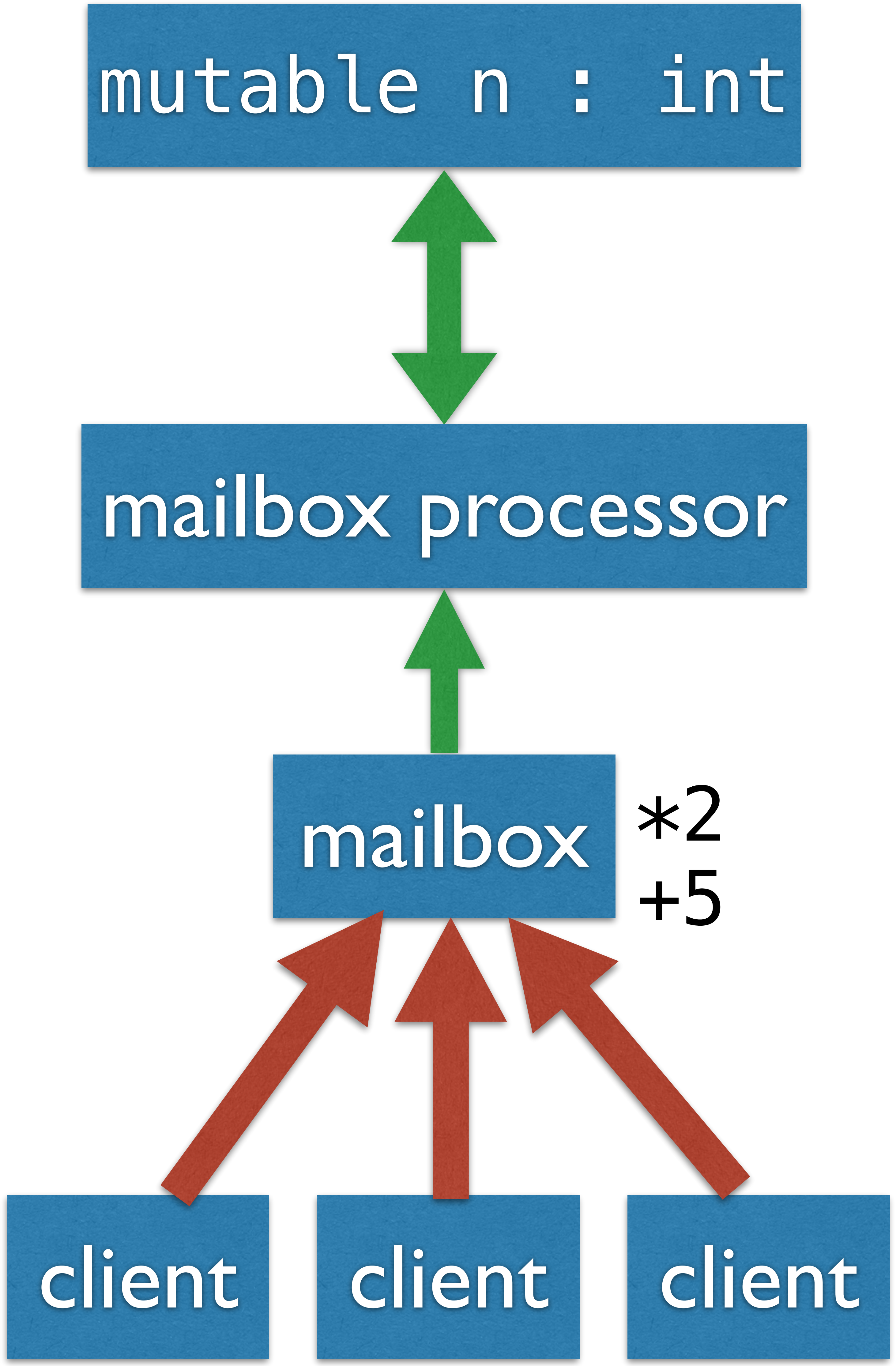
Message-based Synchronisation

- Only one thread (the **mailbox processor**) has access to the *mutable state*
- Other threads (the **clients**) send messages to **mailbox** requesting *changes to the mutable state*
- The mailbox processor works through the messages in the mailbox **one by one** performing the requested *changes*



