Fundamentals: Parallelism & Concurrency

CS 4740: Cloud Computing

Fall 2024

Lecture 3

Yue Cheng

University of Virginia

Parallelism & Concurrency

- Abstraction: Process vs. thread
- Concurrency in Go

What is a process?

What is a process?

- Programs are code (static entity)
- Processes are running programs

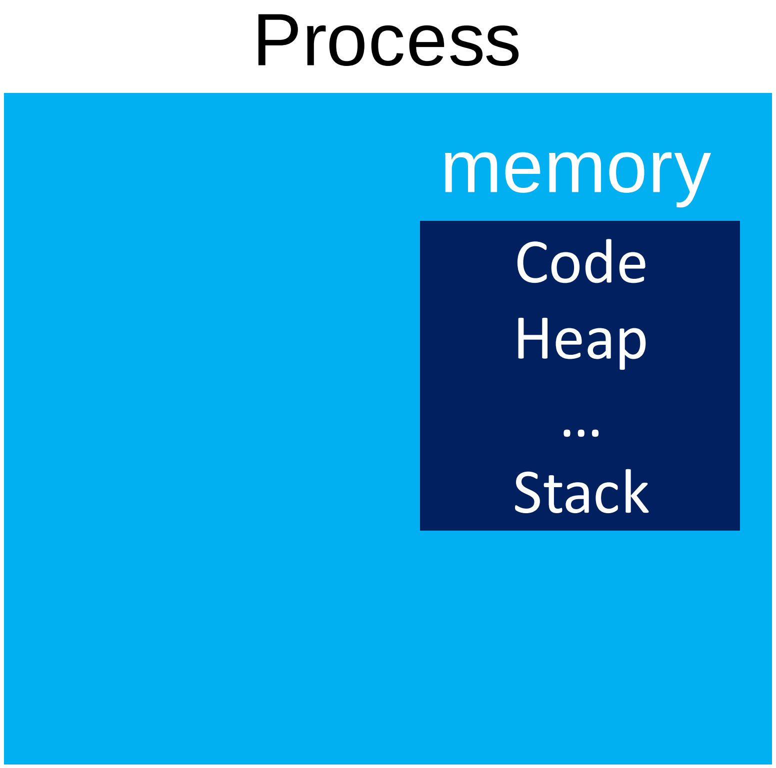
- Java analogy
  - class -> “program”
  - object -> “process”

What is in a process?

Process

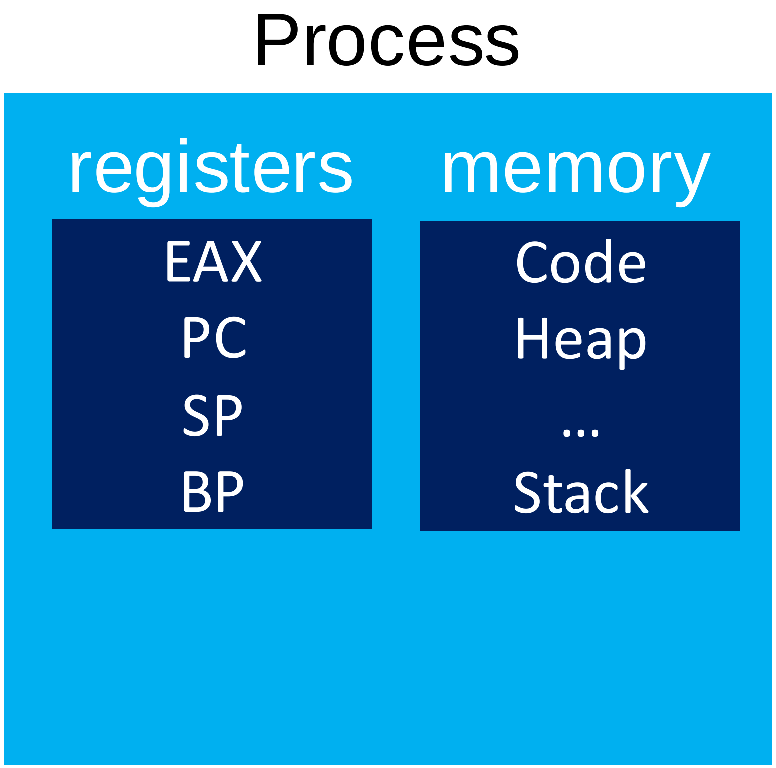
What things change as a program runs?

What is in a process?



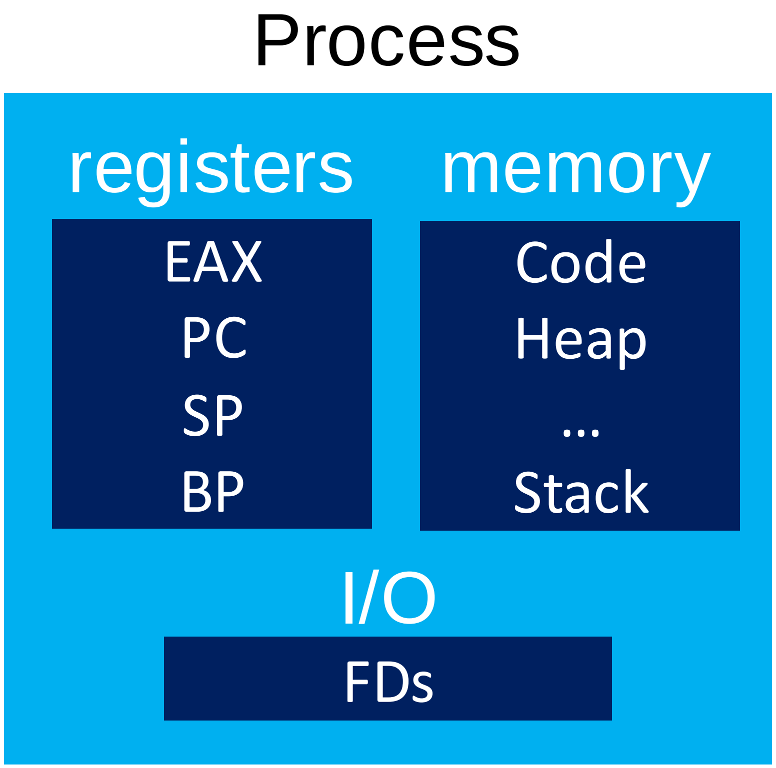
What things change as a program runs?

What is in a process?



What things change as a program runs?

What is in a process?



What things change as a program runs?

Threads

Why thread abstraction?

Process abstraction: Challenge 1

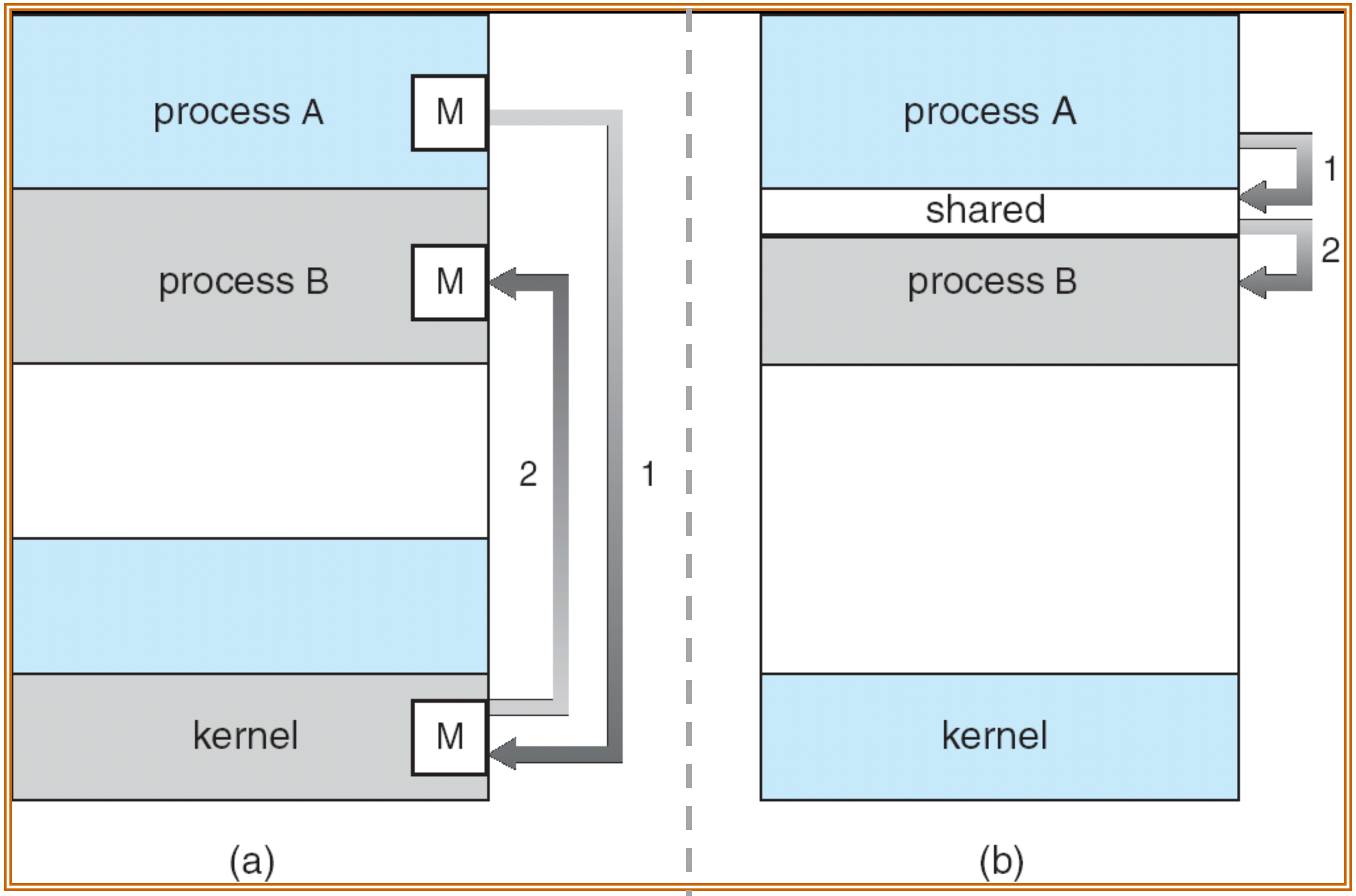
- Inter-process communication (IPC)

