12 - Message Passing

florian.felten@uni.lu

Recap

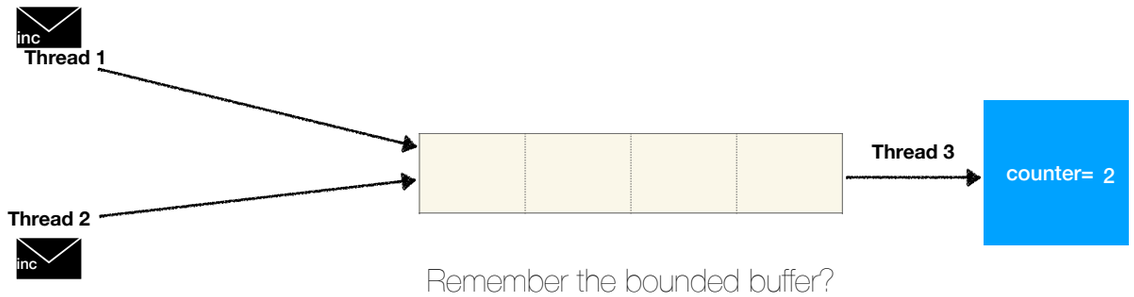
- ✓ We saw that Threads and Domains allow for a program to do multiple things at the same time.
- ✓ We saw that concurrency and parallelism introduce difficulties when modifying shared mutable memory.
- ✓ We fixed the issue by using locks, yet we also saw that it can be very difficult to avoid problems such as deadlocks, livelocks, starvation.

New toys!

Me, telling you the awesome news!

We had a lot of fun (didn't we?) playing with shared state concurrency concepts. But let's be honest, it is not the simplest thing to reason with. Even for small programs, we have seen that it can become quite challenging to keep correct. This lecture will present you with the other common way for dealing with concurrency: message passing!

Message passing



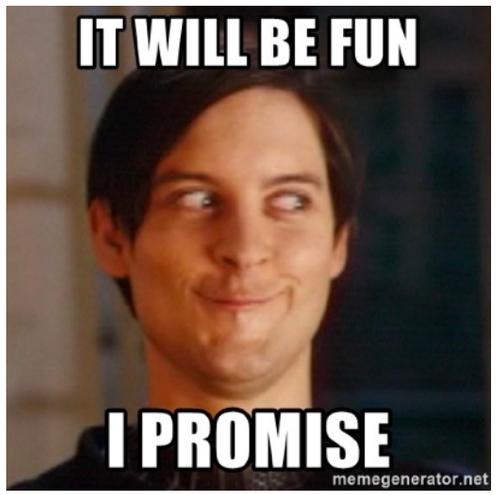
Remember the bounded buffer?

Q: do I need a lock to protect counter?