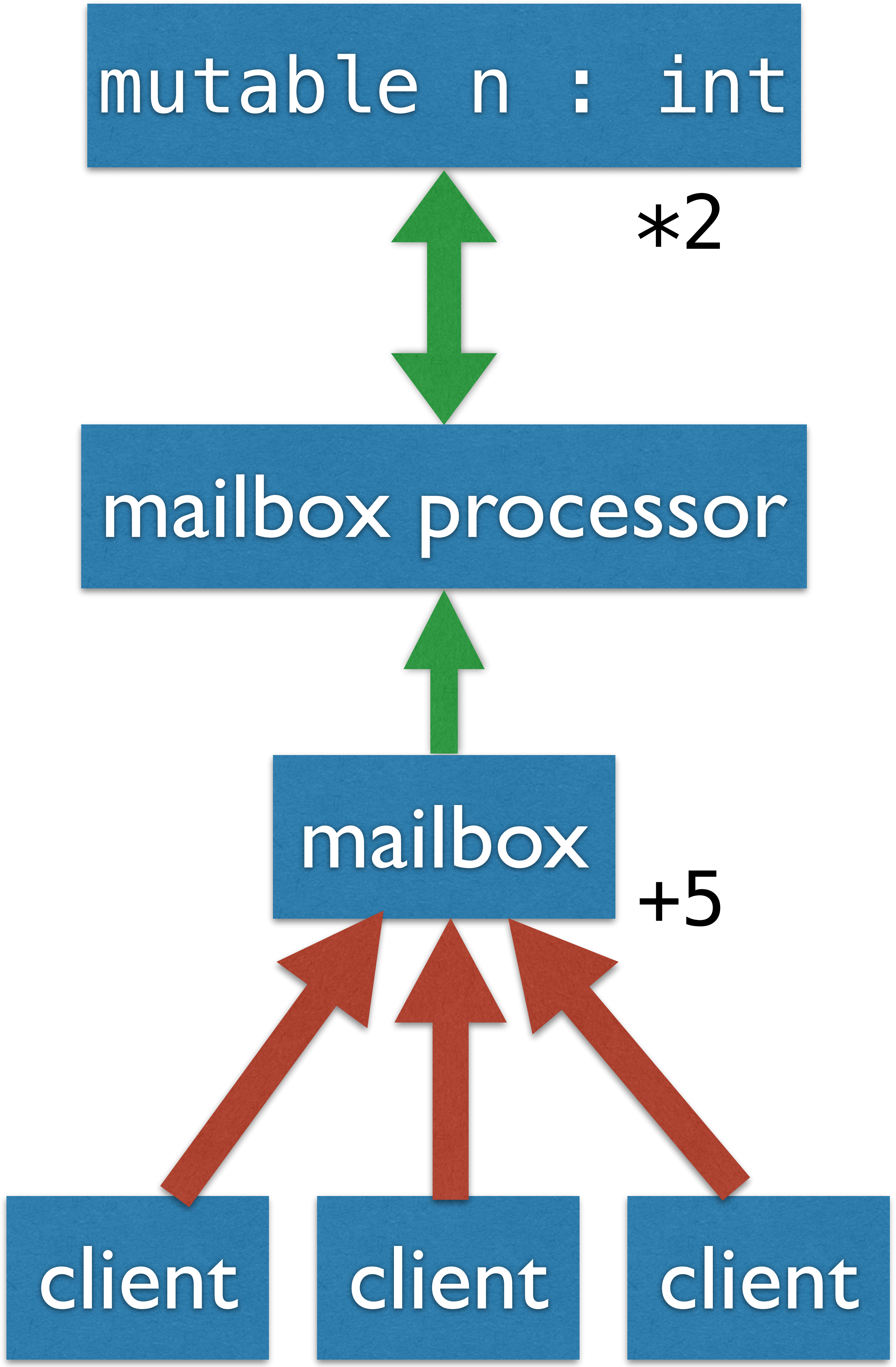
Message-based Synchronisation

- Only one thread (the **mailbox processor**) has access to the *mutable state*
- Other threads (the **clients**) send messages to **mailbox** requesting *changes to the mutable state*
- The mailbox processor works through the messages in the mailbox **one by one** performing the requested *changes*



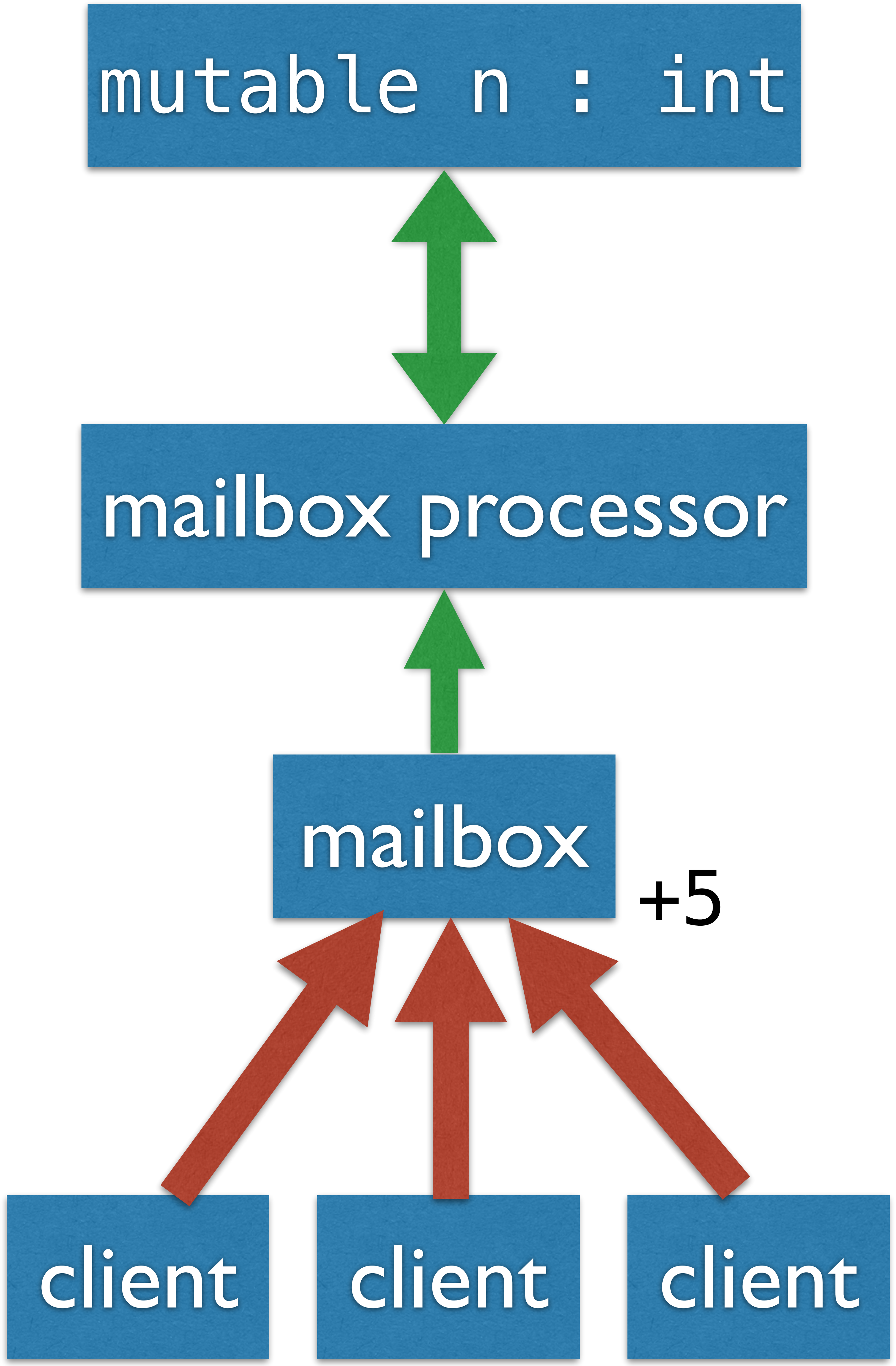
Message-based Synchronisation

- Only one thread (the **mailbox processor**) has access to the *mutable state*
- Other threads (the **clients**) send messages to **mailbox** requesting *changes to the mutable state*
- The mailbox processor works through the messages in the mailbox **one by one** performing the requested *changes*