Inter-process communication

- Mechanism for processes to communicate and to synchronize their actions

- Two models
  - Communication through a shared memory region
  - Communication through message passing

Communication models



Message Passing

Shared Memory

Communication through message passing

- Message passing can be either **blocking** (synchronous) or **non-blocking** (asynchronous)
  - **Blocking Send:** The sending process is blocked until the message is received by the receiving process or by the mailbox
  - **Non-blocking Send:** The sending process resumes the operation as soon as the message is received by the kernel
  - **Blocking Receive:** The receiver blocks until the message is available
  - **Non-blocking Receive:** "Receive" operation does not block; it either returns a valid message or a default value (null) to indicate a non-existing message

Process abstraction: Challenge 1

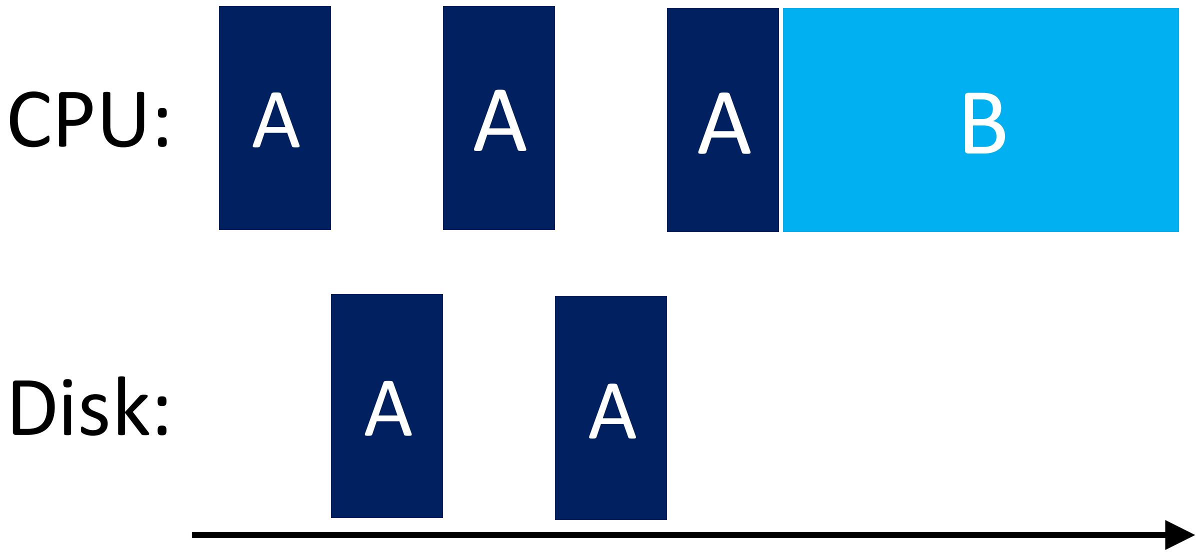
- Inter-process communication (IPC)
  - Cumbersome programming!
  - Copying overheads (inefficient communication)
  - Expensive context switching (why expensive?)
    - As per empirical measurement, the average process-level CFS (Completely Fair Scheduler: Linux's current default scheduler) context-switch overhead as ~7481.4ns

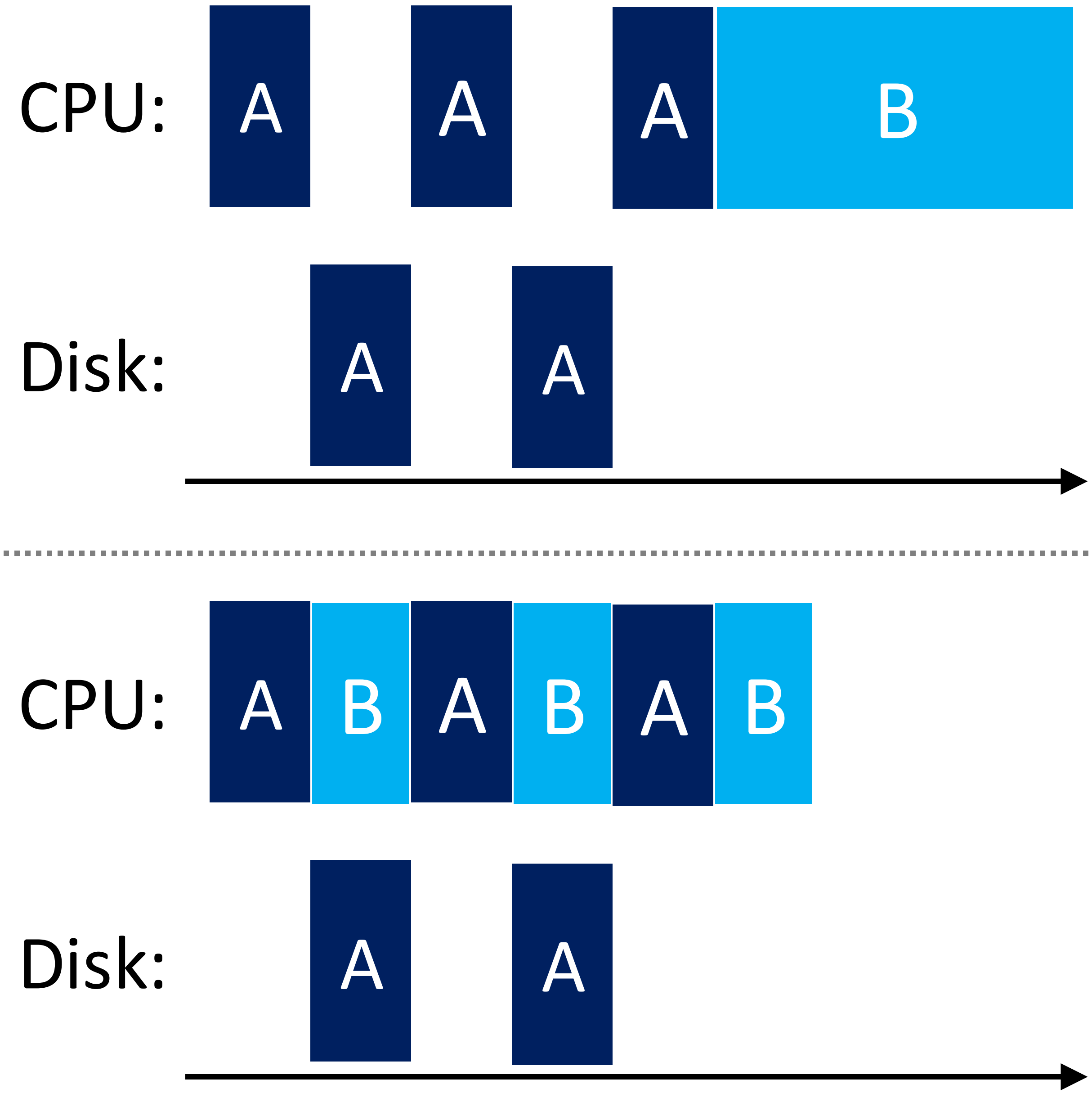
Process abstraction: Challenge 2

- Inter-process communication (IPC)
  - Cumbersome programming!
  - Copying overheads (inefficient communication)
  - Expensive context switching (why expensive?)

