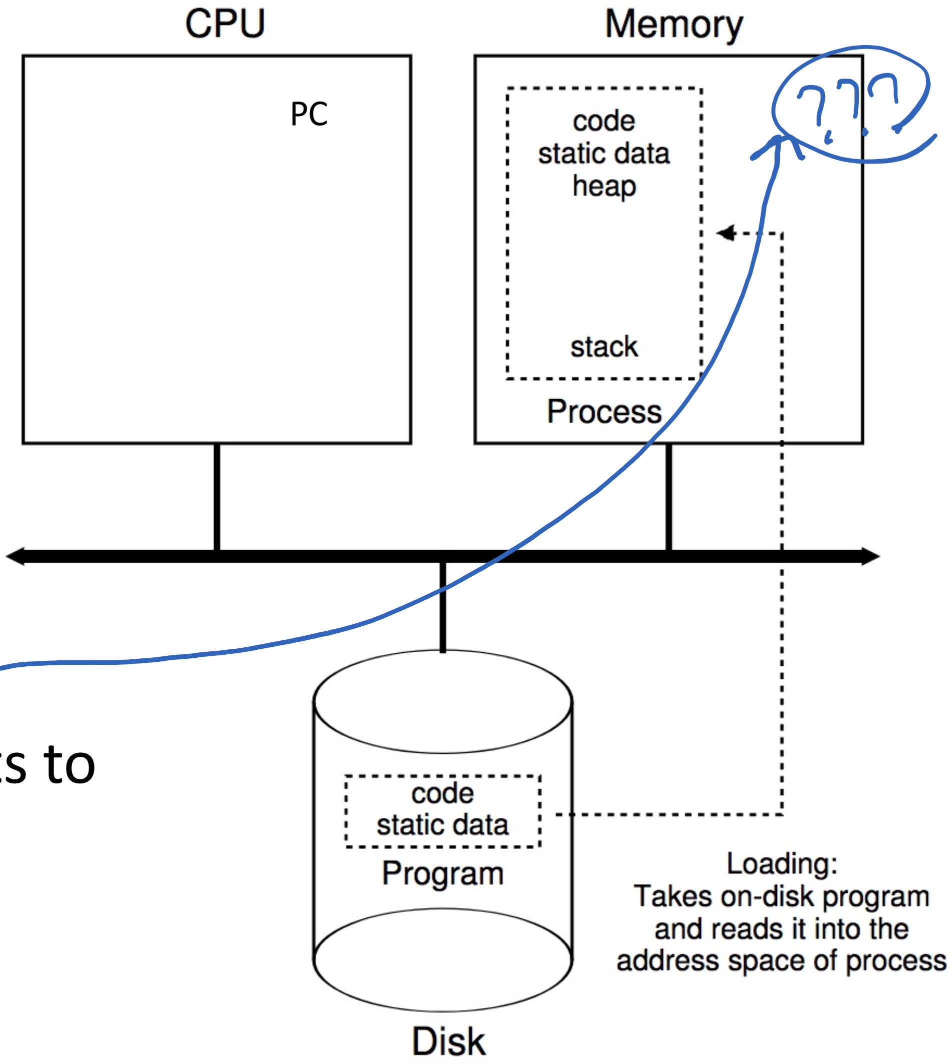
LDE and Threads

CS 571: Operating Systems (Spring 2020)
Lecture 2

Yue Cheng

Some material taken/derived from:
- Wisconsin CS-537 materials created by Remzi Arpaci-Dusseau.

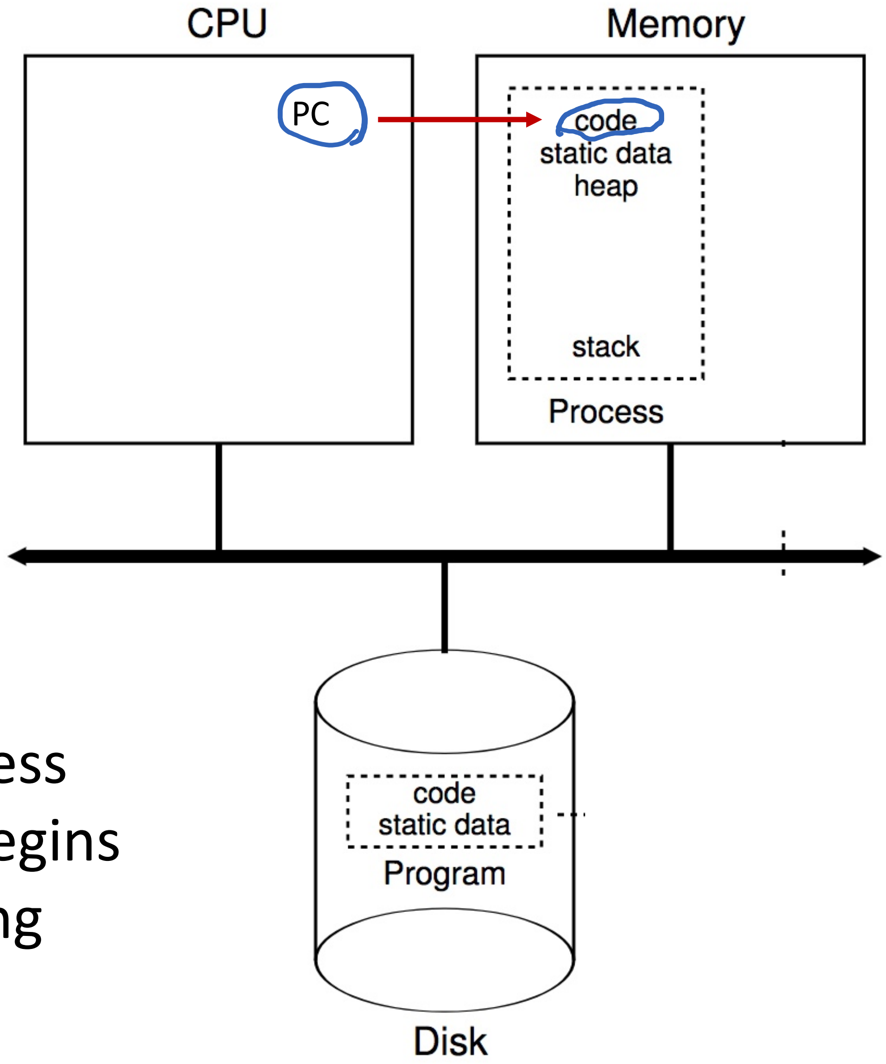
Process Creation



Loading:
Takes on-disk program
and reads it into the
address space of process

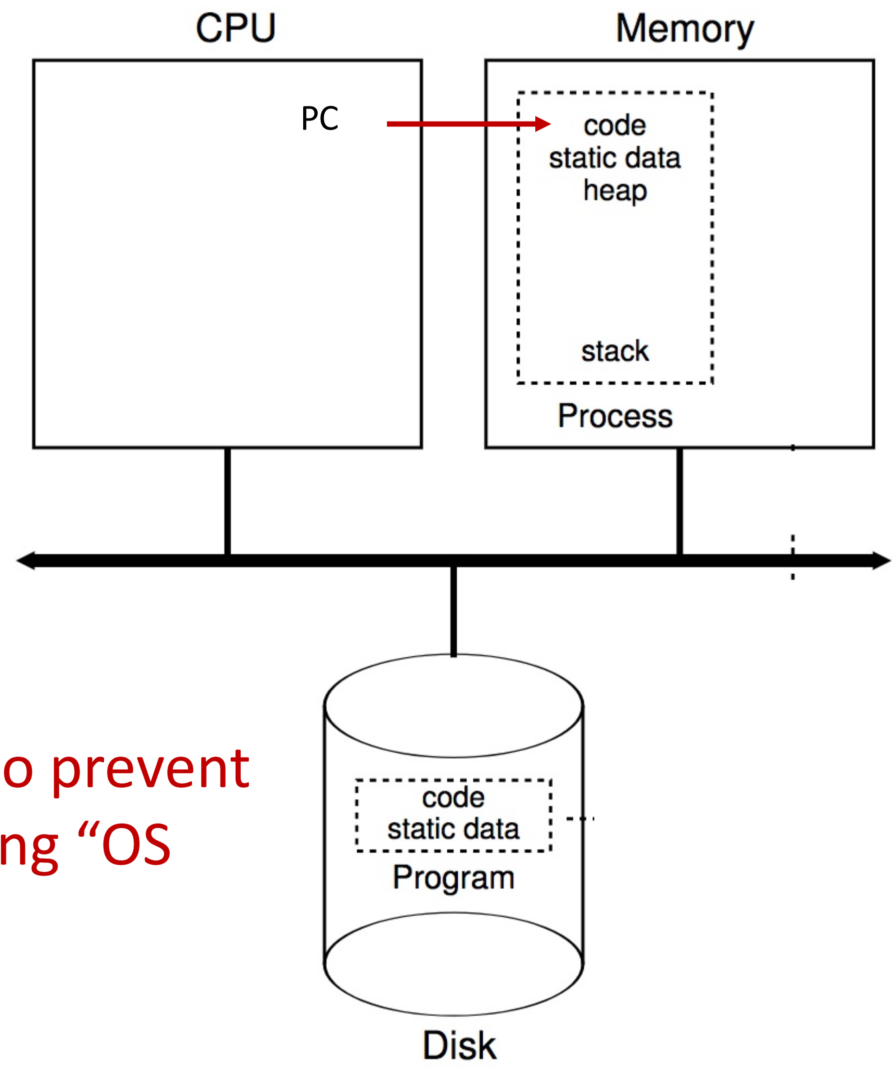
Before, PC points to
kernel code

Process Creation



Now, after process creation, CPU begins directly executing process code

Process Creation



Challenge: how to prevent process from doing “OS kernel stuff”?

Limited Direct Execution (LDE)

Limited Direct Execution (LDE)

- Low-level mechanism that implements the user-kernel space separation

- Usually let processes run with no OS involvement
- Limit what processes can do
- Offer privileged operations through well-defined channels with help of OS

Limited Direct Execution (LDE)

Limited Direct Execution (LDE)

User-level process

LDE mechanism

OS

What to limit?

- General memory access
- Disk I/O
- Certain x86 instructions

How to limit?

- Need hardware support
- Add additional execution mode to CPU

- User mode: restricted, limited capabilities
- Kernel mode: privileged, not restricted

- Processes start in user mode
- OS starts in kernel mode

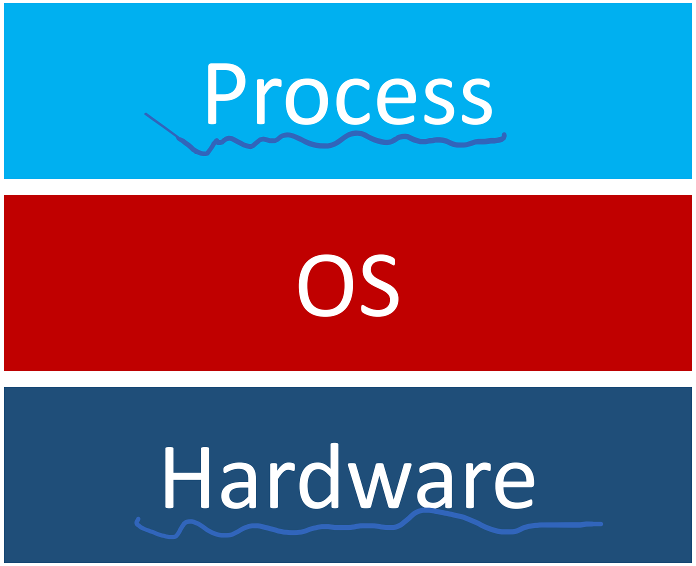
LDE: Remaining Challenges

1. What if process wants to do something privileged?
2. How can OS switch processes (or do anything) if it’s not running?

LDE: Remaining Challenges

1. What if process wants to do something privileged?
2. How can OS switch processes (or do anything) if it's not running?

Taking Turns

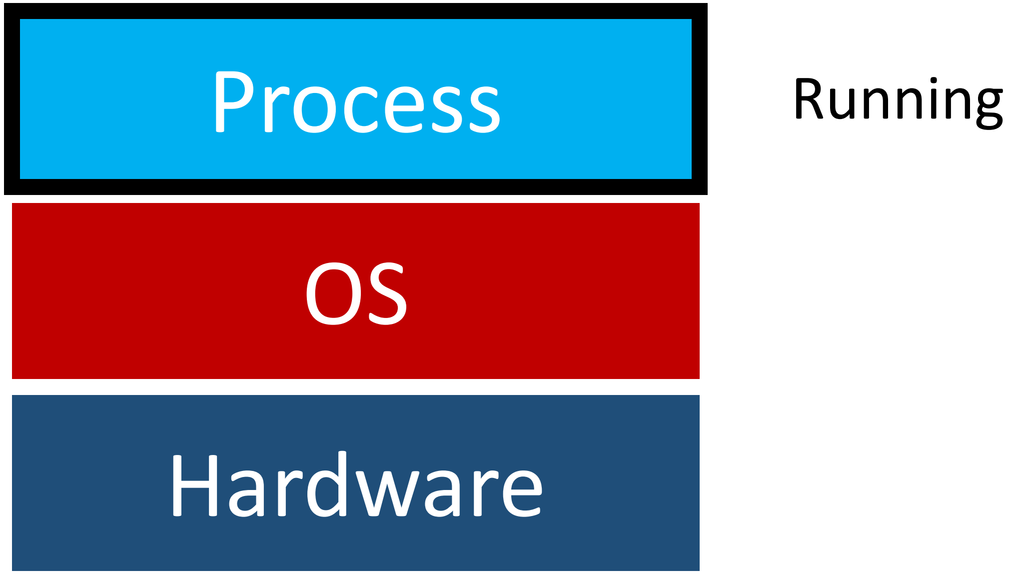


Process

OS

Hardware

Taking Turns

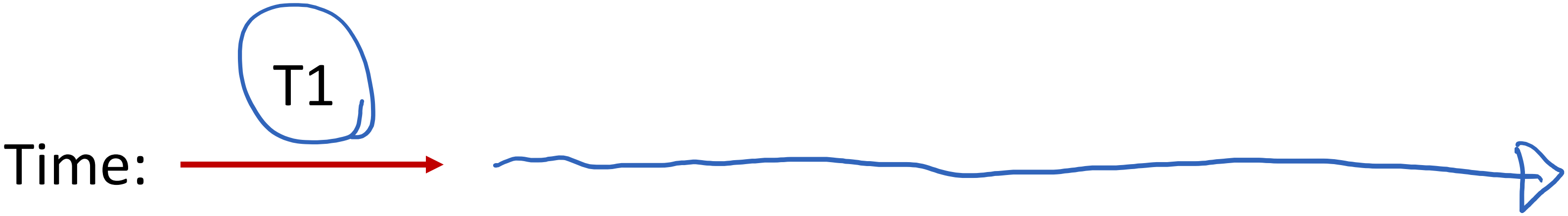


Process

OS

Hardware

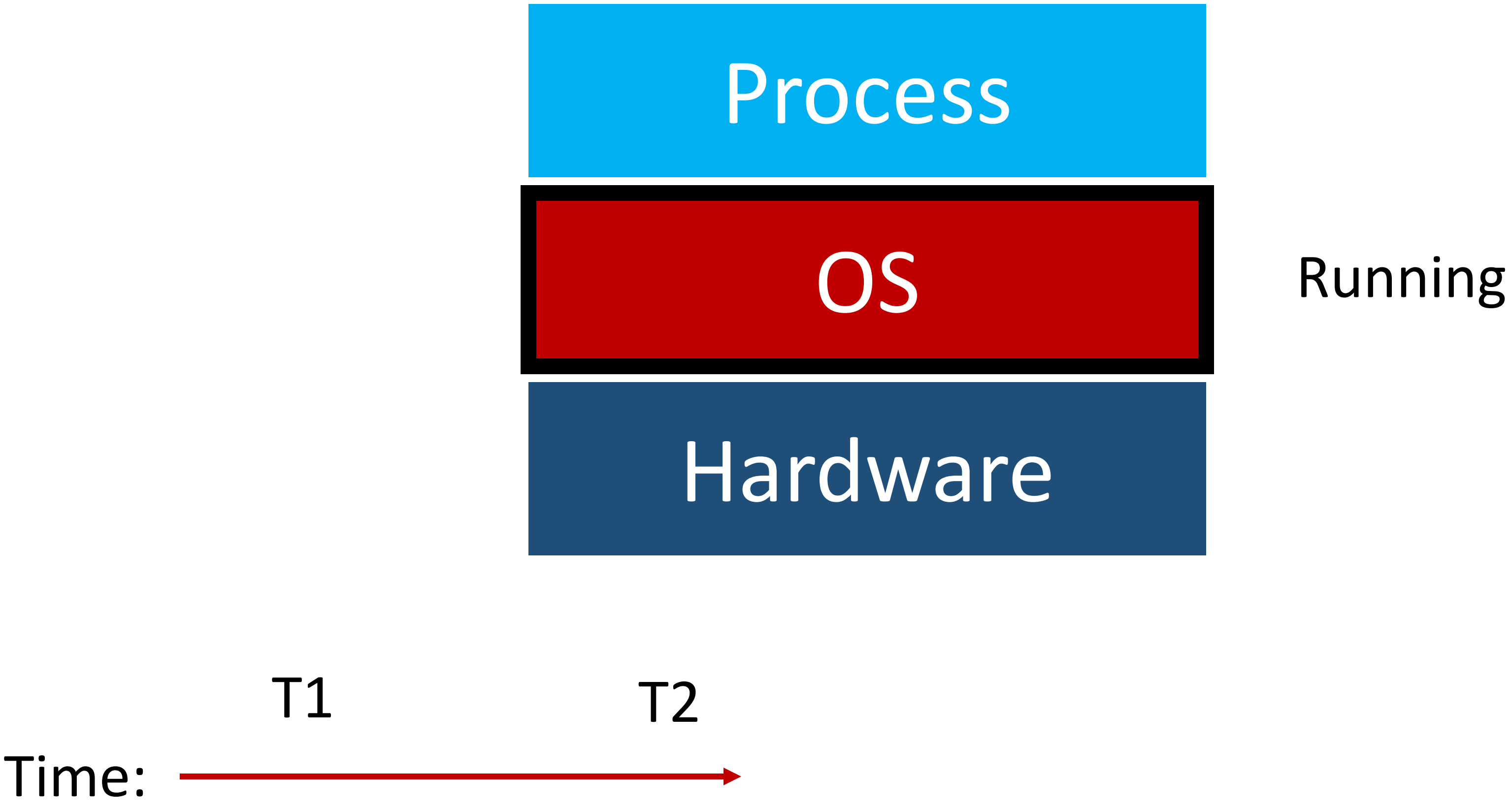
Running



Time:

T1

Taking Turns



Process

OS

Hardware

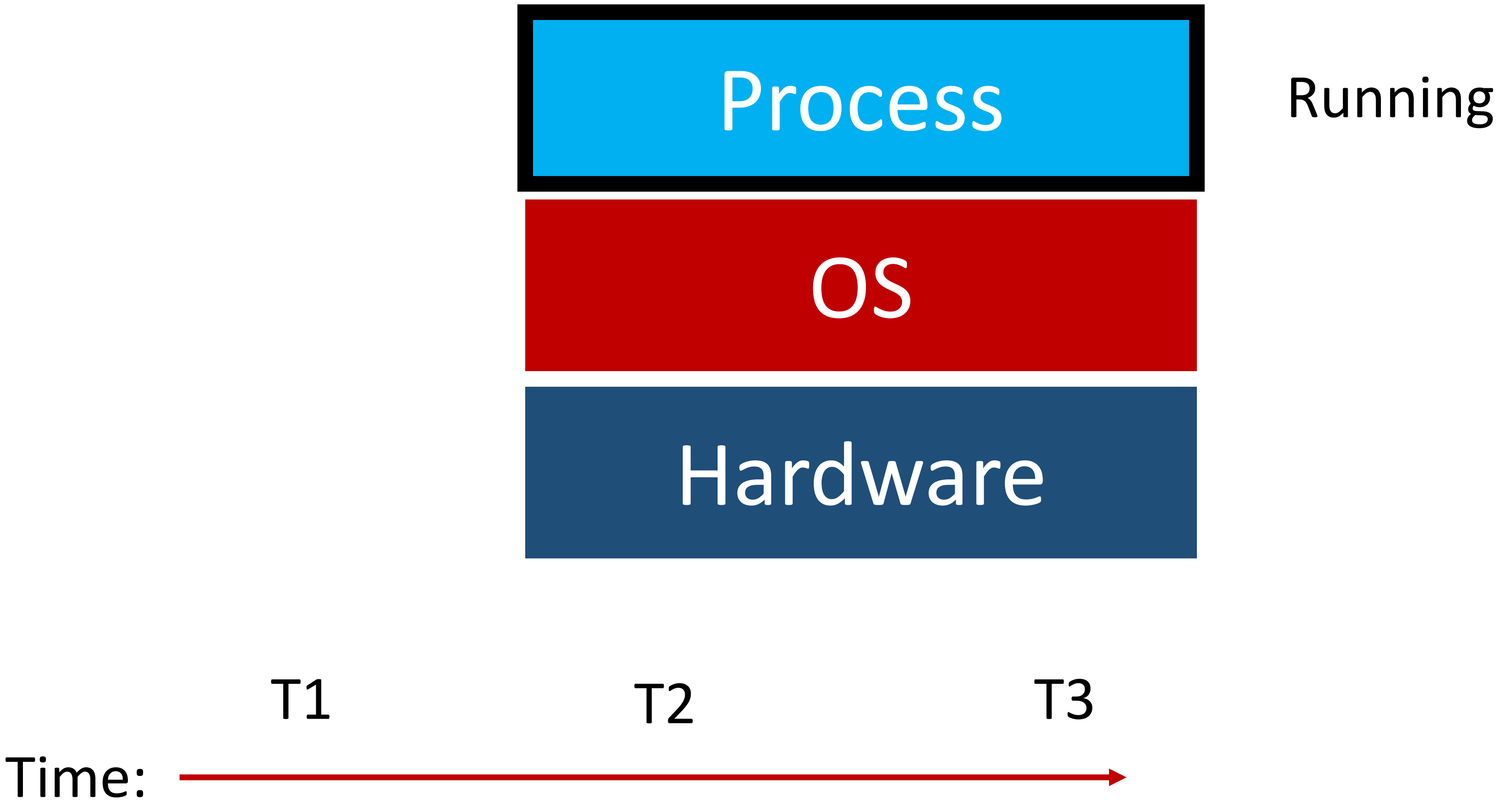
Running

T1

T2

Time:

Taking Turns



Process

Running

OS

Hardware

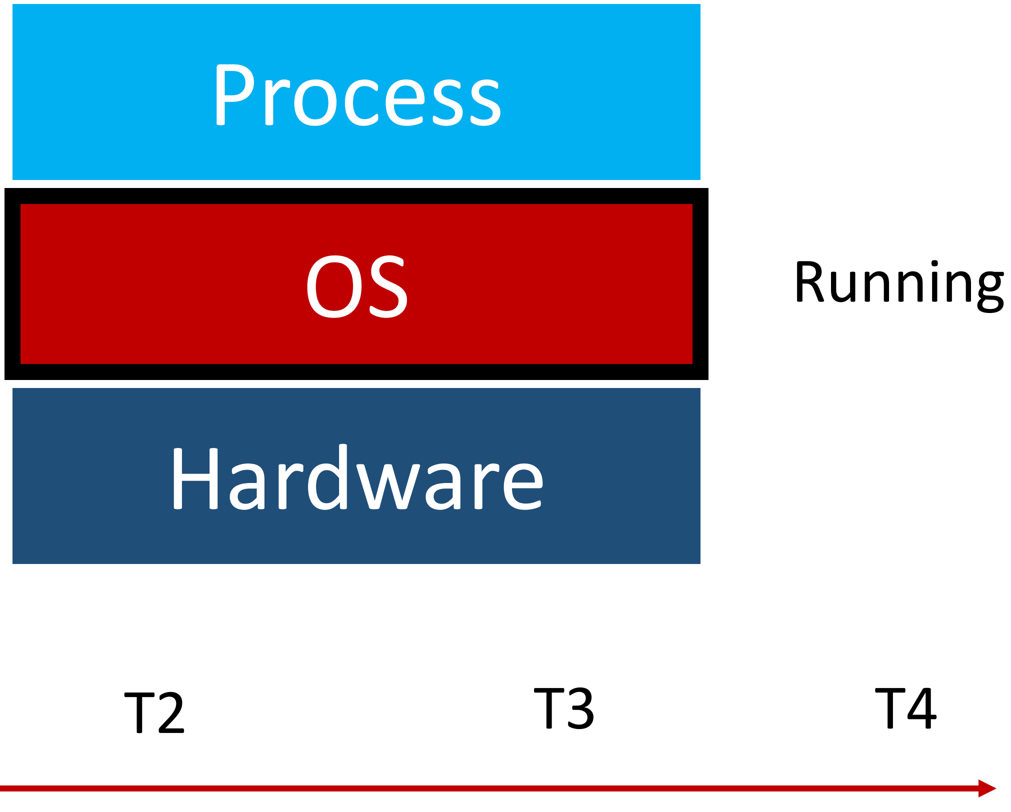
T1

T2

T3

Time:

Taking Turns



Process

OS

Running

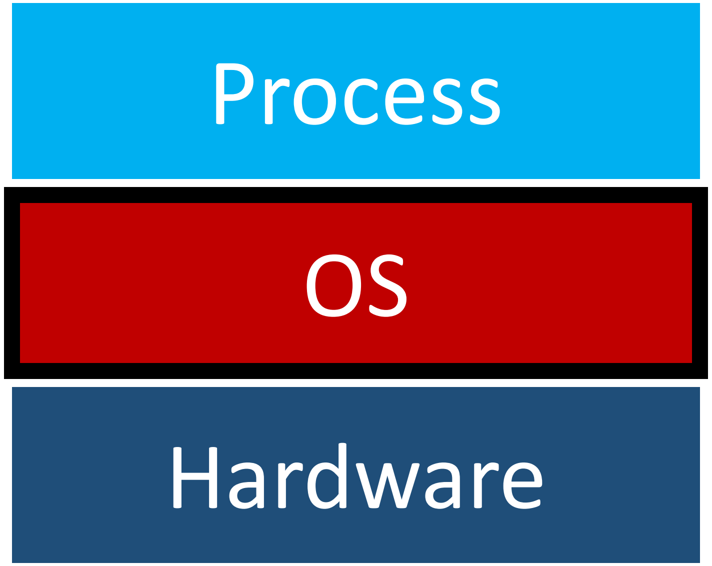
Hardware

T1 T2 T3 T4

Time:

Taking Turns

Question: when/how do we switch to OS?



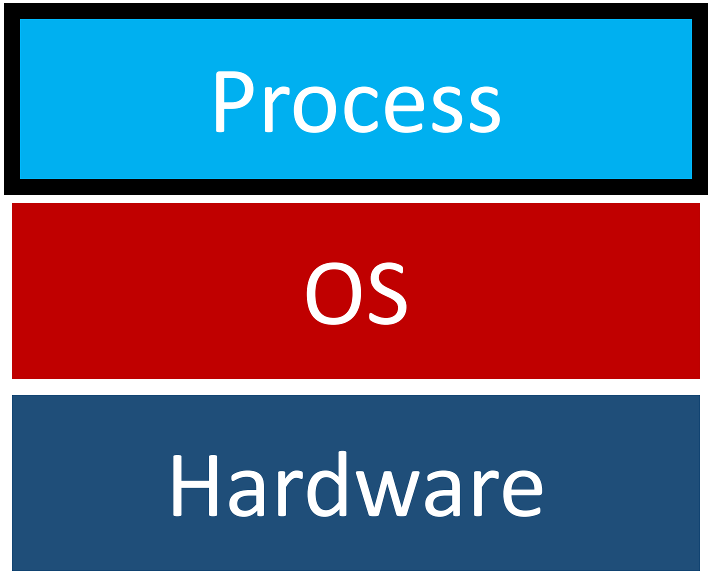
Running

Time:

Exceptions

Interrupt

H/W

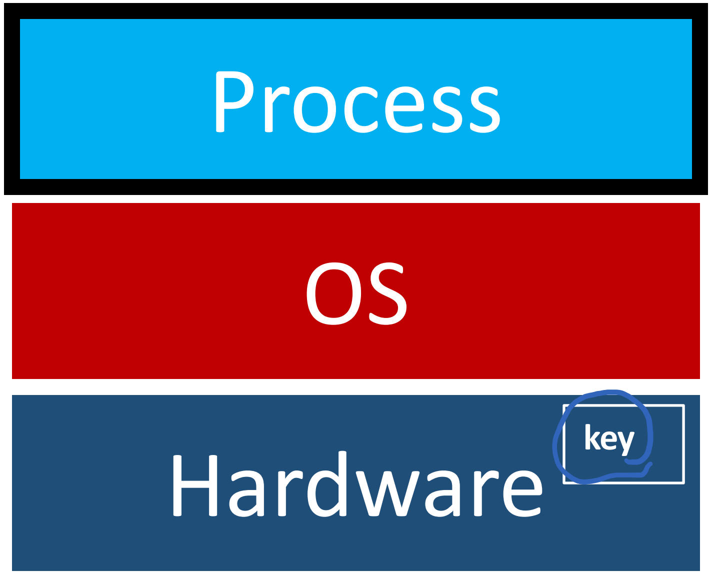


Process

OS

Hardware

Interrupt



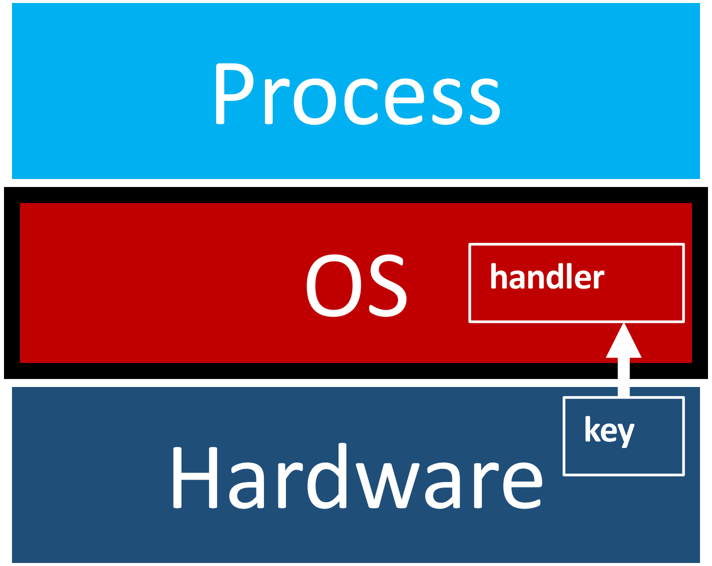
Process

OS

Hardware

key

Interrupt



Process

OS

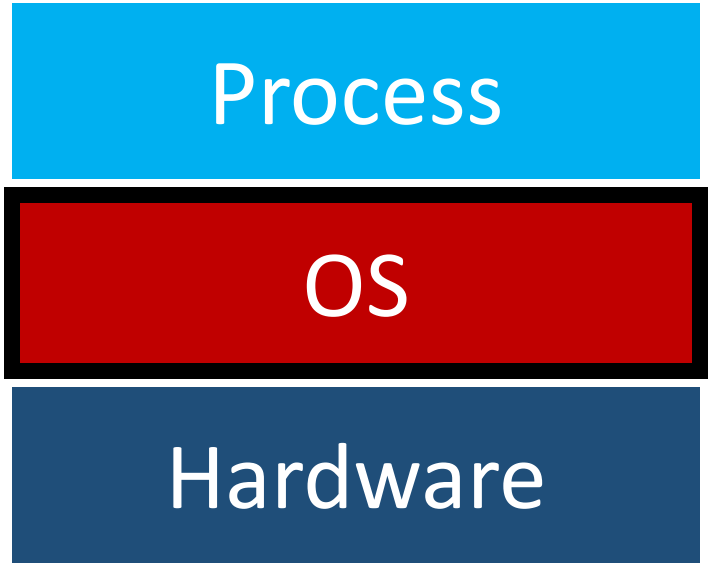
handler

key

Hardware

Hardware interrupt

Interrupt

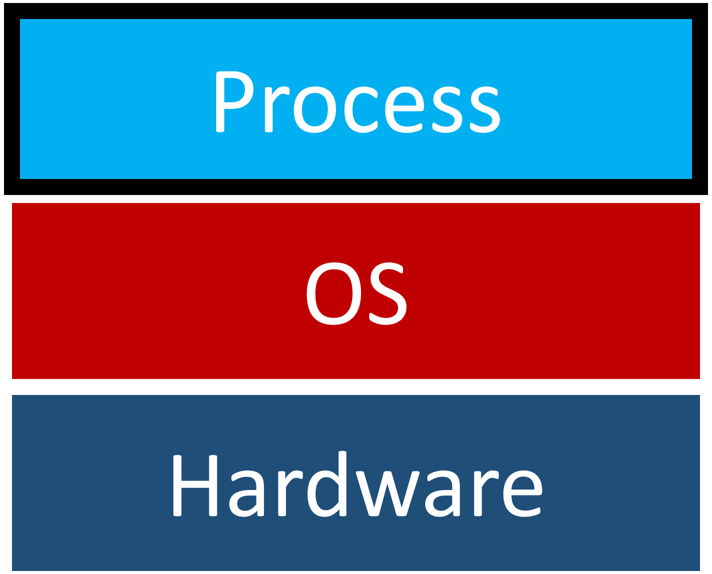

Process

OS

Hardware

System Call

Software Interrupt.

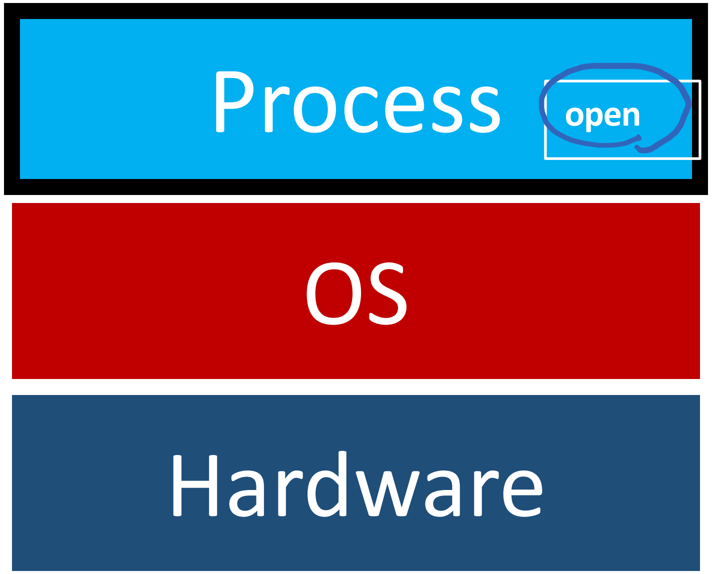


Process

OS

Hardware

System Call

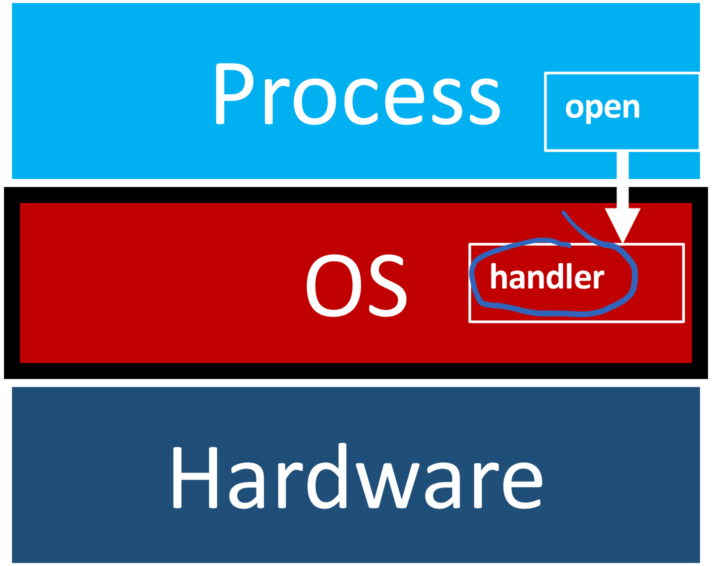

Process

open

OS

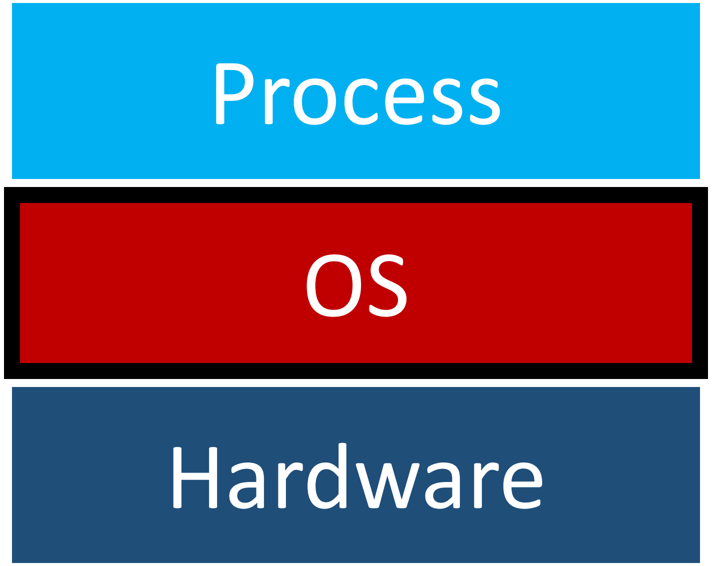
Hardware

System Call



System call "trap"

System Call



Process

OS

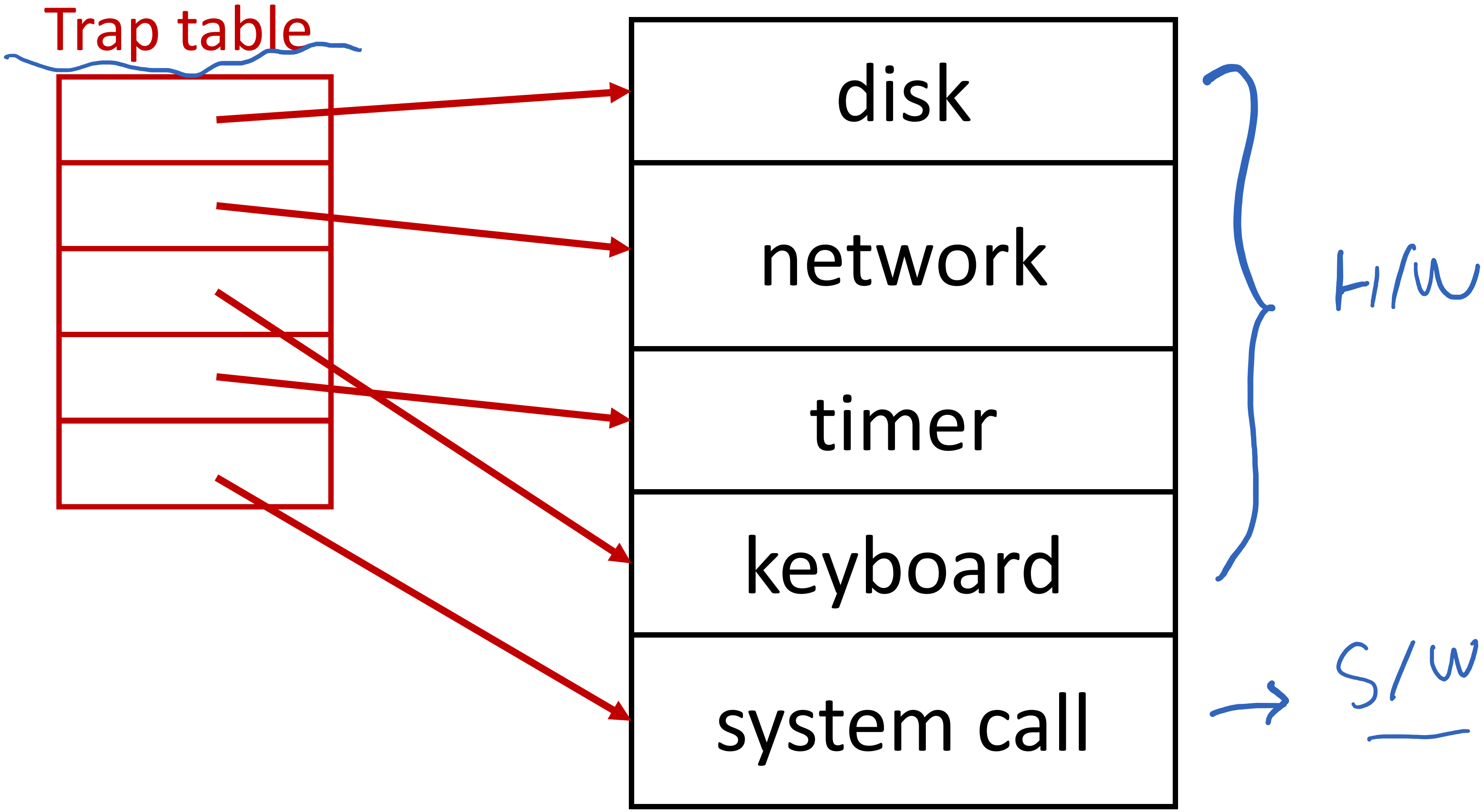
Hardware

Exception Handling

Exception Handling: Implementation

- Goal: Processes and hardware should be able to call functions in the OS

- Corresponding OS functions should be:
  - At well-known locations
  - Safe from processes