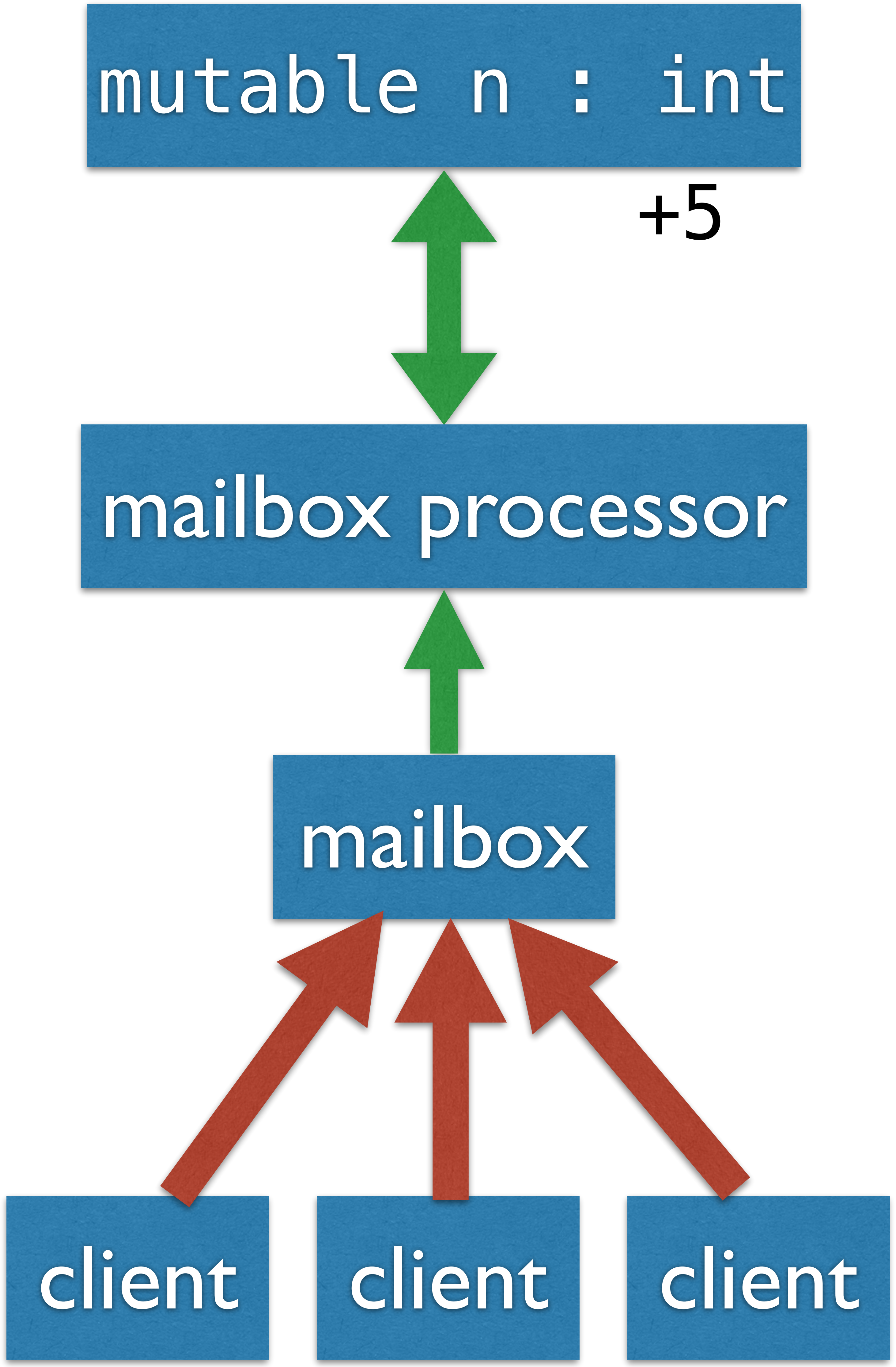
Message-based Synchronisation

- Only one thread (the **mailbox processor**) has access to the *mutable state*
- Other threads (the **clients**) send messages to **mailbox** requesting *changes to the mutable state*
- The mailbox processor works through the messages in the mailbox **one by one** performing the requested *changes*



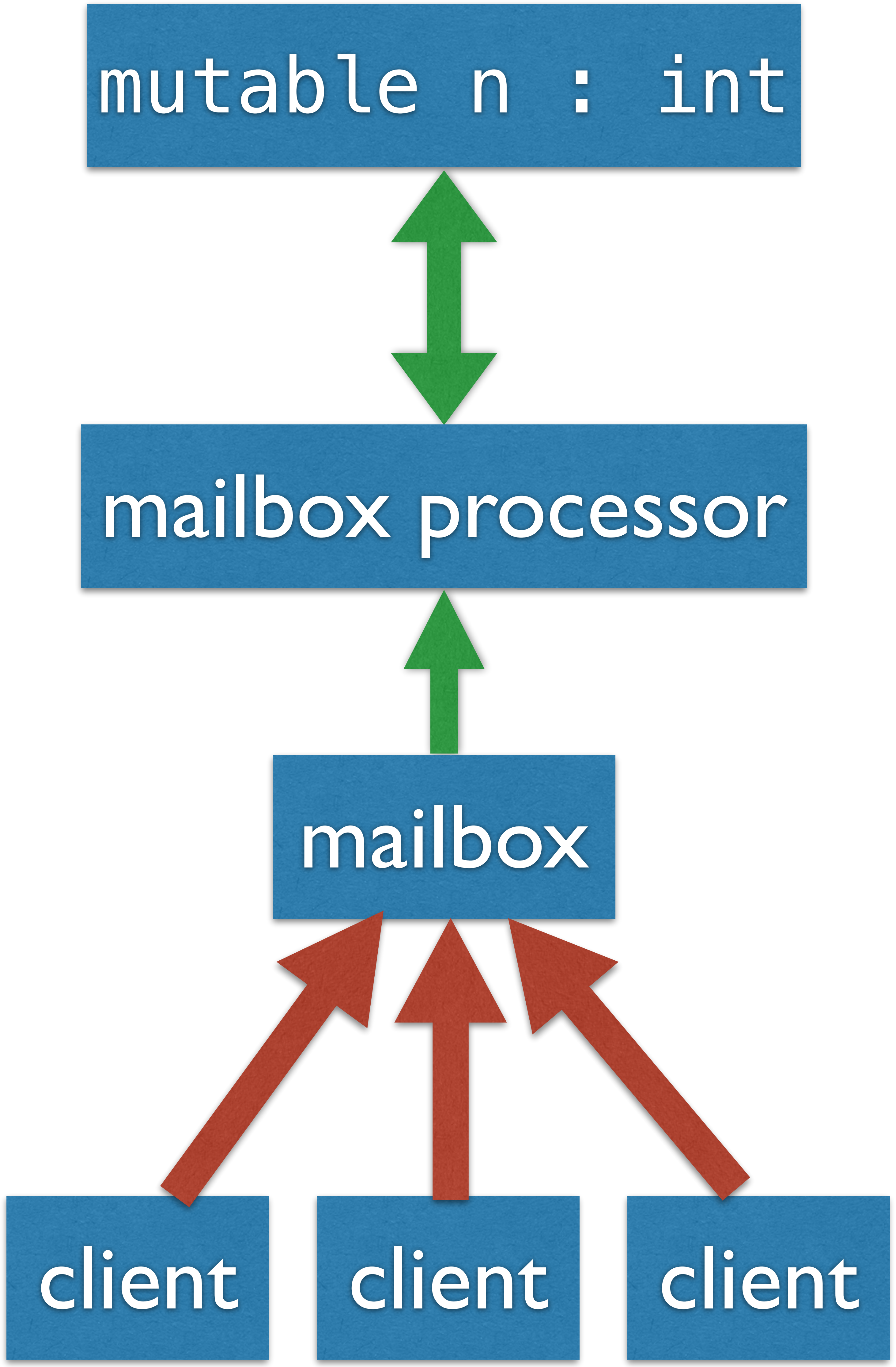
Message-based Synchronisation

- Only one thread (the **mailbox processor**) has access to the *mutable state*
- Other threads (the **clients**) send messages to **mailbox** requesting *changes to the mutable state*
- The mailbox processor works through the messages in the mailbox **one by one** performing the requested *changes*