- CPU utilization

(a) Not interleaved

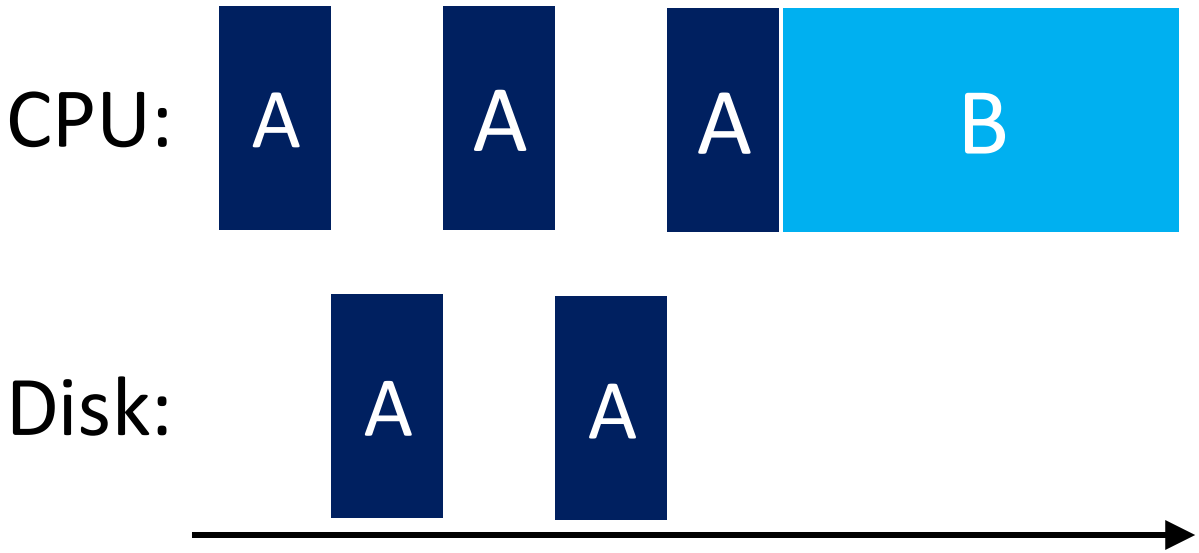




(a) Not interleaved

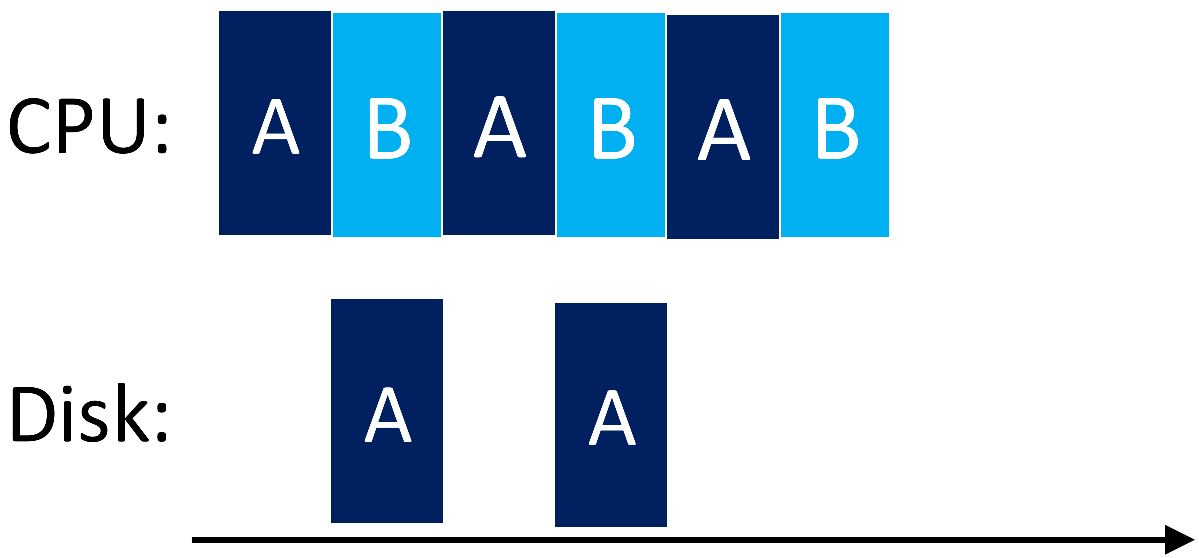
(b) Interleaved

(a) Not interleaved

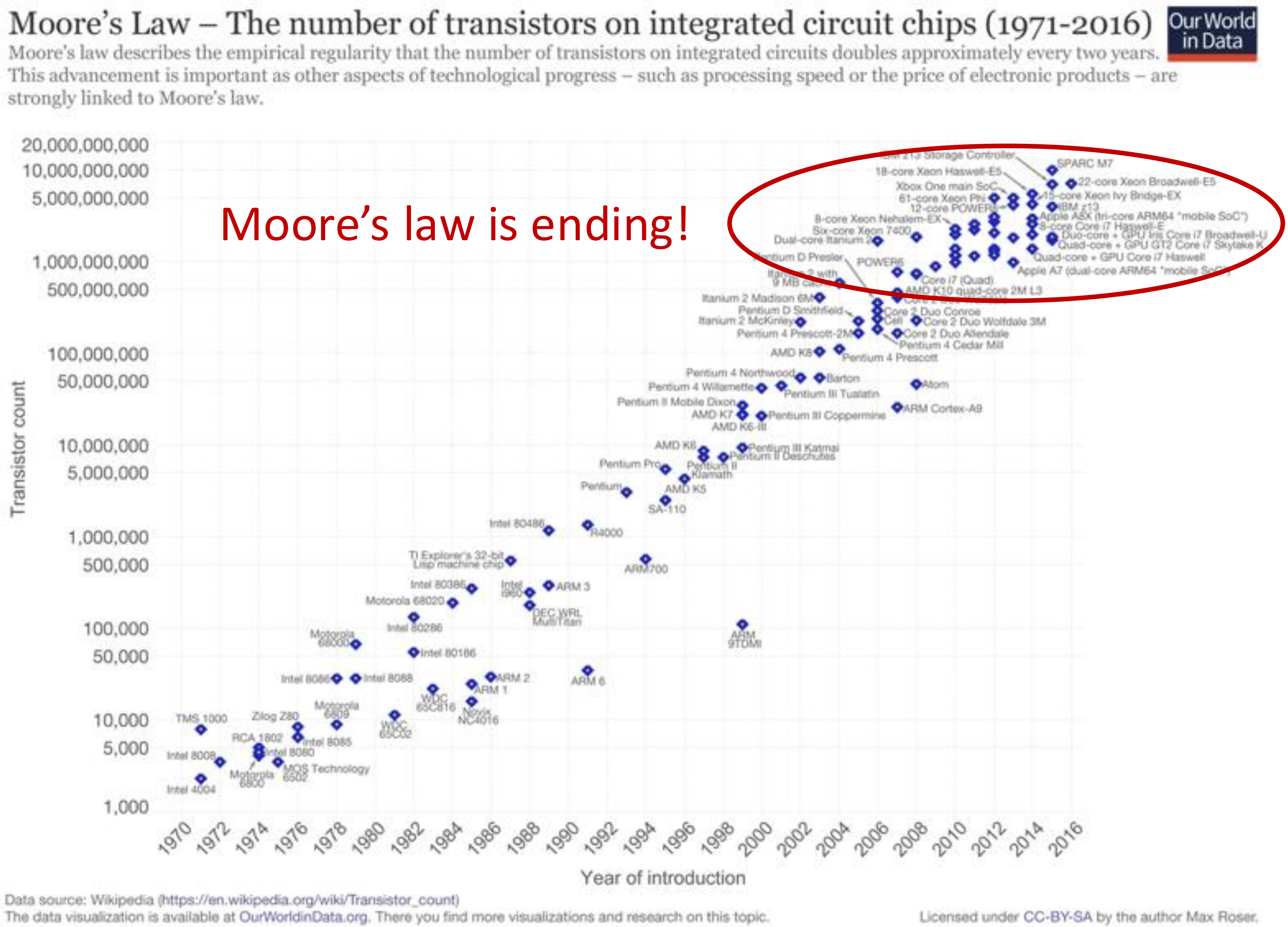


What if there is only one process?

(b) Interleaved



Moore’s law: # transistors doubles every ~2 years



CPU trends – What Moore's Law implies...

- The future
  - Same CPU speed
  - More cores (to scale-up or scale-out)

- Faster programs => concurrent/parallel execution

- Goal: Write applications that fully utilize many CPU cores…

Goal

- Write applications that fully utilize many CPUs…

Strategy 1

- Build applications from many communicating processes
  - Like Chrome (process per tab)
  - Communicate via `pipe()` or similar

- Pros/cons?

Strategy 1

- Build applications from many communicating processes
  - Like Chrome (process per tab)
  - Communicate via `pipe()` or similar

- Pros/cons? – That we've talked about in previous slides
  - Pros:
    - Don't need new abstractions!
    - Better (fault) isolation?
  - Cons:
    - Cumbersome programming using IPC
    - Copying overheads
    - Expensive context switching

Strategy 2

- New abstraction: the thread

Introducing thread abstraction

- New abstraction: the thread
- Threads are just like processes, but threads share the address space

Thread

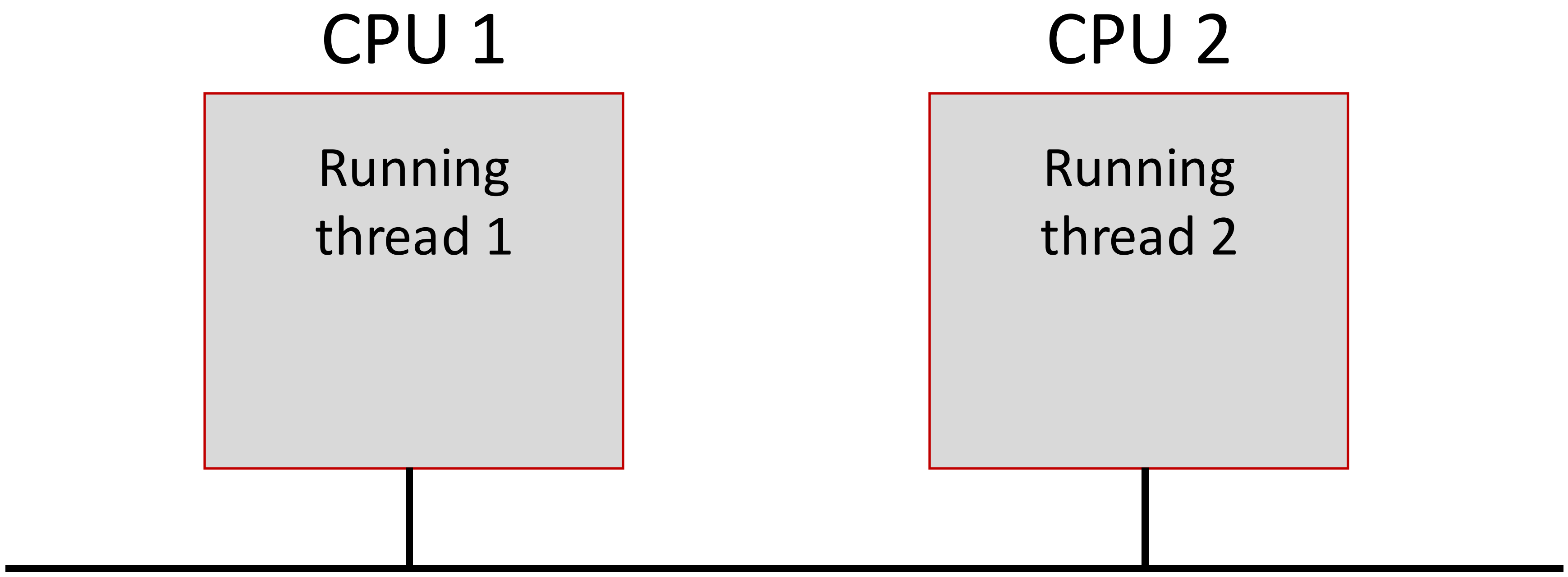
- A process, as defined so far, has only one thread of execution