Trap table

disk

network

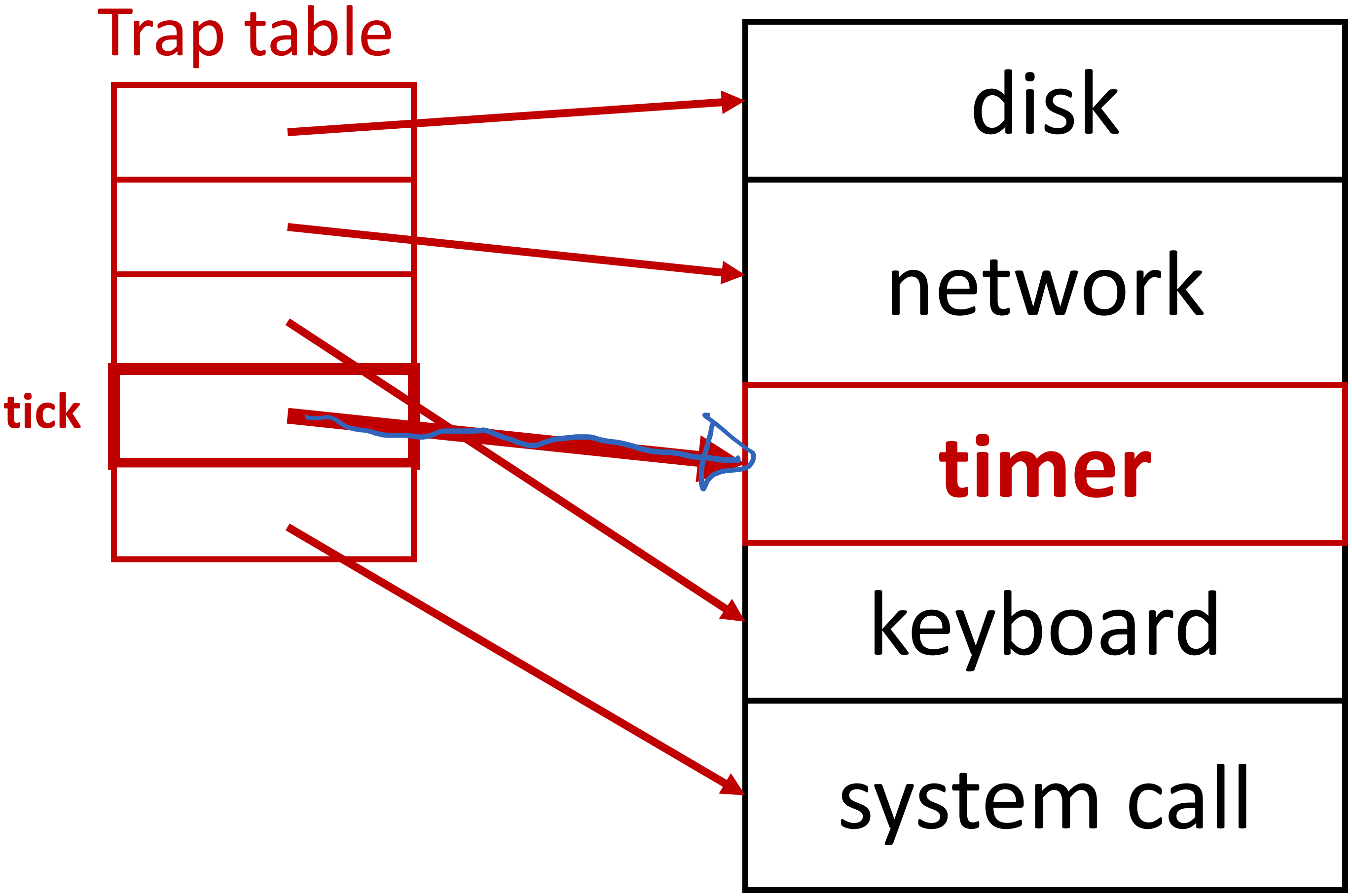
timer

keyboard

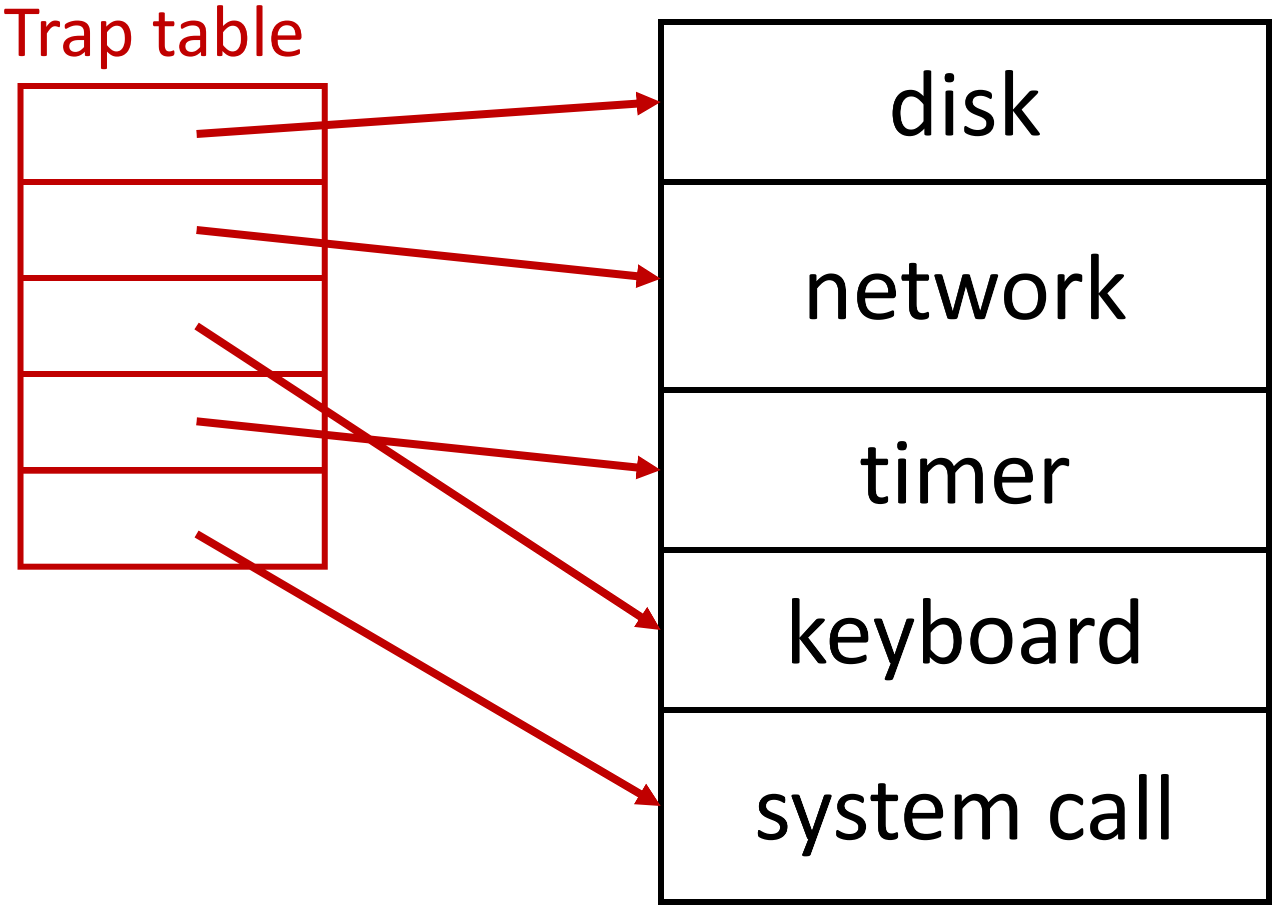
system call

Use array of function pointers to locate OS functions

(**Hardware knows where this is**)



Use array of function pointers to locate OS functions
(Hardware knows this through **`lidt`** instruction)



Trap table

disk

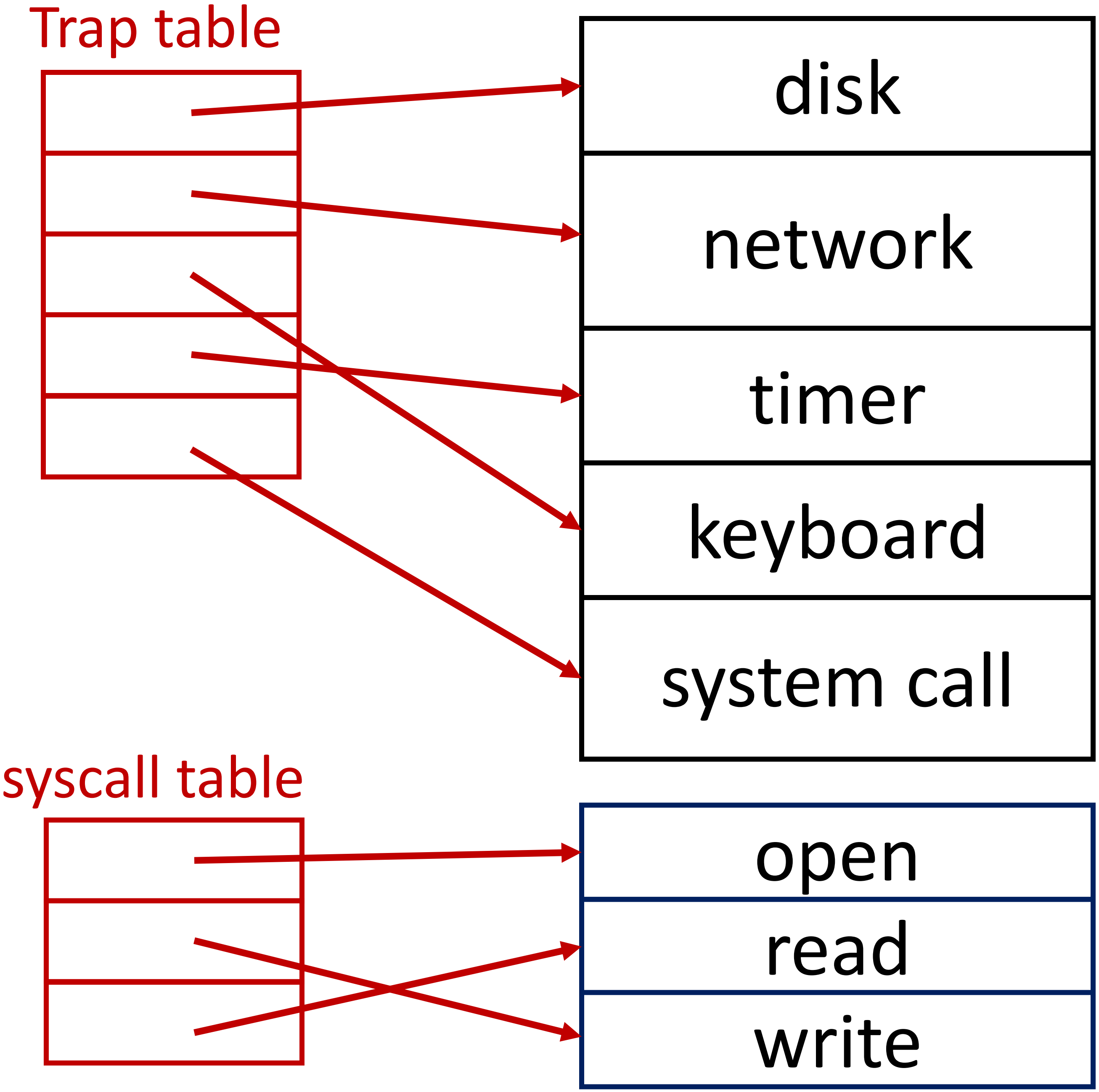
network

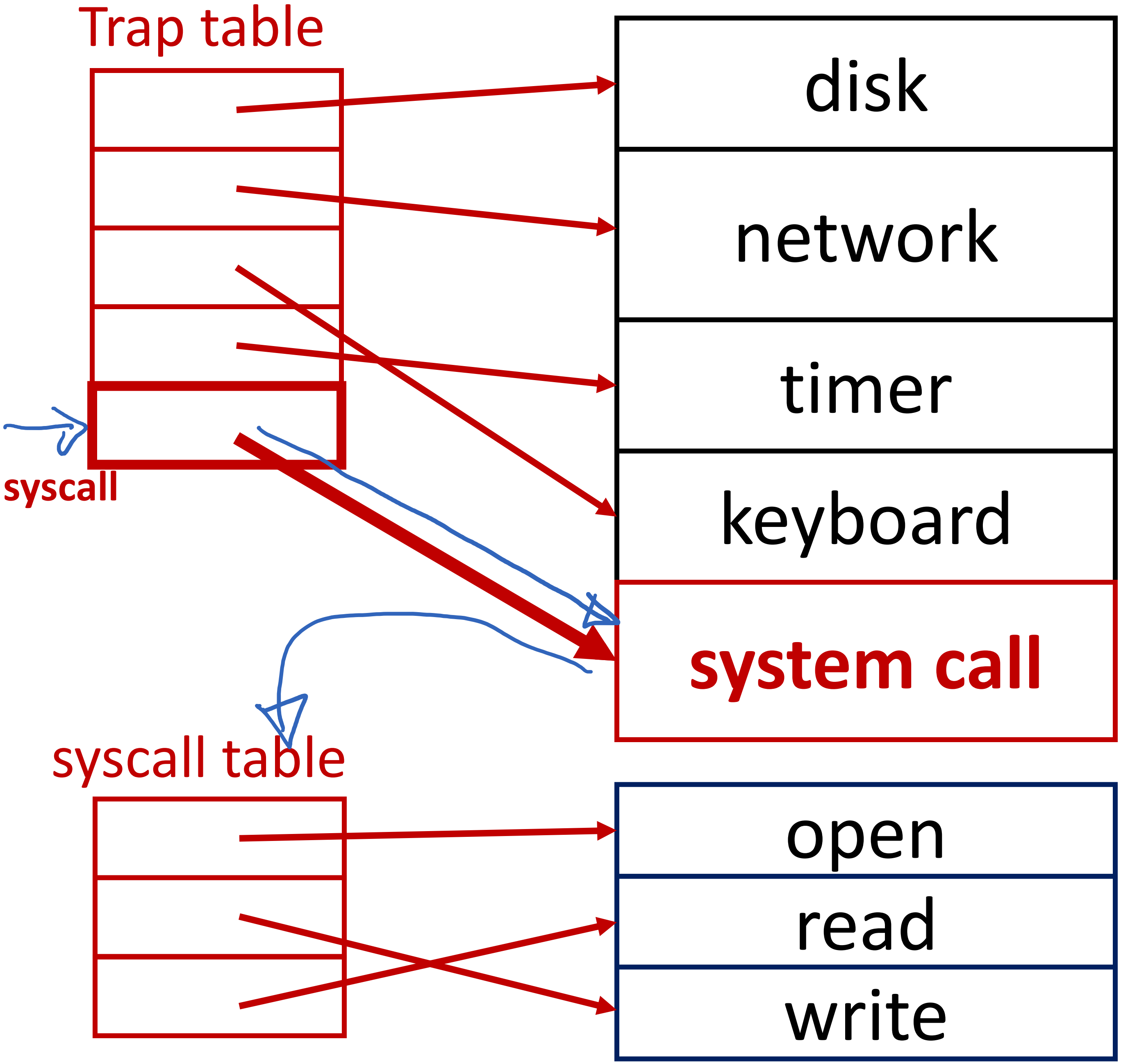
timer

keyboard

system call

How to handle variable number of system calls?


Trap table
disk
network
timer
keyboard
system call
syscall table
open
read
write



Trap table

disk

network

timer

keyboard

system call

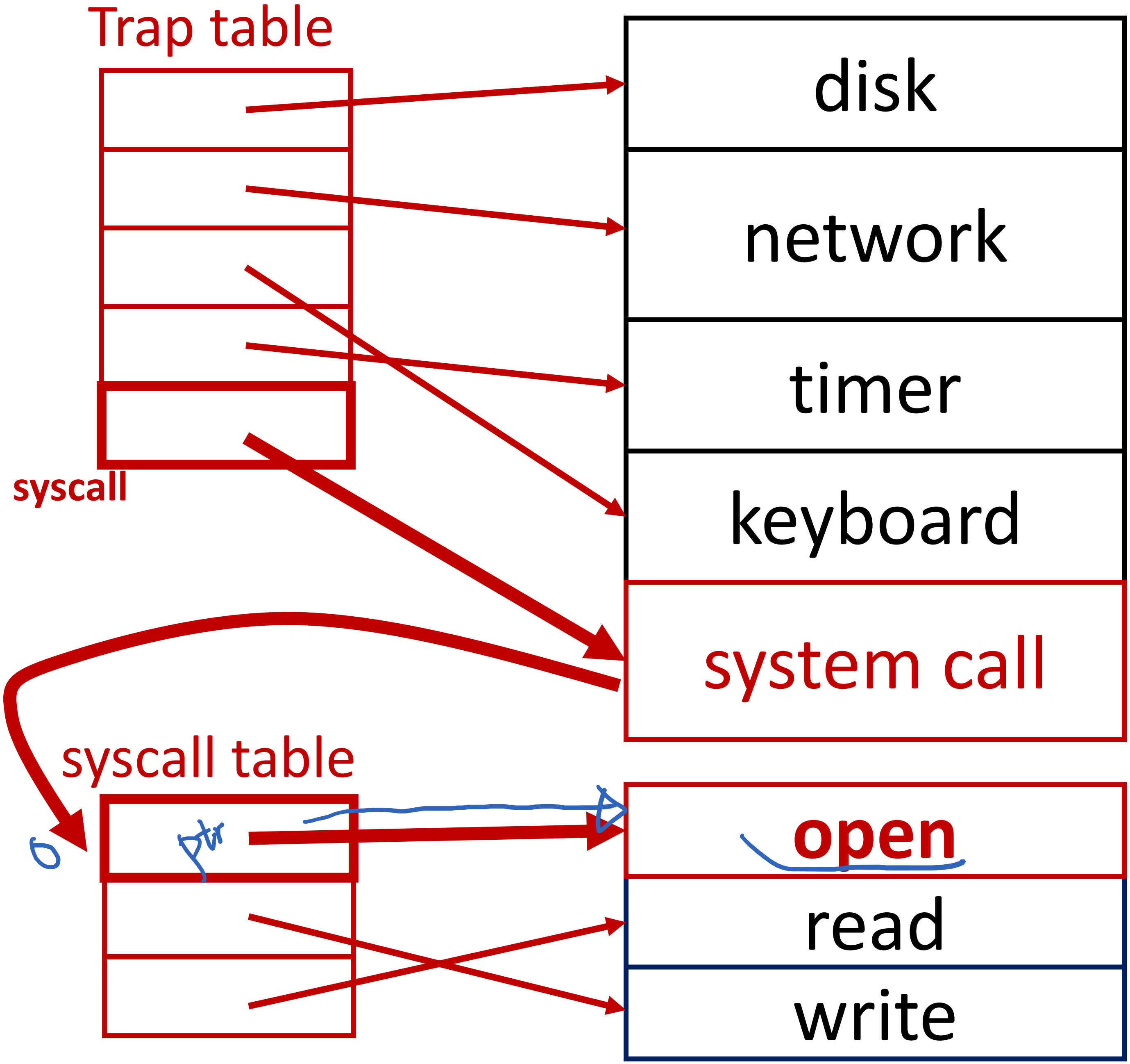
syscall

syscall table

open

read

write



Trap table

syscall

disk

network

timer

keyboard

system call

syscall table

0

ptr

open

read

write

Safe Transfers

- Only certain kernel functions should be callable
- Privileges should escalate at the moment of the call
  - Read/write disk
  - Kill processes
  - Access all memory
  - …

LDE: Remaining Challenges

1. What if process wants to do something privileged?
2. How can OS switch processes (or do anything) if it's not running?

Sharing (virtualizing) the CPU

multiplexing.

How does OS share…

- CPU?

- Memory?

- Disk?

How does OS share…

- CPU? (a: time sharing)
- Memory? (a: space sharing)
- Disk? (a: space sharing)

How does OS share…

- CPU? (a: time sharing)

Today

- Memory? (a: space sharing)
- Disk? (a: space sharing)

How does OS share...

- CPU? (a: time sharing)

Today

- Memory? (a: space sharing)
- Disk? (a: space sharing)

Goal: processes should **not** know they are sharing **(each process will get its own virtual CPU)**

What to do with processes that are not running?

- A: Store context in OS struct

What to do with processes that are not running?