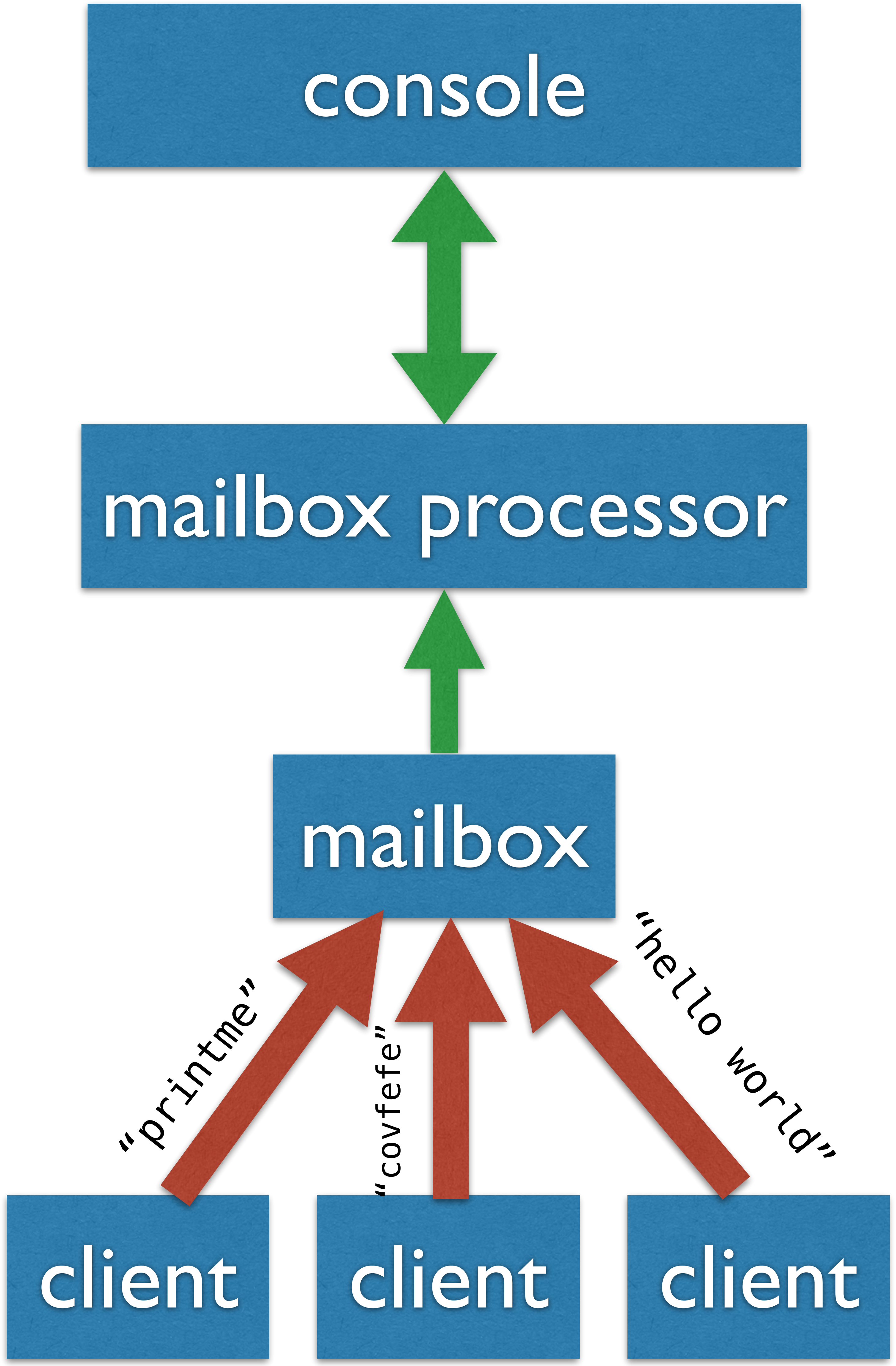
Message-based Synchronisation

- Only one thread (the **mailbox processor**) has access to the *mutable state*
- Other threads (the **clients**) send messages to **mailbox** requesting *changes to the mutable state*
- The mailbox processor works through the messages in the mailbox **one by one** performing the requested *changes*