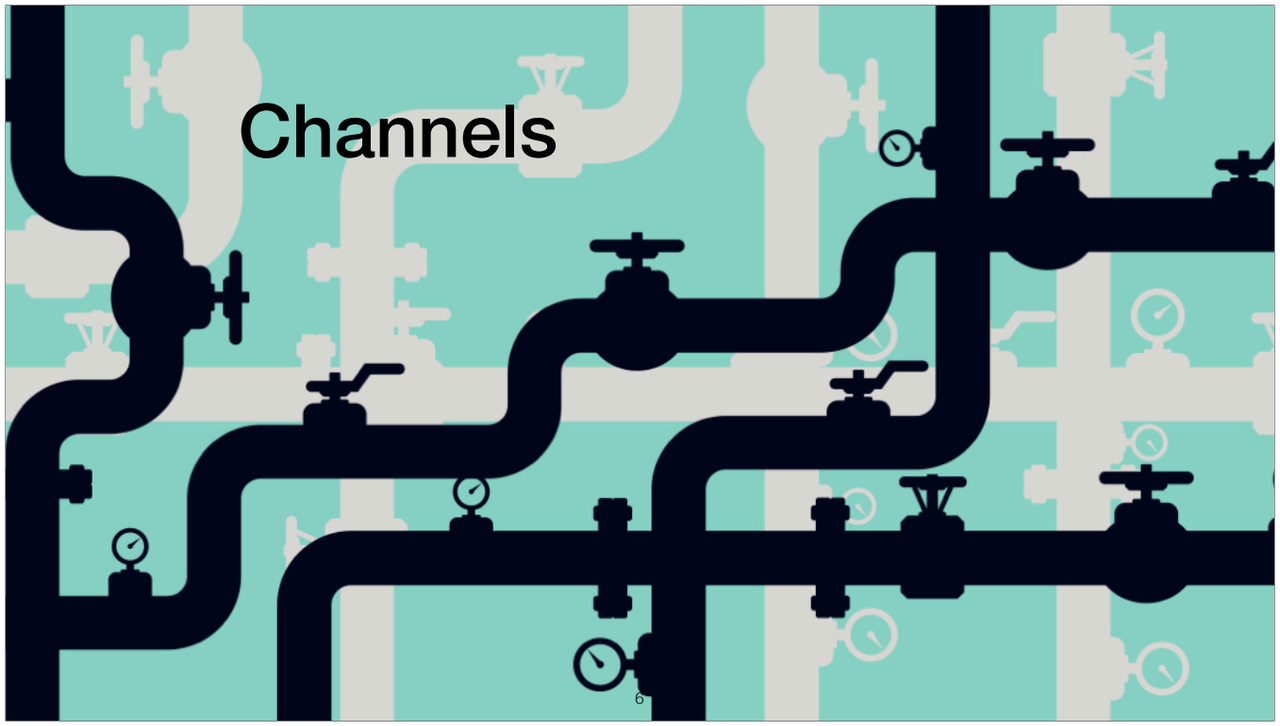
The basic form of message passing relies on a communication medium which is often shared by multiple threads. Most of the time, the messages on the medium are processed one by one, allowing to satisfy the mutual exclusion property by design.

The promise

- You will be introduced to message passing techniques;
- You will see there are many techniques to solve one problem;
- You will understand the differences between message passing and shared state concurrency.



By the end of this course, my promise is that you…


Channels

History

Tony Hoare

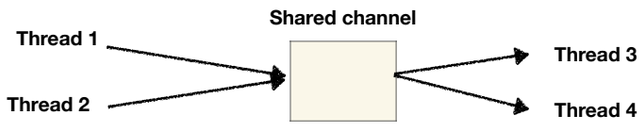
Based on **Communication Sequential Processes** (CSPs): a formal language to describe interactions between concurrent systems. Introduced by Tony Hoare in **1978**.

As of today, the most famous language supporting channels is Go (https://golang.org/), but others also support channels such as occam and Crystal.

Nowadays

Some companies using message passing.

Channels

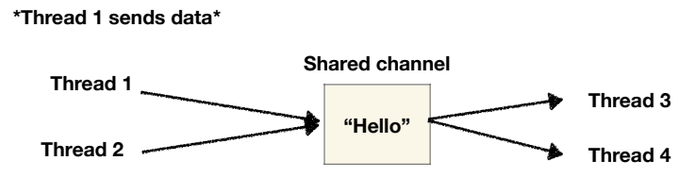


Idea: **share data between threads** through a medium - a channel. **Once a thread has sent the data**, they don't belong to it anymore and it **cannot mutate them**!

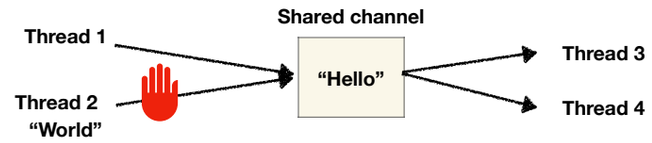
Can be **bounded or not**. Making the channels blocking or not!