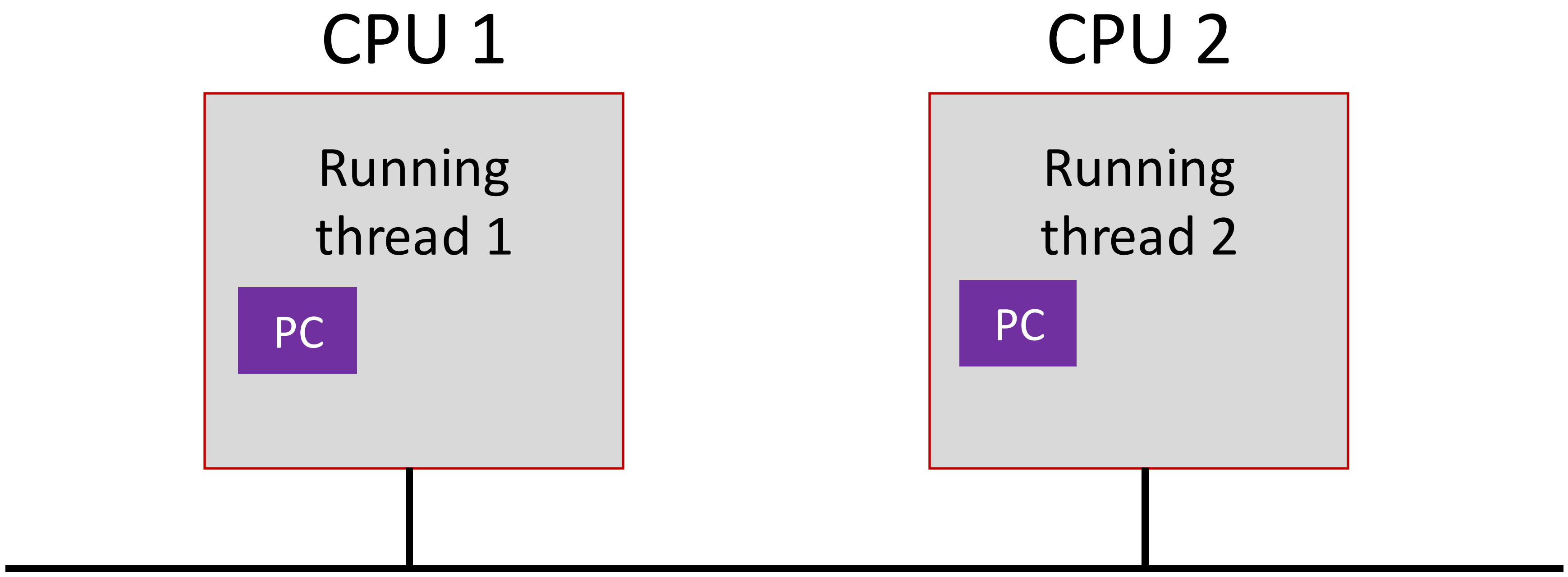
- Idea: Allow multiple threads of concurrently running execution within the same process environment, to a large degree independent of each other
  - Each thread may be executing different code at the same time

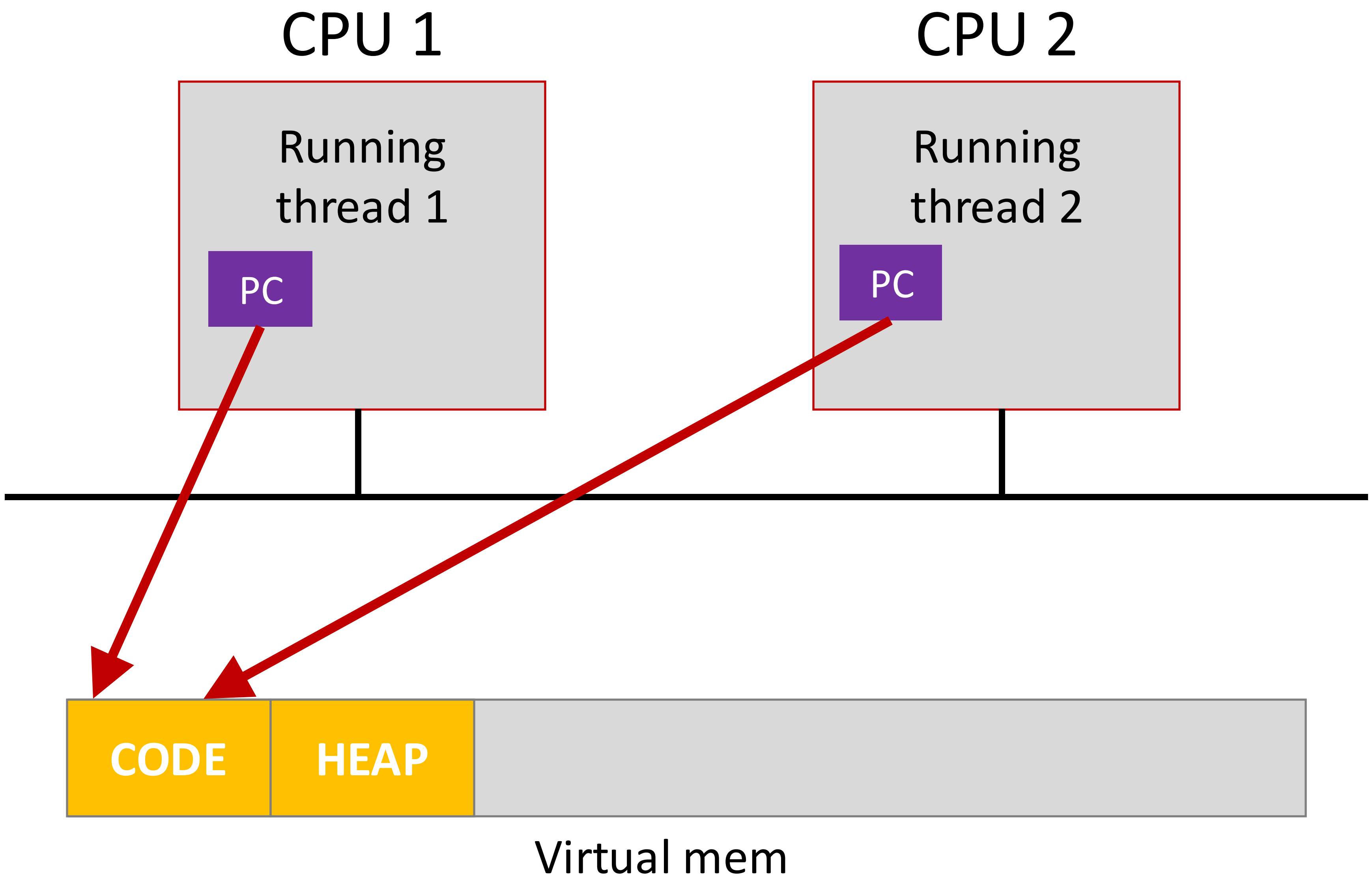
Process vs. thread

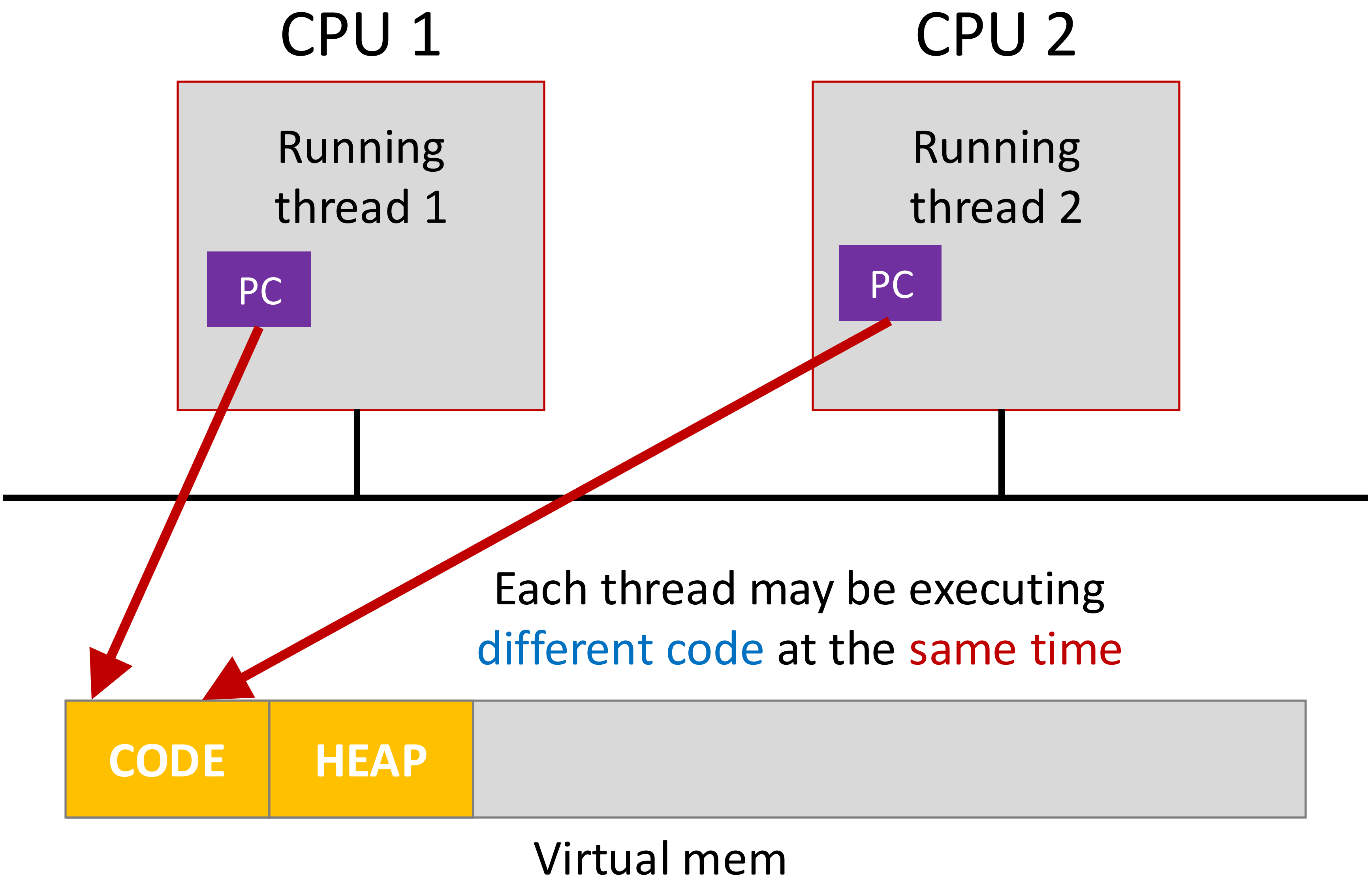
- Multiple threads within a process will share
  - The address space
  - Open files (file descriptors)
  - Other resources

