# Message Passing Concurrency

An alternative to shared memory

# whoami

Hugues (“Hugh”) Evrard

Grenoble INP ENSIMAG

Inria

Imperial College London

GraphicsFuzz

since 2018: Google

*Opinions are my own and not the views of my employer.*

# Agenda

- Shared memory vs. message passing
- Taxonomy of message passing semantics
- Examples: bank account, barrier
- Concurrency: a very brief history
- Synchronous channels (CSP, Golang)
- Actor model (Erlang)
- Hardware

# Definition of *process* for this talk

***Process:*** *an instance executing a program*

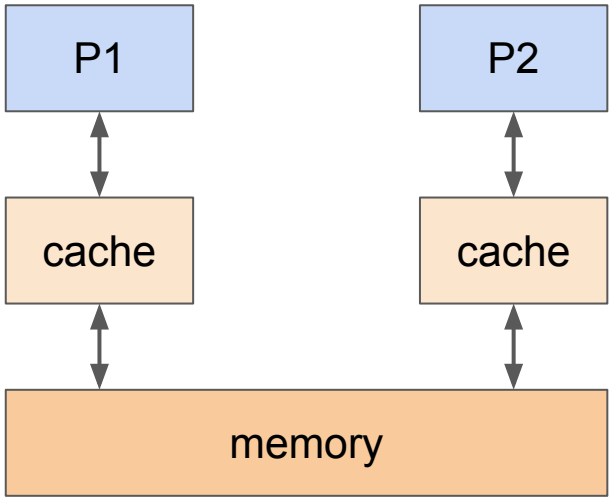
In practice: “a program counter and a stack”

- OS process, thread, programming language runtime routine, etc...

# What is message passing?

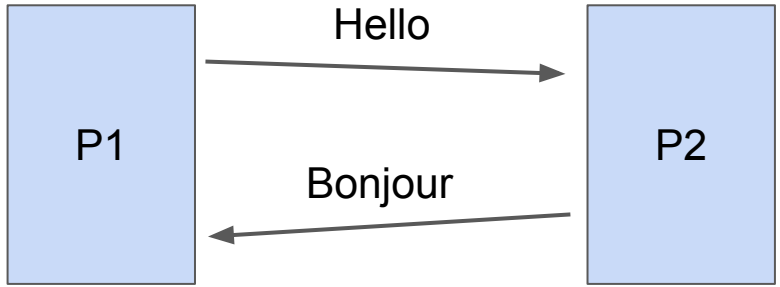
# Inter-process communication & synchronization

Shared memory




primitives: **read / write**

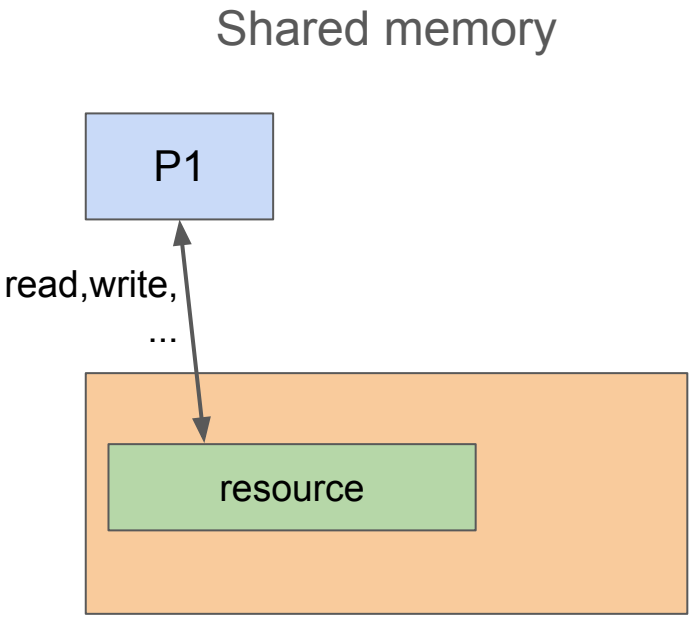
Message Passing



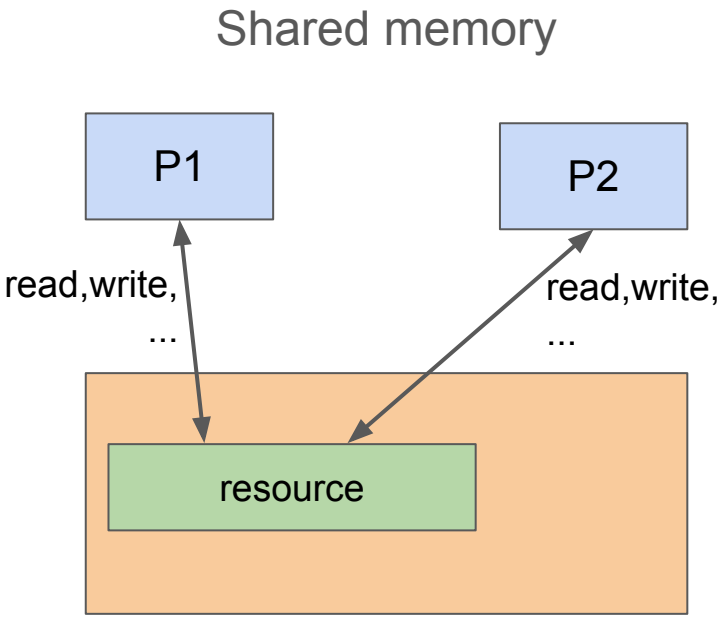

primitives: **send / receive**

# Why use message passing?

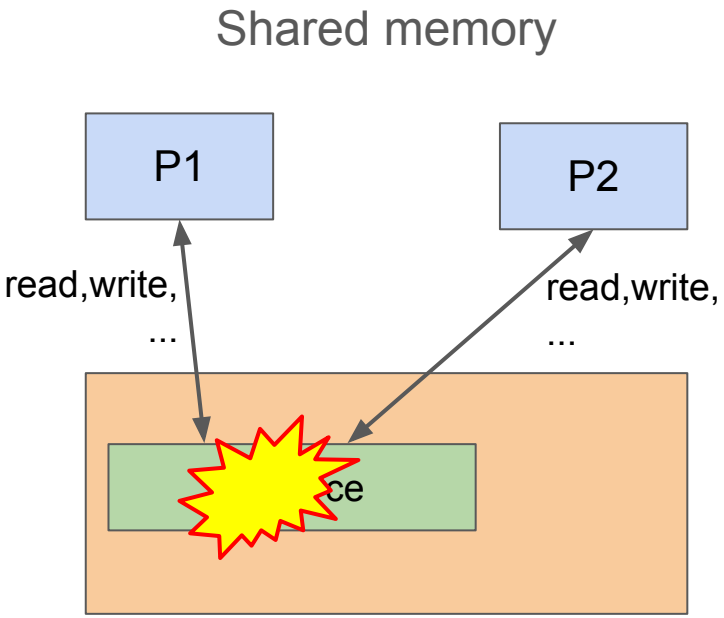
# Inter-process communication & synchronization



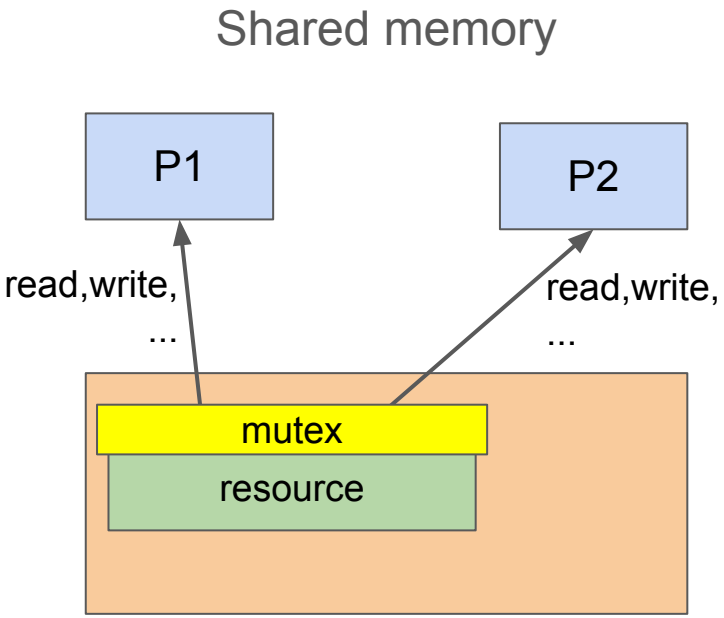
# Inter-process communication & synchronization



# Inter-process communication & synchronization



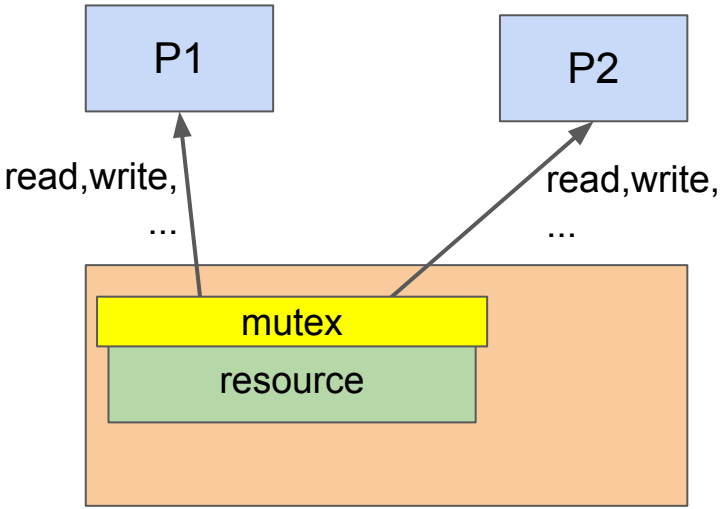
# Inter-process communication & synchronization




**mutex mis-management:**
**deadlocks / data corruption**

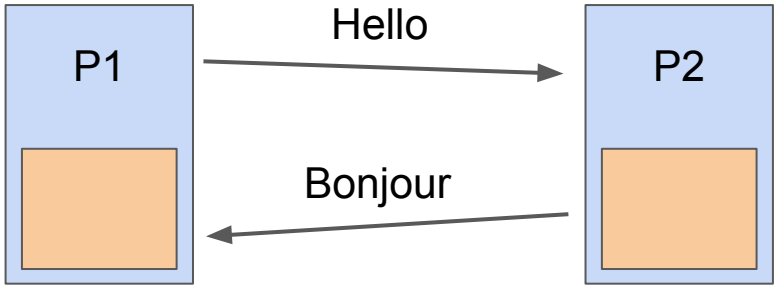
# Inter-process communication & synchronization