Unicast (only one receives)

Bounded channel

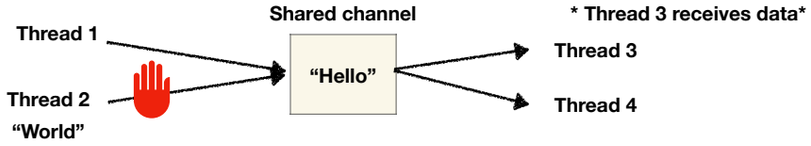


Bounded channel

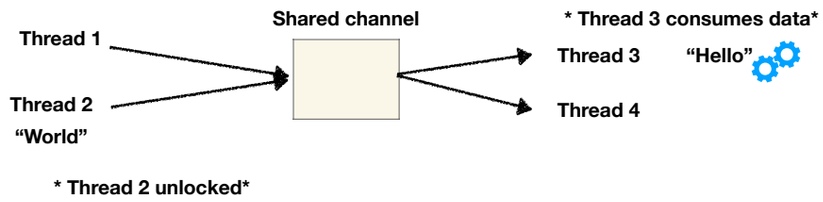


Thread 2 sends data
However, the buffer is full: thread 2 is blocked!

Bounded channel



Bounded channel



Channel size

0: Blocking channel => producer(s) and consumer(s) have to be synchronized ~unlocking a mutex

0 < size < infinity : Bounded buffer/channel =>
If max size is reached, producer(s) blocked
If 0 element, consumer(s) blocked

size = infinity: unbounded channel => max size is given by the memory (careful)

https://github.com/ffelten/ocaml-snippets/tree/main/

hello_world_channel.ml

Channels

`make_bounded: int -> 'a channel`	`Channel.make_bounded n` makes a bounded channel with a buffer of size `n`. With a buffer of size 0, the send operations becomes synchronous.
`make_unbounded: unit -> 'a channel`	Returns an unbounded channel.
`send: 'a channel -> 'a -> unit`	`Channel.send c v` sends the value `v` over the channel `c` . If the channel buffer full then the sending domain blocks until space becomes available.
`send_poll: 'a channel -> 'a -> bool`	`Channel.send_poll c v` sends the value `v` over the channel `c` . If the channel buffer is not full, the message is sent and returns `true.` Otherwise returns `false`.
`recv: 'a channel -> 'a`	`Channel.recv c` returns a value `v` received over the channel. If the channel buffer is empty then the domain blocks until a message is sent on the channel.

https://github.com/ocaml-multicore/domainslib/blob/0.4.2/lib/chan.mli

What kind of data should be sent?

- `int ?`
- `string?`
- `list 'a ?`
- `ref 'a ?`
- `array 'a ?`

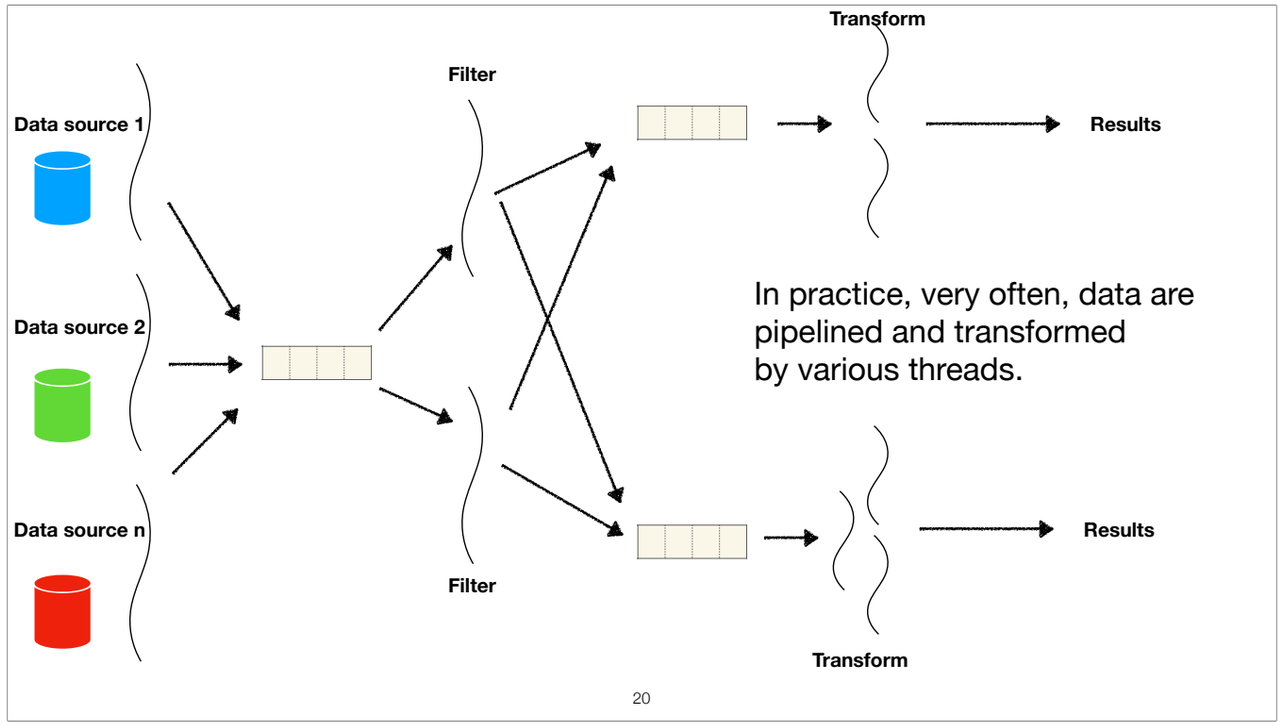
Be very careful with mutability.