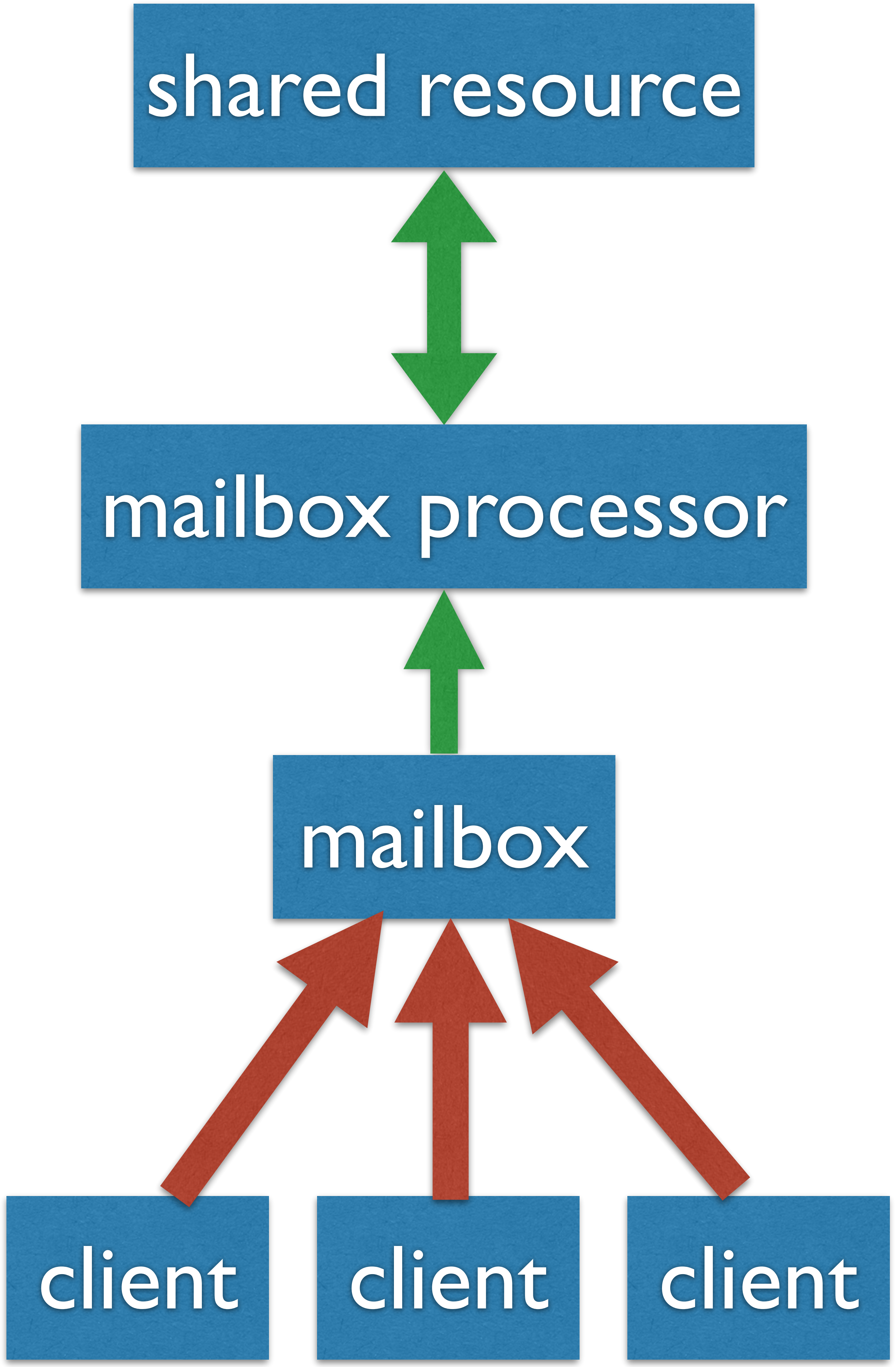
Message-based Synchronisation

- Only one thread (the **mailbox processor**) has access to the *console*
- Other threads (the **clients**) send messages to **mailbox** requesting *to write a string on the console*
- The mailbox processor works through the messages in the mailbox **one by one** *writing the requested strings to the console*



Message-based Synchronisation

- Minimal waiting time for clients:
  - No matter how much time the operations take, sending a message is almost instantaneous
- separation of responsibilities
  - the mailbox processor is responsible for managing access to the shared resource



Console Mailbox

```
let console = MailboxProcessor.Start(fun inbox ->
    let rec messageLoop () = async{
        let! message = inbox.Receive()
        System.Console.WriteLine message
        return! messageLoop ()
        }
    messageLoop ())
```

Console Mailbox

```
MailboxProcessor.Start :
    (MailboxProcessor<'msg> -> Async<unit>)
     -> MailboxProcessor<'msg>
```

```
let console = MailboxProcessor.Start(fun inbox ->
    let rec messageLoop () = async{
        let! message = inbox.Receive()
        System.Console.WriteLine message
        return! messageLoop ()
        }
    messageLoop ())
```

Console Mailbox

```
MailboxProcessor.Start :
    (MailboxProcessor<'msg> -> Async<unit>)
     -> MailboxProcessor<'msg>
```

```
let console = MailboxProcessor.Start(fun inbox ->
    let rec messageLoop () = async{
        let! message = inbox.Receive()
        System.Console.WriteLine message
        return! messageLoop ()
        }
    messageLoop ()
```

```
MailboxProcessor<'msg>
   member Receive : unit -> Async<'msg>
```

Console Mailbox

```
MailboxProcessor.Start :
    (MailboxProcessor<'msg> -> Async<unit>)
     -> MailboxProcessor<'msg>
```

```
console : MailboxProcessor<string>
```

```
let console = MailboxProcessor.Start(fun inbox ->
    let rec messageLoop () = async{
        let! message = inbox.Receive()
        System.Console.WriteLine message
        return! messageLoop ()
        }
    messageLoop ()
```

```
MailboxProcessor<'msg>
   member Receive : unit -> Async<'msg>
```

Console Mailbox

```
let console = MailboxProcessor.Start(fun inbox ->
    let rec messageLoop () = async{
        let! message = inbox.Receive()
        System.Console.WriteLine message
        return! messageLoop ()
        }
    messageLoop ())

seq [1 .. 10]
|> Seq.map (fun i -> async {
   console.Post ("I'm no. " + string i) })
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
```

Console Mailbox

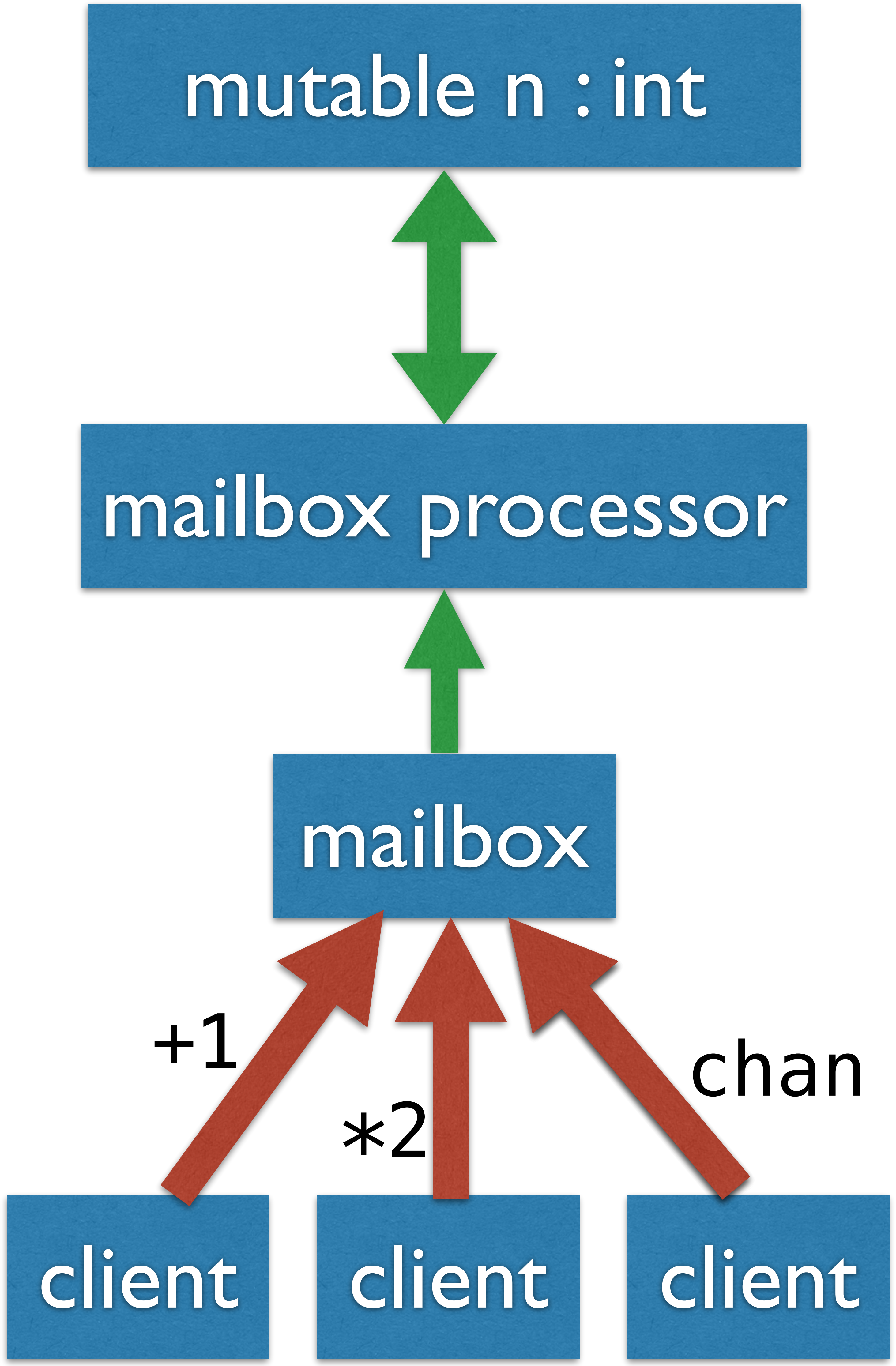
```
let console = MailboxProcessor.Start(fun inb
    let rec messageLoop () = async{
        let! message = inbox.Receive()
        System.Console.WriteLine message
        return! messageLoop ()
        }
    messageLoop ())

seq [1 .. 10]
|> Seq.map (fun i -> async {
   console.Post ("I'm no. " + string i) })
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
```