## Shared memory




**mutex mis-management:**
**deadlocks / data corruption**

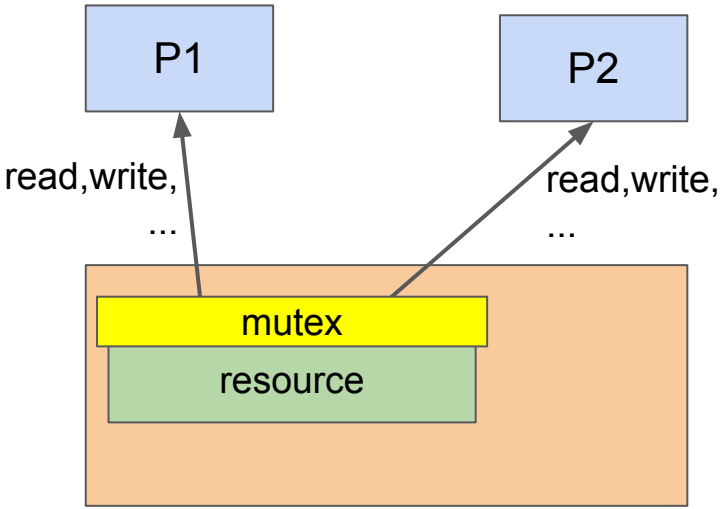
## Message Passing




- Primitives: **send / receive**
- Memory is **isolated, per-process**

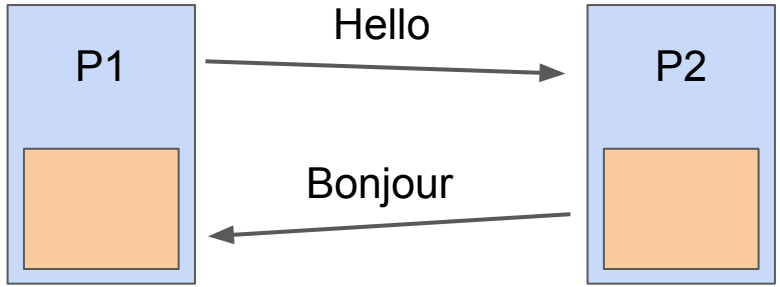
# Inter-process communication & synchronization

## Shared memory




**mutex mis-management:**
**deadlocks / data corruption**

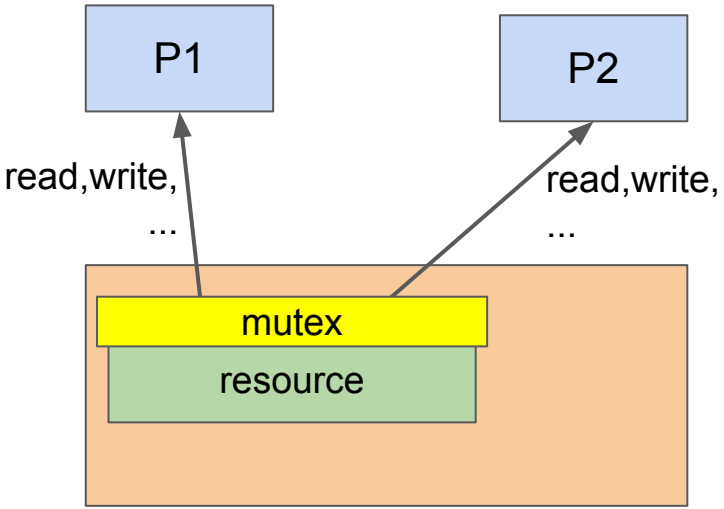
## Message Passing




- Primitives: **send / receive**
- Memory is **isolated, per-process**
- Only possible interaction is MP

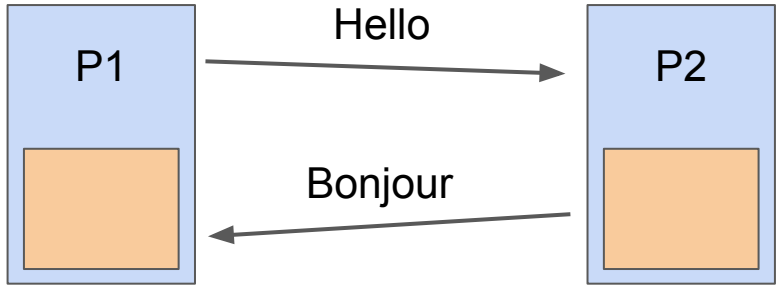
# Inter-process communication & synchronization

## Shared memory




**mutex mis-management: deadlocks / data corruption**

## Message Passing



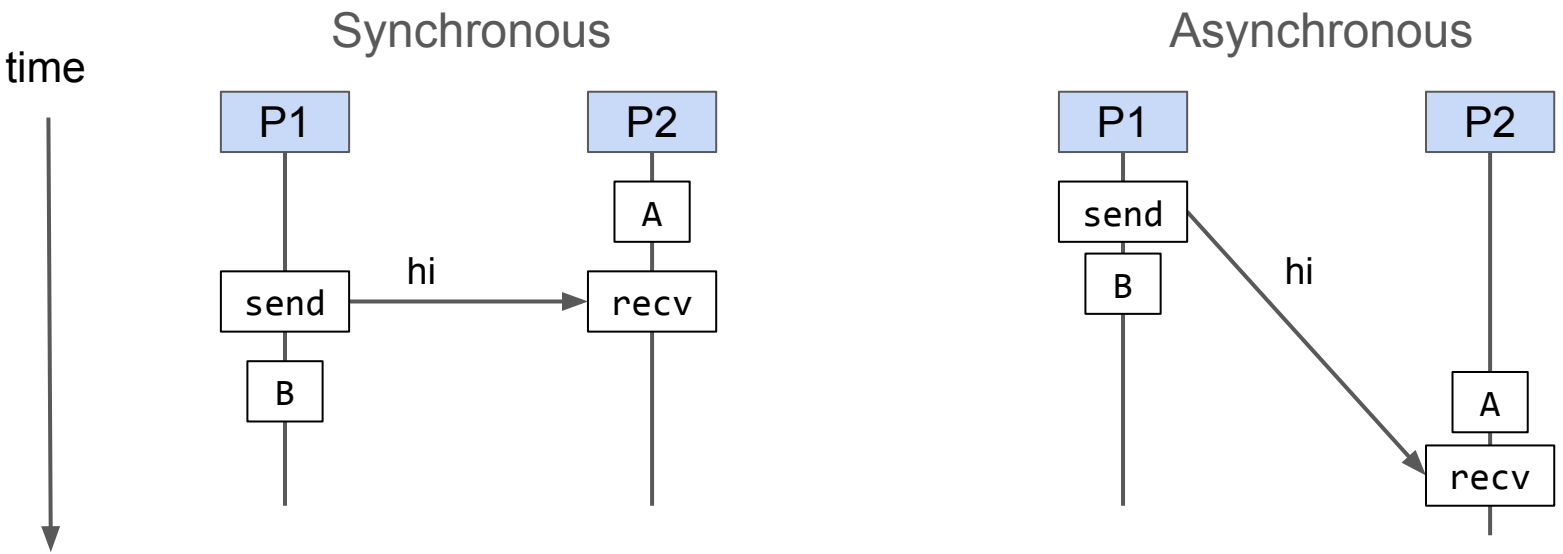

- Primitives: **send / receive**
- Memory is **isolated, per-process**
- Only possible interaction is MP
- **Cannot tamper a process without itself knowing it**

# Message passing semantics taxonomy

# Semantics: synchronous / asynchronous

```
P1 {
  send(P2, “hi”);
  B();
}
```

```
P2 {
  A();
  recv();
}
```



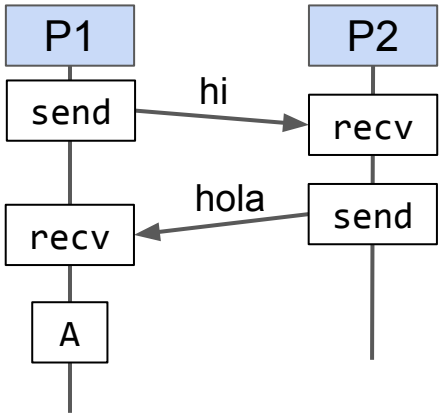
# Semantics: asynchronous: reliable / unreliable

```
P1 {
  send(P2, “hi”);
  recv();
  A();
}
```

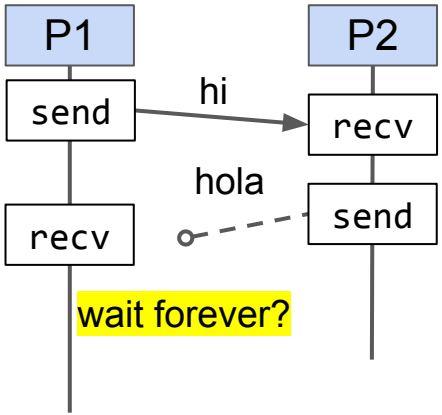
```
P2 {
  recv();
  send(P1, “hola”);
}
```