=> **Only** send **immutable data on the channels**

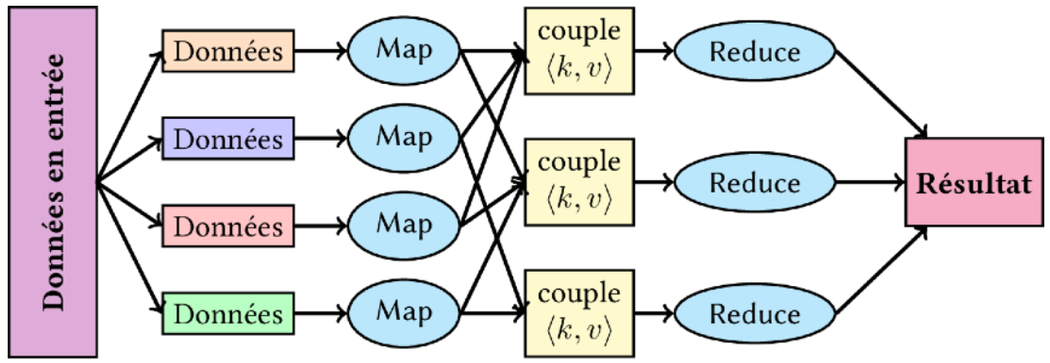
https://github.com/ffelten/ocaml-snippets/tree/main/

hello_world_channel_mut.ml

Pipelining


Transform
Filter
Data source 1
Results
Data source 2
In practice, very often, data are
pipelined and transformed
by various threads.
Data source n
Results
Filter
Transform

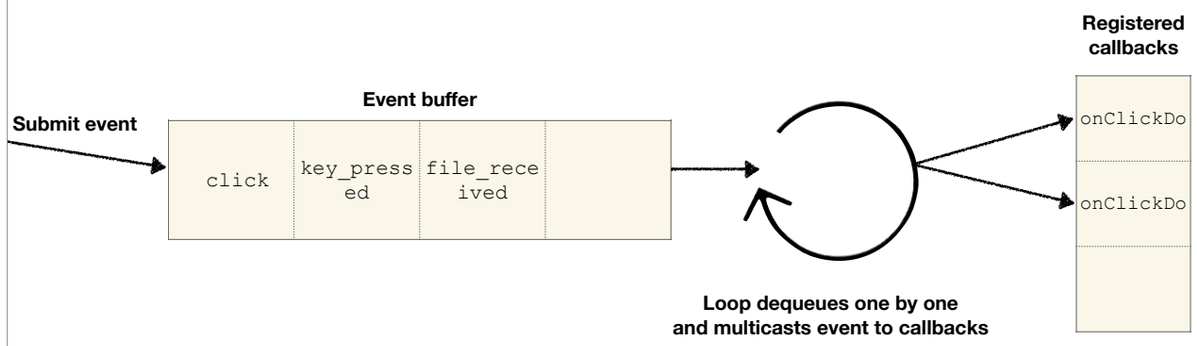
Famous example: MapReduce



Note that MapReduce is even more powerful since it distributes the load over multiple computers (even more cores)