output:

```
I'm no. 4
I'm no. 8
I'm no. 6
I'm no. 1
I'm no. 5
I'm no. 2
I'm no. 7
I'm no. 3
I'm no. 10
I'm no. 9
```

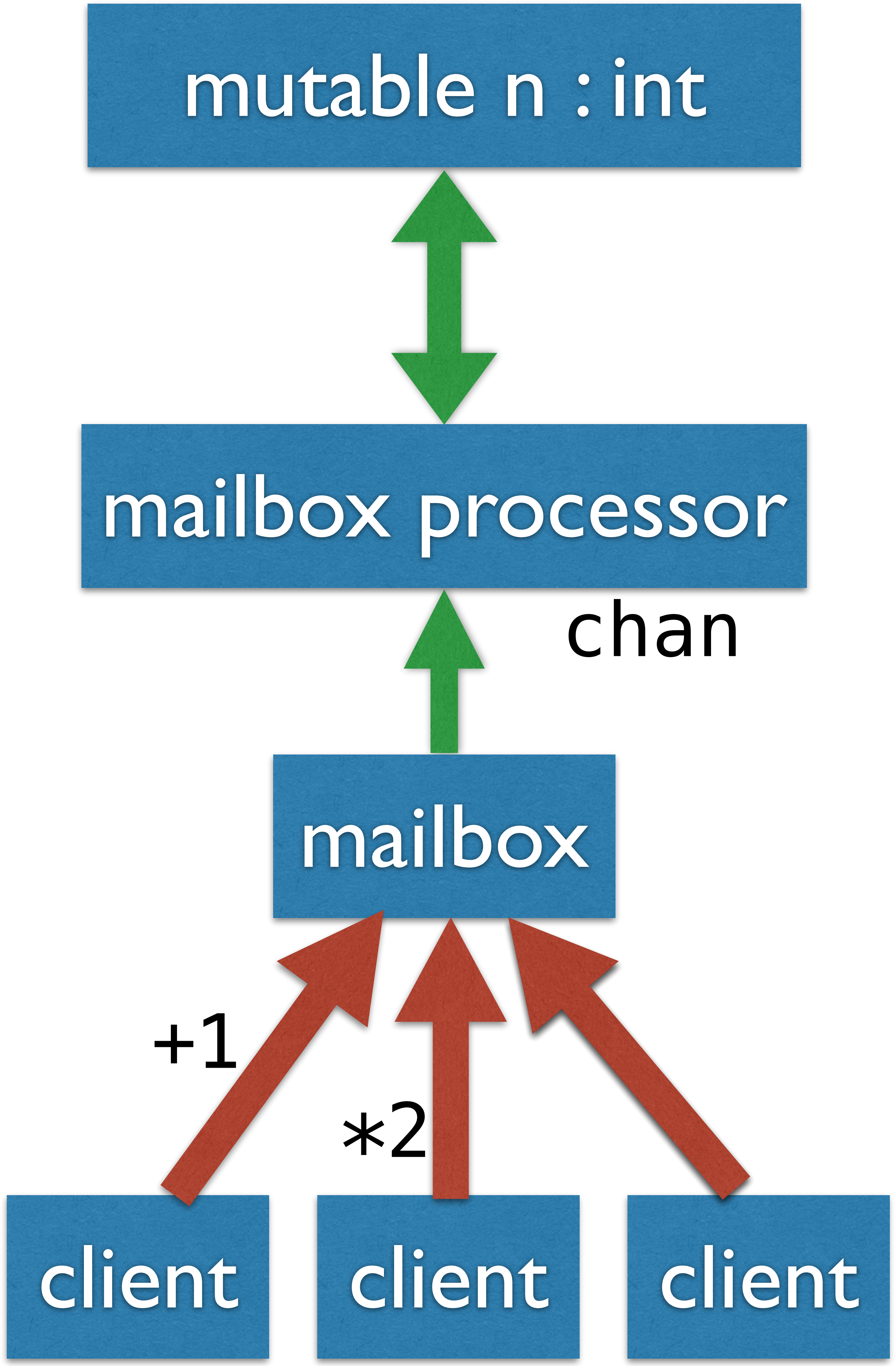
Requesting a Response

- Clients may also **request a response** from the mailbox processor
- To **receive a response**, the client needs to **send a channel**
- The mailbox processor will then respond to the client by **sending data on the channel**