- A: Store context in OS struct

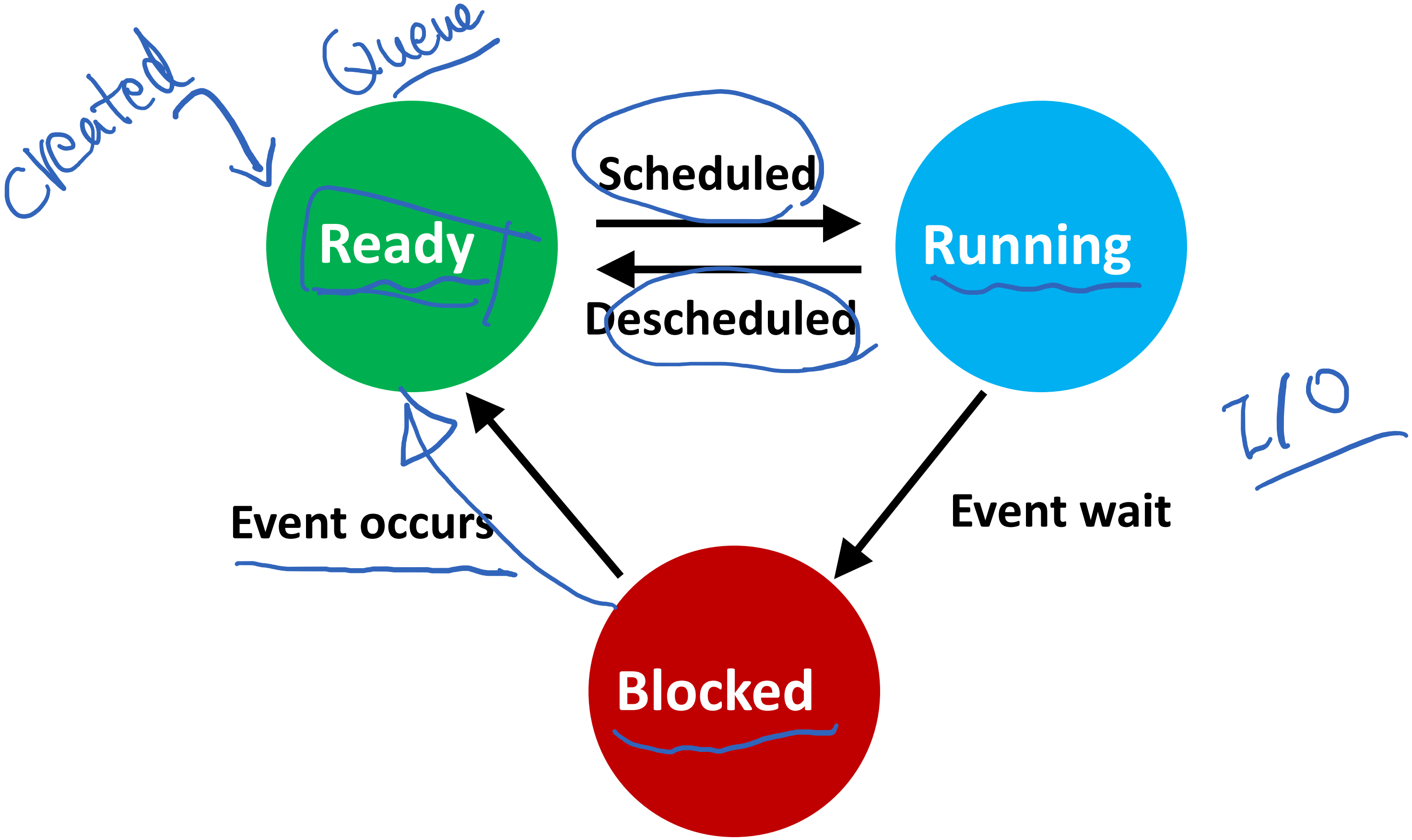
- Context:
    - CPU registers
    - Open file descriptors
    - State (sleeping, running, etc.)

What to do with processes that are not running?

- A: Store context in OS struct

- Context:
    - CPU registers
    - Open file descriptors
    - State (sleeping, running, etc.)

Process State Transitions



Created

Queue

Ready

Scheduled

Descheduled

Running

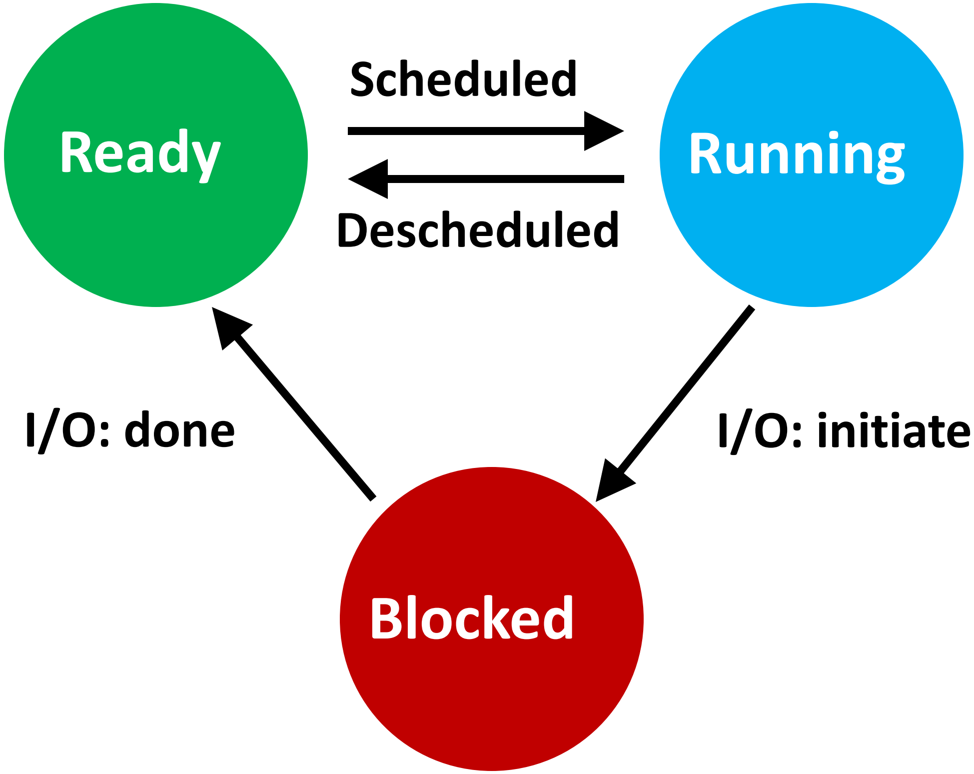
I/O

Event wait

Event occurs

Blocked

Process State Transitions



Ready

Scheduled

Running

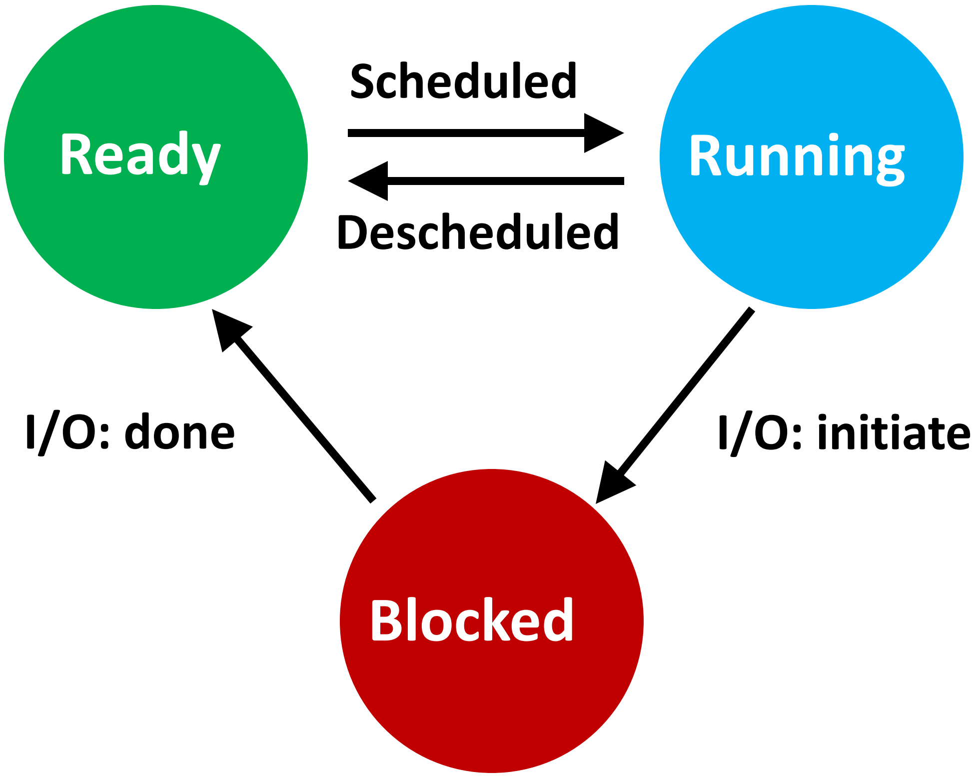
Descheduled

I/O: done

I/O: initiate

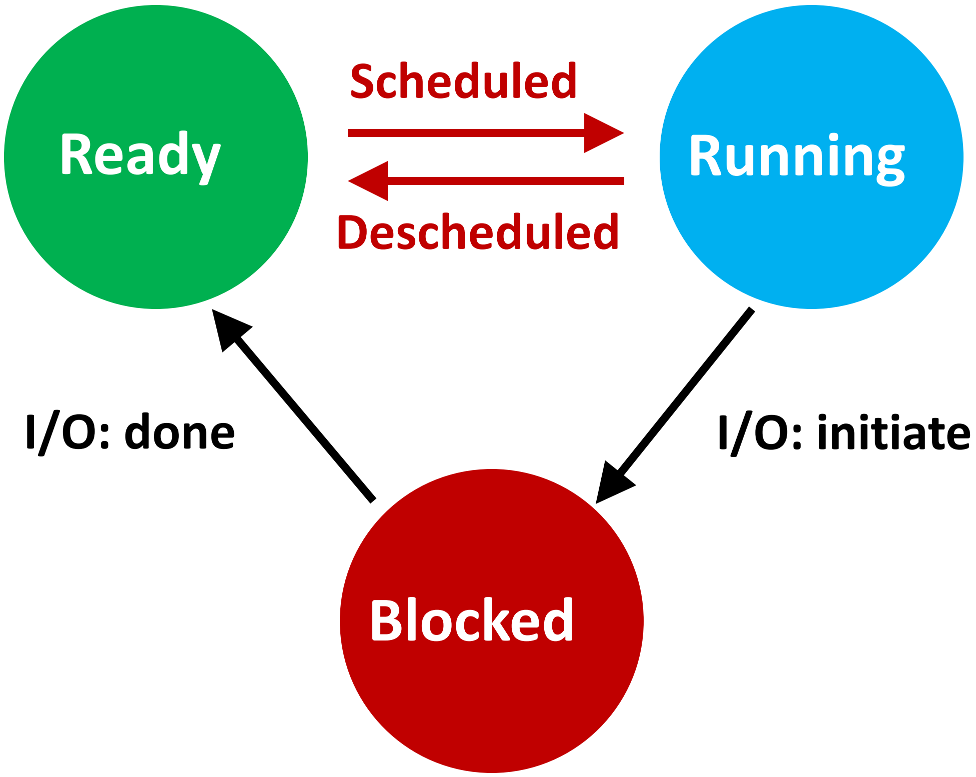
Blocked

Process State Transitions



View process state with "`ps xa`"

How to transition? (mechanism)
When to transition? (policy)



Context Switch

- Problem: When to switch process contexts?
- Direct execution => OS can't run while process runs

- Can OS do anything while it's not running?

Context Switch

- Problem: When to switch process contexts?
- Direct execution => OS can’t run while process runs

- Can OS do anything while it’s not running?
- A: it can’t

Context Switch

- Problem: When to switch process contexts?
- Direct execution => OS can't run while process runs

- Can OS do anything while it's not running?
- A: it can't

- Solution: Switch on interrupts
  - But what interrupt?

Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call

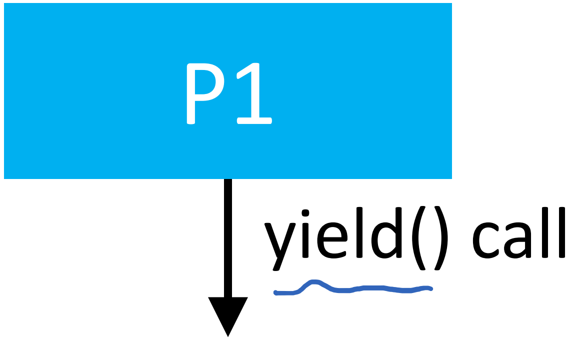
Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call

P1

Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call

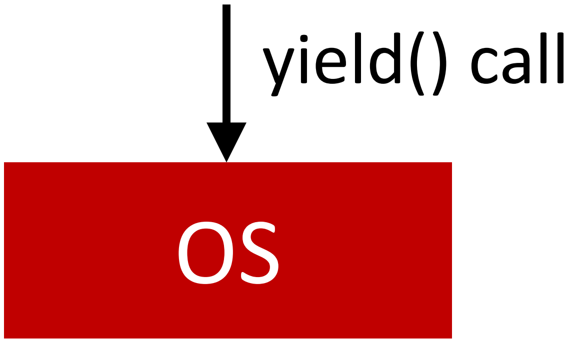


P1

yield() call

Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call



yield() call

OS

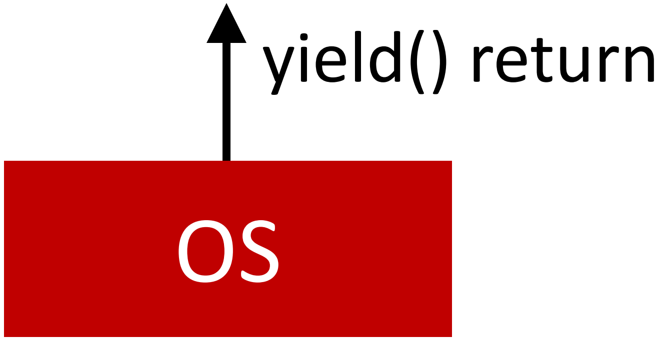
Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call

OS

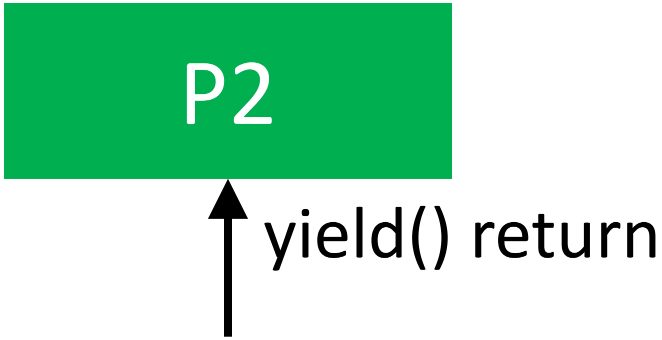
Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call



Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call



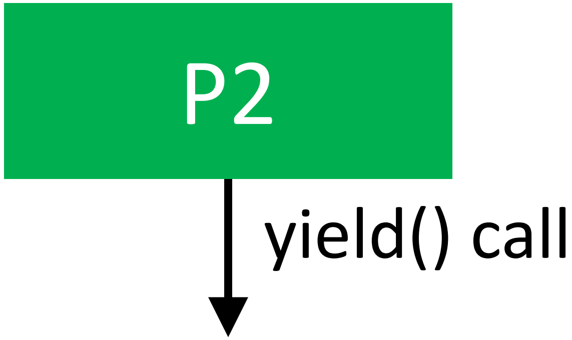
Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call

P2

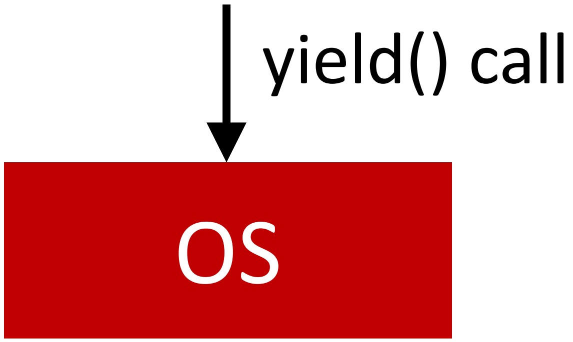
Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call



Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call



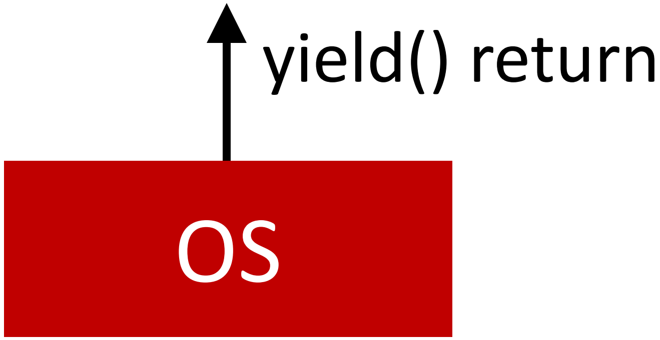
Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call

OS

Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call

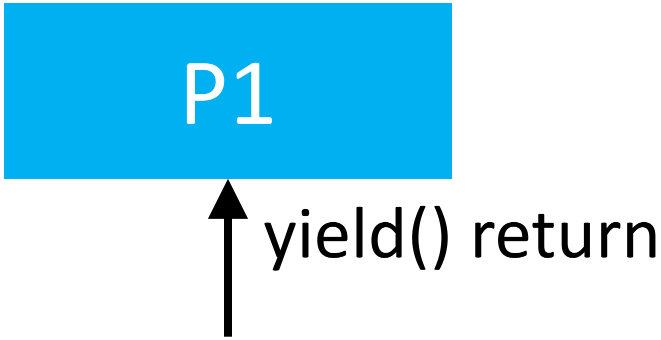


yield() return

OS

Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call



Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call

P1

Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call

P1

Critiques?

Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call
- Cooperative approach is a passive approach

P1

Critiques?

What if P1 never calls `yield()`?

Non-Cooperative Approach

- Switch contexts on timer (hardware) interrupt
- Set up before running any processes
- Hardware does not let processes prevent this
  - Hardware/OS enforces process preemption

Preemptive Sched.

Non-Cooperative Approach

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A …

time

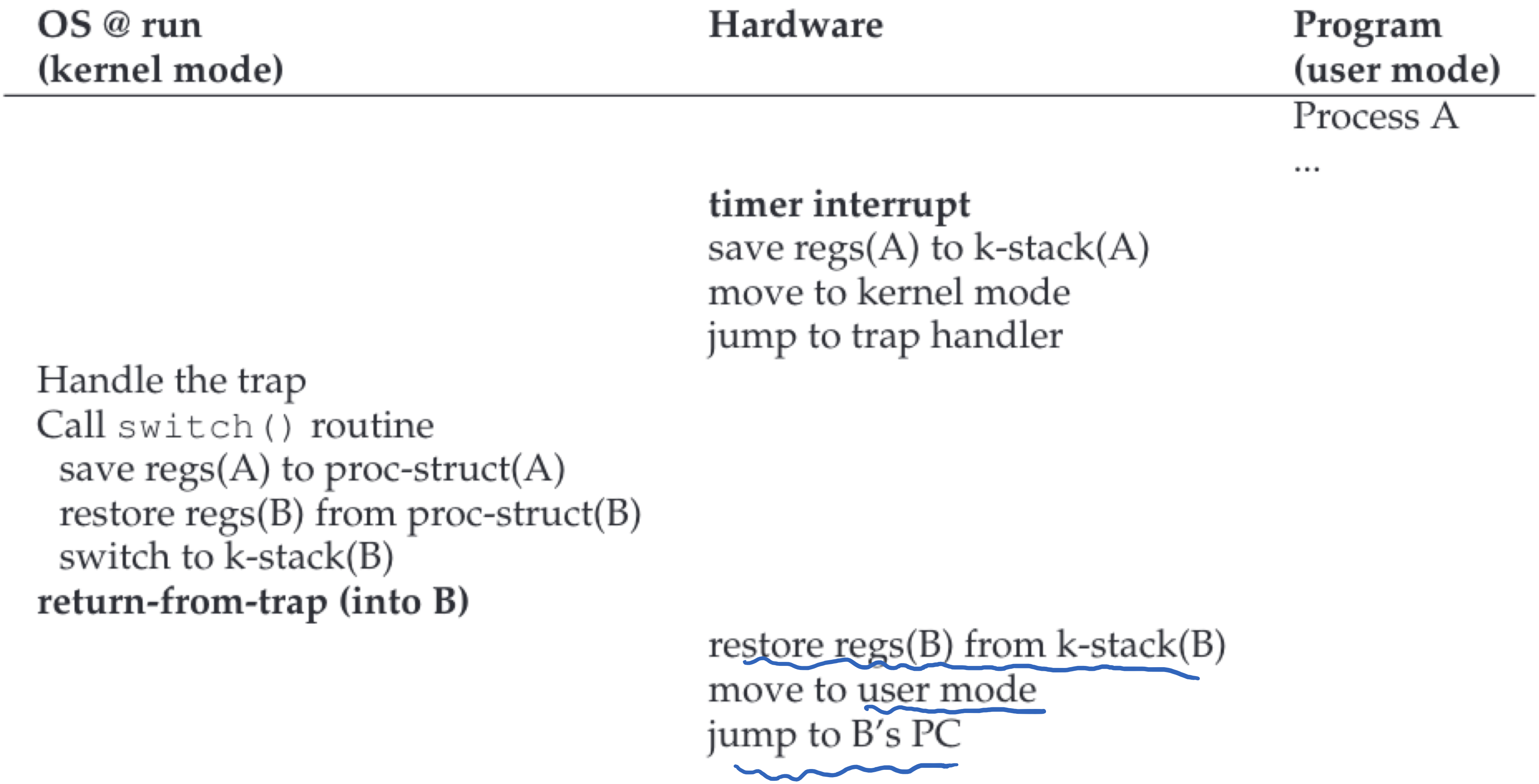
Non-Cooperative Approach

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A ...
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	

Non-Cooperative Approach

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A …
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call `switch()` routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) **return-from-trap (into B)**		