time

Reliable



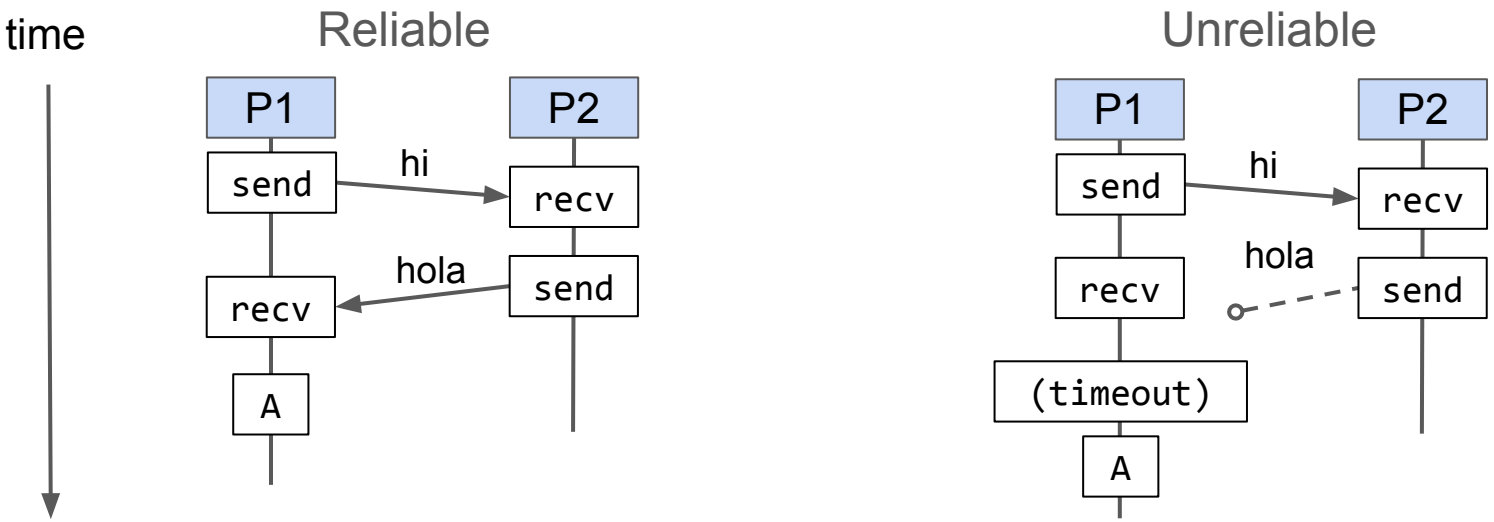

Unreliable



# Semantics: asynchronous: reliable / unreliable

```
P1 {
  send(P2, “hi”);
  recv(timeout=1s);
  A();
}
```

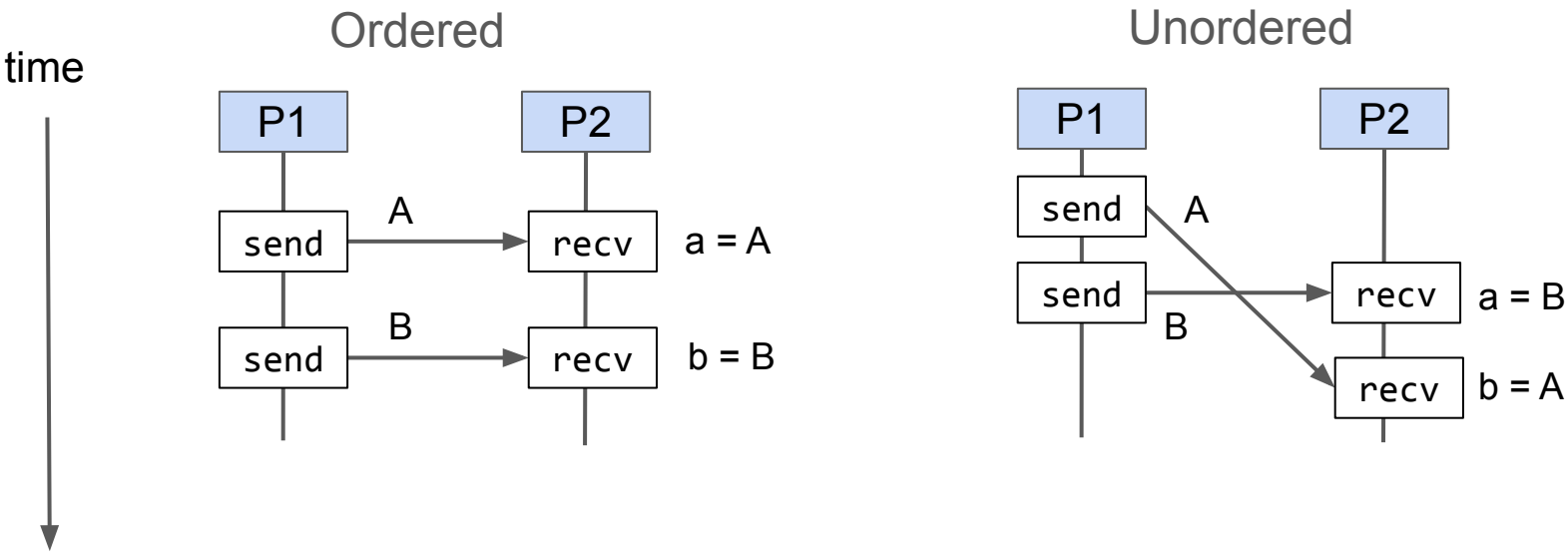
```
P2 {
  recv(timeout=1s);
  send(P1, “hola”);
}
```



# Semantics: asynchronous: ordered / unordered

```
P1 {
  send(P2, “A”);
  send(P2, “B”);
}
```

```
P2 {
  a = recv();
  b = recv();
}
```

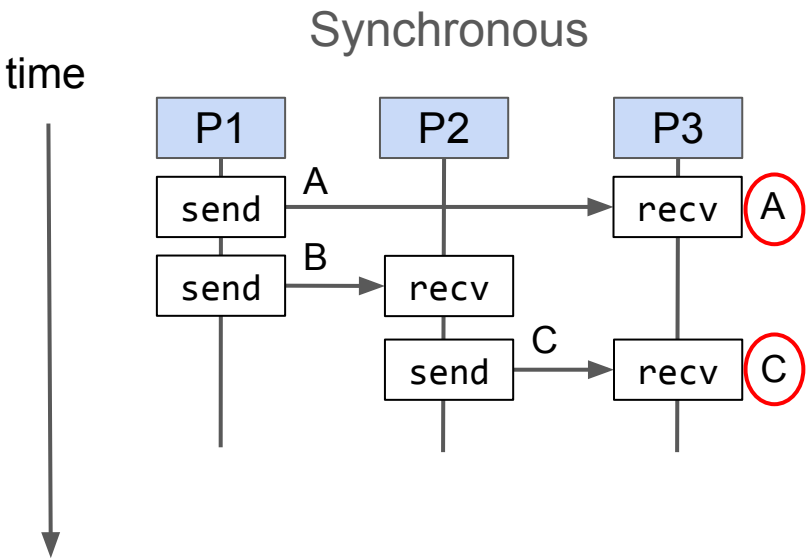


# Message order with more than 2 processes

```
P1 {
  send(P3, “A”);
  send(P2, “B”);
}
```

```
P2 {
  recv();
  send(P3, “C”);
}
```

```
P3 {
  recv();
  recv();
}
```



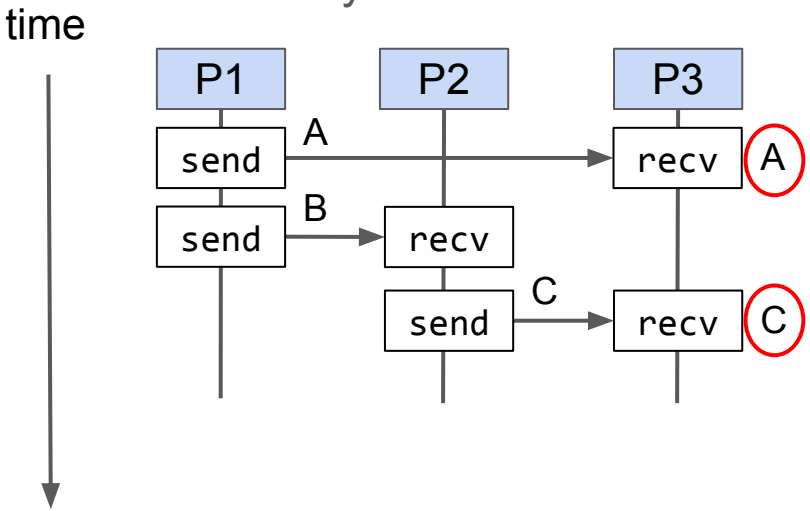
# Message order with more than 2 processes

```
P1 {
  send(P3, “A”);
  send(P2, “B”);
}
```

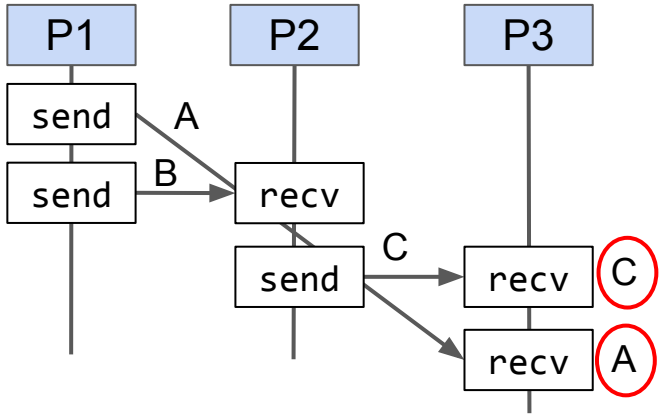
```
P2 {
  recv();
  send(P3, “C”);
}
```

```
P3 {
  recv();
  recv();
}
```

Synchronous




Asynchronous, **ordered**



# What to you send to?

So far, examples use a **process identifier**

```
P1 {
  send(P2, “foo”);
}
```

An alternative is to use **channels**

- named object to which you can send / receive
- channels can be **first-class citizens** (you can assign them to variables and pass them around)
- Unidirectional or bidirectional?

```
P1 (ch: channel) {
  send(ch, “bar”);
  ch2 = recv(ch);
  send(ch2, “baz”);
}
```

# What does recv() means?

- **Causality:** some process did a send() before
- If synchronous:
  - the sender has been blocking on send()
  - the sender is now aware of the reception

# Can you selectively receive?

- General recv(): receive **any** message, from **any** process

# Can you selectively receive?

- General recv(): receive **any** message, from **any** process
- Can you selectively receive:
  - From only a specific process / channel?

```
ch1, ch2, ch3

select {
case recv(ch1): …
case recv(ch2): …
}
recv(ch3);
```

# Can you selectively receive?

- General recv(): receive **any** message, from **any** process
- Can you selectively receive:
  - From only a specific process / channel?
  - Only receive certain values of messages? (“*guarded commands*”)

```
ch1, ch2, ch3

select {
case recv(ch1): …
case recv(ch2): …
}
recv(ch3);
```

```
ch1, ch2, ch3

select {
case i := recv(ch1) where i > 5: // send(ch1, 2) would block
  …
case recv(ch2):
  …
}
recv(ch3);
```

# Message passing semantics: recap

Message passing can be:

- Synchronous
  - Sender/receiver both blocks waiting for the other one
- Asynchronous
  - reliable or not
  - ordered or not
- Send to process identifiers or first-class citizen channels
- Ability to selectively receive, or not

# Message passing semantics: recap

Message passing can be:

- Synchronous
  - Sender/receiver both blocks waiting for the other one
- Asynchronous
  - reliable or not
  - ordered or not
- Send to process identifiers or first-class citizen channels
- Ability to selectively receive, or not
- Also
  - unidirectional / bidirectional, whole/partial message, …
  - Can express asynchronous on top of synchronous (easy) and vice-versa (harder)

# Message passing semantics: recap

Message passing can be:

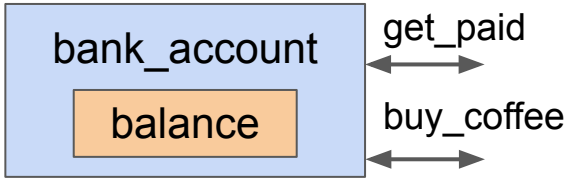
- Synchronous
  - Sender/receiver both blocks wait[illegible]
- Asynchronous
  - reliable or not
  - ordered or not
- Send to process identifiers or first-class citizen channels
- Ability to selectively receive, or not
- Also
  - unidirectional / bidirectional, whole/partial message, …
  - Can express asynchronous on top of synchronous (easy) and vice-versa (harder)

Network protocols:
- TCP: asynchronous, ordered, reliable ( 🤞 )
- UDP: asynchronous, unordered, unreliable

# Examples

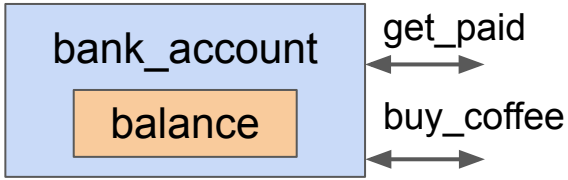
# Example: bank account

```
bank_account (get_paid, buy_coffee: channel int) {
  balance = 0
  for {
    select {

    case i = recv(get_paid):
      balance += i

    case i = recv(buy_coffee):
      balance -= i

    }
  }
}
```



# Example: bank account with guarded reception

```
bank_account (get_paid, buy_coffee: channel int) {
  balance = 0
  for {
    select {

    case i = recv(get_paid):
      balance += i

    case i = recv(buy_coffee) where i <= balance:
      balance -= i

    }
  }
}
```