Other well known usage of message passing concurrency

Event loop

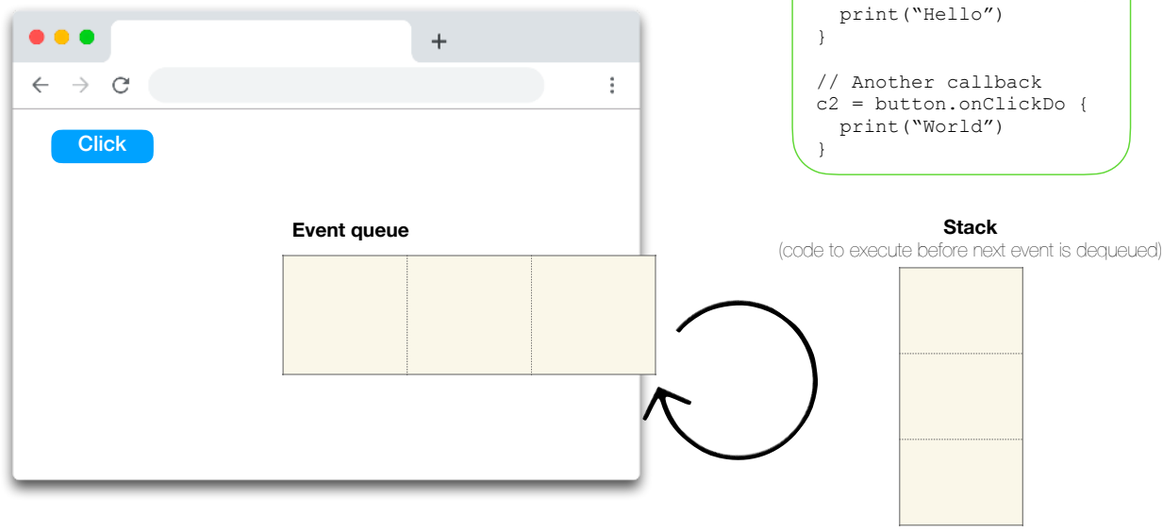


JavaScript only relies on one physical thread, so nothing is done in parallel, it's only context switches

The event loop model is mono processor, it relies on context switches only. In this model, instead of talking about messages, we talk about events. The idea is that threads can submit events to a global buffer. Which will eventually be dequeued one by one and propagated by the event loop. When an event is dequeued, it looks as if it "happens" to the system.

A *callback* is a piece of code which is executed when the event it is attached to happens. Multiple callbacks can be registered to the same kind of event. This means we can have multiple receivers, as opposed to what we saw with channels.

Event loop



Pseudo code

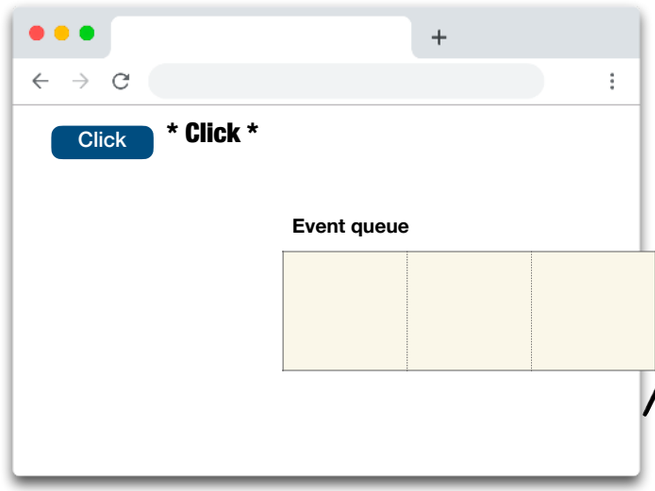
```
// One callback
c1 = button.onClickDo {
  print("Hello")
}

// Another callback
c2 = button.onClickDo {
  print("World")
}
```

Stack

(code to execute before next event is dequeued)

Let's see a concrete example of how the JS event queue works.