Non-Cooperative Approach



OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A ...
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call `switch()` routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) **return-from-trap (into B)**		
	restore regs(B) from k-stack(B) move to user mode jump to B's PC	

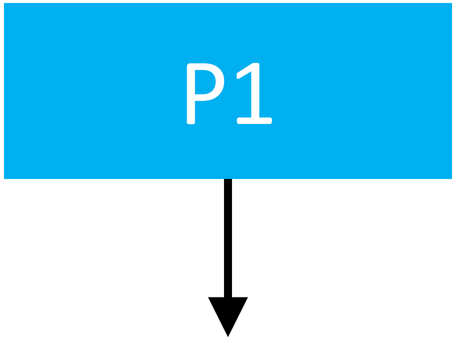
Non-Cooperative Approach

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A …
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call `switch()` routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) **return-from-trap (into B)**		
	restore regs(B) from k-stack(B) move to user mode jump to B's PC	
		Process B …

Preemptive Approach

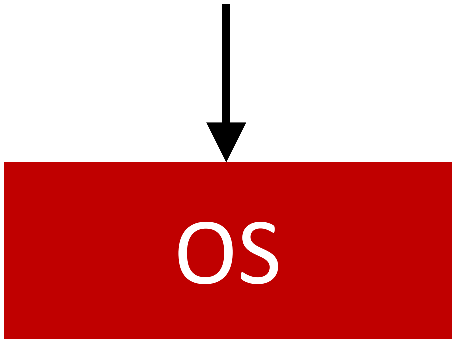
P1

Preemptive Approach



tick

Preemptive Approach



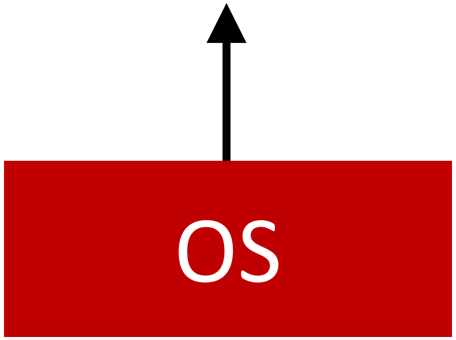
OS

tick

Preemptive Approach

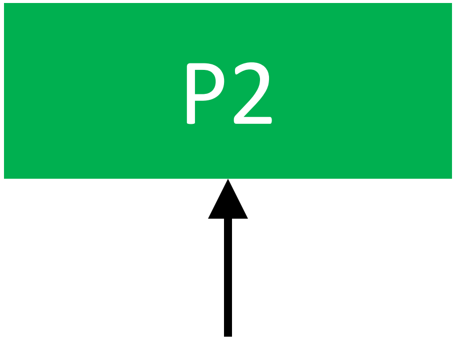
OS

Preemptive Approach



OS

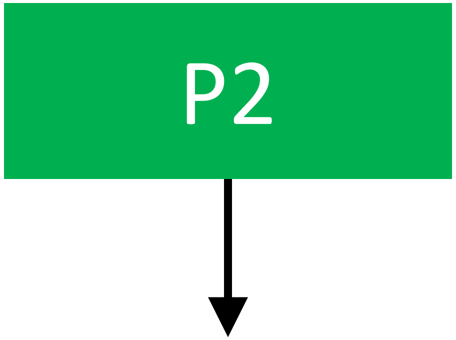
Preemptive Approach



Preemptive Approach

P2

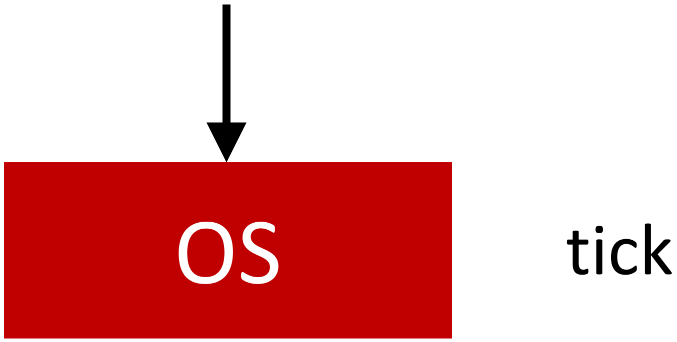
Preemptive Approach



P2

tick

Preemptive Approach



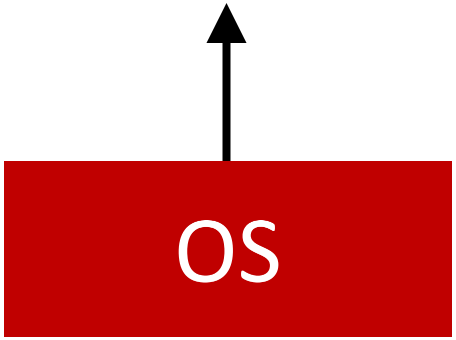
OS

tick

Preemptive Approach

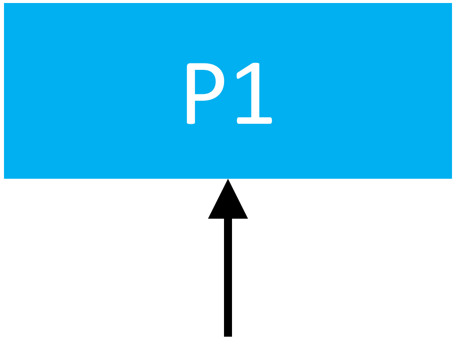
OS

Preemptive Approach



OS

Preemptive Approach



Preemptive Approach

P1

LDE Summary

- Smooth context switching makes each process think it has its own CPU (virtualization!)
- Limited direct execution makes processes fast
- Hardware provides a lot of OS support
  - Limited direct execution
  - Timer interrupt
  - Automatic register saving

Threads

Why Thread Abstraction?

Process Abstraction: Challenge 1

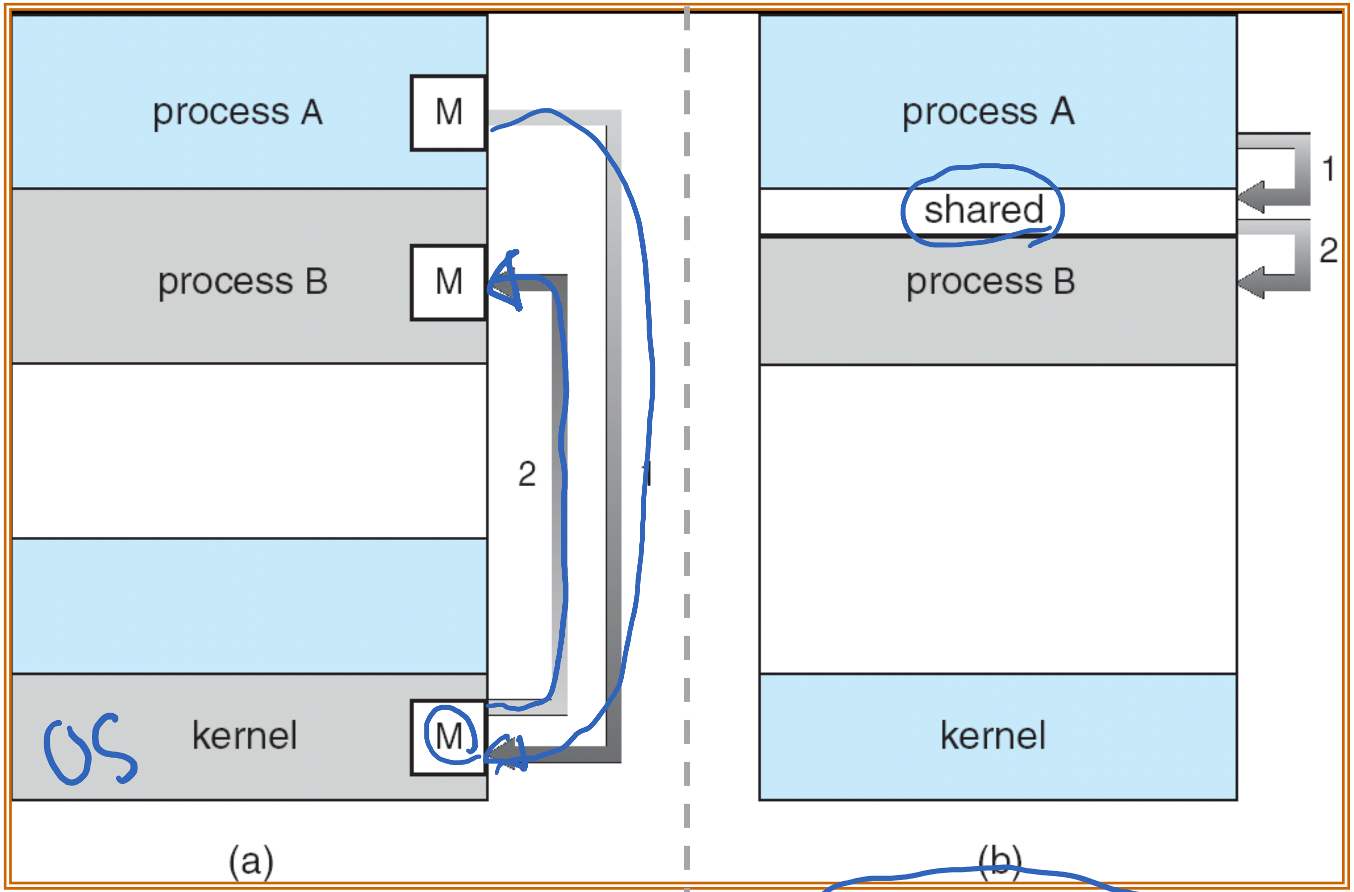
- Inter-process communication (IPC)

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



lock()
unlock()

Message Passing

Shared Memory

Previously, in a distributed system, message-passing was the only possible communication model. However, remote direct memory access (RDMA) technique bridges this gap by providing remote memory access through network.

Communication through Message Passing

- Message system – processes communicate with each other without resorting to shared variables

- A message-passing facility must provide at least two operations:
  - `send(message, recipient)`
  - `receive(message, recipient)`

- With **indirect** communication, the messages are sent to and received from mailboxes (or, ports)
  - `send(A, message) /* A is a mailbox */`
  - `receive(A, message)`

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

Communication through Shared Memory

- The memory region to be shared must be explicitly defined
- System calls (Linux):
  - `shmget` creates a shared memory block
  - `shmat` maps/attaches an existing shared memory block into a process's address space
  - `shmdt` removes ("**`unmaps`**") a shared memory block from the process's address space
  - `shmctl` is a general-purpose function allowing various operations on the shared block (receive information about the block, set the permissions, lock in memory, …)
- Problems with simultaneous access to the shared variables
- Compilers for concurrent programming languages can provide direct support when declaring variables (e.g., "**`shared int buffer`**")

Process Abstraction: Challenge 1

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

Process Abstraction: Challenge 2

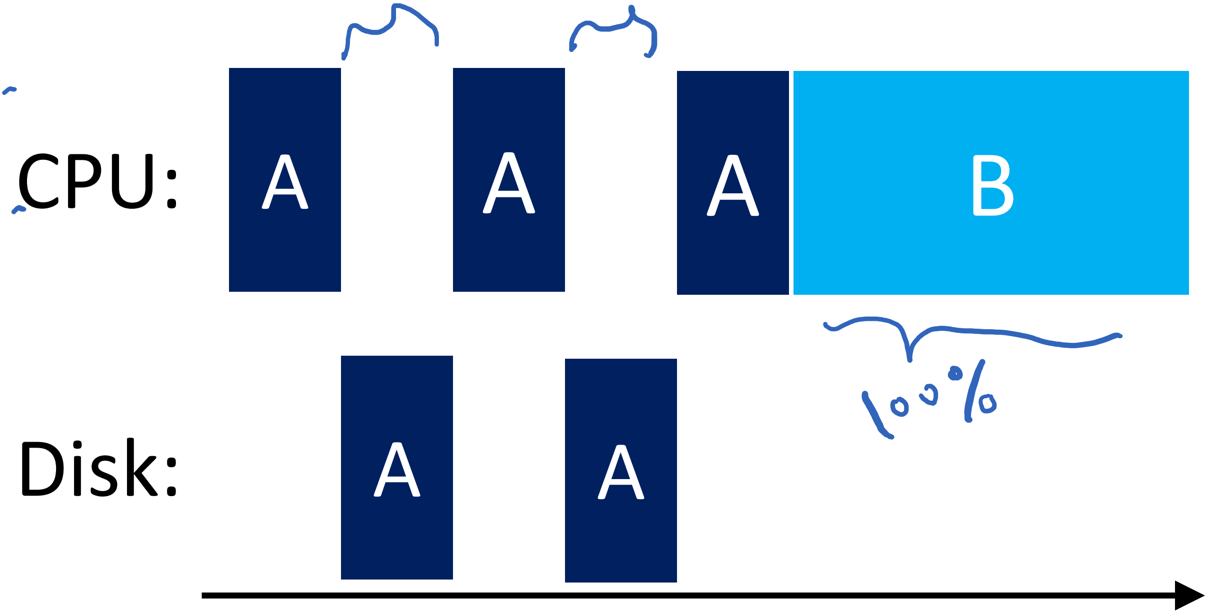
- Inter-process communication (IPC)
  - Cumbersome programming!
  - Copying overheads (inefficient communication)
  - Expensive context switching (why expensive?)