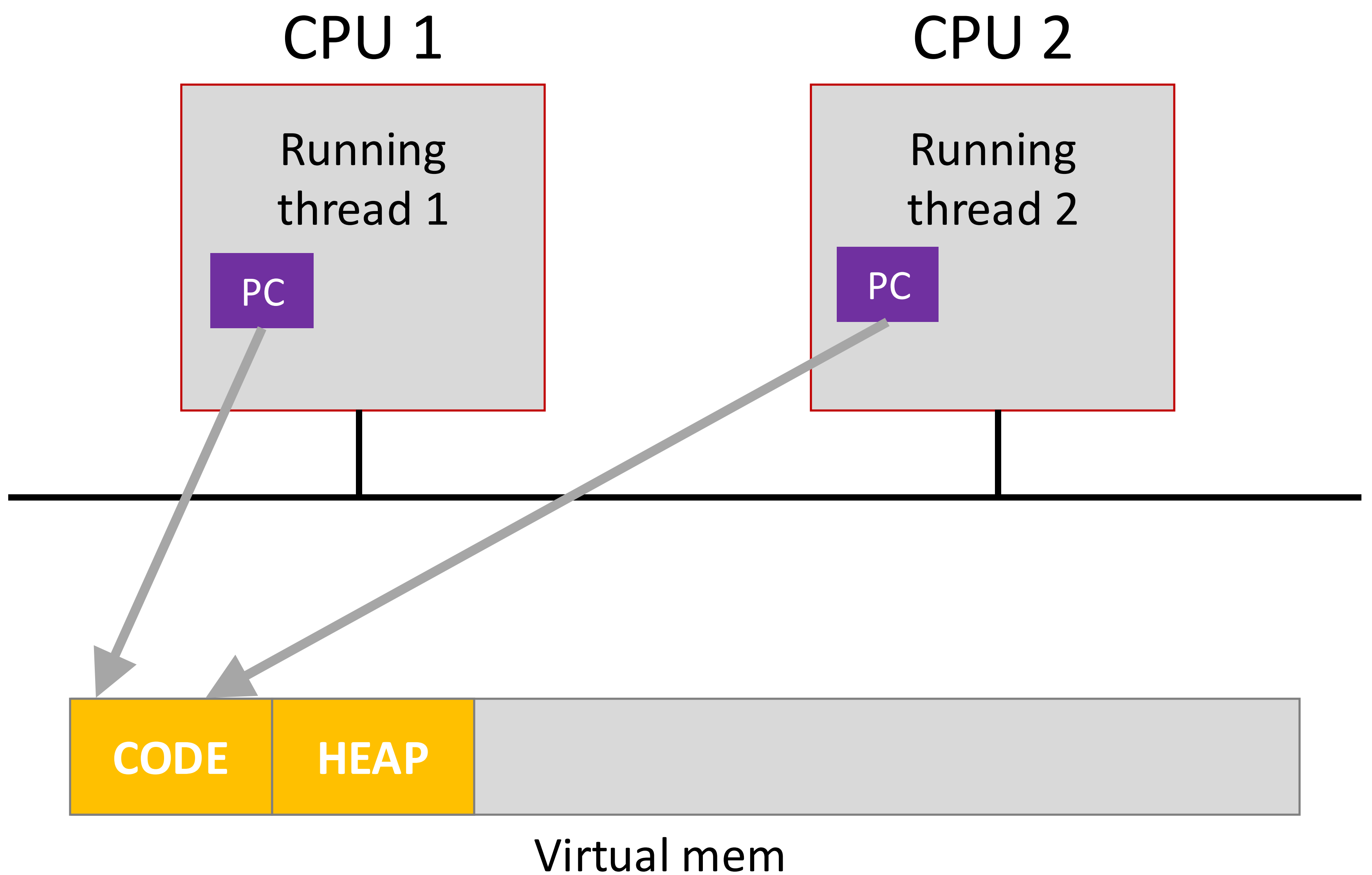
- Thread
  - Efficient and fast resource sharing
  - Efficient utilization of many CPU cores with only one process
  - Less context switching overheads