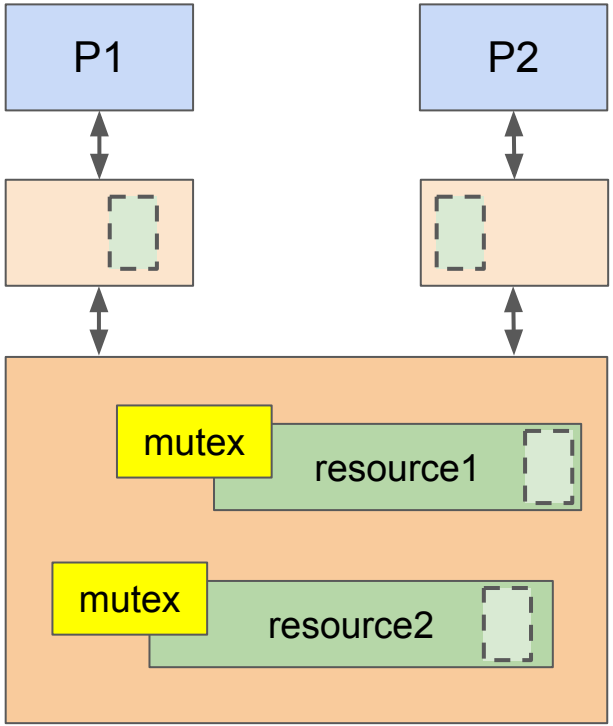


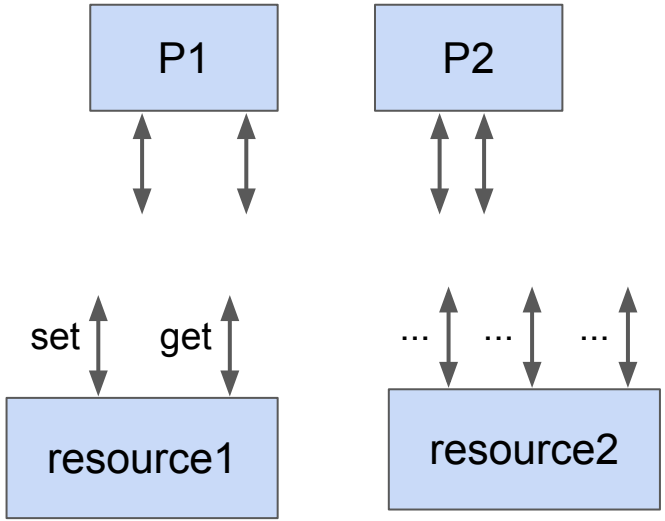

*Avoids negative balance*

*But: if synchronous, sender will block!*

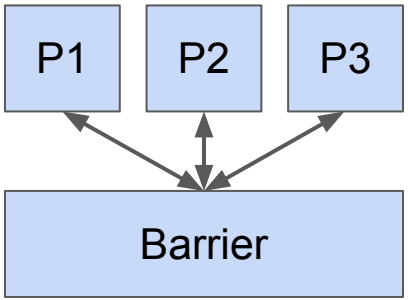
*It may be OK, as it will block until get_paid adds enough money to unblock.*

# Shared resources become processes



# Example: barrier




```
P (ch: channel) {
  ...
  // barrier synchronization
  send(ch)
  recv(ch)
  ...
}
```

```
Barrier (n: int, ch: channel) {
  for (i = 0; i < n; i++) {
    recv(ch)
  }
  for (i = 0; i < n; i++) {
    send(ch)
  }
}
```

# Step back:
# Why concurrent programming?

# Why concurrent programming? A very brief history

- Big bang: many things happen concurrently

https://upload.wikimedia.org/wikipedia/commons/thumb/a/ae/Bigbang.png/1024px-Bigbang.png, Pablo Carlos Budassi

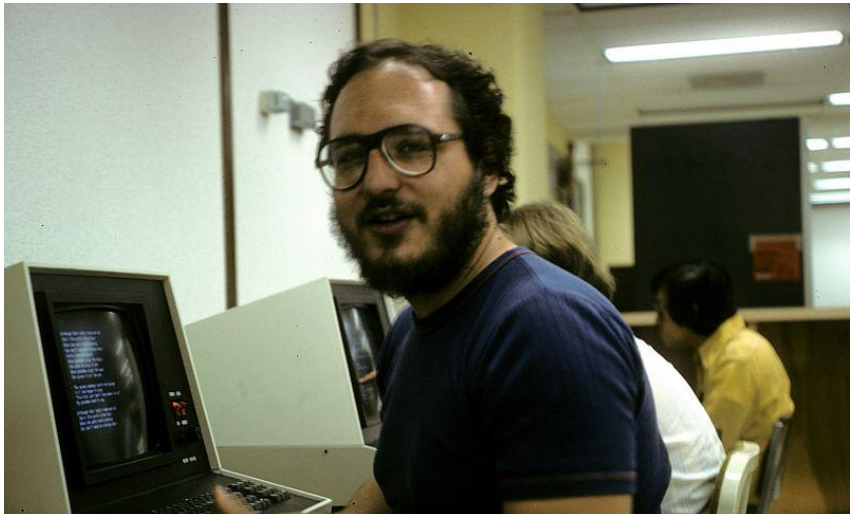
# Why concurrent programming? A very brief history

- Big bang: many things happen concurrently
- ...
- 40s: single processor, “batch” execution of single program

IBM 1401, http://ed-thelen.org/comp-hist/BRL61-0526.jpg

# Why concurrent programming? A very brief history

- Big bang: many things happen concurrently
- ...
- 40s: single processor, “batch” execution of single program
- 60s: ***time sharing*** on a single processor



Dave Winer, https://en.wikipedia.org/wiki/File:Unix_Timesharing_UW-Madison_1978.jpeg

# Why concurrent programming? A very brief history

- Big bang: many things happen concurrently
- ...
- 40s: single processor, “batch” execution of single program
- 60s: ***time sharing*** on a single processor
- 65: Dijkstra’s *Solution of a problem in concurrent programming control*

# Why concurrent programming? A very brief history

- Big bang: many things happen concurrently
- ...
- 40s: single processor, “batch” execution of single program
- 60s: ***time sharing*** on a single processor
- 65: Dijkstra’s *Solution of a problem in concurrent programming control*
- 70s: single processors talk over networks
- 78: Hoare’s *Communicating Sequential Processes* (also: Milner’s CCS,…)
- 87: Erlang
- 2000s: multi-core processors with shared memory

# Why concurrent programming? A very brief history

- Big bang: many things happen concurrently
- ...
- 40s: single processor, “batch” execution of single program
- 60s: ***time sharing*** on a single processor
- 65: Dijkstra’s *Solution of a problem in concurrent programming control*
- 70s: single processors talk over networks
- 78: Hoare’s *Communicating Sequential Processes* (also: Milner’s CCS,…)
- 87: Erlang
- 2000s: multi-core processors with shared memory

**Decades** before shared-memory multi-core processors, concurrency mattered as **a way to design programs**.

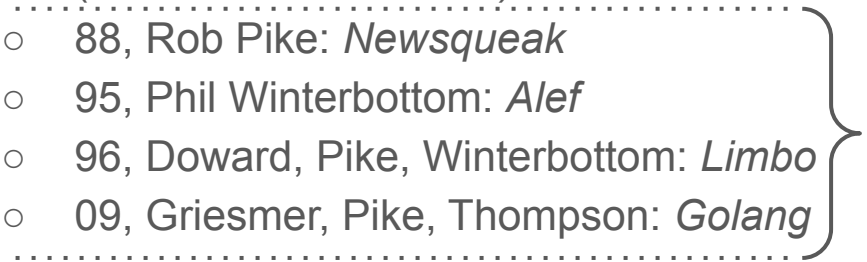
# Synchronous channels

# Programming languages with synchronous channels

- Incomplete timeline:
  - ...
  - 78, Tony Hoare: *CSP*
  - 83, David May: *occam*
  - (83, Jean Ichbiah: *Ada)*
  - 88, Rob Pike: *Newsqueak*
  - 95, Phil Winterbottom: *Alef*
  - 96, Doward, Pike, Winterbottom: *Limbo*
  - 09, Griesmer, Pike, Thompson: *Golang*

# Programming languages with synchronous channels

- Incomplete timeline:
  - ...
  - 78, Tony Hoare: *CSP*
  - 83, David May: *occam*
  - (83, Jean Ichbiah: *Ada)*
  - 88, Rob Pike: *Newsqueak*
  - 95, Phil Winterbottom: *Alef*
  - 96, Doward, Pike, Winterbottom: *Limbo*
  - 09, Griesmer, Pike, Thompson: *Golang*



“Bell Labs” family

# Programming languages with synchronous channels

- Incomplete timeline:
  - ...
  - 78, Tony Hoare: *CSP*
  - 83, David May: *occam*
  - (83, Jean Ichbiah: *Ada)*
  - 88, Rob Pike: *Newsqueak*
  - 95, Phil Winterbottom: *Alef*
  - 96, Doward, Pike, Winterbottom: *Limbo*
  - 09, Griesmer, Pike, Thompson: *Golang*

“Bell Labs” family

- As a library:
  - mid-90s: “Java CSP”
  - 2013: Clojure “core.async”
  - …

# Programming languages with synchronous channels

- Incomplete timeline:
  - ...
  - 78, Tony Hoare: *CSP*
  - 83, David May: *occam*
  - ...
  - mid 90s: Java CSP
  - 2013: Clojure core.async
  - ...
- As a library:
  - mid 90s: Java CSP
  - 2013: Clojure core.async
  - ...

***The roots of this style*** *go back at least as far as Hoare's Communicating Sequential Processes (**CSP**), followed by realizations and extensions in e.g. occam, Java CSP and the Go programming language.*

*[...] the notion of a channel becomes **first class** […]*

*A key characteristic of channels is that they are **blocking**. […]*

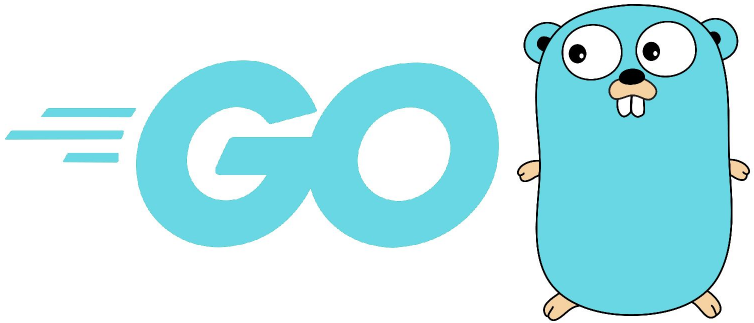
https://clojure.org/news/2013/06/28/clojure-clore-async-channels

# Programming languages with synchronous channels

- Incomplete timeline:
  - ...
  - 78, Tony Hoare: *CSP*
  - 83, David May: *occam*
  - (83, Jean Ichbiah: *Ada)*
  - 88, Rob Pike: *Newsqueak*
  - 95, Phil Winterbottom: *Alef*
  - 96, Doward, Pike, Winterbottom: *Limbo*
  - 09, Griesmer, Pike, Thompson: *Golang*
- Golang channels: bidirectional, typed, optionally buffered (i.e. asynchronous), no guarded receive.

- As a library:
  - mid-90s: “Java CSP”
  - 2013: Clojure “core.async”
  - …

“Bell Labs” family



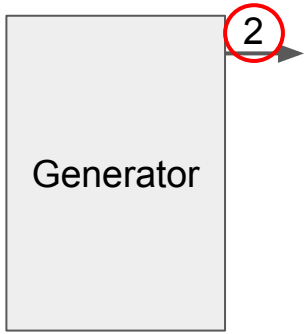
# Programming languages with synchronous channels

- Incomplete timeline:
  - ...
  - 78, Tony Hoare: *CSP*
  - 83, David May: *occam*
  - (83, Jean Ichbiah: *Ada)*
  - 88, Rob Pike: *Newsqueak*
  - 95, Phil Winterbottom: *Alef*
  - 96, Doward, Pike, Winterbottom: *Limbo*
  - 09, Griesmer, Pike, Thompson: *Golang*
- Golang channels: bidirectional, typed, optionally buffered (i.e. asynchronous), no guarded receive.