Event loop



Pseudo code

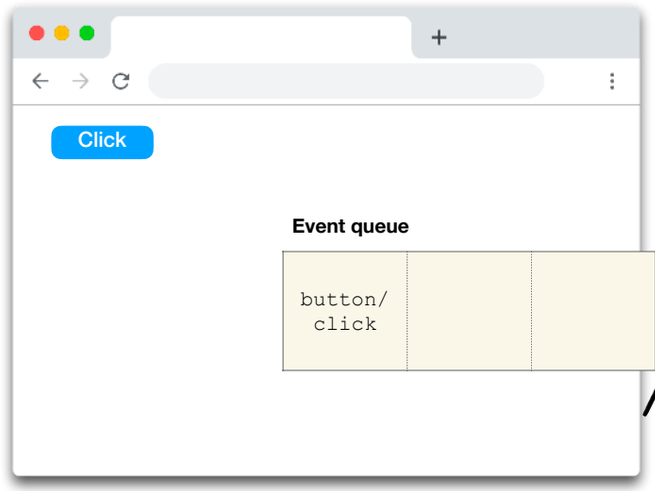
```
// One callback
c1 = button.onClickDo {
  print("Hello")
}

// Another callback
c2 = button.onClickDo {
  print("World")
}
```

Stack

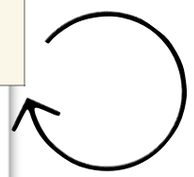
(code to execute before next event is dequeued)

The user clicks the button

Event loop



Event queue



Pseudo code

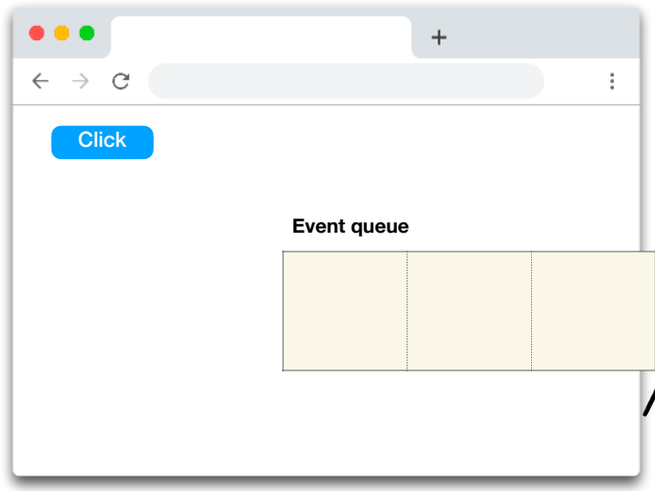
```
// One callback
c1 = button.onClickDo {
  print("Hello")
}

// Another callback
c2 = button.onClickDo {
  print("World")
}
```

Stack

(code to execute before next event is dequeued)

The event is added to the queue

Event loop



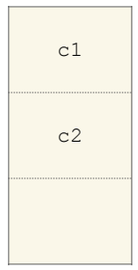
Event queue

Pseudo code

```
// One callback
c1 = button.onClickDo {
  print("Hello")
}

// Another callback
c2 = button.onClickDo {
  print("World")
}
```

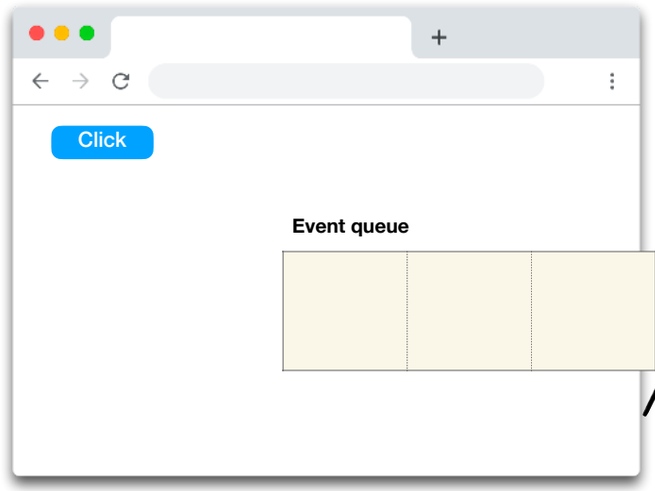
Stack

(code to execute before next event is dequeued)

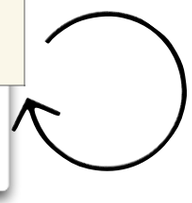


The event is dequeued by the event loop, adding all its related callbacks to the stack.