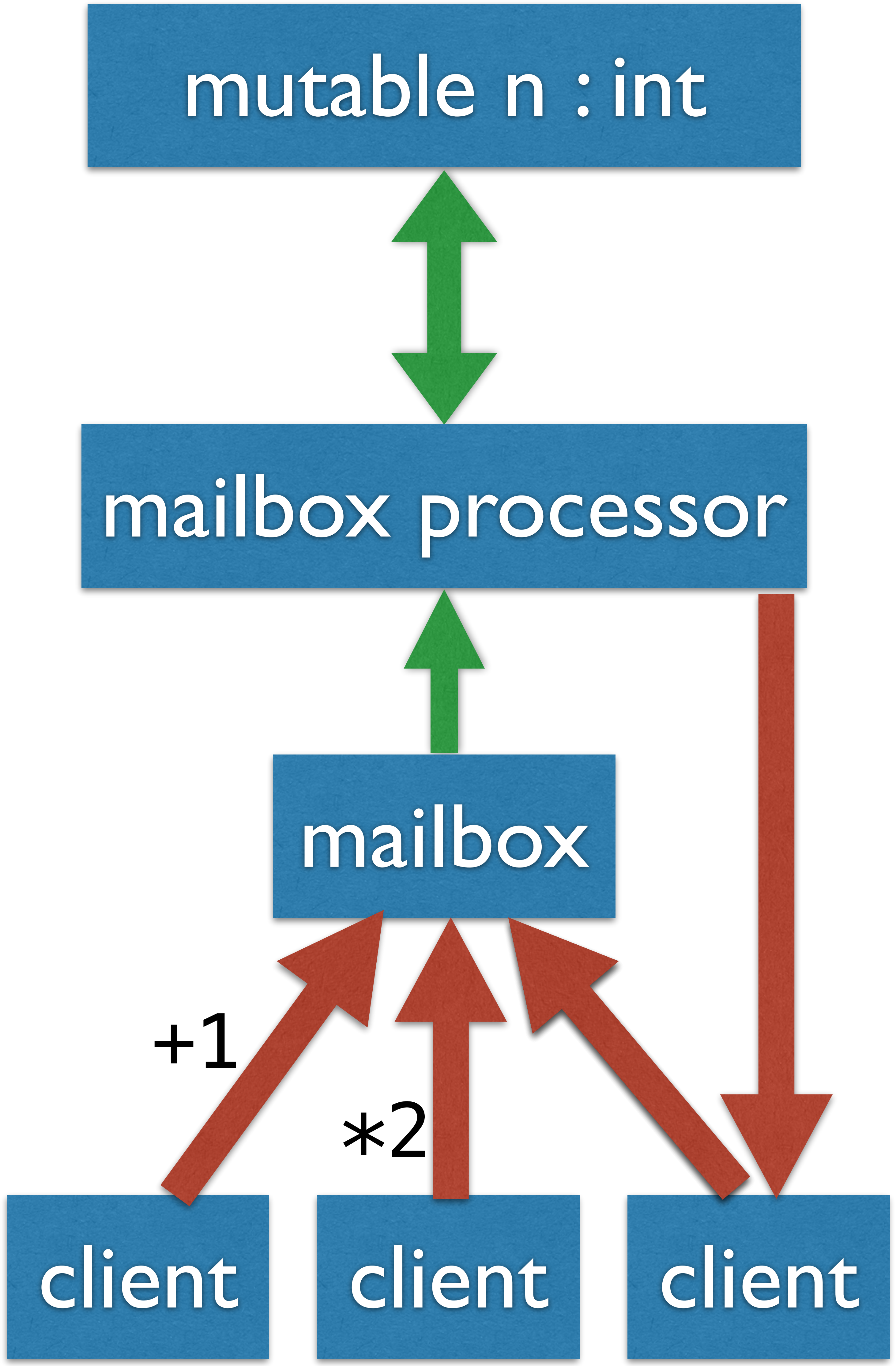
Requesting a Response

- Clients may also **request a response** from the mailbox processor
- To **receive a response**, the client needs to **send a channel**
- The mailbox processor will then respond to the client by **sending data on the channel**



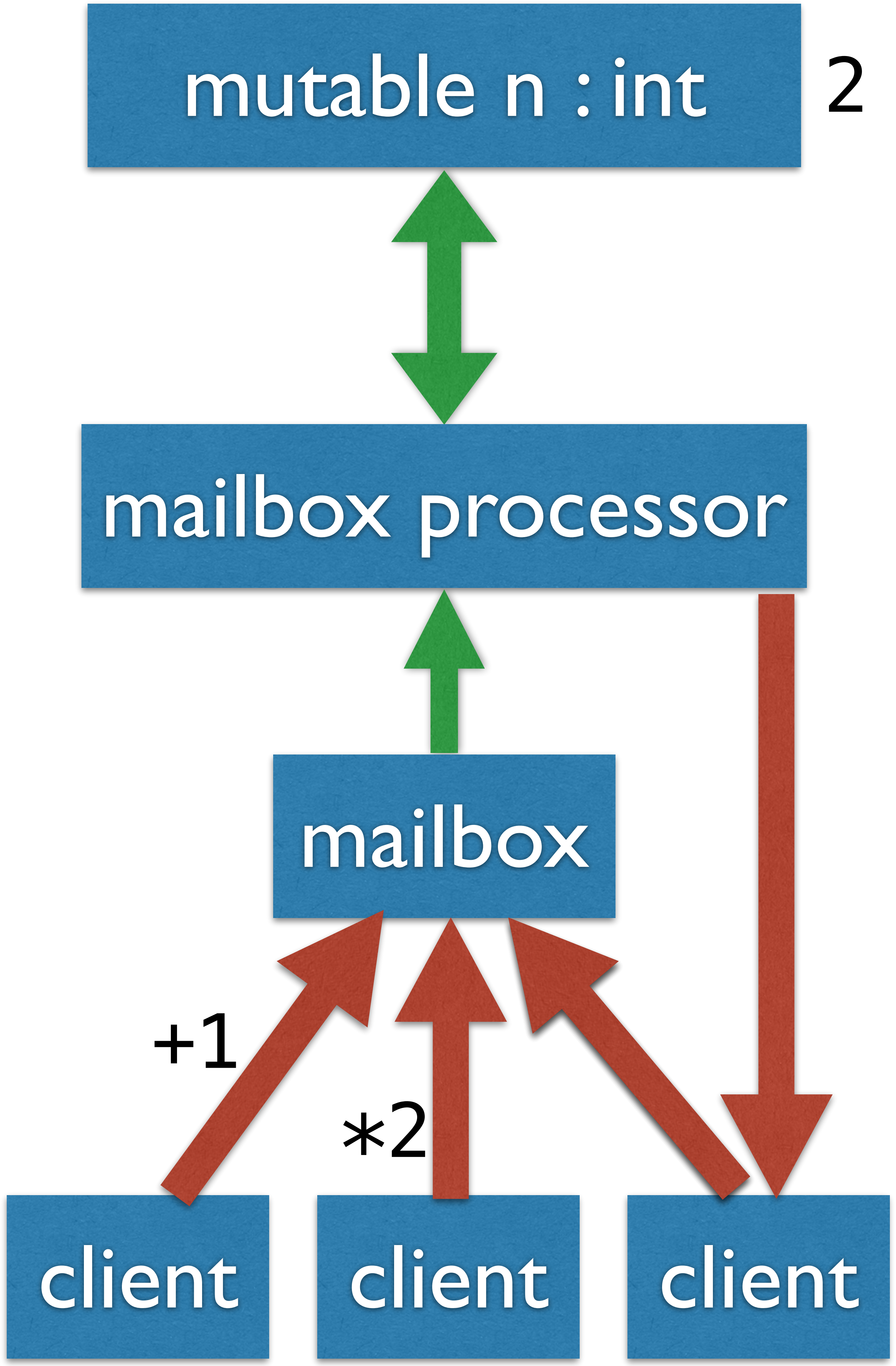
Requesting a Response

- Clients may also **request a response** from the mailbox processor
- To **receive a response**, the client needs to **send a channel**
- The mailbox processor will then respond to the client by **sending data on the channel**