- CPU utilization

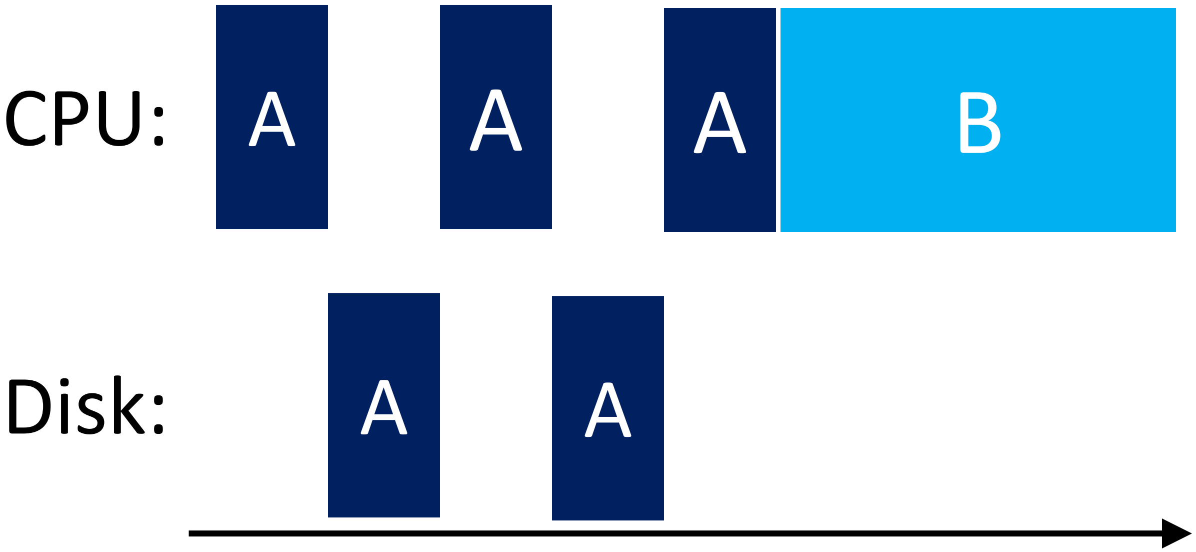
A: I/O intensive

B: CPU intensive

(a) Not interleaved

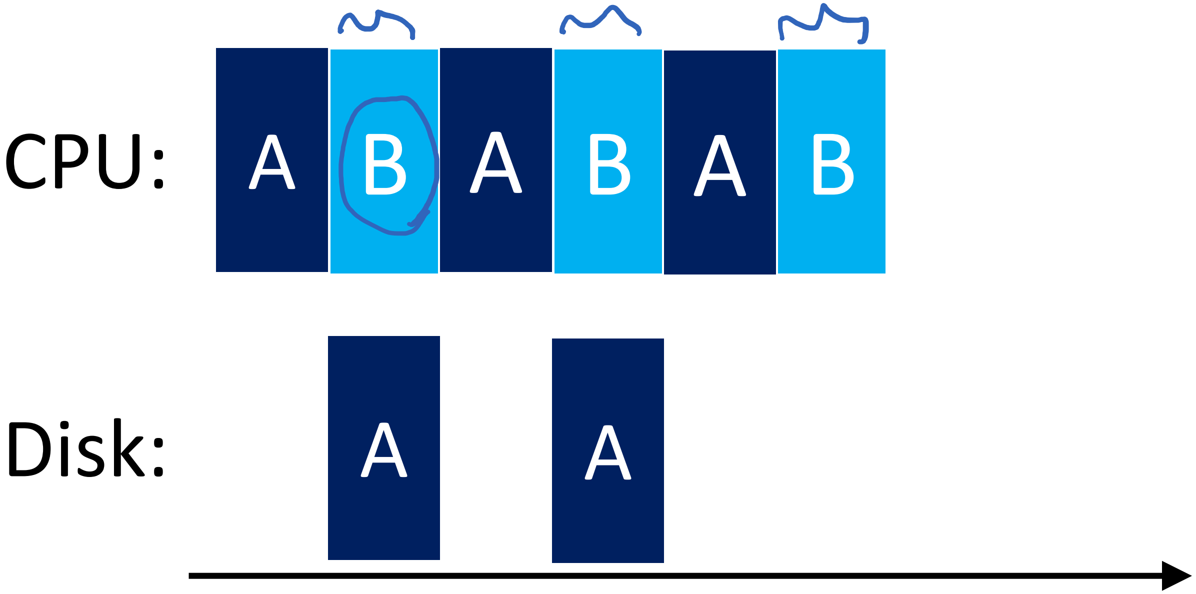


100%

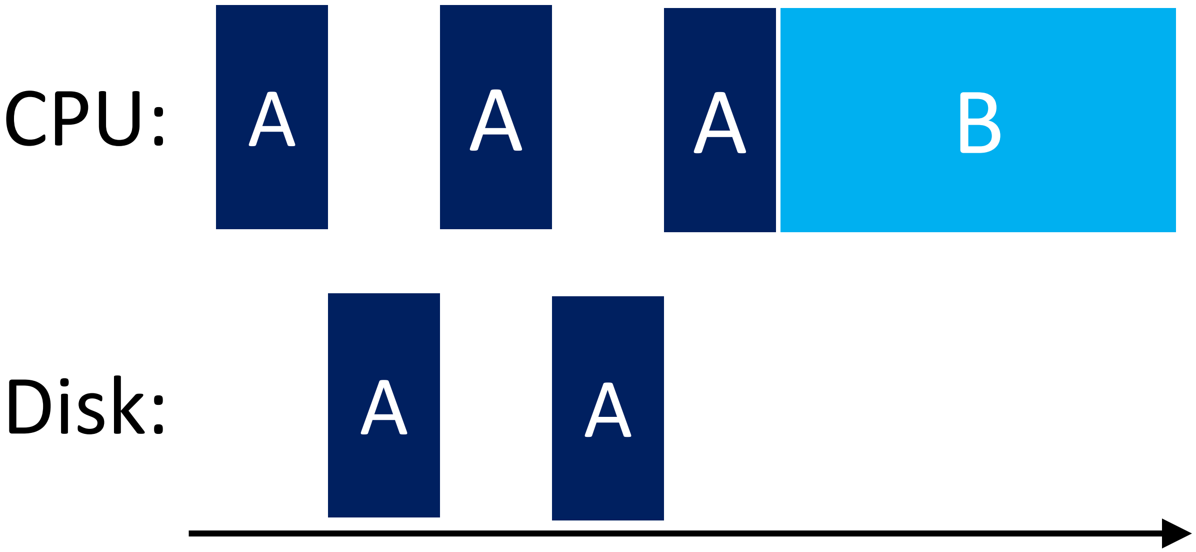


(a) Not interleaved

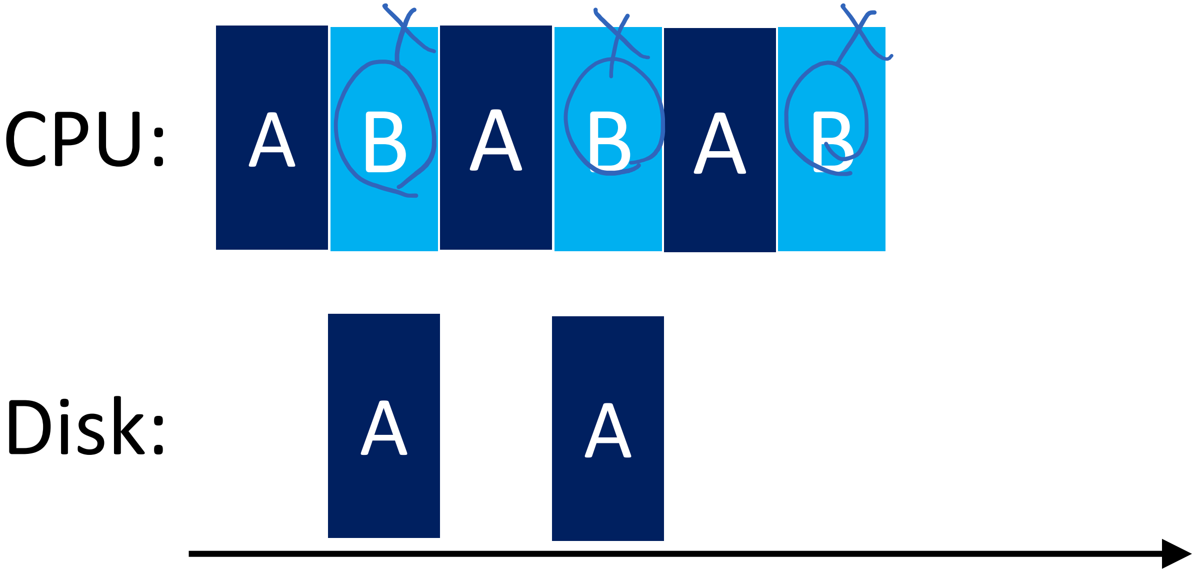
100%



(b) Interleaved



(a) Not interleaved



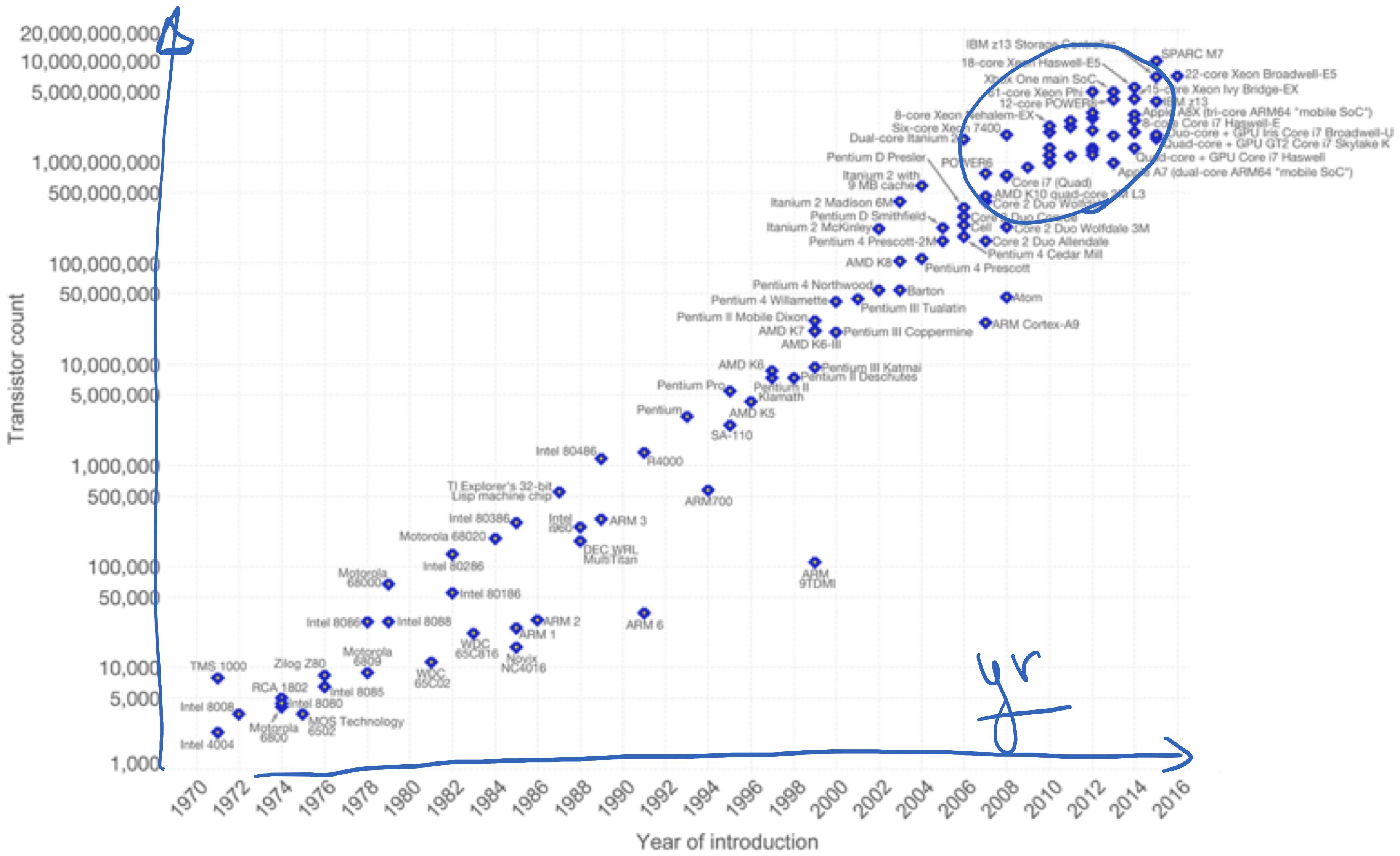
(b) Interleaved

What if there is only one process?

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

Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)

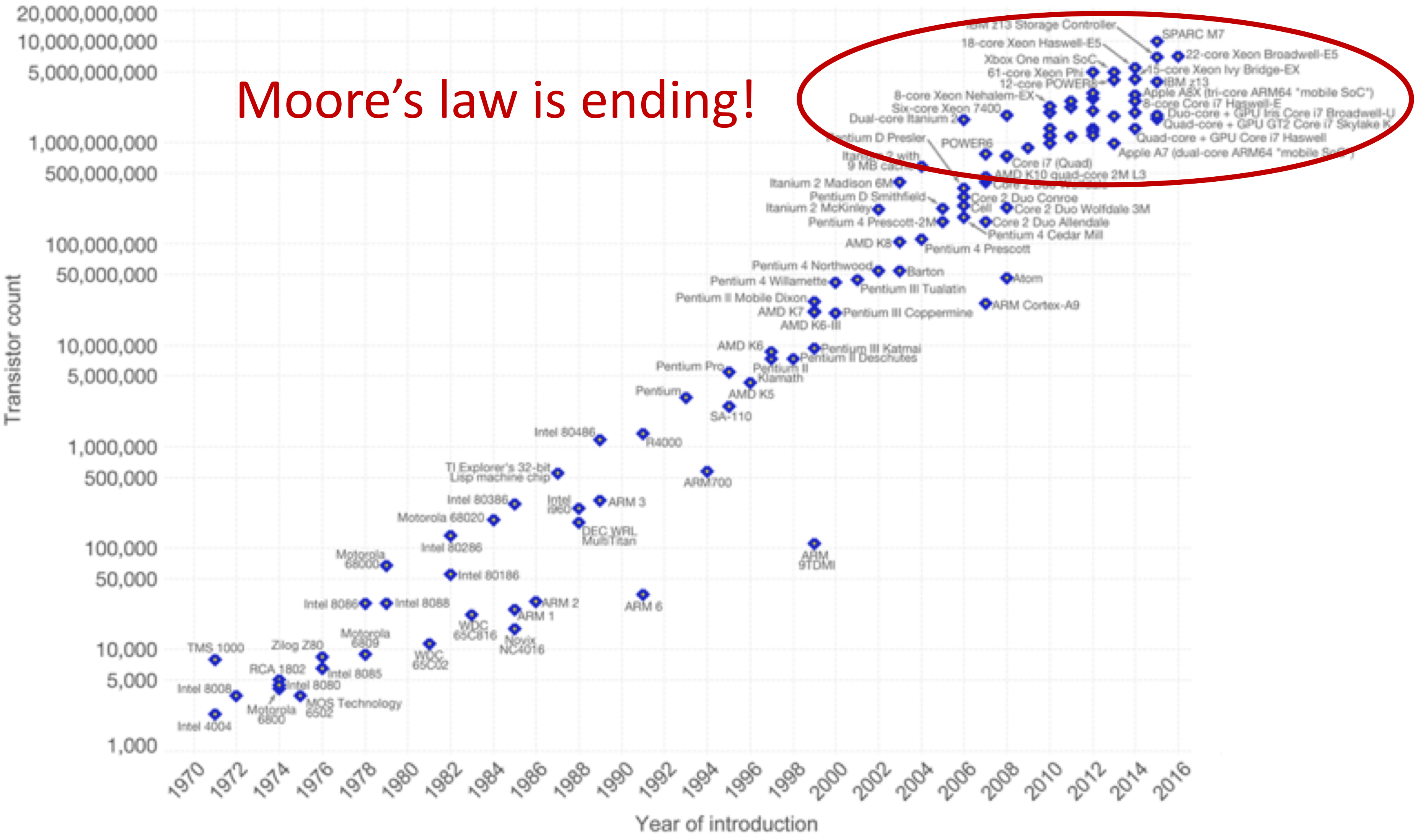
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

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

Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.

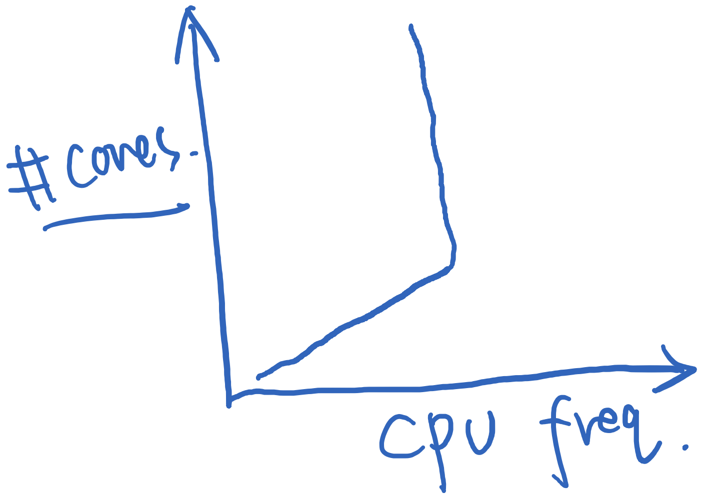


Moore's law is ending!

Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

CPU Trends – What Moore's Law Implies…

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



- Faster programs => concurrent execution

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

Goal

- Write applications that fully utilize many CPUs…

Strategy 1

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

- Pros/cons?

Strategy 1

- Build applications from many communication 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!
  - 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

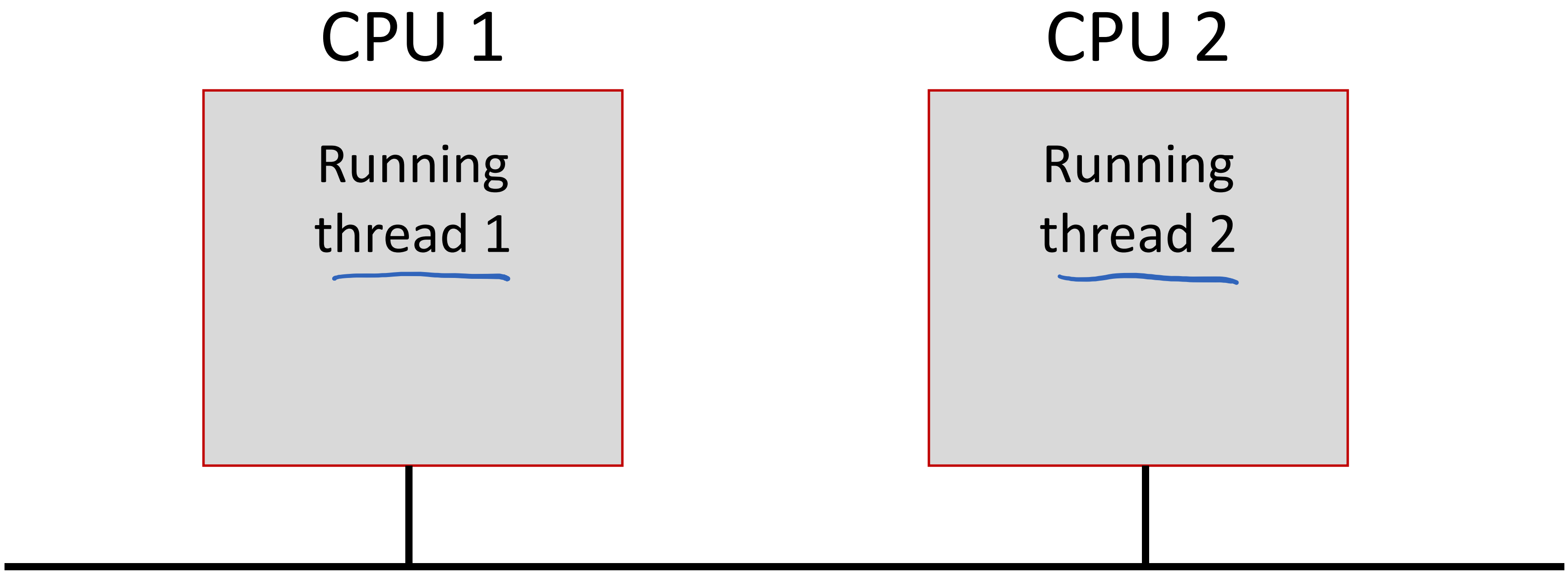
- A process, as defined so far, has only one thread of execution