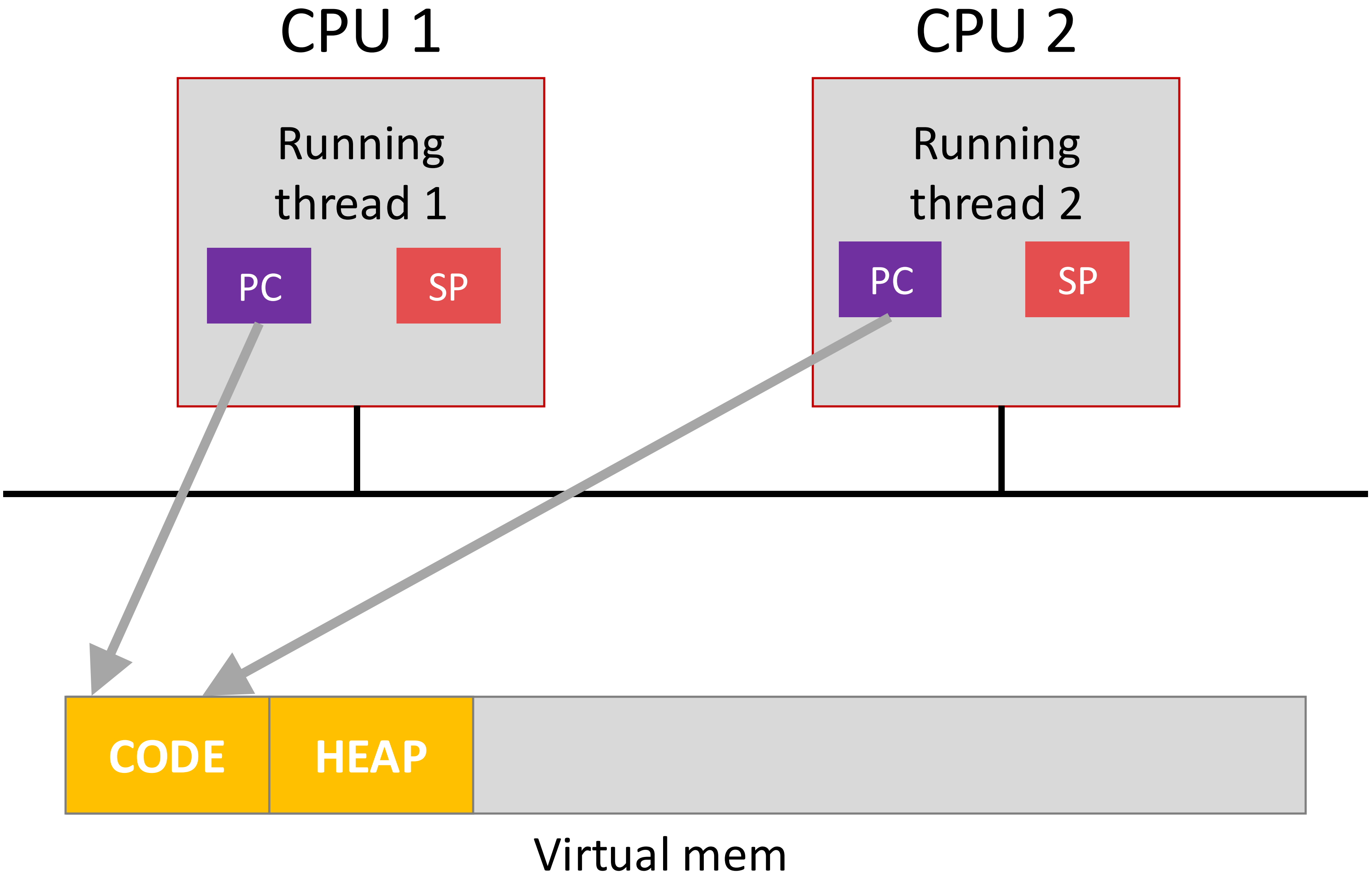

CPU 1
CPU 2
Running thread 1
Running thread 2

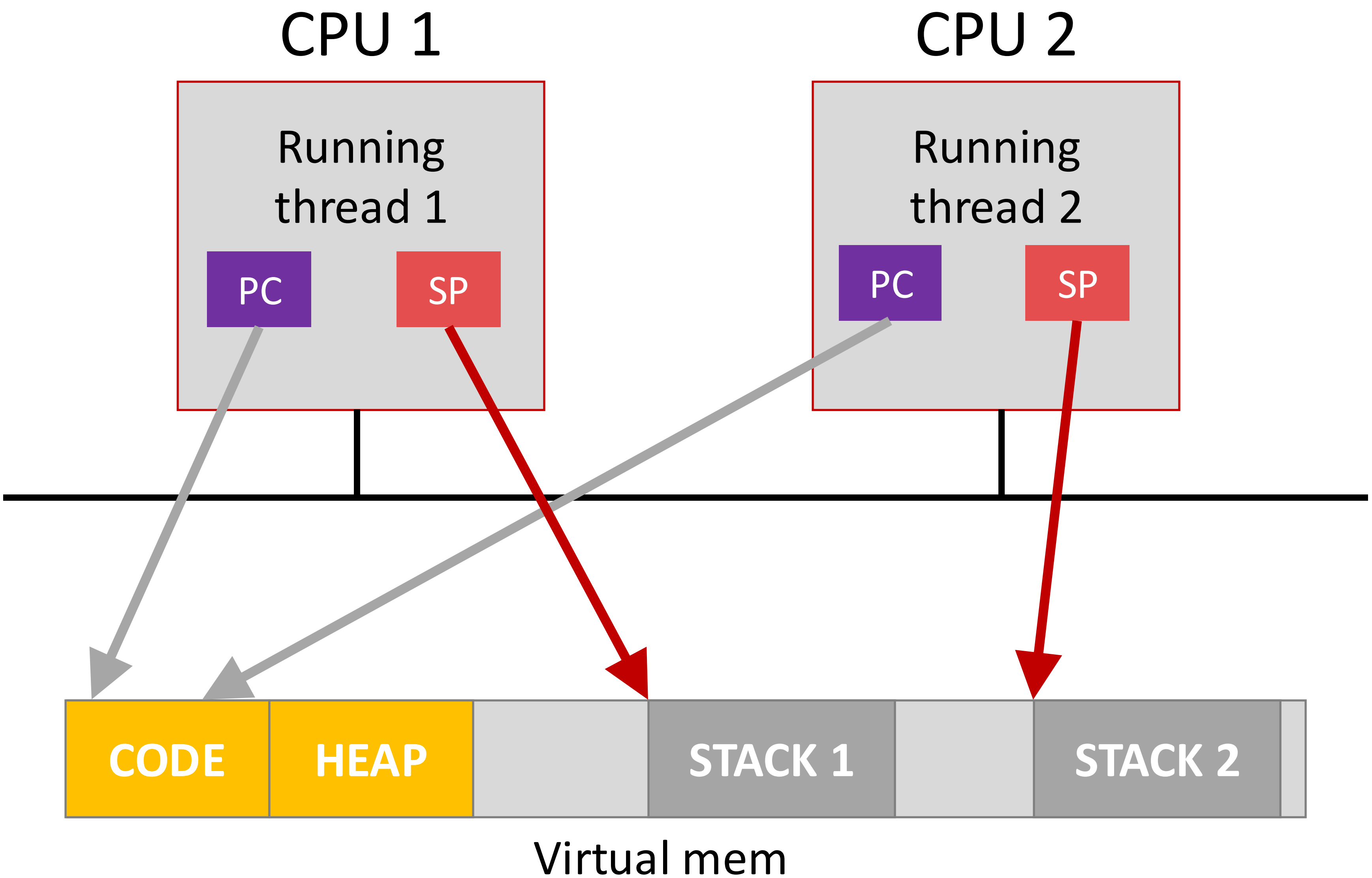

CPU 1
CPU 2
Running thread 1
PC
Running thread 2
PC


CPU 1
CPU 2
Running thread 1
Running thread 2
PC
PC
CODE
HEAP
Virtual mem

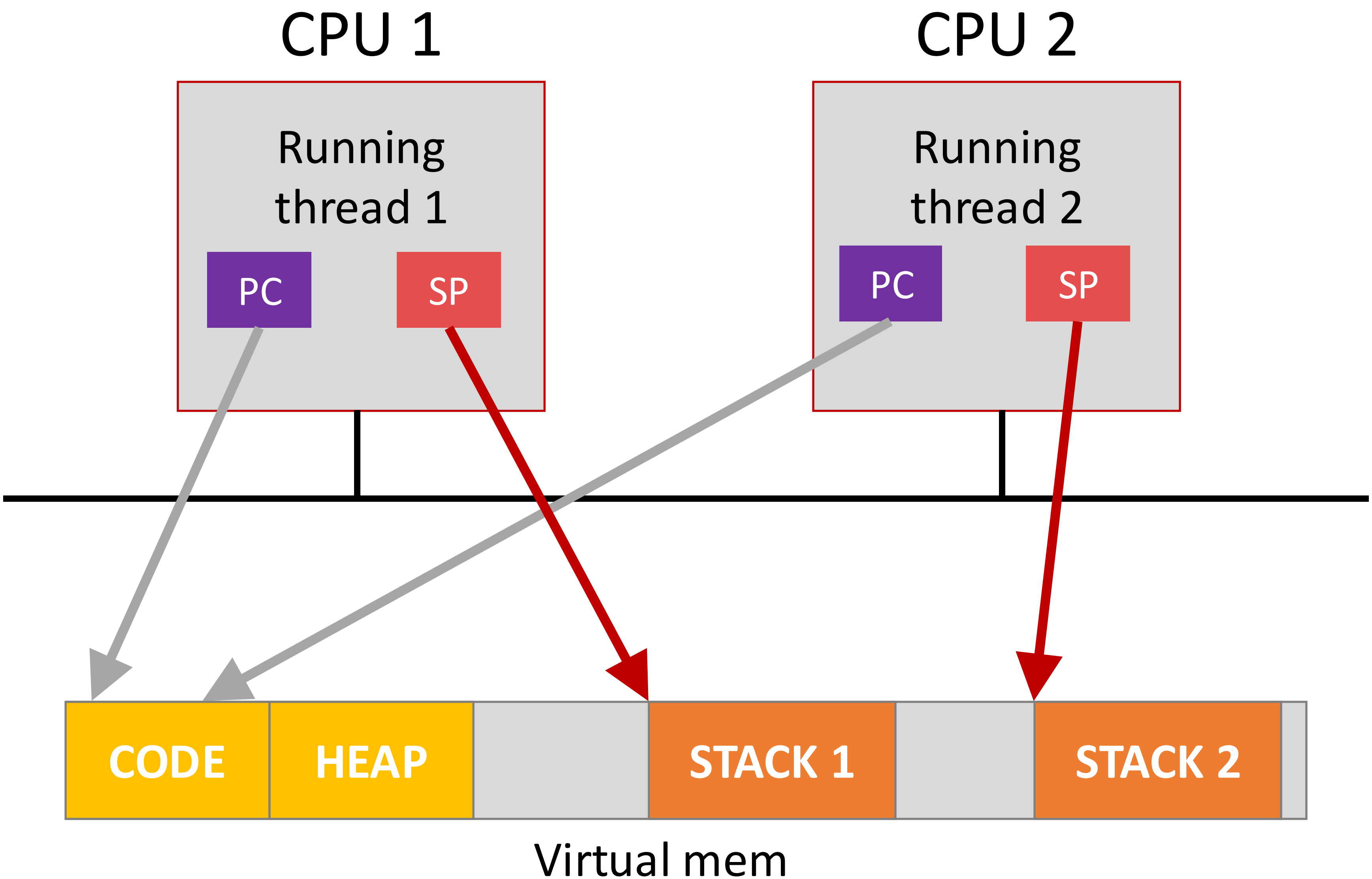

CPU 1
CPU 2
Running thread 1
Running thread 2
PC
PC
Each thread may be executing different code at the same time
CODE
HEAP
Virtual mem


CPU 1
CPU 2
Running thread 1
Running thread 2
PC
PC
CODE
HEAP
Virtual mem


CPU 1
CPU 2
Running thread 1
Running thread 2
PC
SP
PC
SP
CODE
HEAP
Virtual mem


CPU 1
CPU 2
Running thread 1
Running thread 2
PC
SP
PC
SP
CODE
HEAP
STACK 1
STACK 2
Virtual mem

Thread executing different functions need different stacks



Parallelism & Concurrency

- Abstraction: Process vs. thread

- Concurrency in Go

Go keywords

```
break      case    chan     const        continue
default    defer   else     fallthrough  for
func       go      goto     if           import
interface  map     package  range        return
select     struct  switch   type         var
```

This lecture covers **go, chan, select**

```
break      case    chan     const        continue
default    defer   else     fallthrough  for
func       go      goto     if           import
interface  map     package  range        return
select     struct  switch   type         var
```

Concurrency

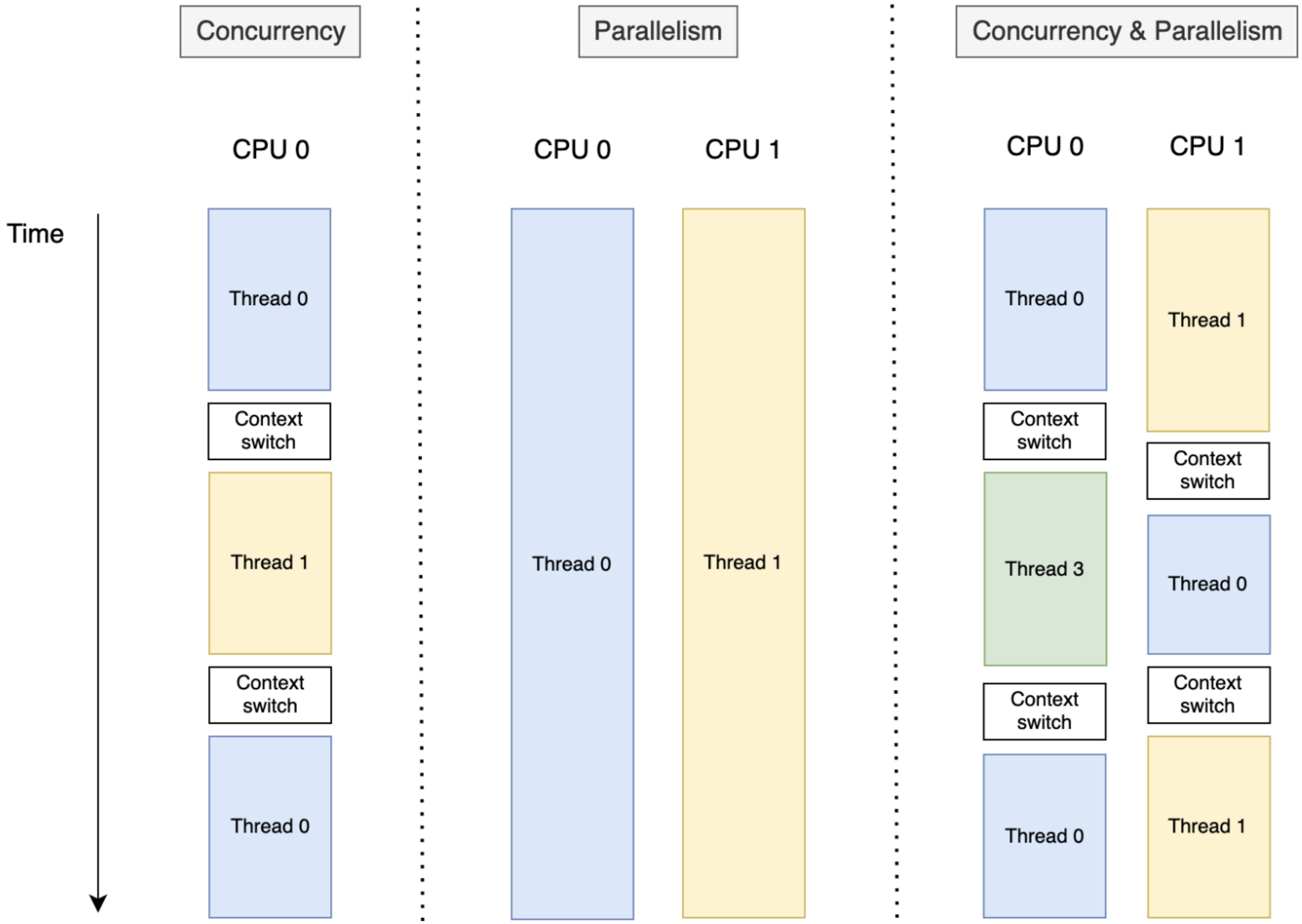
Concurrency vs. parallelism

"Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once."

-- Rob Pike: Concurrency is not Parallelism

- Concurrency is possible with even a single CPU core
  - Parallelism is not
- Backbone of concurrency in Go:
  - Goroutines
  - Channels
  - `select` construct

Concurrency vs. parallelism



Goroutines

Goroutines

- Core concepts in Go
- Basically, lightweight threads
  - Managed by Go's runtime
    - Limited context switching and interaction with the OS
    - Goroutine scheduler is able to better optimize the workload
  - Generally cheap to spawn
    - Initial stack size is smaller compared to POSIX threads (8KB vs. 8MB)
    - But do not get the false sense you can spawn infinite number of them, it is still a resource
      - Up to tens/hundreds of thousands are fine
    - Internally multiplexed across on kernel thread pool (M:N)

Goroutines

```
package main

import (
    "fmt"
    "time"
)

func print(s string) {
    for range 5 {
        fmt.Printf("Hello from %s!\n", s)
        time.Sleep(500 * time.Millisecond)
    }
}

func main() {
    go print("first")
    go print("second")
    print("main")
}
```

Runtime

Runtime

- Just a library
  - Same as the **`libc`** library for C
- Statically linked with your program upon compilation

```
func main() {
    fmt.Printf("Logical CPUs (\"P\"s): %d\n", runtime.NumCPU())
    runtime.GC() // Invokes garbage collector
    fmt.Printf("GOMAXPROCS: %d\n", runtime.GOMAXPROCS(8))
}
```

[Go’s runtime package]

Scheduler

- Runs goroutines
- Pauses and resumes them
- Preemptive since Go 1.14
  - Goroutines are preempted after 10ms
  - Sysmon
- Work-stealing
- Coordinates system calls, I/O operations, runtime tasks, etc.