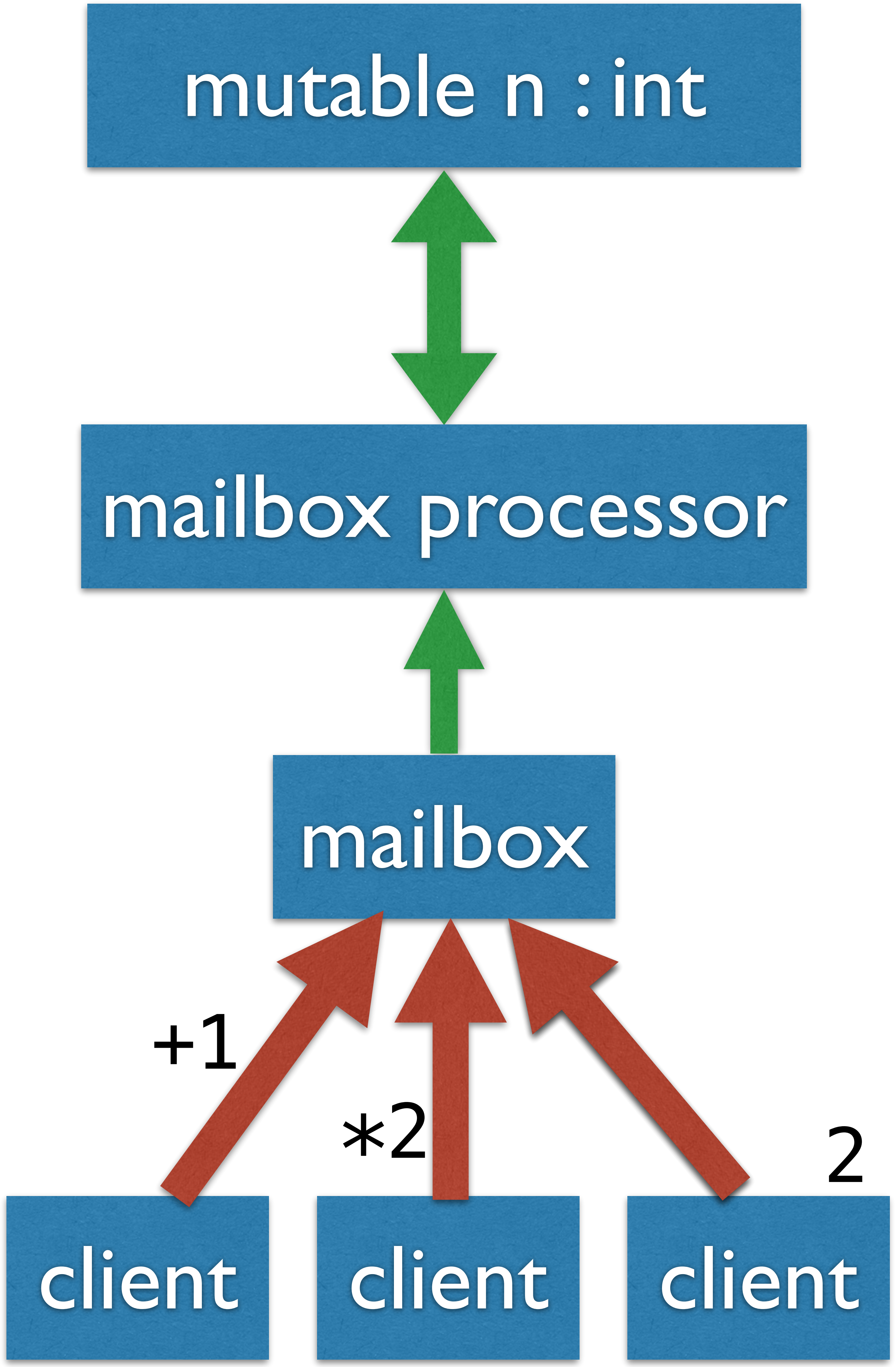
Requesting a Response

- Clients may also **request a response** from the mailbox processor
- To **receive a response**, the client needs to **send a channel**
- The mailbox processor will then respond to the client by **sending data on the channel**



Requesting a Response

- Clients may also **request a response** from the mailbox processor
- To **receive a response**, the client needs to **send a channel**
- The mailbox processor will then respond to the client by **sending data on the channel**



Counter Mailbox

```
type msg =
    | Add of int
    | Get of AsyncReplyChannel<int>

let mutable n = 0
let counter =
  MailboxProcessor.Start(fun inbox ->
  let rec loop () = async {
      let! msg = inbox.Receive ()
      match msg with
      | Add x -> n <- n + x
      | Get ch -> ch.Reply n
      return! loop () }
  loop ())
```

Counter Mailbox

```
type msg =
    | Add of int
    | Get of AsyncReplyChannel<int>

let mutable n = 0
let counter =
  MailboxProcessor.Start(fun inbox ->
  let rec loop () = async {
      let! msg = inbox.Receive ()
      match msg with
      | Add x -> n <- n + x
      | Get ch -> ch.Reply n
      return! loop () }
  loop ())
```

```
counter.PostAndReply
   (fun ch -> Get ch);;
 val it : int = 0
```

Counter Mailbox

```
type msg =
    | Add of int
    | Get of AsyncReplyChannel<int>

let mutable n = 0
let counter =
  MailboxProcessor.Start(fun inbox ->
  let rec loop () = async {
      let! msg = inbox.Receive ()
      match msg with
      | Add x -> n <- n + x
      | Get ch -> ch.Reply n
      return! loop () }
  loop ())
```

```
counter.PostAndReply
   (fun ch -> Get ch);;
 val it : int = 0

counter.Post(Add 7);;
 val it : unit = ()
```

Counter Mailbox

```
type msg =
    | Add of int
    | Get of AsyncReplyChannel<int>

let mutable n = 0
let counter =
  MailboxProcessor.Start(fun inbox ->
  let rec loop () = async {
      let! msg = inbox.Receive ()
      match msg with
      | Add x -> n <- n + x
      | Get ch -> ch.Reply n
      return! loop () }
  loop ())
```

```
counter.PostAndReply
   (fun ch -> Get ch);;
 val it : int = 0

counter.Post(Add 7);;
 val it : unit = ()

counter.PostAndReply
   (fun ch -> Get ch);;
 val it : int = 7
```

Summary

- Task parallelism vs. data parallelism

Summary

- Task parallelism vs. data parallelism
- Task parallelism with `Async` & `Task`

Summary

- Task parallelism vs. data parallelism
- Task parallelism with Async & Task
- Data parallelism with Array.Parallel

Summary

- Task parallelism vs. data parallelism
- Task parallelism with Async & Task
- Data parallelism with Array.Parallel
- Race conditions & deadlocks

Summary

- Task parallelism vs. data parallelism
- Task parallelism with `Async` & `Task`
- Data parallelism with Array.Parallel
- Race conditions & deadlocks
- Message-based synchronisation using `MailboxProcessor`