- 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

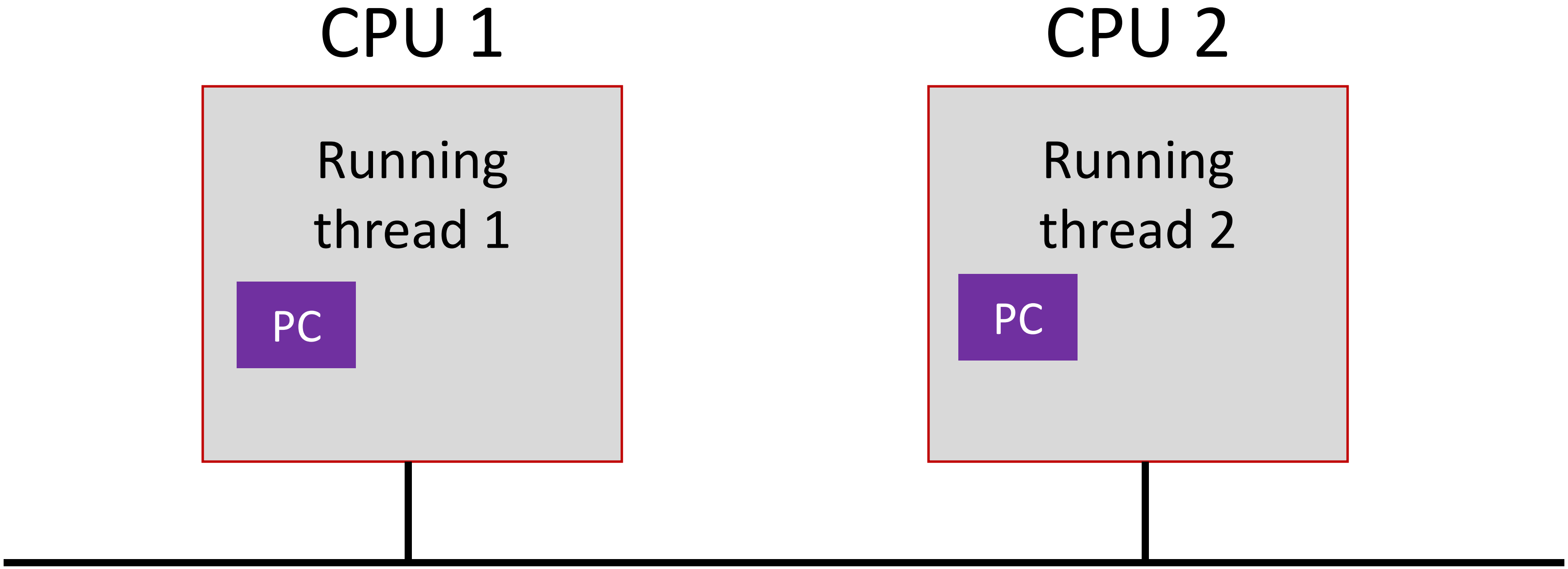


CPU 1

Running thread 1

CPU 2

Running thread 2



CPU 1

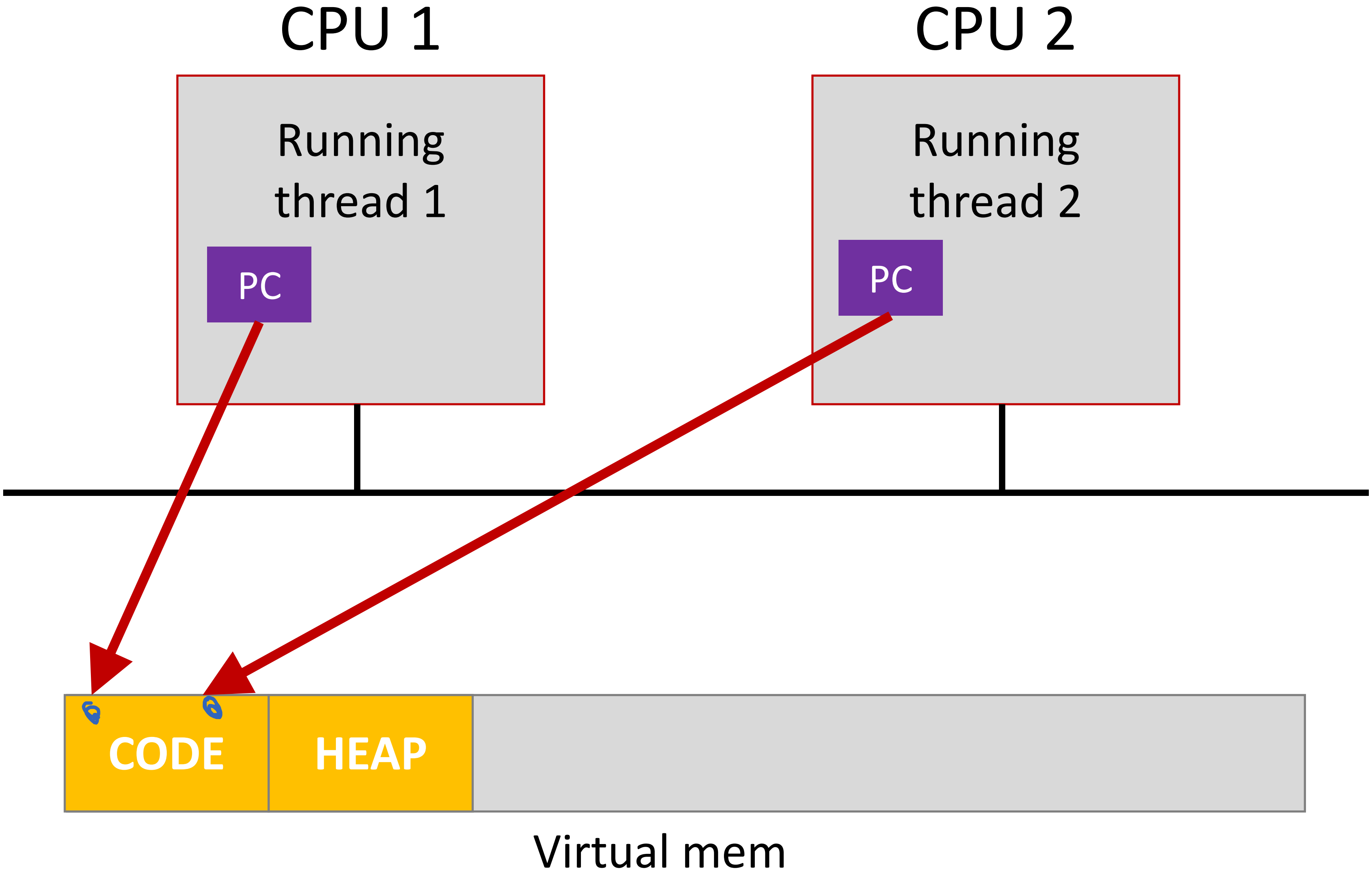
Running thread 1

PC

CPU 2

Running thread 2

PC



CPU 1

Running thread 1

PC

CPU 2

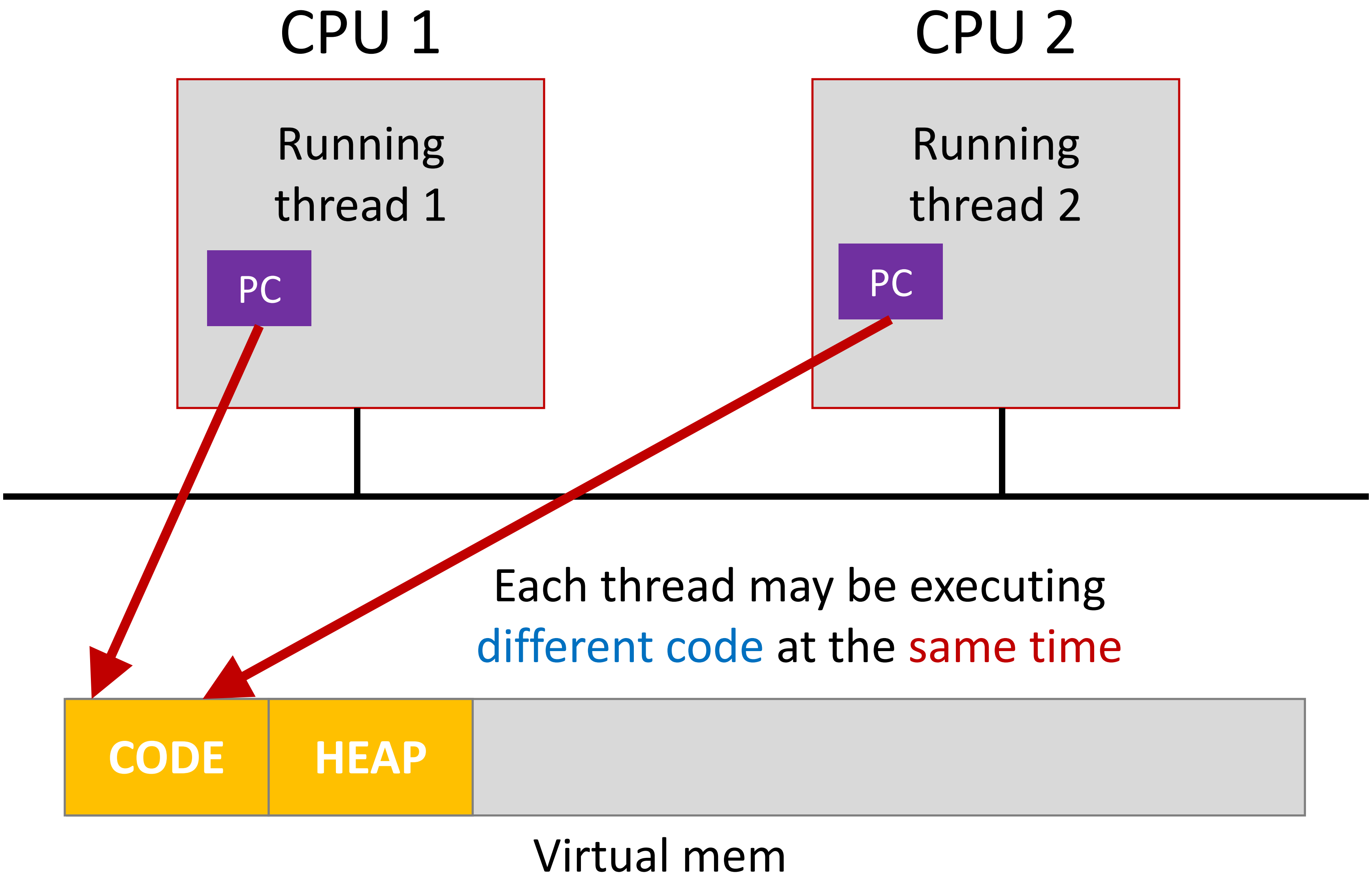
Running thread 2

PC

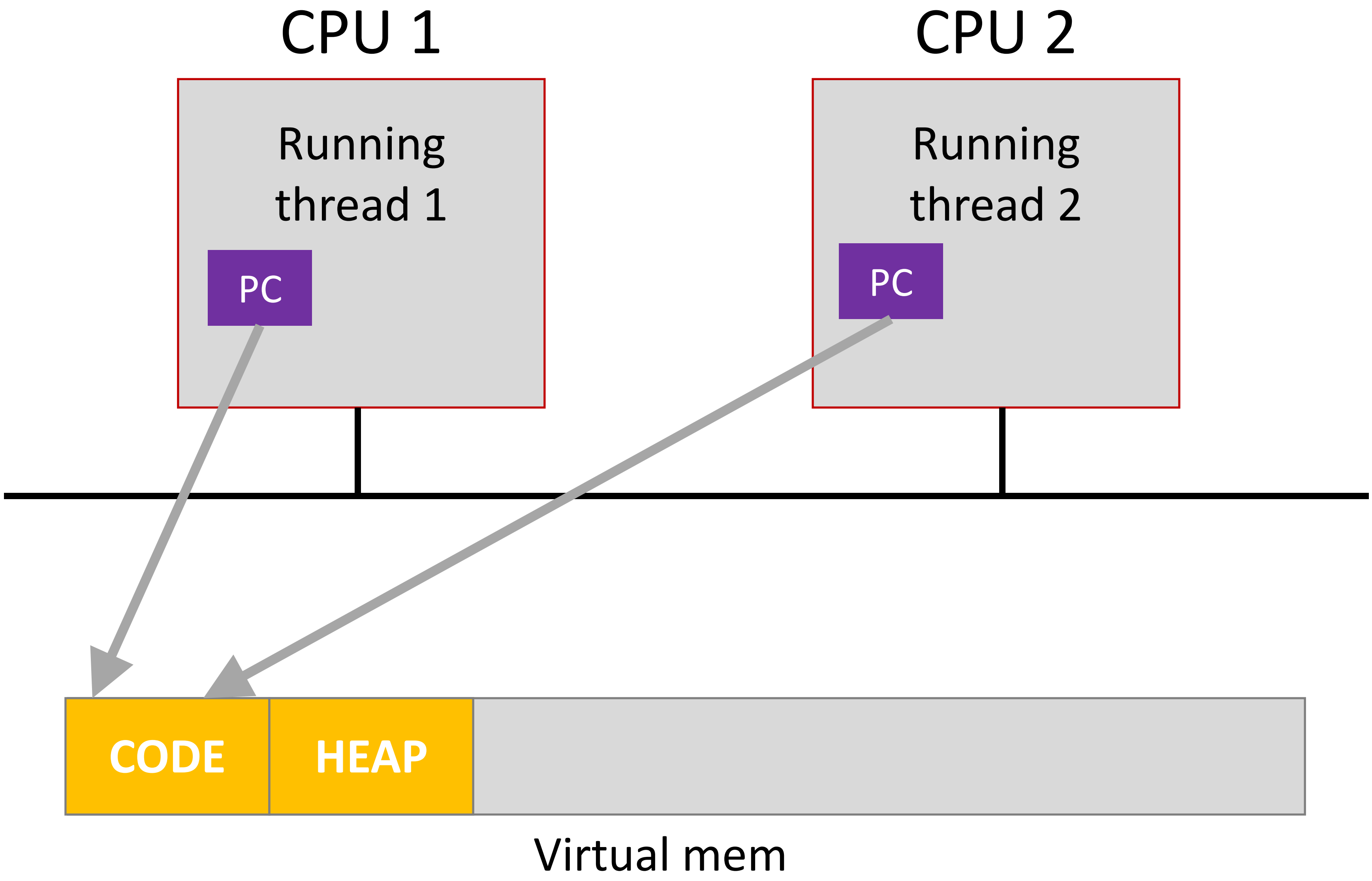
CODE

HEAP

Virtual mem



Each thread may be executing different code at the same time



CPU 1

Running thread 1

PC

CPU 2

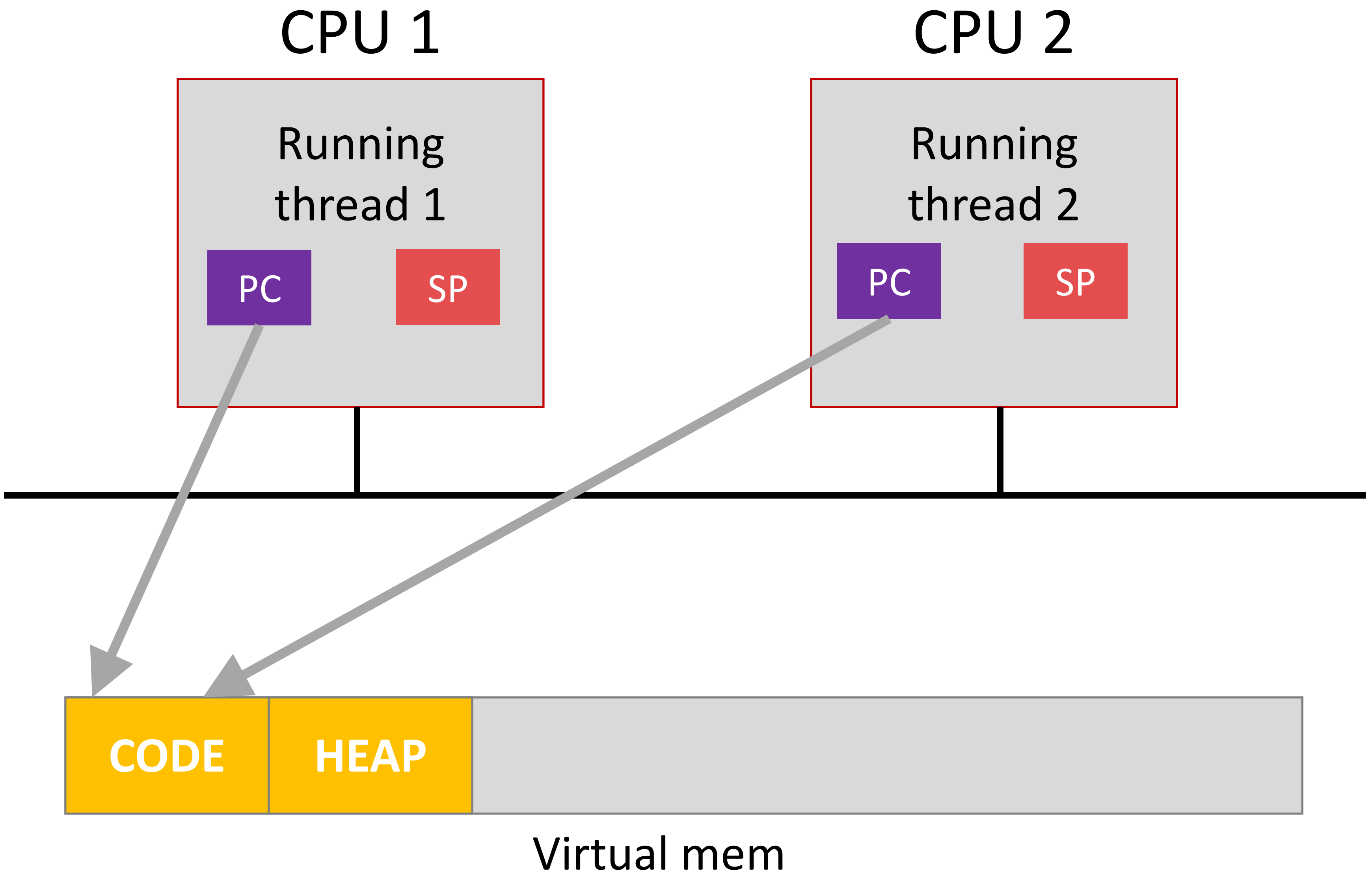
Running thread 2

PC

CODE

HEAP

Virtual mem



CPU 1

Running thread 1

PC

SP

CPU 2

Running thread 2

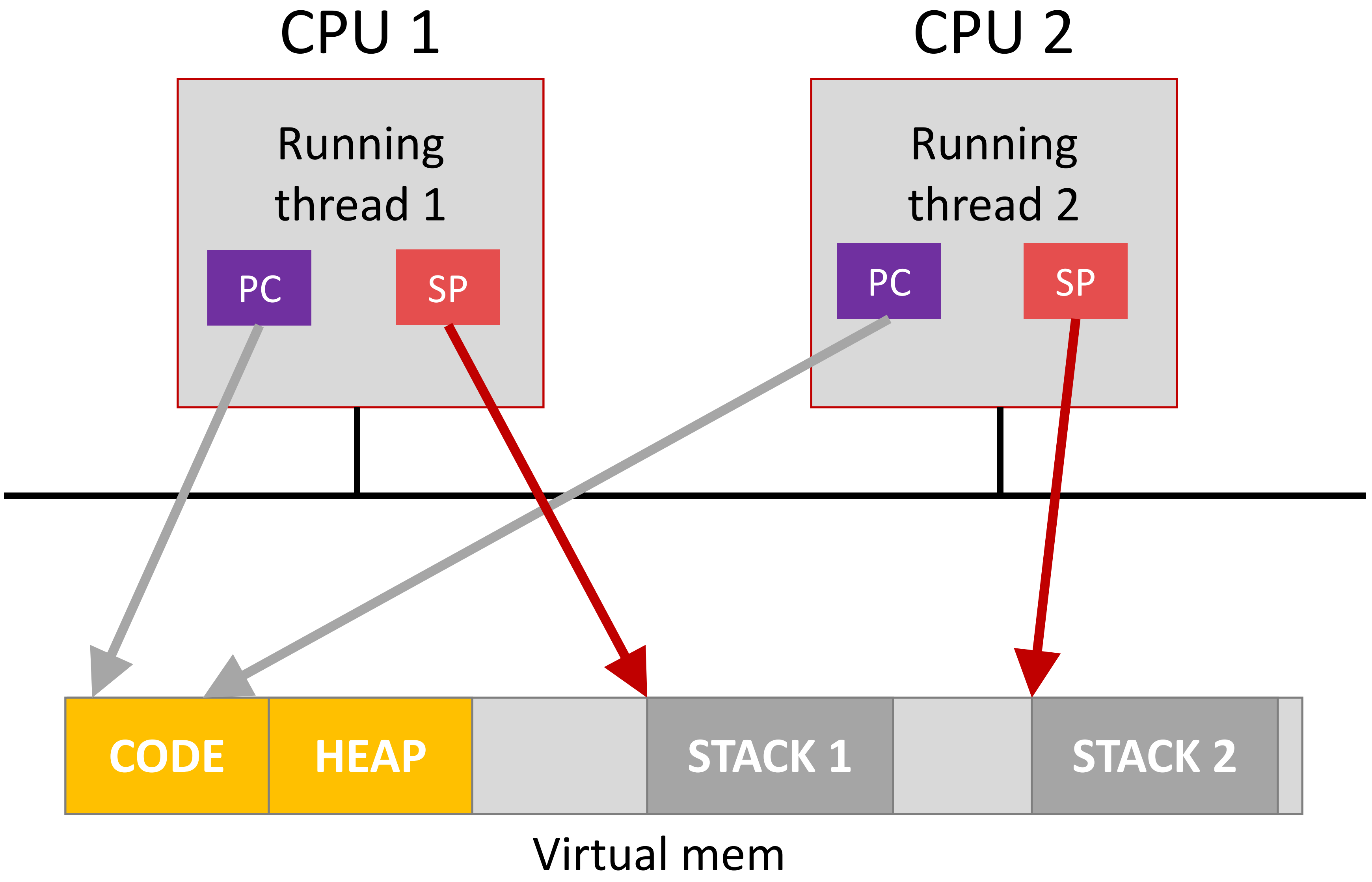
PC

SP

CODE

HEAP

Virtual mem



CPU 1

Running thread 1

PC

SP

CPU 2

Running thread 2

PC

SP

CODE

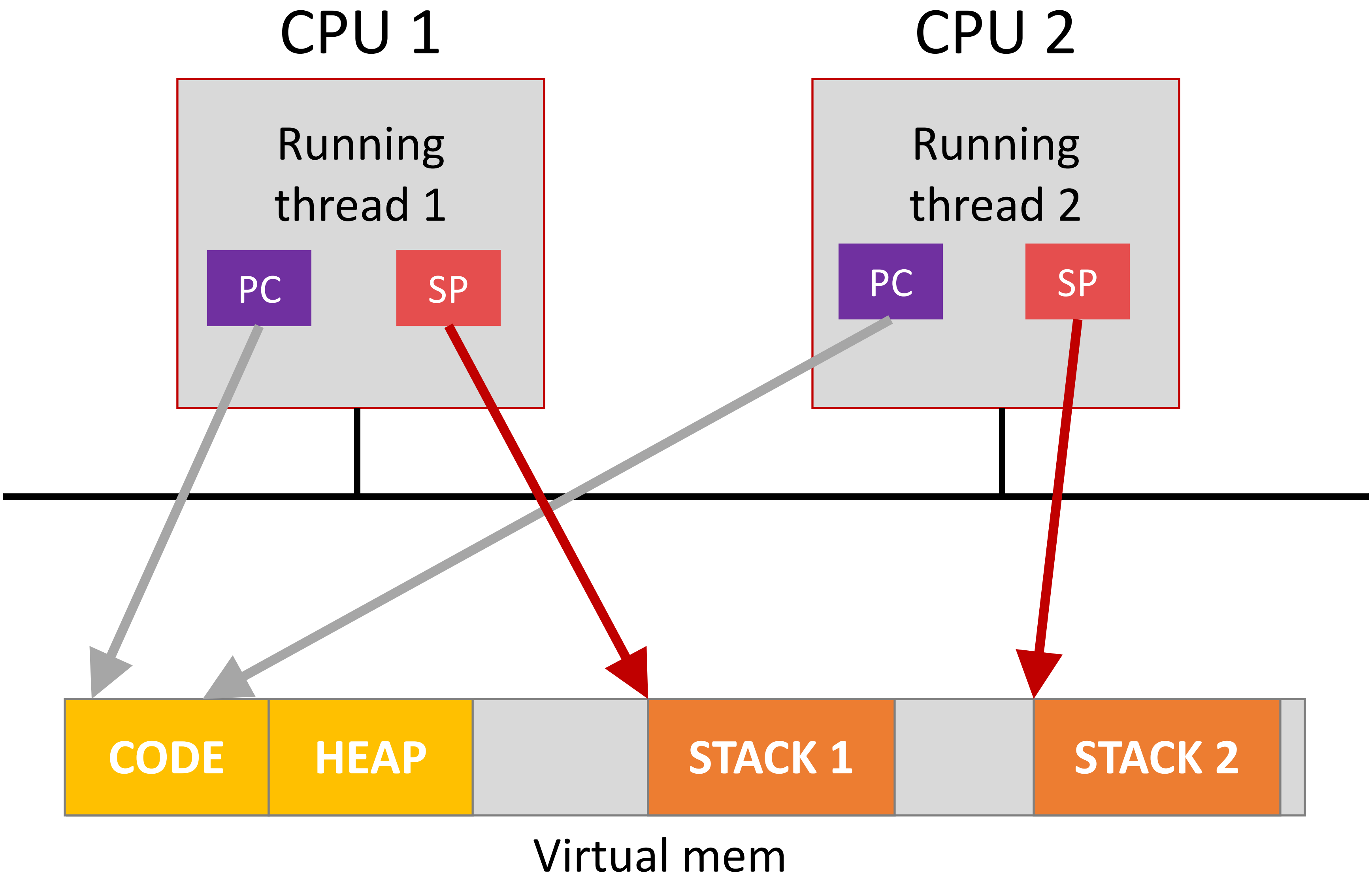
HEAP

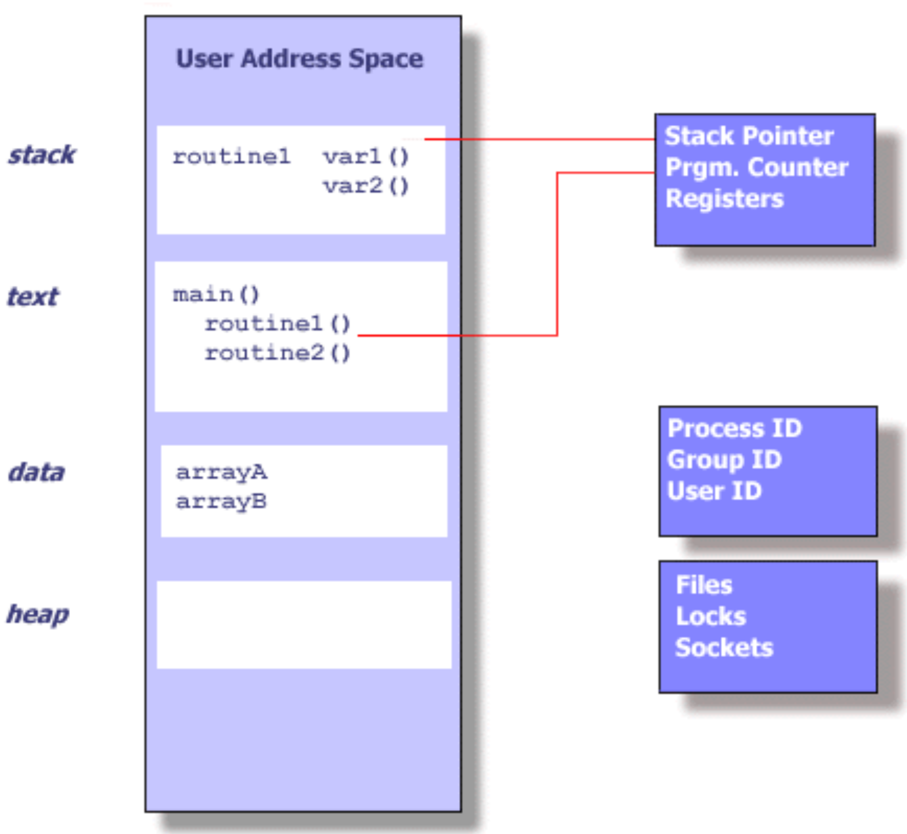
STACK 1

STACK 2

Virtual mem

Thread executing different functions need different stacks





User Address Space

stack

routine1 var1()
var2()

text

main()
routine1()
routine2()

data

arrayA
arrayB

heap

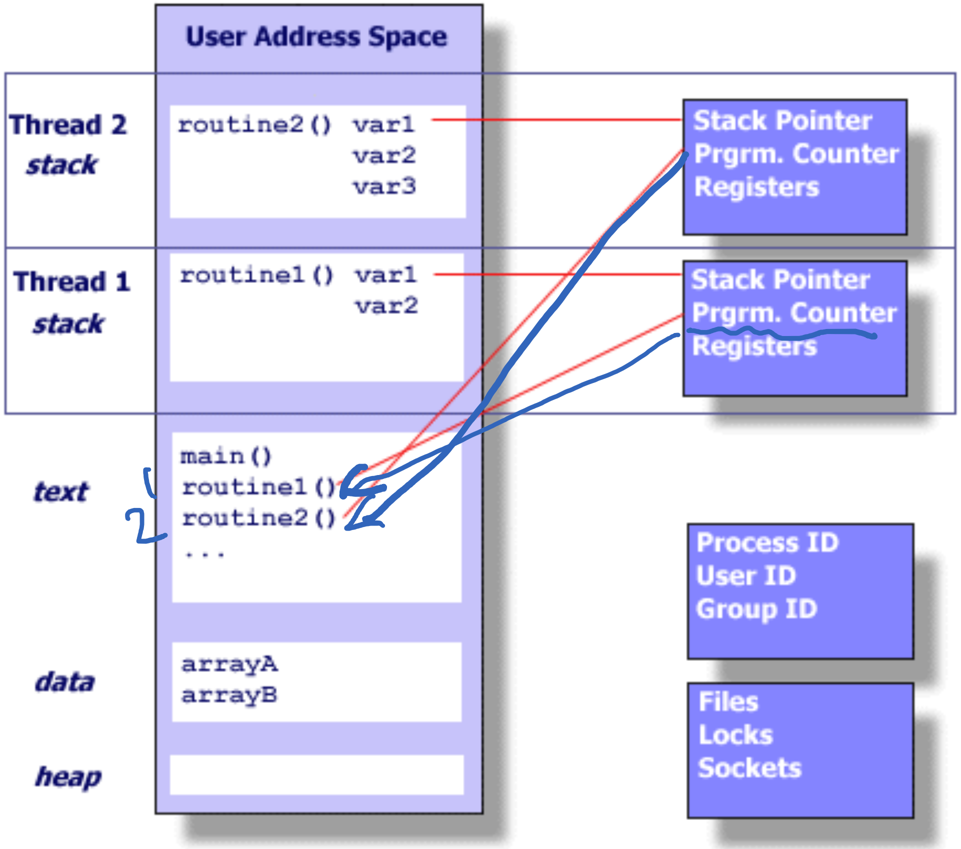
Stack Pointer
Prgm. Counter
Registers

Process ID
Group ID
User ID

Files
Locks
Sockets

Linux process

vs



User Address Space

Thread 2
stack

routine2() var1
var2
var3

Thread 1
stack

routine1() var1
var2

text

main()
1 routine1()
2 routine2()
...