Ardan Labs: Scheduling in Go

Goroutine scheduling states

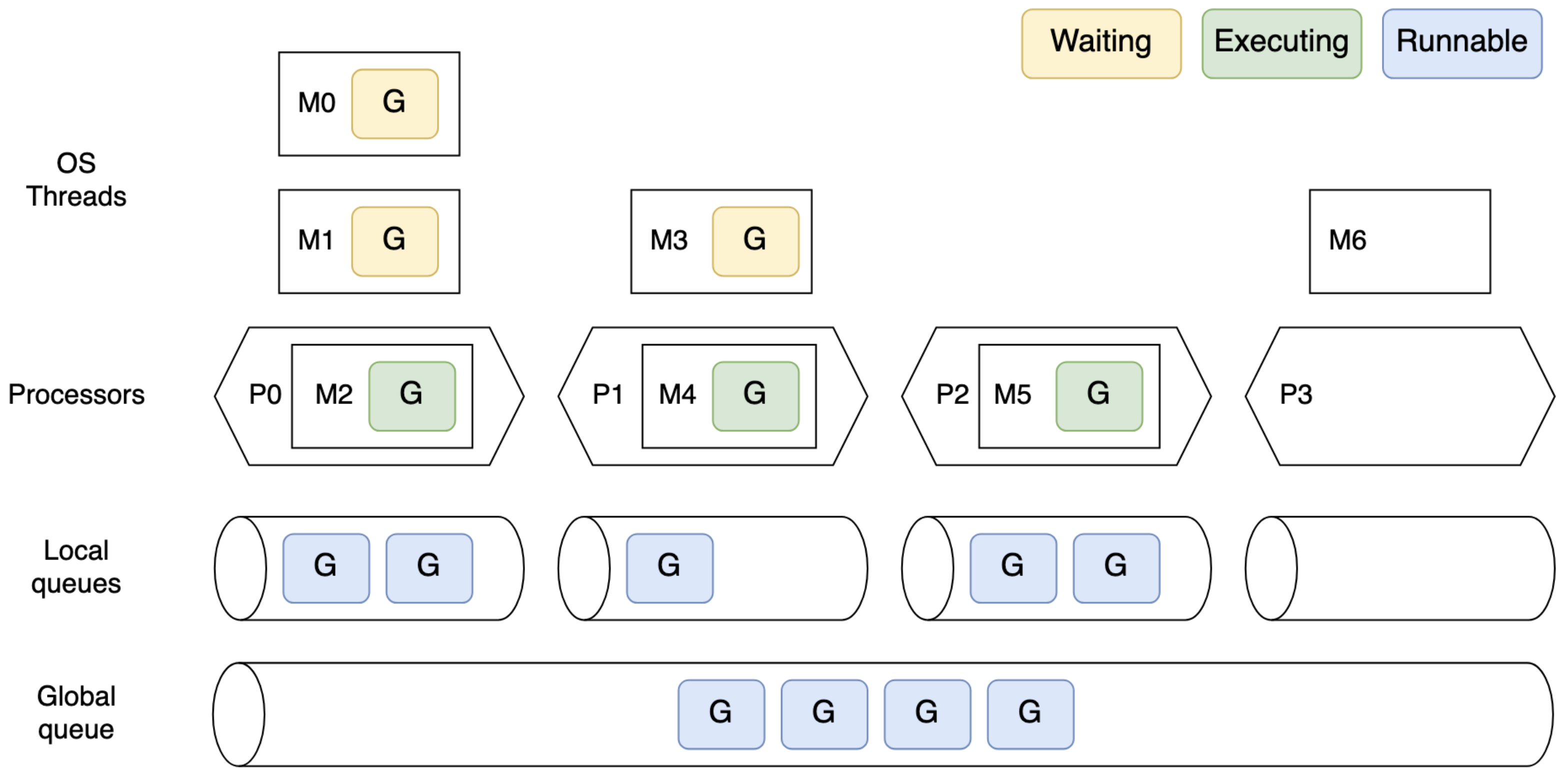
- Runnable
  - Can be run but is not assigned to a CPU core
- Executing
  - Currently running
- Waiting
  - System calls
  - Synchronous calls
  - I/O operations

Goroutine scheduling illustration

P (Processor): Logical processor
M (Machine): OS thread

- Initially, each P gets assigned one M
- More can be spawned by the runtime

G (Goroutine): Goroutine



Goroutine scheduling algorithm

```
runtime.schedule() {
    // only 1/61 of the time, check the global runnable queue for a G.
    // if not found, check the local queue.
    // if not found,
    //     try to steal from other Ps.
    //     if not, check the global runnable queue.
    //     if not found, poll network.
}
```

Jaana Dogan: Scheduler (CC BY SA 4.0)

Channels

Channels

- Way to transfer data between Goroutines
- Data type is a part of the channel type
- Buffered and unbuffered
- Channels can be created only with **`make`**

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

- New operator **`<-`** used to send and receive messages from channels

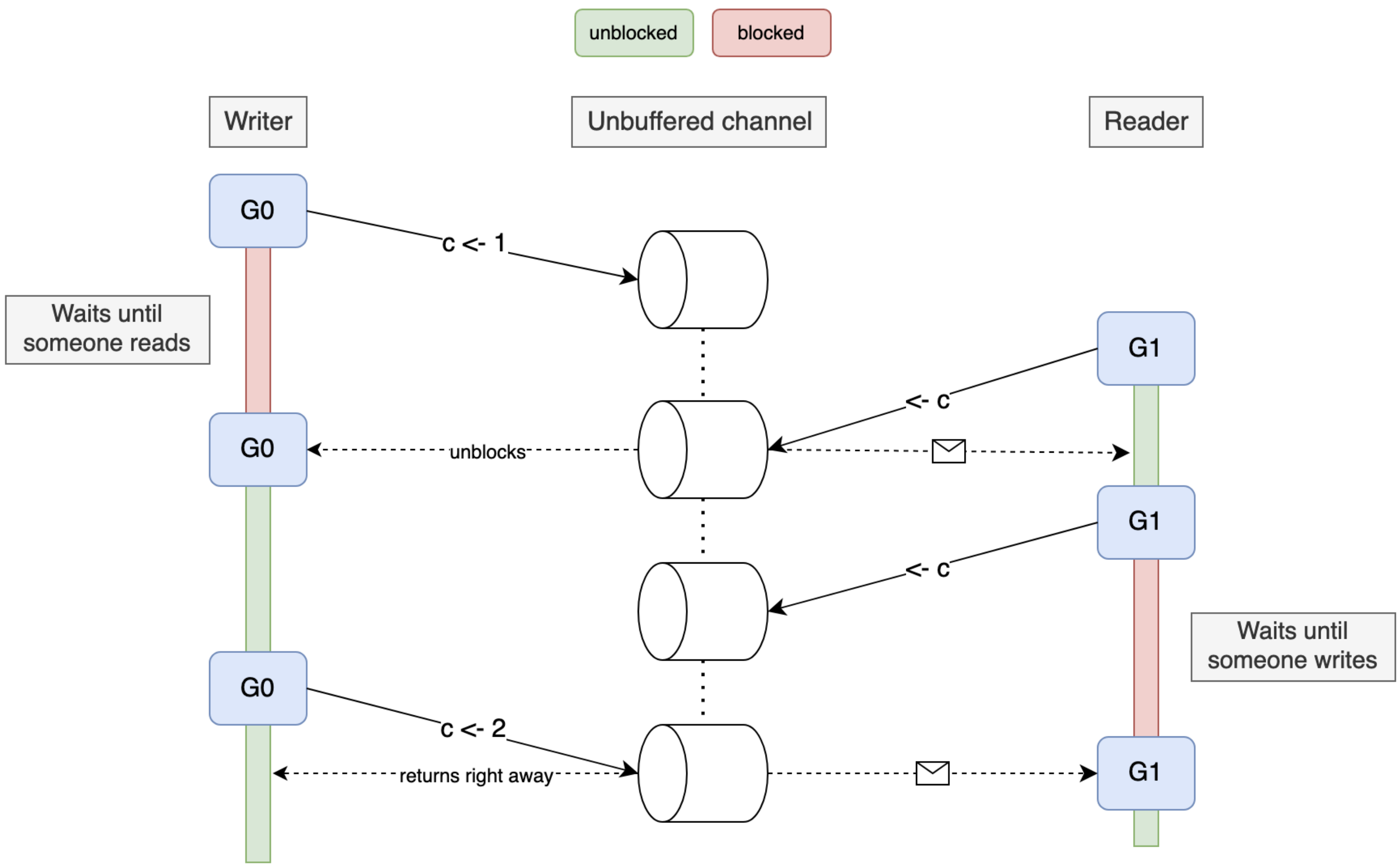
```
value := <-ch        // read
ch<-value            // write
```

Unbuffered channels

Note that this example is racy

```
package main

import "fmt"

func readAndPrint(c <-chan int) { // c is read-only channel
    value := <-c
    fmt.Println("Received", value)
}

func main() {
    c := make(chan int)
    fmt.Println("Channel length:", len(c))
    fmt.Println("Channel capacity:", cap(c))

    go readAndPrint(c)
    c <- 5
}
```

Unbuffered channels



Channel deadlocks

Unbuffered channels do block

- Buffered channels also block when full or empty
- Go kindly detects deadlocks

```
package main

import "fmt"

func readAndPrint(c <-chan int) {
    value := <-c
    fmt.Println("Received", value)
}

func main() {
    c := make(chan int)
    c <- 5 // blocks
    go readAndPrint(c)
}
```