Event loop



Event queue

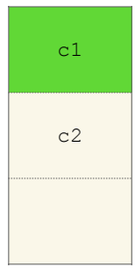


Pseudo code

```
// One callback
c1 = button.onClickDo {
  print("Hello")
}

// Another callback
c2 = button.onClickDo {
  print("World")
}
```

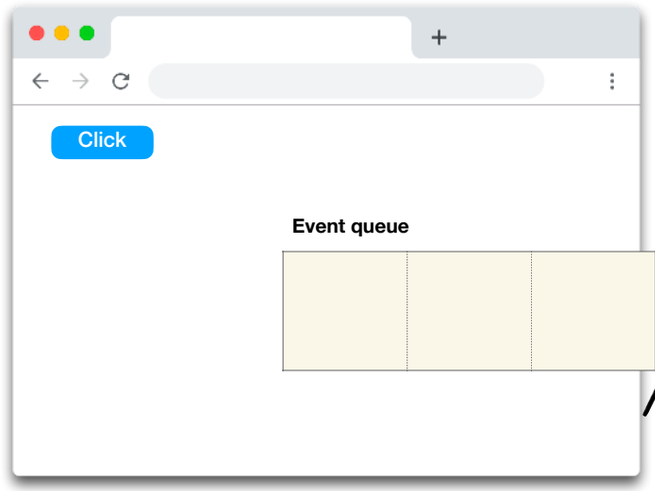
Stack

(code to execute before next event is dequeued)

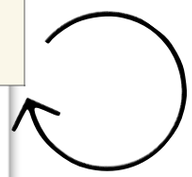


The functions on the stack are executed one by one until the stack becomes empty

Event loop



Event queue

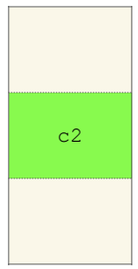


Pseudo code

```
// One callback
c1 = button.onClickDo {
  print("Hello")
}

// Another callback
c2 = button.onClickDo {
  print("World")
}
```

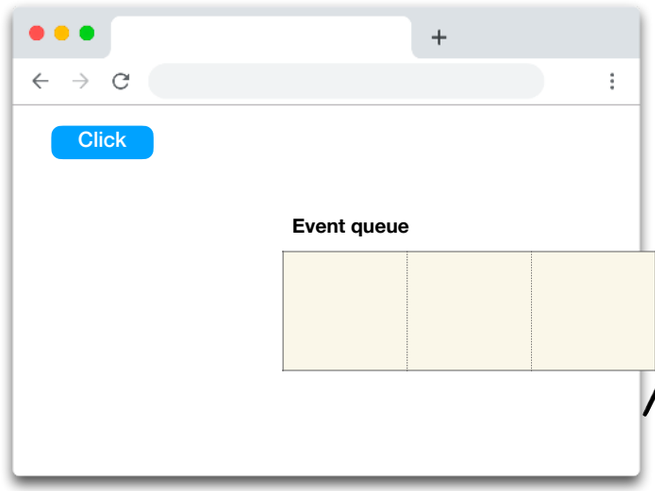
Stack

(code to execute before next event is dequeued)



The functions on the stack are executed one by one until the stack becomes empty

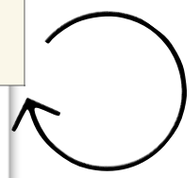
Event loop



Pseudo code

```
// One callback
c1 = button.onClickDo {
  print("Hello")
}

// Another callback
c2 = button.onClickDo {
  print("World")
}
```