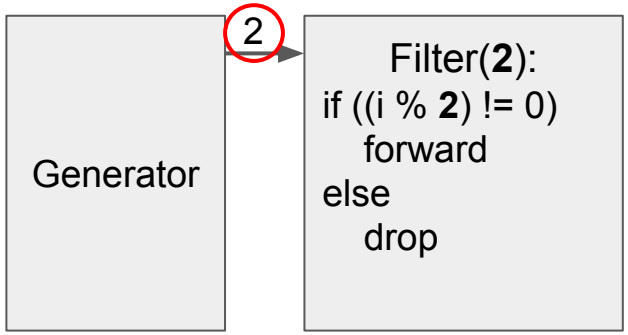
```
// This is Golang code

func counter(ch chan int) {
  i := 1;
  for {
    i++;
    ch <- i  // send
  }
}

func main() {
  ch := make(chan int) // create a channel
  go counter(ch);      // launch a process
  j <- ch              // recv, j == 2
  j <- ch              // j == 3
  j <- ch              // j == 4
}
```

https://play.golang.org/p/1PE4jTg1tNa

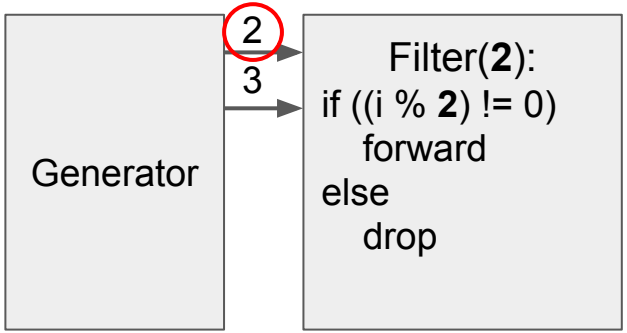
# Example: prime sieve (McIlroy)



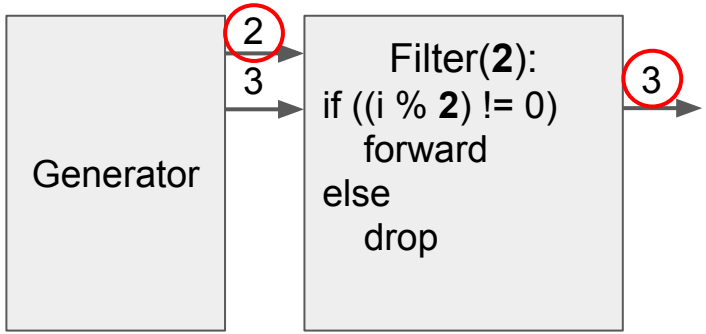
# Example: prime sieve (McIlroy)



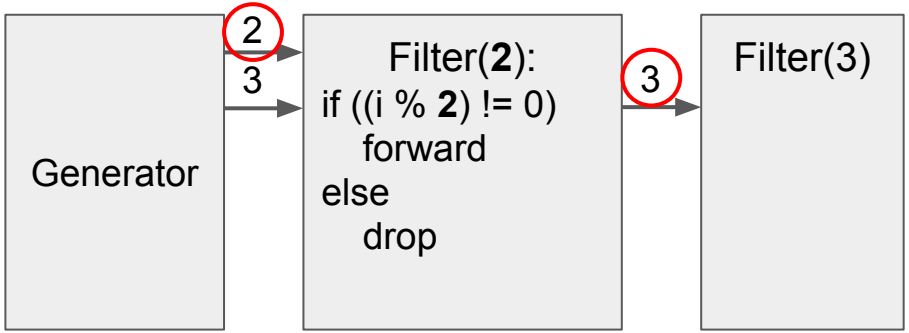
# Example: prime sieve (McIlroy)



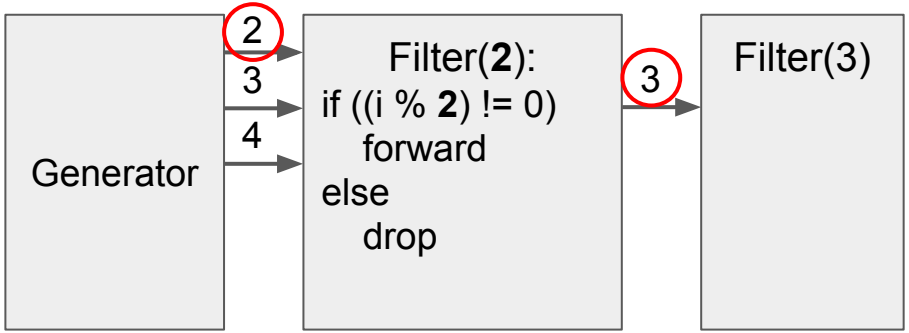
# Example: prime sieve (McIlroy)



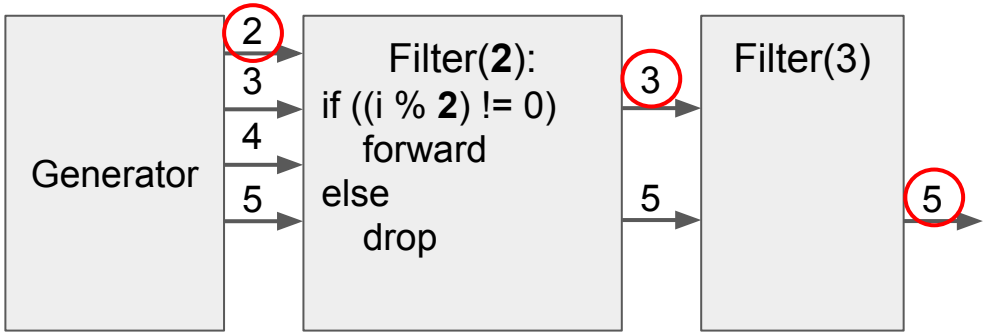
# Example: prime sieve (McIlroy)



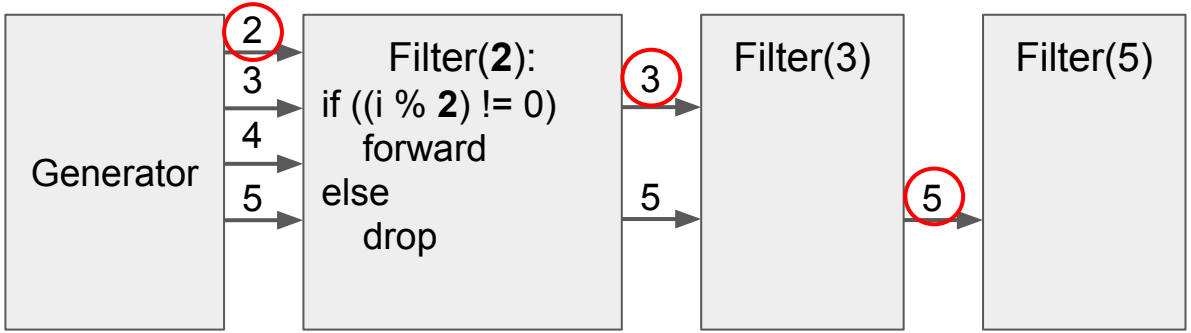
# Example: prime sieve (McIlroy)



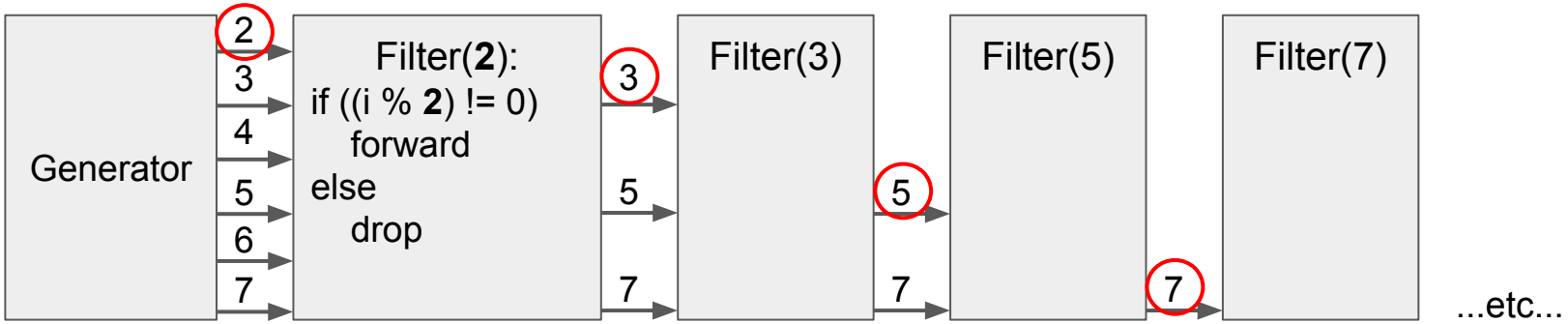
# Example: prime sieve (McIlroy)



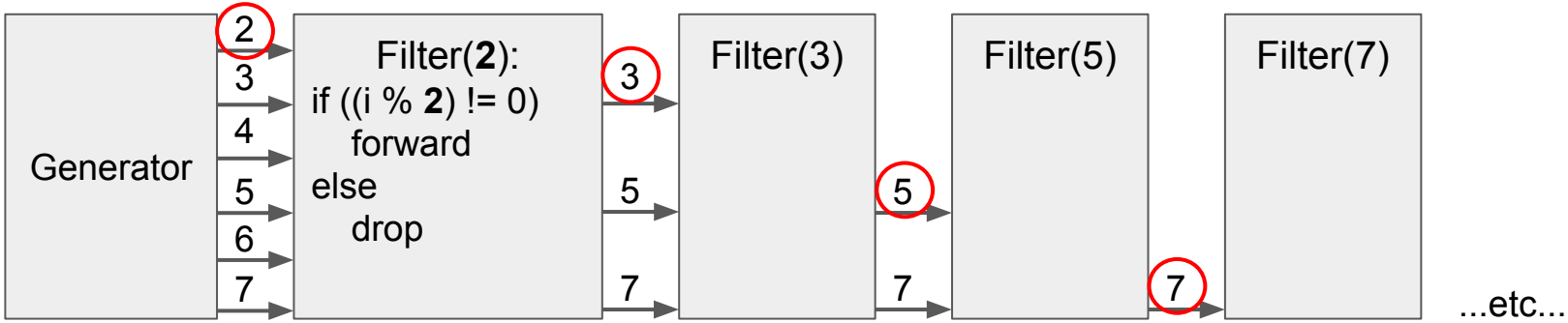
# Example: prime sieve (McIlroy)



# Example: prime sieve (McIlroy)

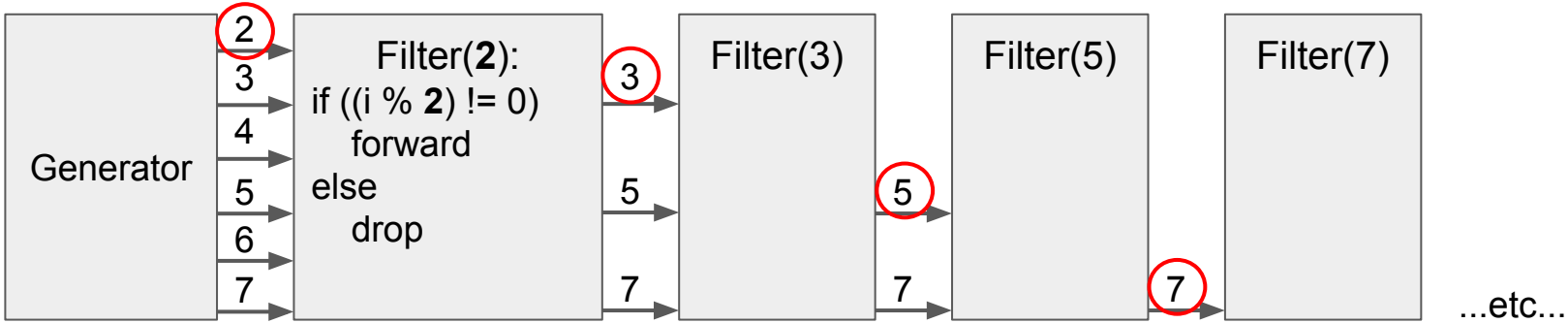


# Example: prime sieve (McIlroy)




```
func filter(prime int,
            src, next chan int) {
  for {
    i := <-src
    if (i % prime) != 0 {
      next <- i
    }
  }
}
```

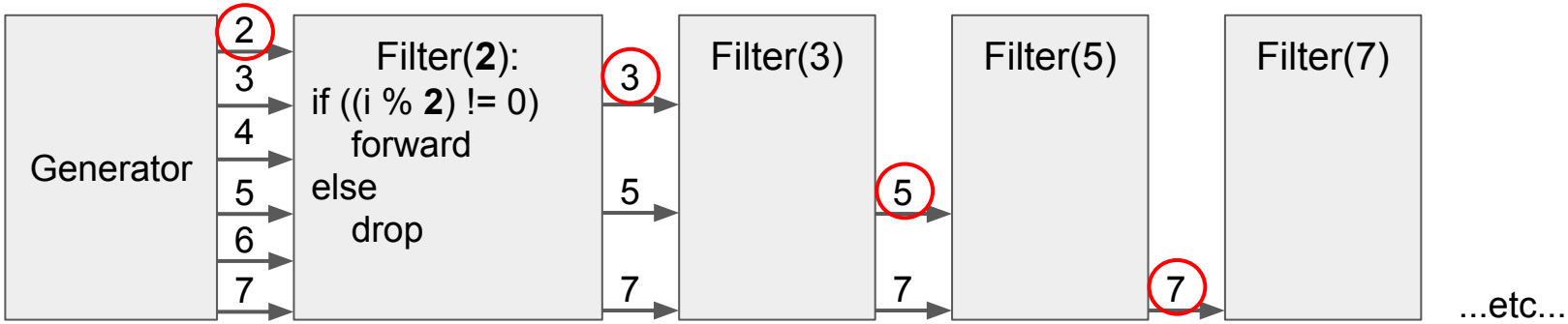
# Example: prime sieve (McIlroy)




```
func filter(prime int,
            src, next chan int) {
  for {
    i := <-src
    if (i % prime) != 0 {
      next <- i
    }
  }
}
```

```
func sieve(result chan int) {
  ch := make(chan int)
  go counter(ch)
  for {
    i := <-ch
    result <- i
    next := make(chan int)
    go filter(i, ch, next)
    ch = next
  }
}
```

# Example: prime sieve (McIlroy)




```
func filter(prime int,
            src, next chan int) {
  for {
    i := <-src
    if (i % prime) != 0 {
      next <- i
    }
  }
}
```

```
func sieve(result chan int) {
  ch := make(chan int)
  go counter(ch)
  for {
    i := <-ch
    result <- i
    next := make(chan int)
    go filter(i, ch, next)
    ch = next
  }
}
```

```
func main() {
  r := make(chan int)
  go sieve(r)
  <-r // 2
  <-r // 3
  <-r // 5
  <-r // 7
  <-r // 11
  <-r // 13
}
```

https://play.golang.org/p/xDKhkm6ZDIa

# Golang is not “purely” message passing

- You can have shared memory, mutexes, etc
- This is discouraged

# Golang is not “purely” message passing

- You can have shared memory, mutexes, etc
- This is discouraged

> *Do not communicate by sharing memory; instead, share memory by communicating.*

https://blog.golang.org/codelab-share

# Asynchronous, “pure” message passing

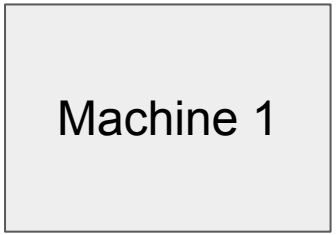
(Actor model)
Concurrency for reliability

# Asynchronous, “pure” message passing

(Actor model)
Concurrency for reliability

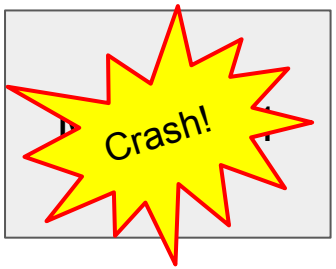
# Origin: ***reliability***

*Thesis: “Making reliable distributed systems in the presence of software errors”*



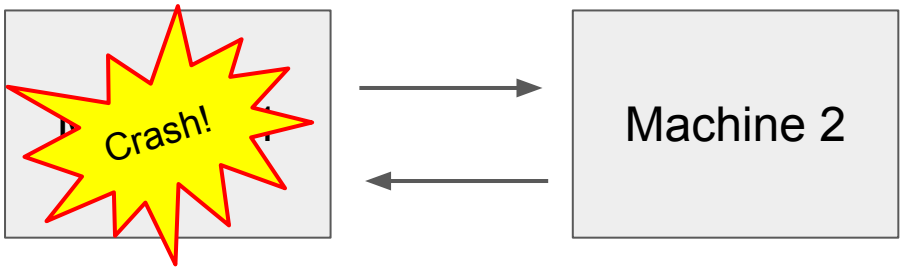
# Origin: ***reliability***

*Thesis: “Making reliable distributed systems in the presence of software errors”*



# Origin: ***reliability***

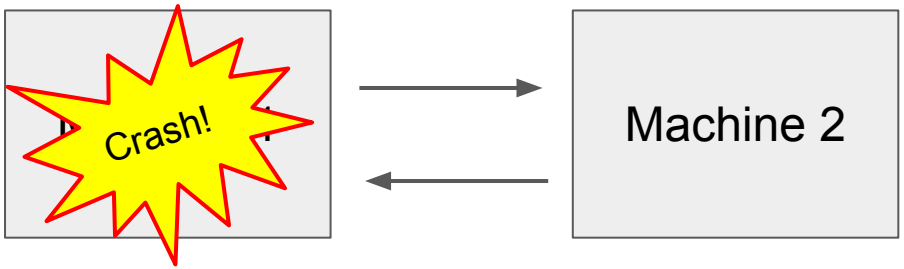
*Thesis: “Making reliable distributed systems in the presence of software errors”*




- Need at least 2 separate machines

# Origin: ***reliability***

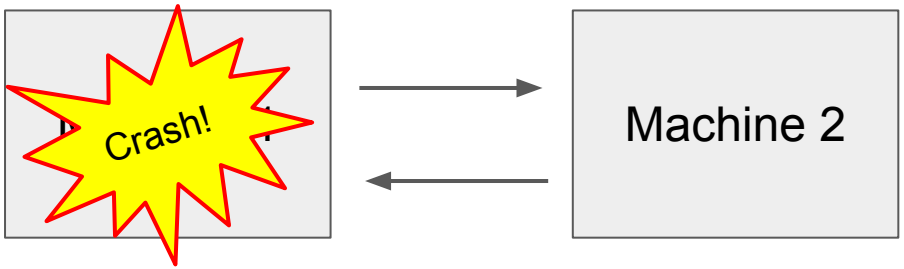
*Thesis: “Making reliable distributed systems in the presence of software errors”*




- Need at least 2 separate machines
- Do not share memory
- Send messages over a network

# Origin: ***reliability***

*Thesis: “Making reliable distributed systems in the presence of software errors”*




- Need at least 2 separate machines
- Do not share memory
- Send messages over a network
- Synchronous? No! If sender/receiver crashes, the other deadlocks
- *Asynchronous message passing between isolated processes*

# Origin: ***reliability***

*Thesis: "Making reliable distributed systems in the presence of software errors"*

*The process provides a clean unit of modularity, service, fault containment and failure. Fault containment through fail-fast software modules. The process achieves fault containment by sharing no state with other processes; its only contact with other processes is via messages carried by a kernel message system.*

1985, Jim Gray: Why do computers stop and what can be done about it?

- Synchronous? No! If sender/receiver crashes, the other deadlocks
- *Asynchronous message passing between isolated processes*

# A taste of Erlang

- Sequential part: functional programming

```
foo() -> ...        // define func foo with arity 0
```

# A taste of Erlang

- Sequential part: functional programming
- MP is only possible interaction, strictly no shared memory
- MP: asynchronous, ordered, unreliable
  - *“Send and pray”*

```
foo() -> ...          // define func foo with arity 0

Pid = spawn(foo/0) // launch a process, get its ID

Pid ! msg             // send a message to Pid
```

# A taste of Erlang

- Sequential part: functional programming
- MP is only possible interaction, strictly no shared memory
- MP: asynchronous, ordered, unreliable
  - *“Send and pray”*
- Powerful receive primitive:
  - Pattern-matching
  - Guards
  - Timeout

```
foo() -> …          // define func foo with arity 0

Pid = spawn(foo/0) // launch a process, get its ID

Pid ! msg           // send a message to Pid


receive             // pattern-matching guarded recv
   Pattern [when Guard] -> …
   Pattern [when Guard] -> …
   after timeout -> …
end
```

# A taste of Erlang

- Sequential part: functional programming
- MP is only possible interaction, strictly no shared memory
- MP: asynchronous, ordered, unreliable
  - *“Send and pray”*
- Powerful receive primitive:
  - Pattern-matching
  - Guards
  - Timeout
- Fail-fast, “*let it crash*”
- Crash propagated to *linked* processes
- Crash notification as a message to *monitoring* processes

```
foo() -> …          // define func foo with arity 0

Pid = spawn(foo/0) // launch a process, get its ID

Pid ! msg           // send a message to Pid


receive             // pattern-matching guarded recv
   Pattern [when Guard] -> …
   Pattern [when Guard] -> …
   after timeout -> …
end


P = spawn_link(foo/0) // P crash ⇒ local crash

Ref = monitor(P)      // if P crash, local recv msg
```

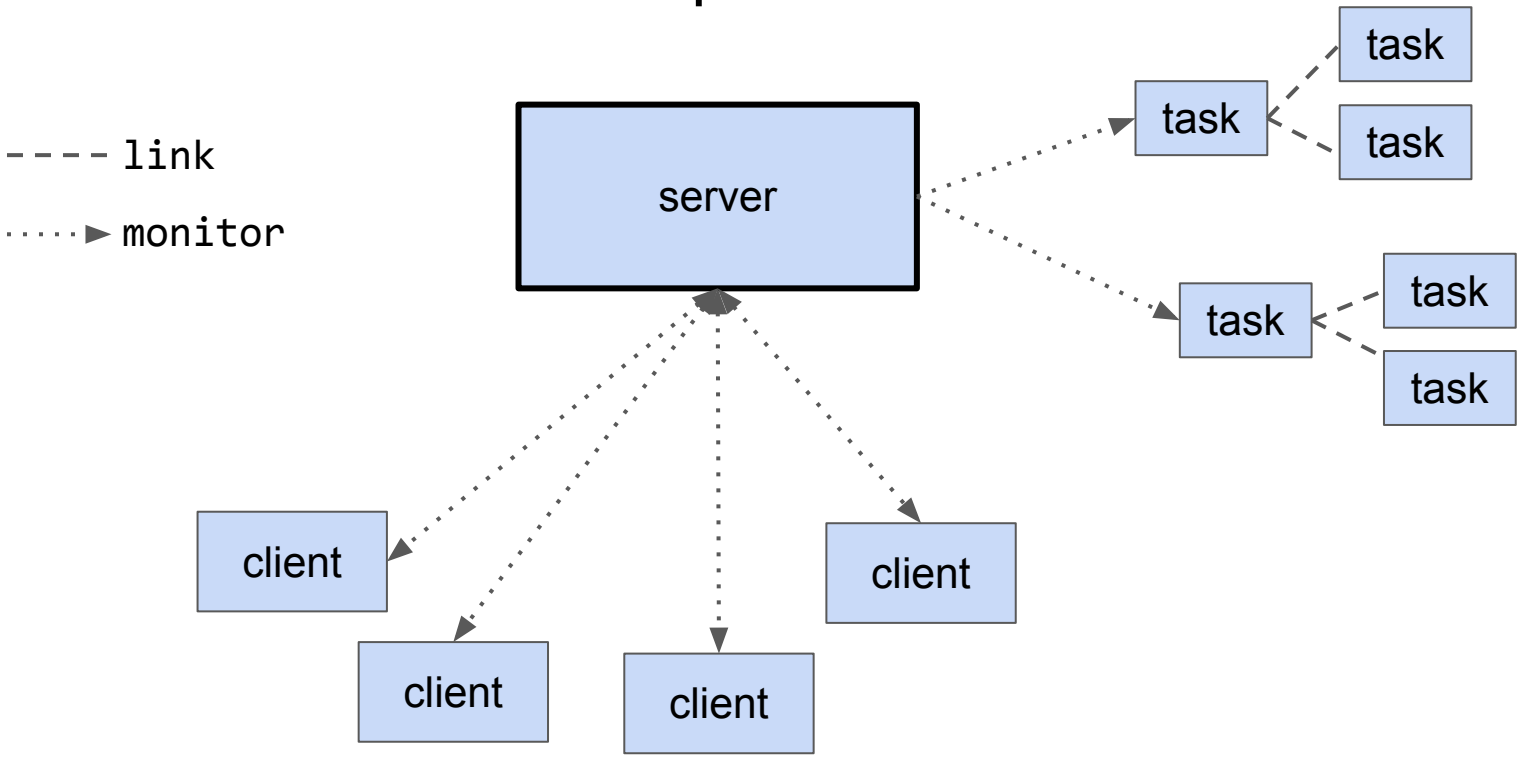
# A taste of ~~Erlang~~ Elixir

The new cool kid on top of Erlang VM!
(nicer syntax, similar concepts)

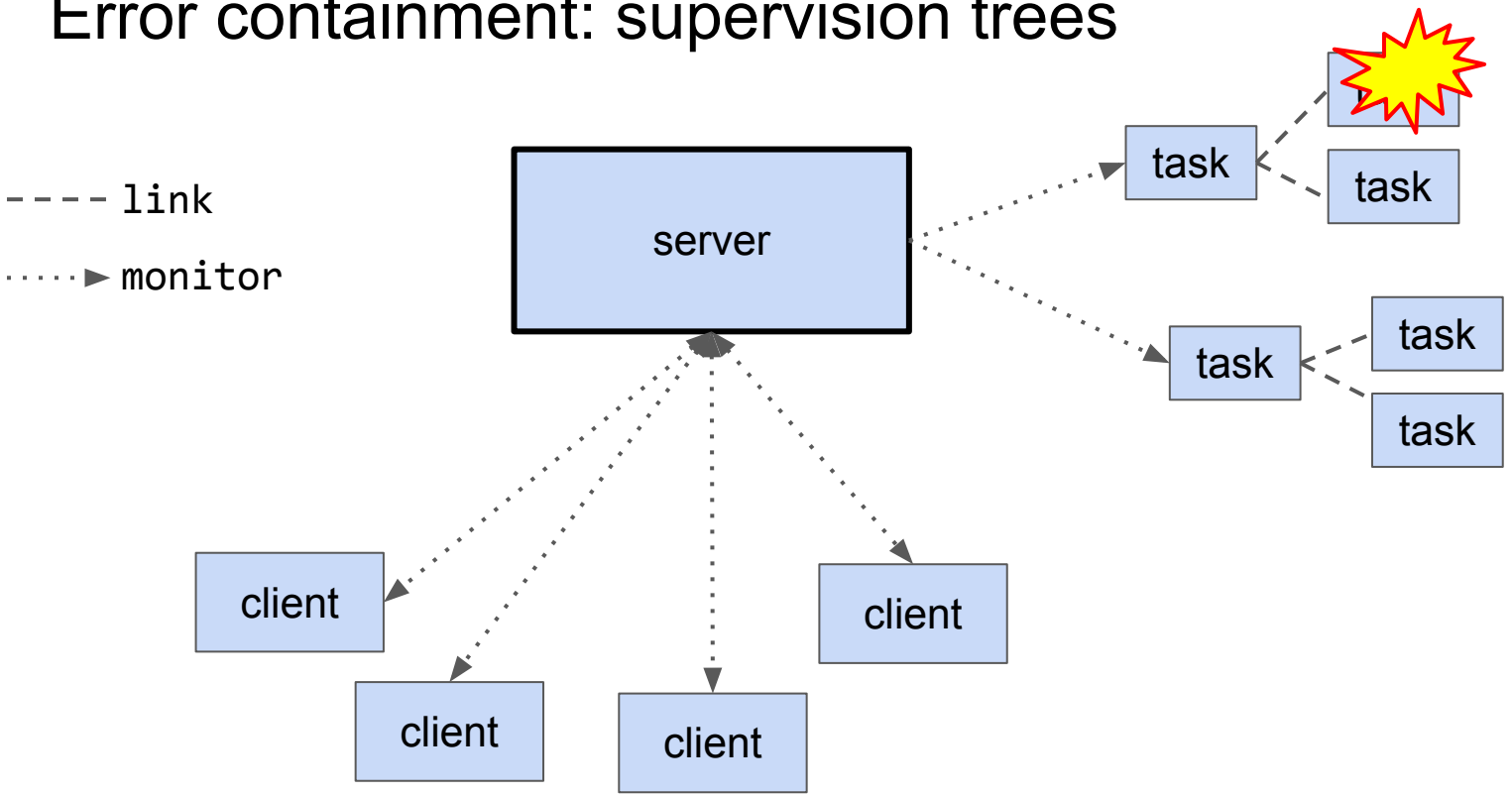
- Sequential part: functional programm
- MP is only possible interaction, strictly no shared memory
- MP: asynchronous, ordered, unreliab
  - *“Send and pray”*
- Powerful receive primitive:
  - Pattern-matching
  - Guards
  - Timeout
- Fail-fast, “*let it crash*”
- Crash propagated to *linked* processes
- Crash notification as a message to *monitoring* processes


```
rity 0

its ID

ed recv

   Pattern [when Guard] -> ...
   after timeout -> ...
end


P = spawn_link(foo/0) // P crash ⇒ local crash

Ref = monitor(P)      // if P crash, local recv msg
```

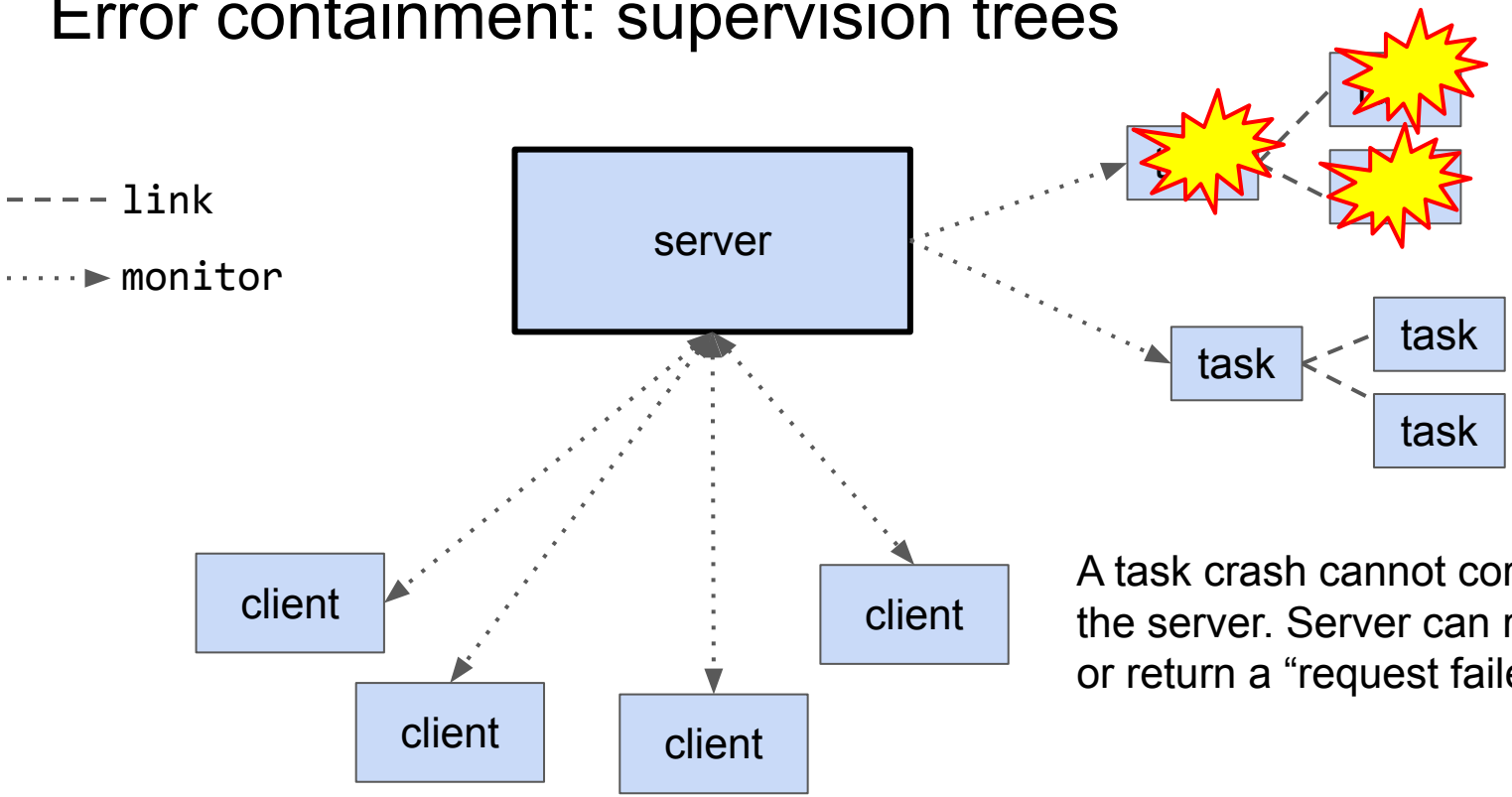
# Error containment: supervision trees



# Error containment: supervision trees



# Error containment: supervision trees




A task crash cannot corrupt the state of the server. Server can restart the task, or return a “request failed” to the client.

# Is Erlang used in industry?


- Telecom companies
  - **Ericsson** (this is where it started! **Er**icsson **Lang**uage)
  - **Nortel**, **T-Mobile**, …
  - Reliability first! 99,999% uptime needed
  - Erlang VM can do hot code reload, no downtime



- **WhatsApp**, 2015: **50 engineers, 900 millions users** (Wired article)
- **Heroku** (Elixir)


- **AliBaba**

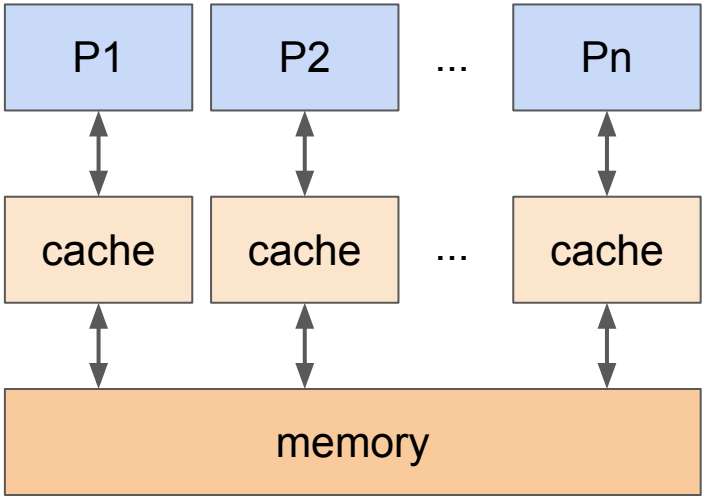

- **EaseMob** (Chinese comm framework, reportedly 1 billion users)


- **RabbitMQ**


- …

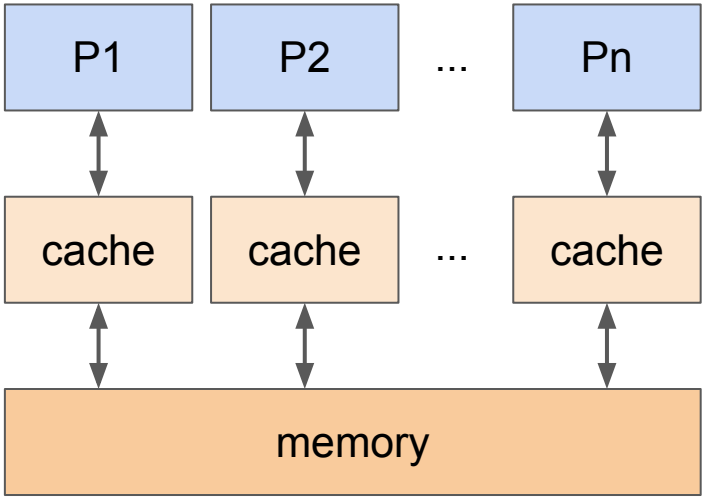
# Hardware

# Message passing at the hardware level

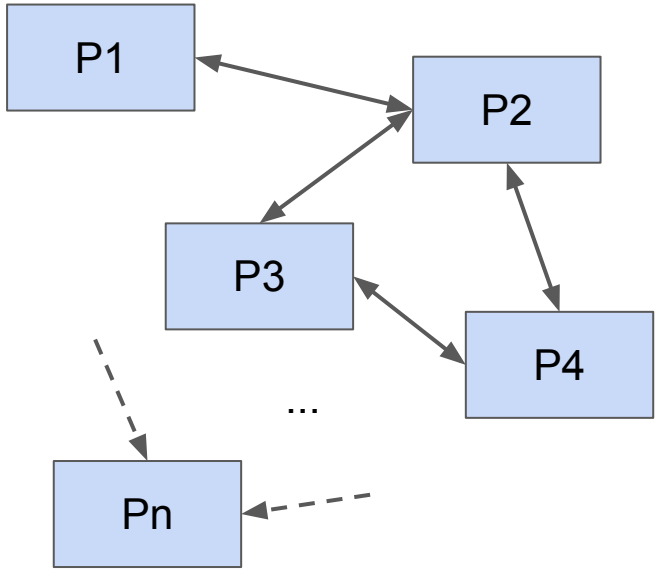



Memory coherency: *hard to scale!*
Nowadays, **n < 100**

# Message passing at the hardware level

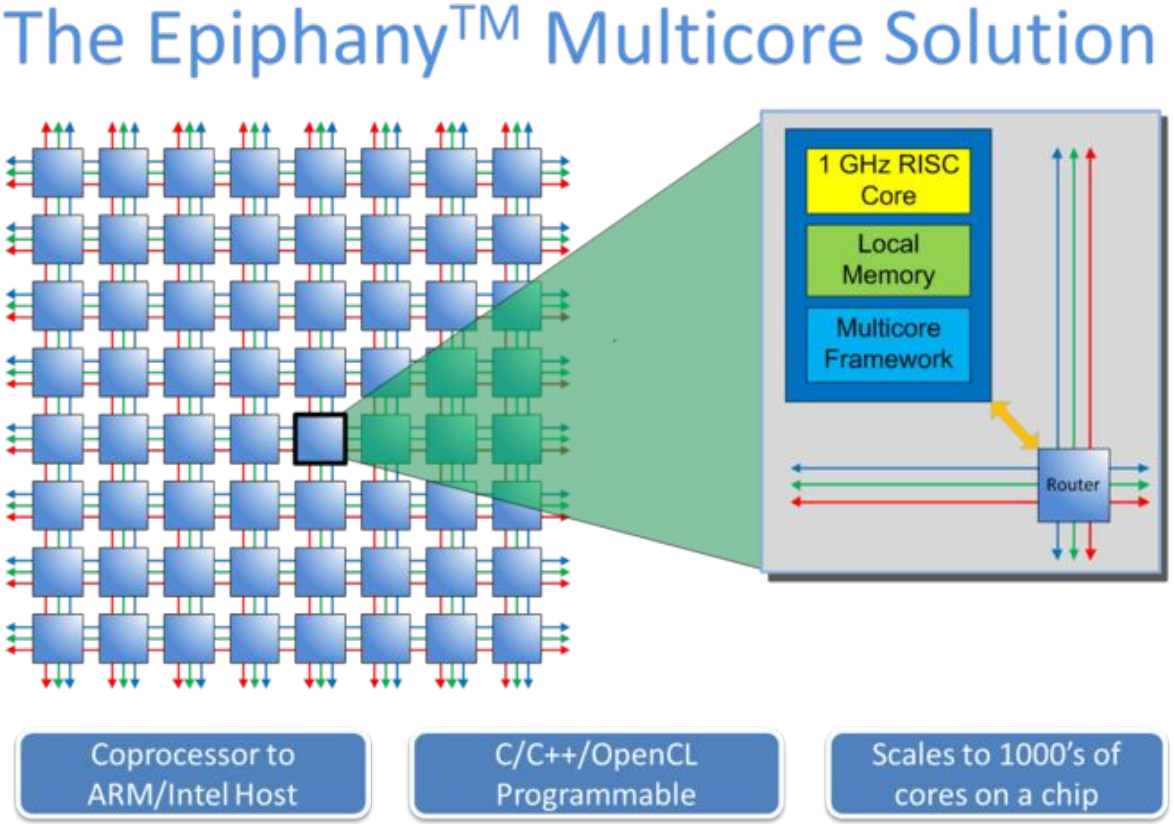



Memory coherency: *hard to scale!*
Nowadays, **n < 100**




On the same chip / over a network
**n > 100**

# Massively parallel processor array (MPPA)




https://spellfoundry.com/wp/wp-content/uploads/2013/05/Parallela.png

# Hardware with non-shared memory

An non-exhaustive list:

- 80’s: Inmos Transputer (programmed in occam, ?? cores)
- 90’s, Intel: Paragon (2048 cores “in a 2D grid”)
- 2000’s, IBM & Rapport: Kilocore (1024 cores)
- 2010’s, Tilera: TileGx (72 cores)
- 2010’s Adapteva: Epiphany (up to 4096 cores in theory, Parallela board 64 cores)
- 2006-2015, UC Davis: AsAP (36, then 167 cores)
- since 2008, Kalray: MPPA (256 cores)
- since 2012, Green Arrays: GA144 (144 cores) - asynchronous, no clock signal!
- 2018, Sunway: SW26010 (260 cores) (TaihuLight supercomputer #1 in 2018)

# The elephant in the room

Processes interacting by sending messages over a network…

# The elephant in the room

Processes interacting by sending messages over a network…

**The internet!**

Distributed systems. “The cloud.”
A software **designed** using isolated process & MP can **scale**.

# The elephant in the room

Processes interacting by sending messages over a network…

**The internet!**

Distributed systems. “The cloud.”
A software **designed** using isolated process & MP can **scale**.

High performance computing (HPC): Open MPI (*Message Passing Interface*)

# Conclusion

# Conclusion

- Message passing semantics
- Use concurrency for software *design*
- Use isolated processes for software *reliability*
- Not mainstream (yet?), but used in the industry
- The world is concurrent, you can map it to concurrent processes

- Not a silver bullet!
- Try it yourself :)

# Process isolation for reliability

## Introducing Site Isolation in Firefox

Anny Gakhokidze and Neha Kochar | May 18, 2021

When two major vulnerabilities known as Meltdown and Spectre were disclosed by security researchers in early 2018, Firefox promptly added security mitigations to keep you safe. Going forward, however, it was clear that with the evolving techniques of malicious actors on the web, we needed to
and to kee

We are excite
fundamental
operating sys
a separate operating system process makes it even harder for malicious sites to read another site's secret or private data.

Isolating each site into a separate operating system process makes it even harder for malicious sites to read another site's secret or private data.

https://blog.mozilla.org/security/2021/05/18/introducing-site-isolation-in-firefox/

# Further reading / watching

- Videos
  - **Joe Armstrong: How we program multicores** https://youtu.be/bo5WL5IQAd0
  - **Rob Pike: Concurrency/message passing Newsqueak** https://youtu.be/hB05UFqOtFA
  - (Both are good speakers, check out their other talks!)
- Russ Cox: ”Bell Labs and CSP Threads” https://swtch.com/~rsc/thread/
- Joe Armstrong’s PhD thesis (2003): http://erlang.org/download/armstrong_thesis_2003.pdf
- Fred Hébert (Heroku): https://learnyousomeerlang.com/ https://www.erlang-in-anger.com/
- Actor model as a library in Java: Akka https://akka.io
- Formal reasoning on concurrent programs using CSP, CCS, process calculus, etc…
- Multiway rendezvous: synchronous message passing between N processes
  - Garavel, Serwe: The Unheralded Value of the Multiway Rendezvous hal.archives-ouvertes.fr/hal-01511847
- More generally: “Coders at Work” by Peter Seibel, interviews of famous programmers

# Thanks!

Questions?

# Unix, OSes in general

- *Unix file descriptors* are, arguably, channels

# DJB

Qmail: 20+ years running, no major bug.

Structure: OS processes talking through pipes. Well isolate. Messages. Concurrent.

# TODO

- Erlang: Armstrong PhD, …
  - WhatsApp, AliBaba, Easemob (1B)
  - hot code reload
- DJB aaron schwartz http://www.aaronsw.com/weblog/djb
- COP: unix pipelines, etc
- Shared memory doesn't scale easily, at SoC level (caches etc), or at distributed levels. COP does.
- OpenMP, MPI, etc
- gen_server, etc.
- Smalltalk
- protocol buffers
- Scalability of designed based on MP
- idempotent
- not a silver bullet