Goroutine synchronization

Unbuffered channels can be used to synchronize goroutines

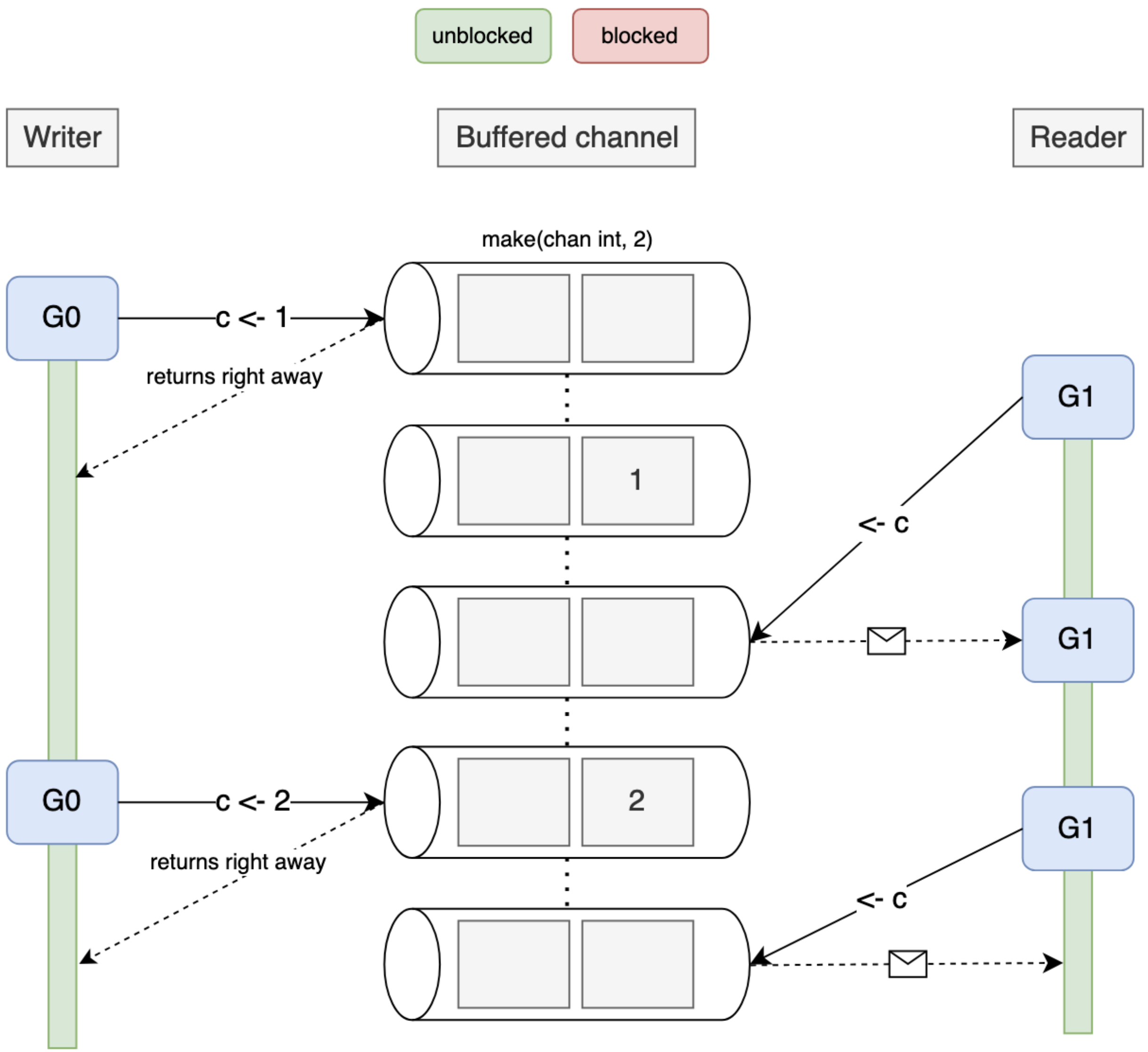
```
func process(done chan<- struct{}) { // done is write-only channel
    fmt.Println("Processing...")
    time.Sleep(2 * time.Second)
    fmt.Println("Finished!")
    done <- struct{}{}
}

func main() {
    done := make(chan struct{})
    go process(done)

    fmt.Println("Waiting for processing...")
    <-done // Blocks until `process` finishes
    fmt.Println("Continuing in main")
}
```

Buffered channels

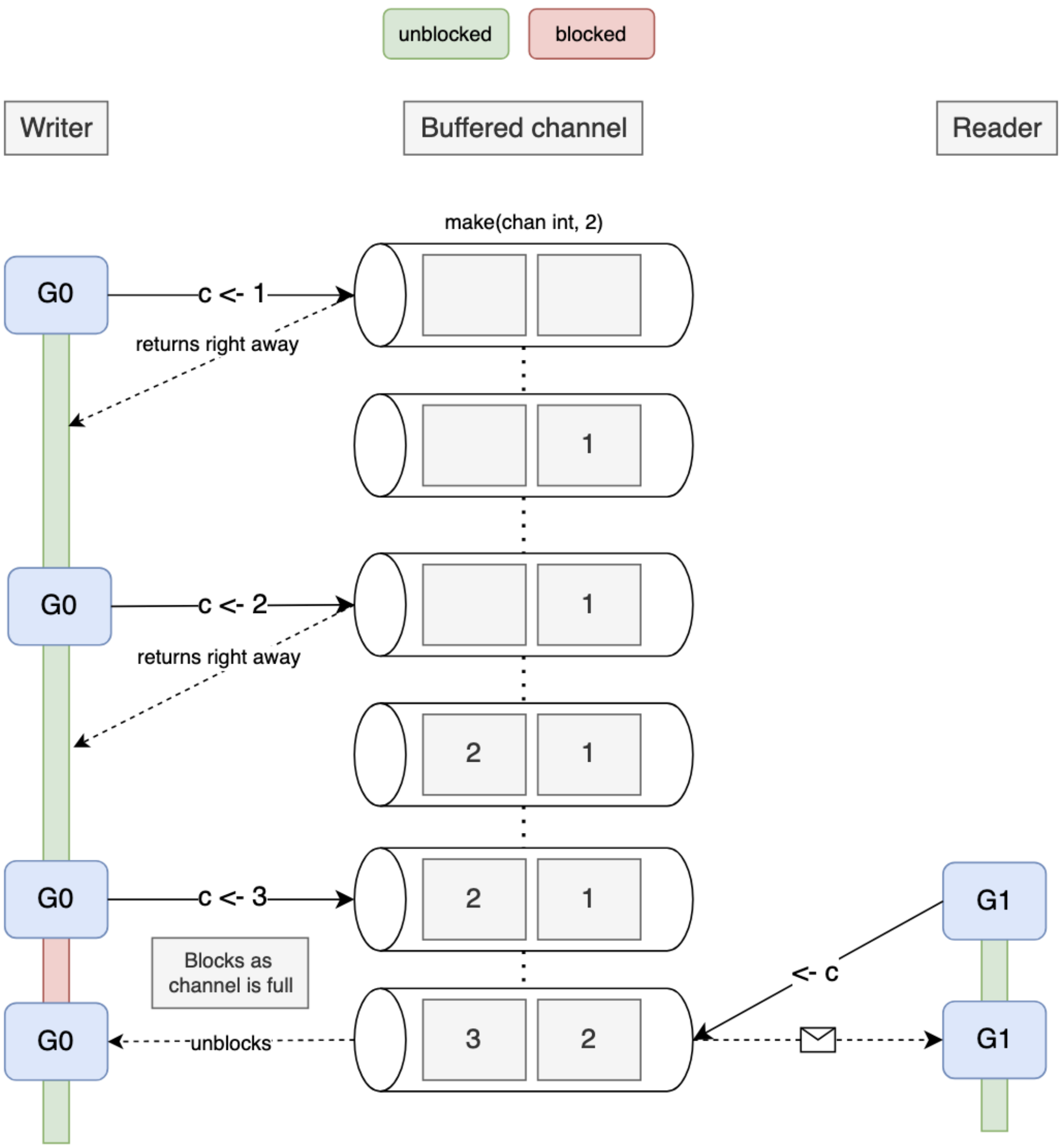
The size of the channel is provided as the second argument to `make`

```
func readAndPrint(c <-chan int) {
    value := <-c
    fmt.Println("Received", value)
}

func main() {
    c := make(chan int, 1)
    fmt.Println("Channel length:", len(c))
    fmt.Println("Channel capacity:", cap(c))

    c <- 5 // note that now it does not block
    fmt.Println("Channel length:", len(c))
    fmt.Println("Channel capacity:", cap(c))

    go readAndPrint(c)
    time.Sleep(time.Second) // need to wait
}
```

Buffered channels



Buffered channels



Select

Select

- Syntanctically similar to the `switch` statement

- Helps us manipulate multiple channels at the same time
  - You can read on/write to numerous channels at once
  - Prevents reads/writes that would otherwise block

Select

- The `select` statement always chooses a case that does not block
- Both of the channels in the following example are ready to be read from
  - Therefore, the `select` chooses one of them at random

```
func main() {
    chanA := make(chan int, 1)
    chanB := make(chan int, 1)
    chanA <- 0
    chanB <- 0

    select {
    case <-chanA:
        fmt.Println("Read from A")
    case <-chanB:
        fmt.Println("Read from B")
    }
    fmt.Println("All done")
}
```

Select

- The same works for writes
  - Neither channels is full
  - Writing to them would not block

```
func main() {
    chanA := make(chan int, 1)
    chanB := make(chan int, 1)

    select {
    case chanA <- 0:
        fmt.Println("Wrote to A")
    case chanB <- 1:
        fmt.Println("Wrote to B")
    }
    fmt.Println("All done")
}
```

Select

- The `chanB` would block on read
  - The `select` therefore always chooses the `chanA`

```
func main() {
    chanA := make(chan int, 1)
    chanB := make(chan int, 1)
    chanA <- 0

    select {
    case <-chanA:
        fmt.Println("Read from A")
    case <-chanB:
        fmt.Println("Read from B")
    }
    fmt.Println("All done") }
```

Select default

- All channel reads block
  - `select` would panic
- We can leverage the `default` case
  - The `default` gets selected only when all the cases would block

```
func main() {
    chanA := make(chan int, 1)
    chanB := make(chan int, 1)

    select {
    case <-chanA:
        fmt.Println("Read from A")
    case <-chanB:
        fmt.Println("Read from B")
    default:
        fmt.Println("Fallback")
    }
    fmt.Println("All done")
}
```

Next Monday: MapReduce