data

arrayA
arrayB

heap

Stack Pointer
Prgm. Counter
Registers

Stack Pointer
Prgm. Counter
Registers

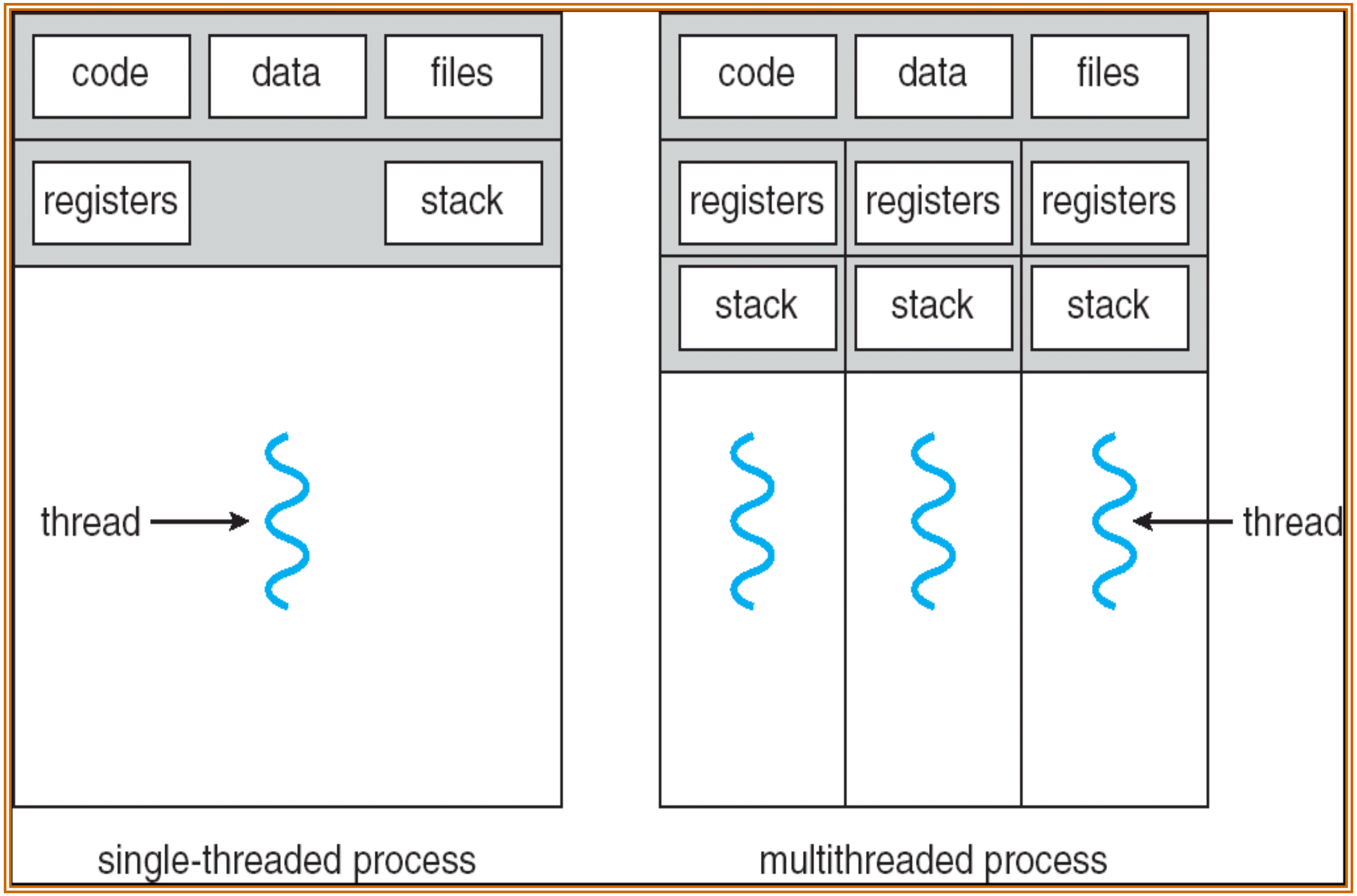
Process ID
User ID
Group ID

Files
Locks
Sockets

Threads within a Linux process

*: https://computing.llnl.gov/tutorials/pthreads/

Single- vs. Multi-threaded Process



single-threaded process

multithreaded process

Using Threads

- Processes usually start with a single thread
- Usually, library procedures are invoked to manage threads
  - thread_create: typically specifies the name of the procedure for the new thread to run
  - thread_exit
  - thread_join: blocks the calling thread until another (specific) thread has exited
  - thread_yield: voluntarily gives up the CPU to let another thread run

Pthread

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX (e.g., Linux) OSes

Pthread APIs

Thread Call	Description
`pthread_create`	Create a new thread in the caller's address space
`pthread_exit`	Terminate the calling thread
`pthread_join`	Wait for a thread to terminate
`pthread_mutex_init`	Create a new mutex
`pthread_mutex_destroy`	Destroy a mutex
`pthread_mutex_lock`	Lock a mutex
`pthread_mutex_unlock`	Unlock a mutex
`pthread_cond_init`	Create a condition variable
`pthread_cond_destroy`	Destroy a condition variable
`pthread_cond_wait`	Wait on a condition variable
`pthread_cond_signal`	Release one thread waiting on a condition variable

Pthread APIs

Thread Call	Description
`pthread_create`	Create a new thread in the caller's address space
`pthread_exit`	Terminate the calling thread
`pthread_join`	Wait for a thread to terminate
`pthread_mutex_init`	Create a new mutex
`pthread_mutex_destroy`	Destroy a mutex
`pthread_mutex_lock`	Lock a mutex
`pthread_mutex_unlock`	Unlock a mutex
`pthread_cond_init`	Create a condition variable
`pthread_cond_destroy`	Destroy a condition variable
`pthread_cond_wait`	Wait on a condition variable
`pthread_cond_signal`	Release one thread waiting on a condition variable

Thread creation

Thread lock

Thread CV

Example of Using Pthread

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

ptr

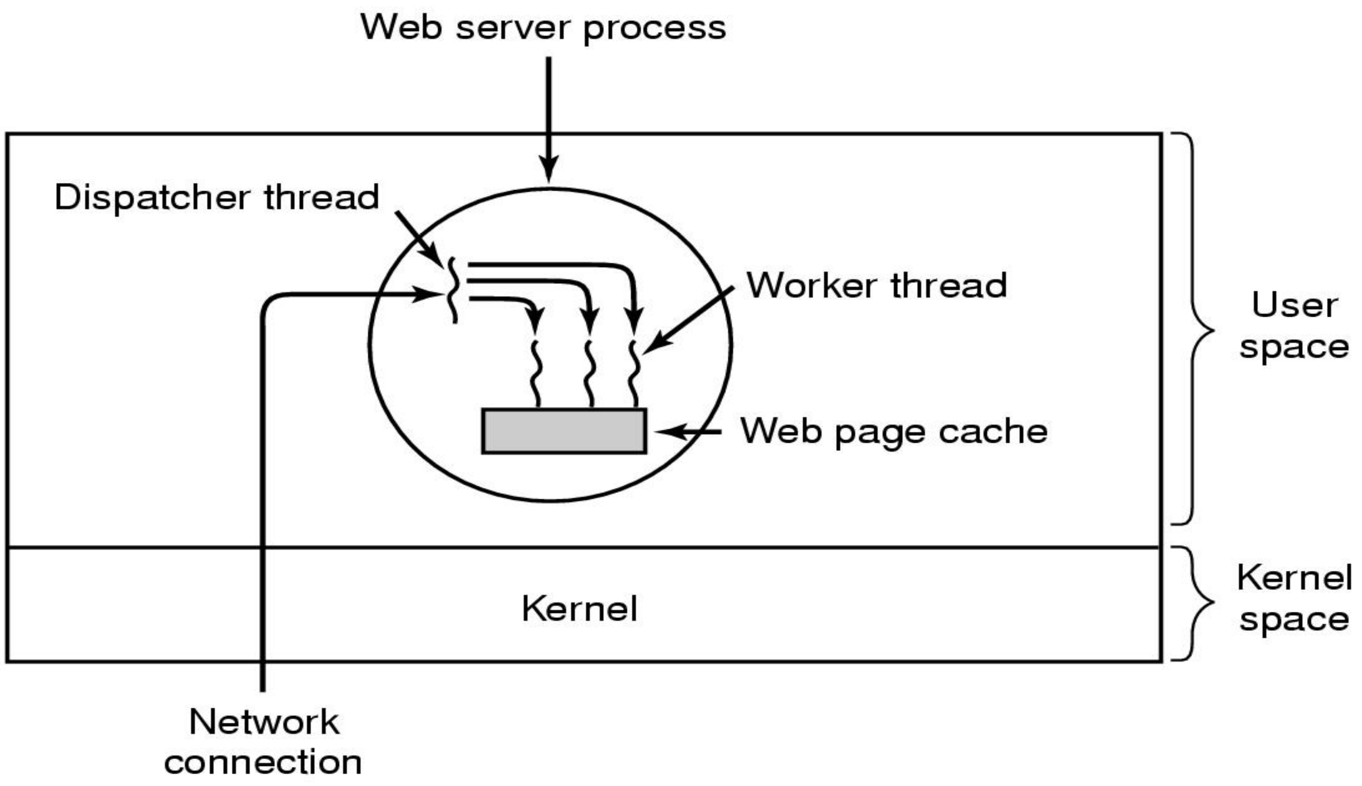
Demo: Basic Threads

- Fork the demo code repo at: https://github.com/tddg/demo-ostep-code

- In today's lecture, we showed the demo in dir: thread-api

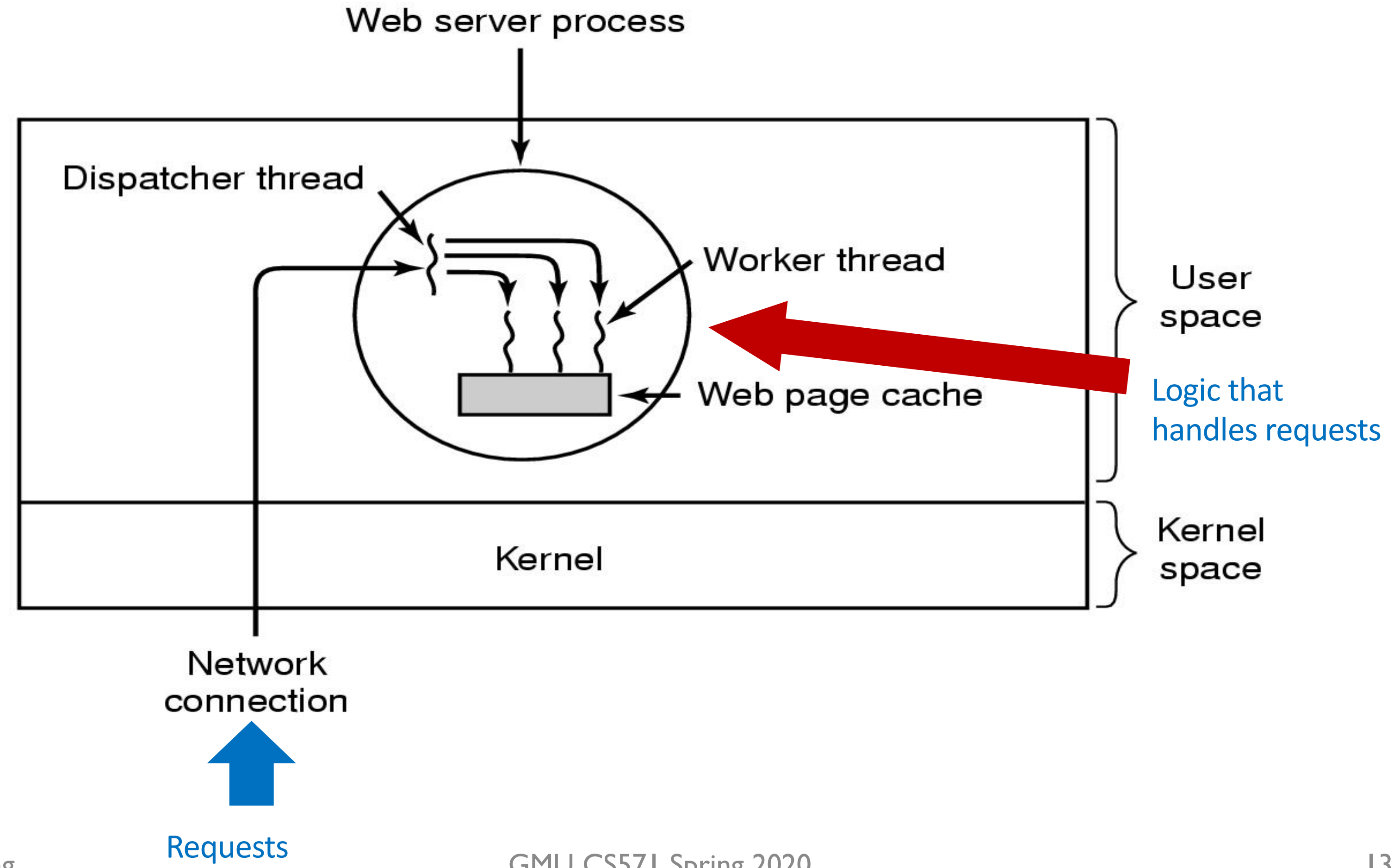
Example Multithreaded Applications

A multithreaded web server



Example Multithreaded Applications

A multithreaded web server



Logic that handles requests

Requests

Code Sketch

```
 while (TRUE) {
   get_next_request(&buf);
  handoff_work(&buf);
}
```

(a) Dispatcher thread

```
   while (TRUE) {
     wait_for_work(&buf);
    check_cache(&buf; &page);
     if (not_in_cache)
read_from_disk(&buf,  &page);
    return_page(&page);
   }
```

(b) Worker thread

Benefits of Multi-threading

- Resource sharing
  - Sharing the address space and other resources may result in high degree of cooperation
- Economy
  - Creating/managing processes much more time consuming than managing threads: e.g., context switch
- Better utilization of multicore architectures
  - Threads are doing job concurrently (in parallel)
  - Multithreading an interactive application may allow a program to continue running even if part of it is blocked or performing a lengthy operation

Real-world Example: Memcached

- Memcached—A high-performance memory-based caching system
  - 14k lines of C source code
  - https://memcached.org/
- A typical multithreaded server implementation
  - `Pthread` + `libevent`
  - A dispatcher thread dispatches newly coming connections to the worker threads in a round-robin manner
  - Event-driven: Each worker thread is responsible for serving requests from the established connections

Multithreading vs. Multi-processes

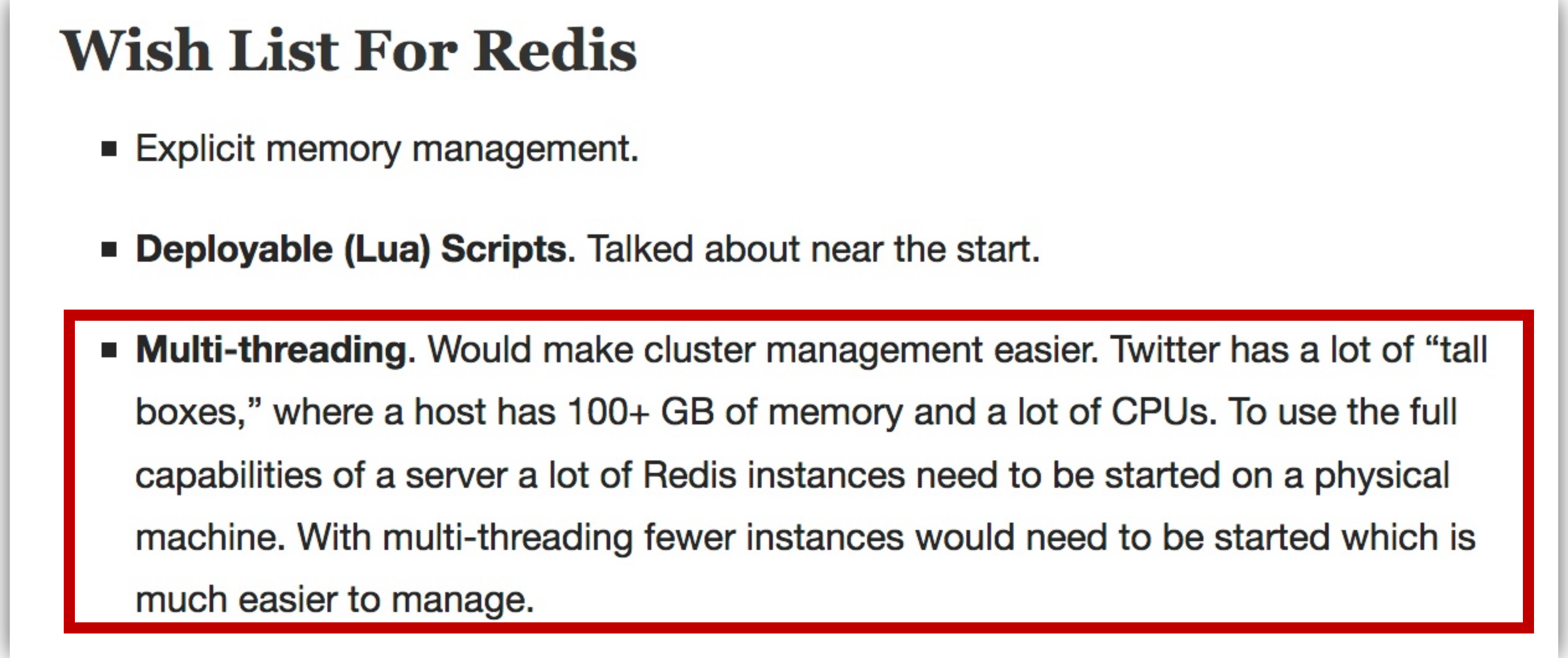
- Real-world debate
  - Multithreading vs. Multi-processes
  - Memcached vs. Redis
- Redis—A single-threaded memory-based data store
  - https://redis.io/

Memcached

redis

Wish List for Redis…

http://goo.gl/N9UTKD



Wish List For Redis

- Explicit memory management.
- **Deployable (Lua) Scripts**. Talked about near the start.
- **Multi-threading**. Would make cluster management easier. Twitter has a lot of “tall boxes,” where a host has 100+ GB of memory and a lot of CPUs. To use the full capabilities of a server a lot of Redis instances need to be started on a physical machine. With multi-threading fewer instances would need to be started which is much easier to manage.