Event queue

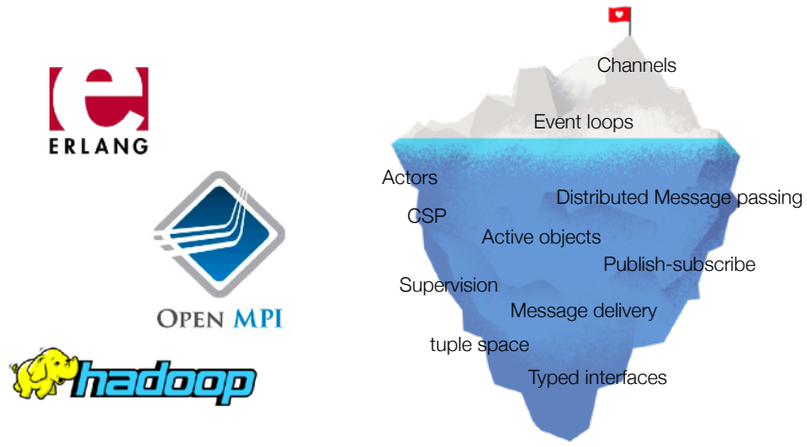


Stack

(code to execute before next event is dequeued)

When the stack becomes empty, the loop tries to dequeue the next event (or waits until there is a next event).

There is more…



Of course, there is more to see and the world is big. If you are interested to read further, here are some links:

Publish subscribe is a multi-threaded variant of event loop: https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern

Communication sequential processes are the basic idea behind channels: https://en.wikipedia.org/wiki/Communicating_sequential_processes

Some enhanced forms of the Actor model have an interesting form of error handling called supervision: https://doc.akka.io/docs/akka/current/general/supervision.html

There is no best…

Concept	Shared-state	Message Passing
Communication between threads	Shared memory region	Immutable messages sent
Thread-safety	Requires locking	No locking required (yet we can synchronise threads with blocking buffers)
Level of abstraction	Low	Medium
Communication speed	Fast	Can be slow (depends on implementation)
Debugging	Can be hard	Usually easier than with shared-state

From experiences, the best fitted model for building concurrent applications seems to be message passing. However, a lot of real world application rely on shared state concurrency. In fact, there is a good chance it is what you will see in the industry.

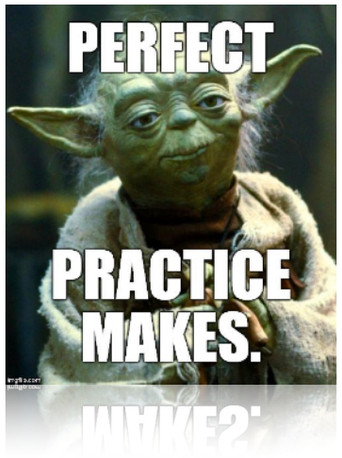
In the end, there is no “best” way to reason. The paradigms we saw are different, yet equivalent in terms of expressivity.

What you should take home is that there are many ways to solve problems. Message passing is better than shared state in some cases, while it is the opposite in others. It is up to you to choose the model which fits the most to your problem, as well as the one you feel most comfortable with.

Take aways - did I hold my promise?

- ✓ We saw message passing - another way to build concurrent programs
- ✓ We presented the Channel abstraction
- ✓ We showed in practice how a pipeline architecture could look like.
- ✓ We briefly presented other forms of message passing: event loop of JS.
- ✓ There is no “best”

Exercises



Producer consumer in Go

Creates channel of integers

```
channel := make(chan int)
```

“go” Starts a thread executing a function

Synchronously pushes element on the channel

```
// producer
go func() {
  for {
    channel <- produceElement()
  }
}()
```

Synchronously takes element from the channel

```
// consumer
for {
  i := <- channel
  fmt.Printf("i=%v\n", i)
}
```

Here is an example of the producer consumer problem with a buffer of size one written in Go. Quite elegant, isn’t it?

Resources

Google :-)

OCaml API doc: https://github.com/ocaml-multicore/domainslib/blob/0.4.2/lib/chan.mli

Going further with Message Passing: An excellent video explaining the actor model: https://www.youtube.com/watch?v=7erJ1DV_Tlo&ab_channel=jasonofthel33t

Understanding the broader scope: