CS 471 Operating Systems

Yue Cheng

George Mason University
Spring 2019

Announcement

- Reminder to complete the Google Form for OS/161 team composition

Intro of OS/161

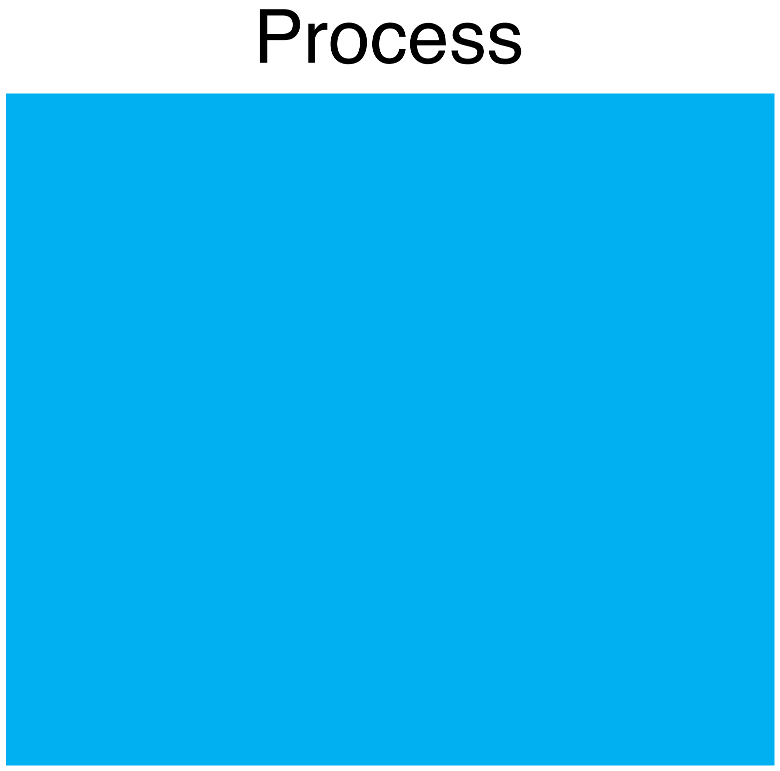
What is a Process?

What is a Process?

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

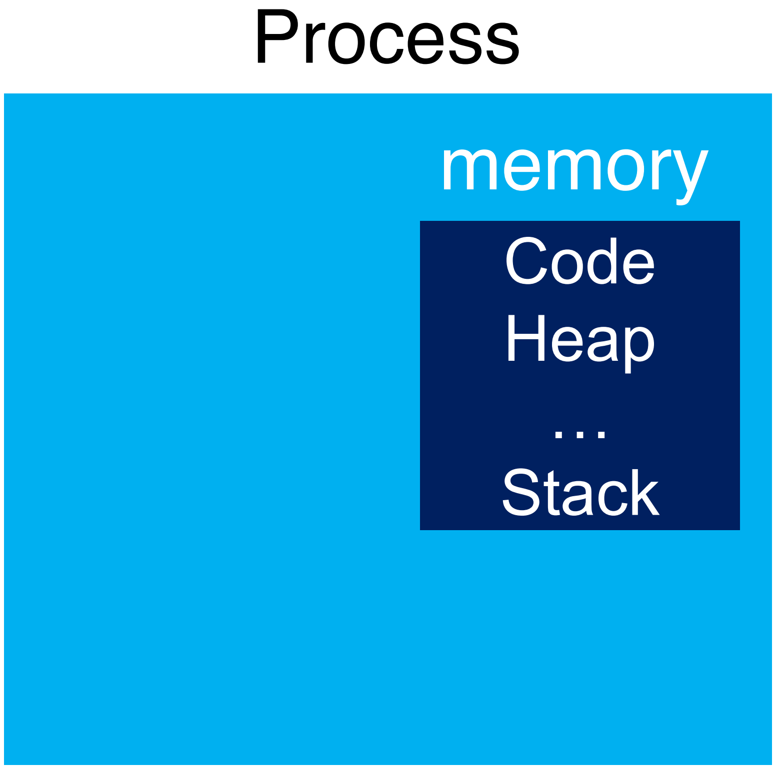
- Java analogy
  - class -> “program”
  - object -> “process”

What is in a Process?



What things change as a program runs?

What is in a Process?



Process

memory

Code

Heap

…

Stack

What things change as a program runs?

What is in a Process?

Process

registers

EAX
PC
SP
BP

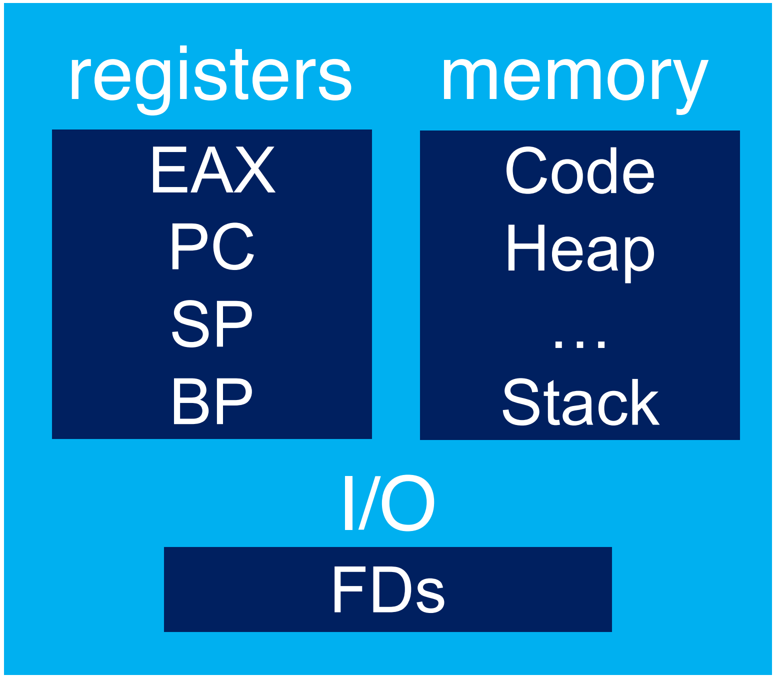
memory

Code
Heap
…
Stack

What things change as a program runs?

What is in a Process?

Process



registers

EAX
PC
SP
BP

memory

Code
Heap
…
Stack

I/O

FDs

What things change as a program runs?

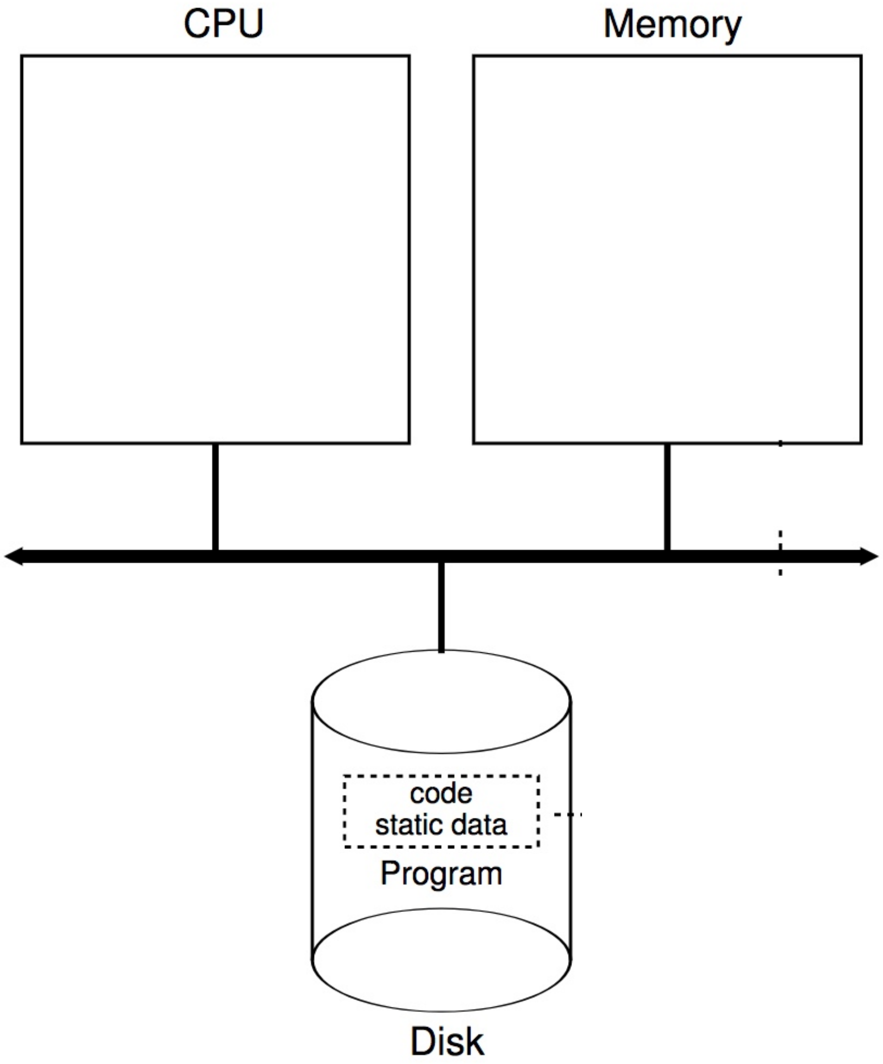
Peeking Inside

- Processes share code, but each has its own "context"
- CPU
  - Instruction pointer (Program Counter)
  - Stack pointer
- Memory
  - Set of memory addresses ("address space")
  - `cat /proc/<PID>/maps`
- Disk
  - Set of file descriptors
  - `cat /proc/<PID>/fdinfo/*`

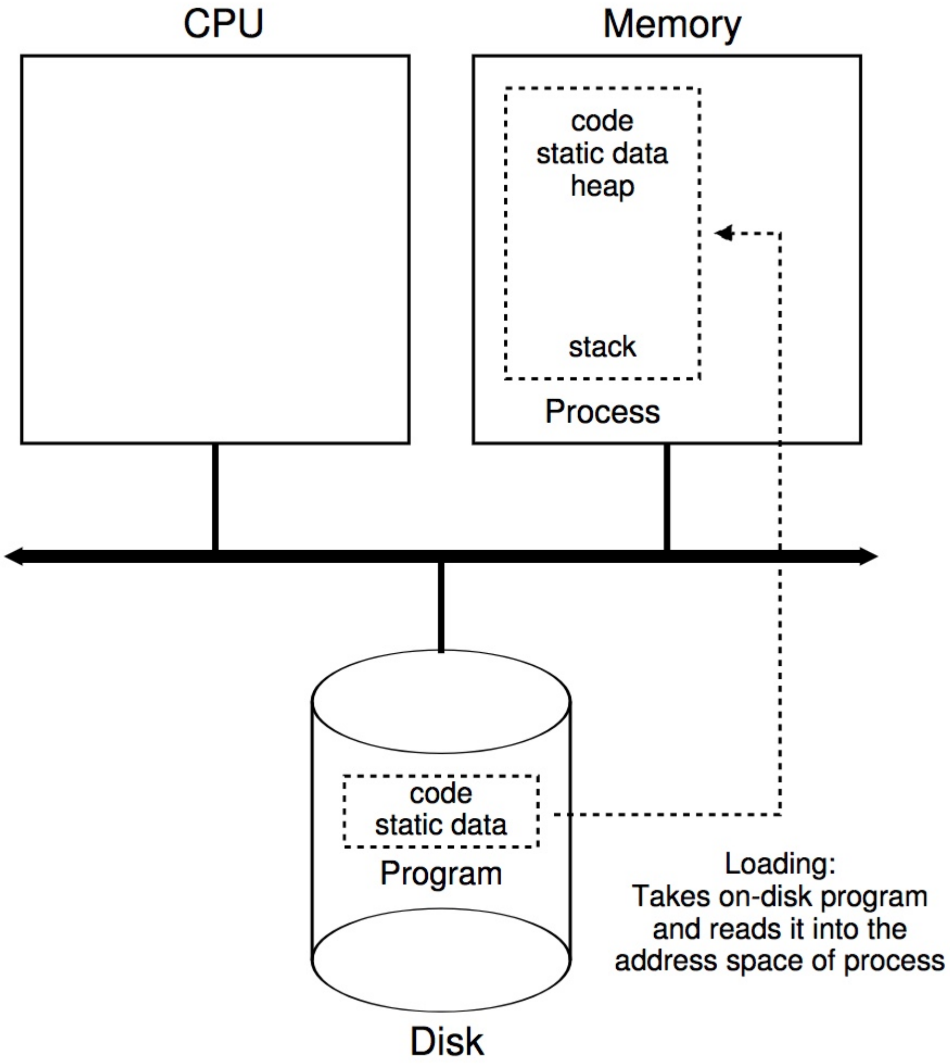
Process Creation

- Principle events that cause process creation
  - System initialization
  - Execution of a process creation system call by a running process
  - User request to create a process

Process Creation

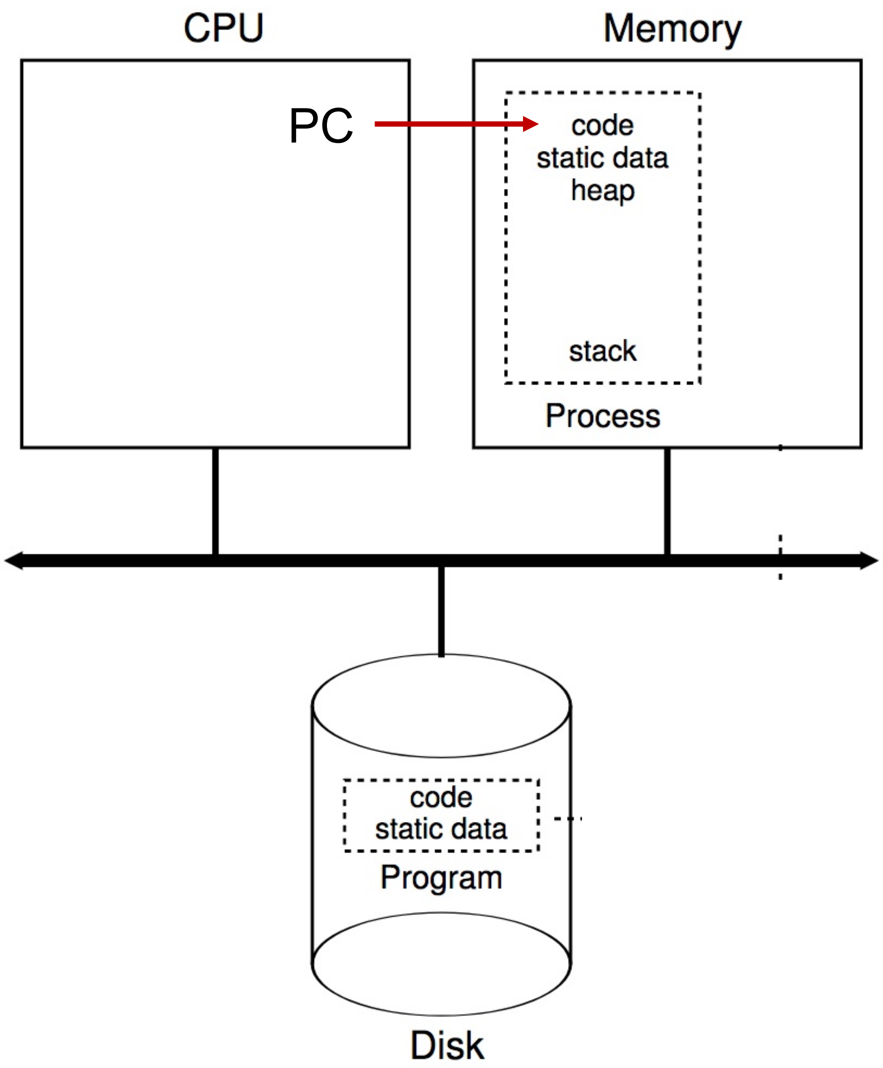


Process Creation



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

Process Creation

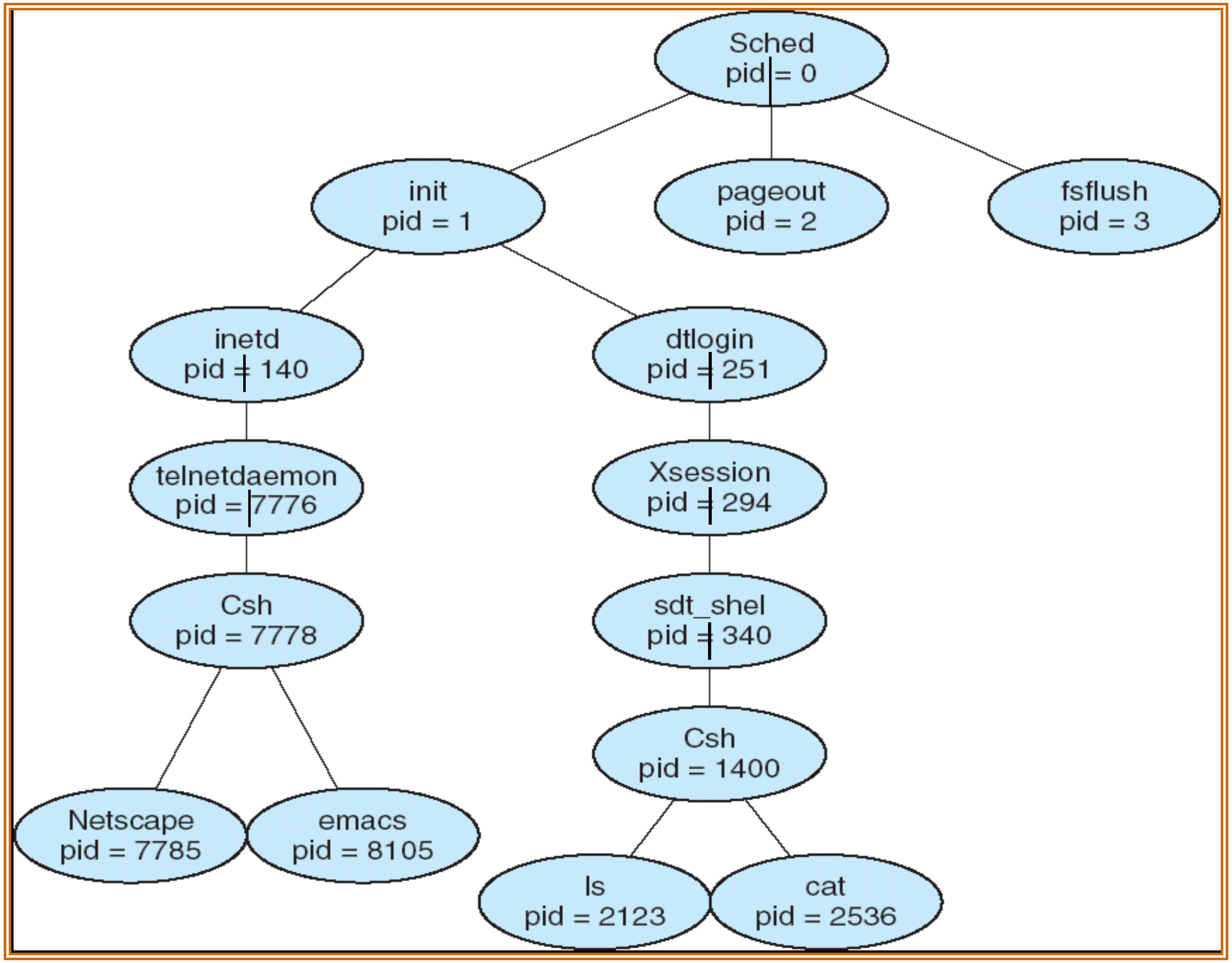


Process Creation (cont.)

- Parent process creates children processes, which, in turn create other processes, forming a tree (**hierarchy**) of processes

- **Questions**:
  - Will the parent and child execute concurrently?
  - How will the address space of the child be related to that of the parent?
  - Will the parent and child share some resources?

An Example Process Tree



How to View Process Tree in Linux?

- `% ps auxf`
  - ‘`f`’ is the option to show the process tree

- `% pstree`

Process Creation in Linux

- Each process has a process identifier (pid)
- The parent executes `fork()` system call to spawn a child
- The child process has a separate copy of the parent's address space
- Both the parent and the child continue execution at the instruction following the `fork()` system call. The return value for the `fork()` system call is
  - **zero** for the new (**child**) process
  - **non-zero** pid for the parent process
- Typically, a process can execute a system call like `execl()` to load a binary file into memory

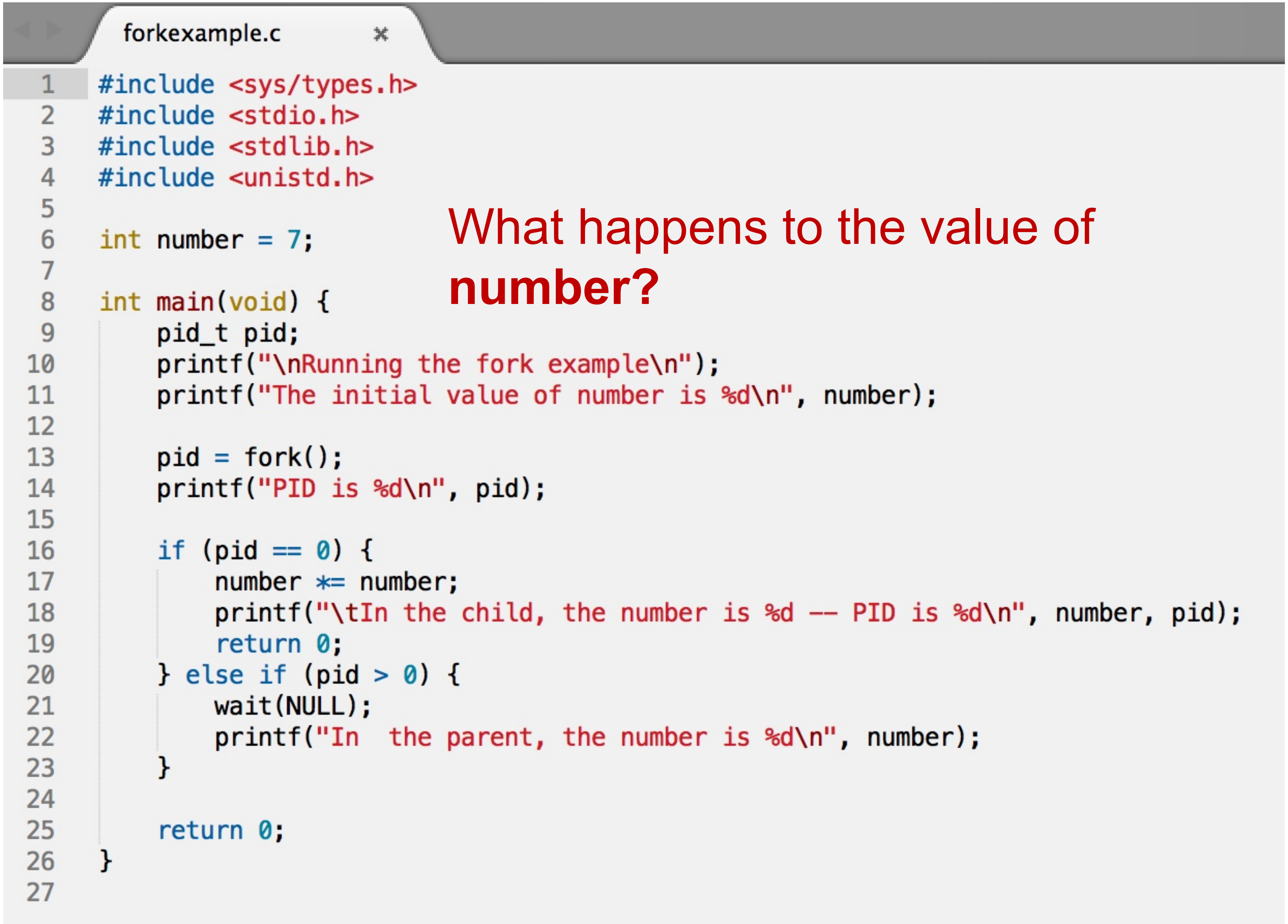
Example Program with “fork”

```
void main () {
    int pid;

    pid = fork();
    if  (pid < 0) {/* error_msg */}
    else if (pid == 0) {  /* child process */
          execl(“/bin/ls”, “ls”, NULL); /* execute ls */
    } else {                      /* parent process */
          /* parent will wait for the child to complete */
          wait(NULL);
          exit(0);
    }
    return;
}
```

A Very Simple Shell using “fork”

```
while (1) {
        type_prompt();
        read_command(cmd);
        pid = fork();
         if  (pid < 0) {/* error_msg */}
         else if (pid == 0) { /* child process */
             execute_command(cmd);
         } else {               /* parent process */
              wait(NULL);
         }
}
```

Example: fork 1



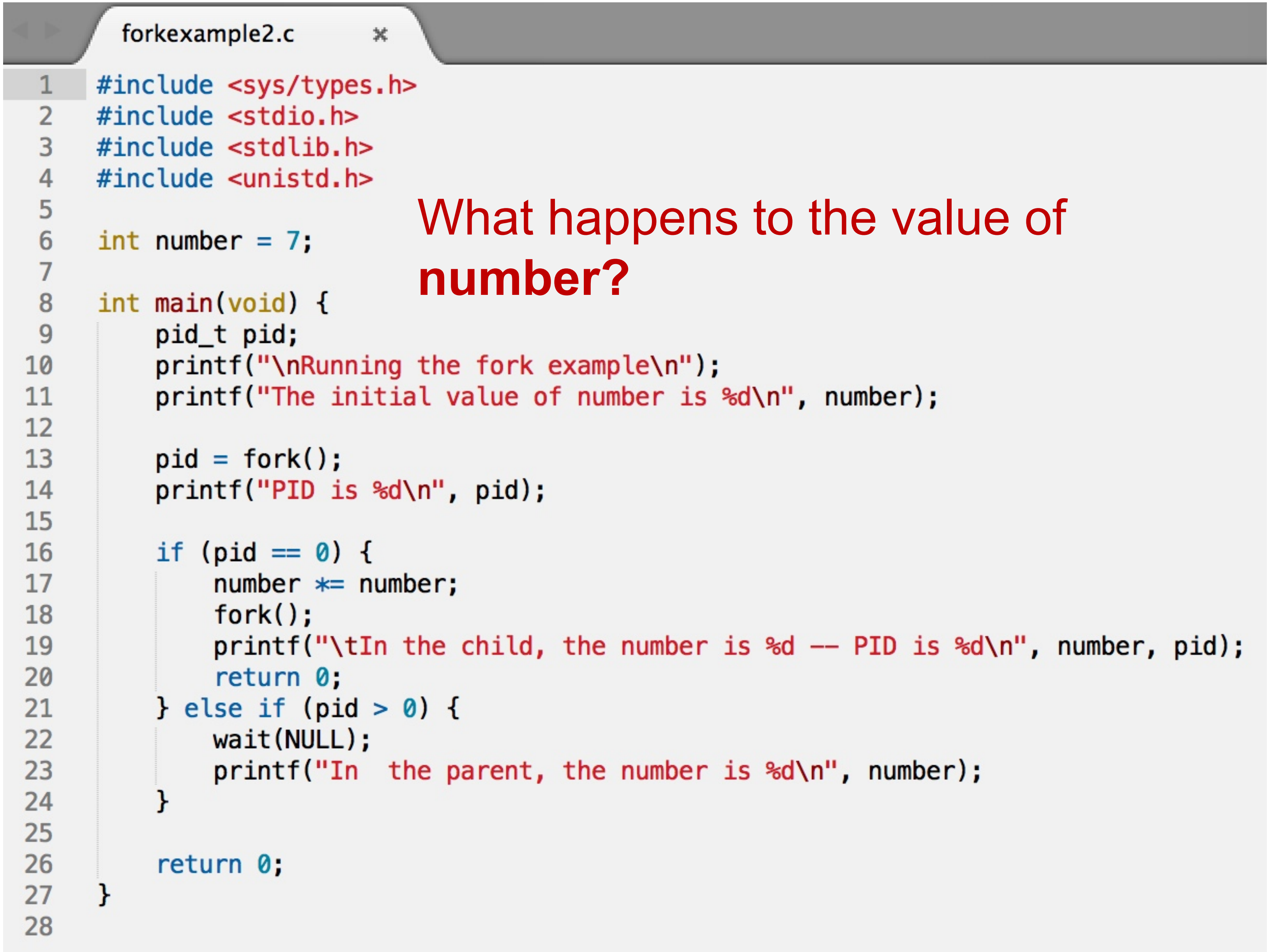
```c
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int number = 7;

int main(void) {
    pid_t pid;
    printf("\nRunning the fork example\n");
    printf("The initial value of number is %d\n", number);

    pid = fork();
    printf("PID is %d\n", pid);

    if (pid == 0) {
        number *= number;
        printf("\tIn the child, the number is %d -- PID is %d\n", number, pid);
        return 0;
    } else if (pid > 0) {
        wait(NULL);
        printf("In  the parent, the number is %d\n", number);
    }

    return 0;
}

```

What happens to the value of **number?**

Results

```
./forkexample1

Running the fork example
The initial value of number is 7
 PID is 2137
 PID is 0
         In the child, the number is 49  PID is 0
In the parent, the number is 7
```

Example: fork 2



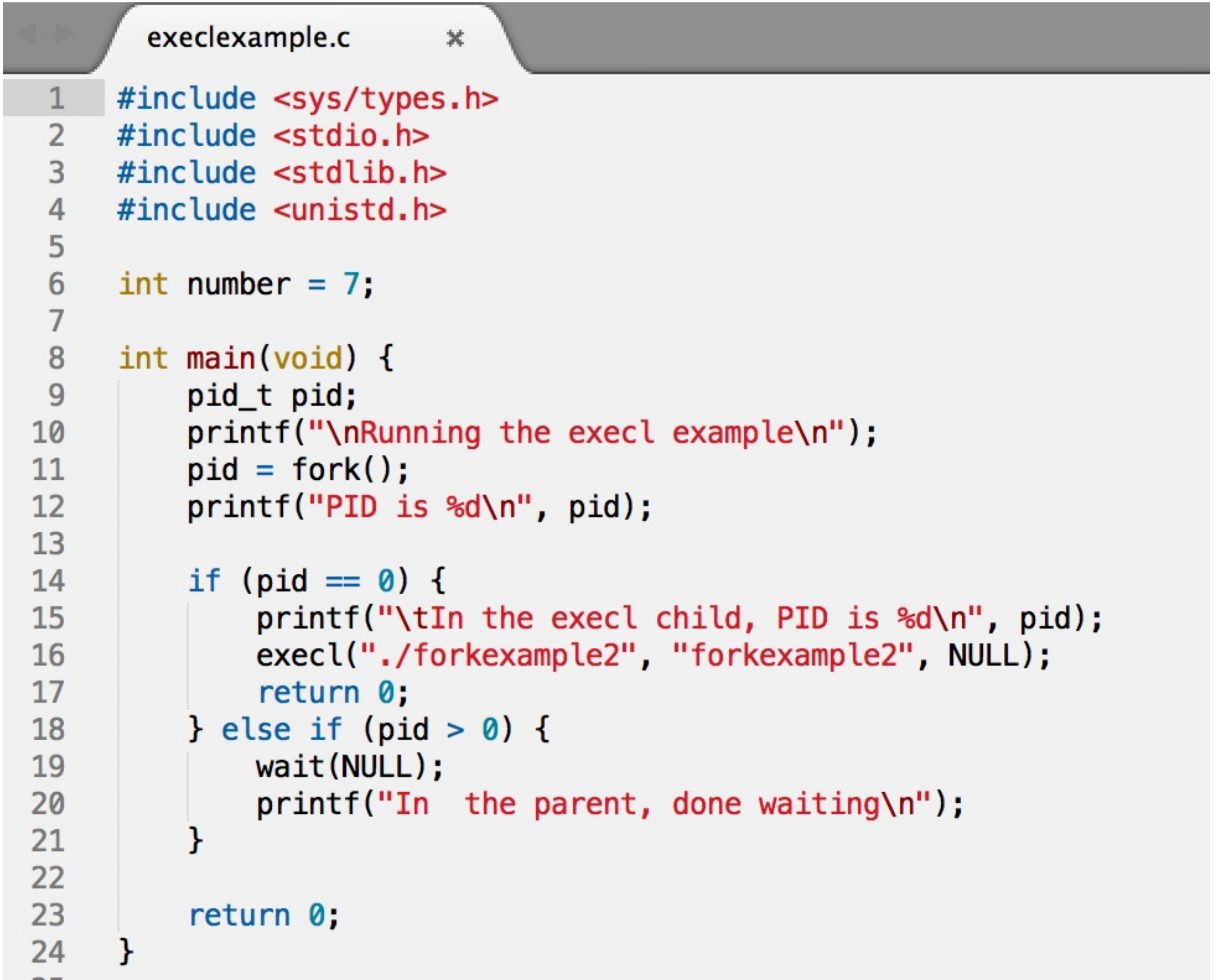
```c
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int number = 7;

int main(void) {
    pid_t pid;
    printf("\nRunning the fork example\n");
    printf("The initial value of number is %d\n", number);

    pid = fork();
    printf("PID is %d\n", pid);

    if (pid == 0) {
        number *= number;
        fork();
        printf("\tIn the child, the number is %d -- PID is %d\n", number, pid);
        return 0;
    } else if (pid > 0) {
        wait(NULL);
        printf("In  the parent, the number is %d\n", number);
    }

    return 0;
}

```

What happens to the value of **number?**

Results

```
./forkexample2

Running the fork example
The initial value of number is 7
 PID is 2164
 PID is 0
         In the child, the number is 49  PID is 0
         In the child, the number is 49  PID is 0
In the parent, the number is 7
```

execl vs. fork



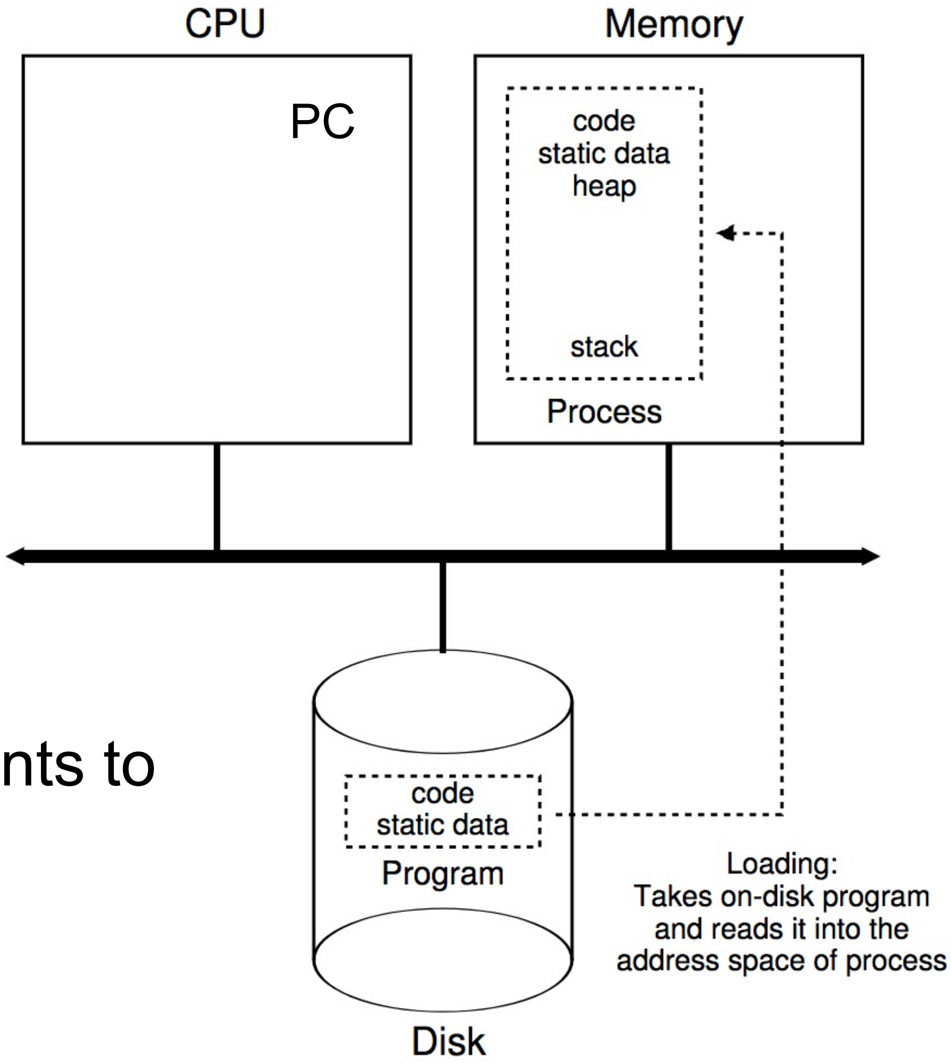
```c
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int number = 7;

int main(void) {
    pid_t pid;
    printf("\nRunning the execl example\n");
    pid = fork();
    printf("PID is %d\n", pid);

    if (pid == 0) {
        printf("\tIn the execl child, PID is %d\n", pid);
        execl("./forkexample2", "forkexample2", NULL);
        return 0;
    } else if (pid > 0) {
        wait(NULL);
        printf("In  the parent, done waiting\n");
    }

    return 0;
}
```

Results

```
./execlexample
Running execl code
 PID is 2179
 PID is 0

                In the execl child,   PID is 0

Running the fork example
The initial value of number is 7
 PID is 2180
 PID is 0

                In the child, the number is 49  PID is 0
                In the child, the number is 49  PID is 0
In the parent, the number is 7
In the parent, done waiting
```

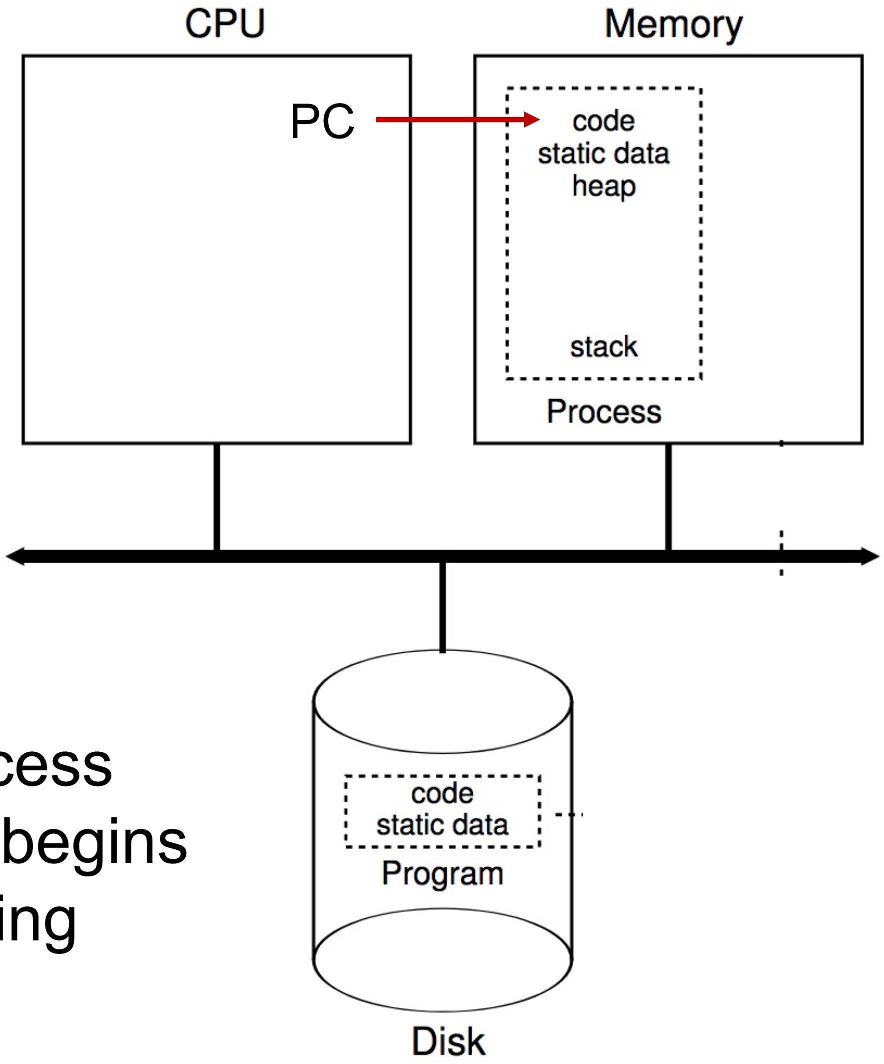
forkexample2

Process Creation



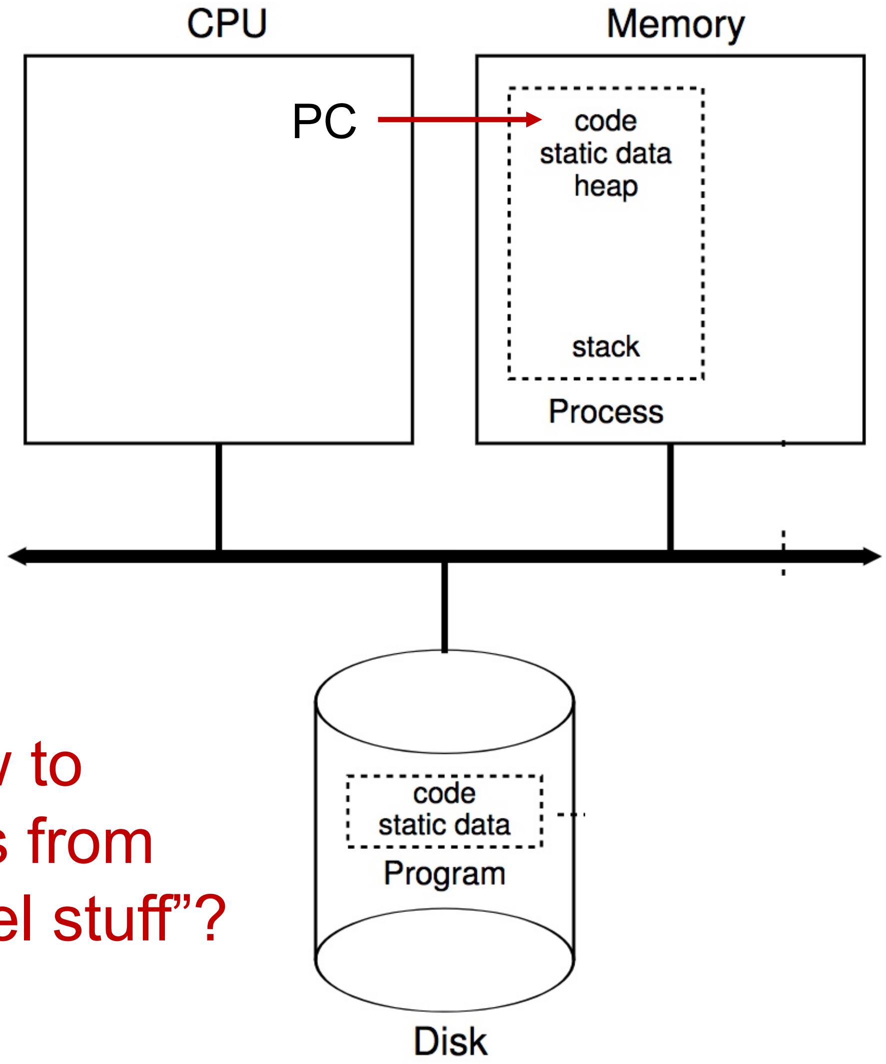
Before, PC points to kernel code

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

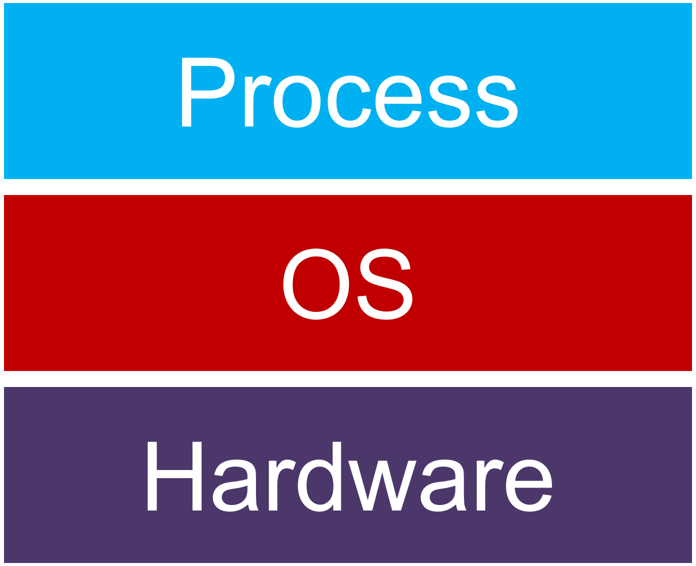
LDE: Remaining Challenges

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

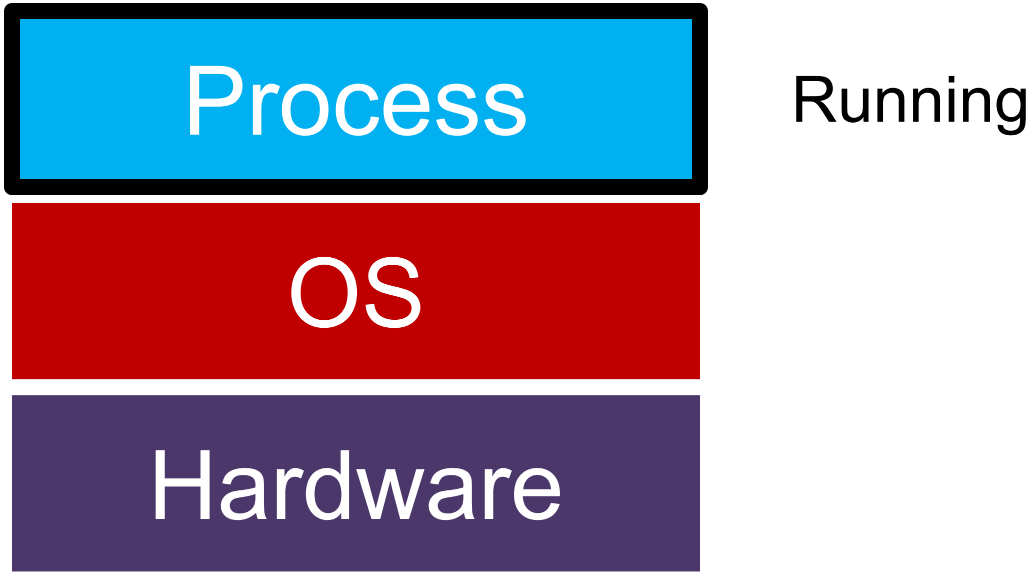


Process

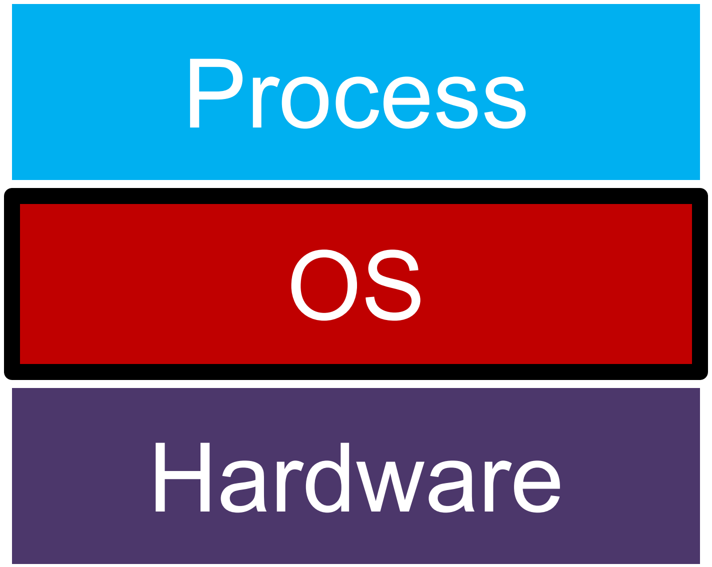
OS

Hardware

Taking Turns



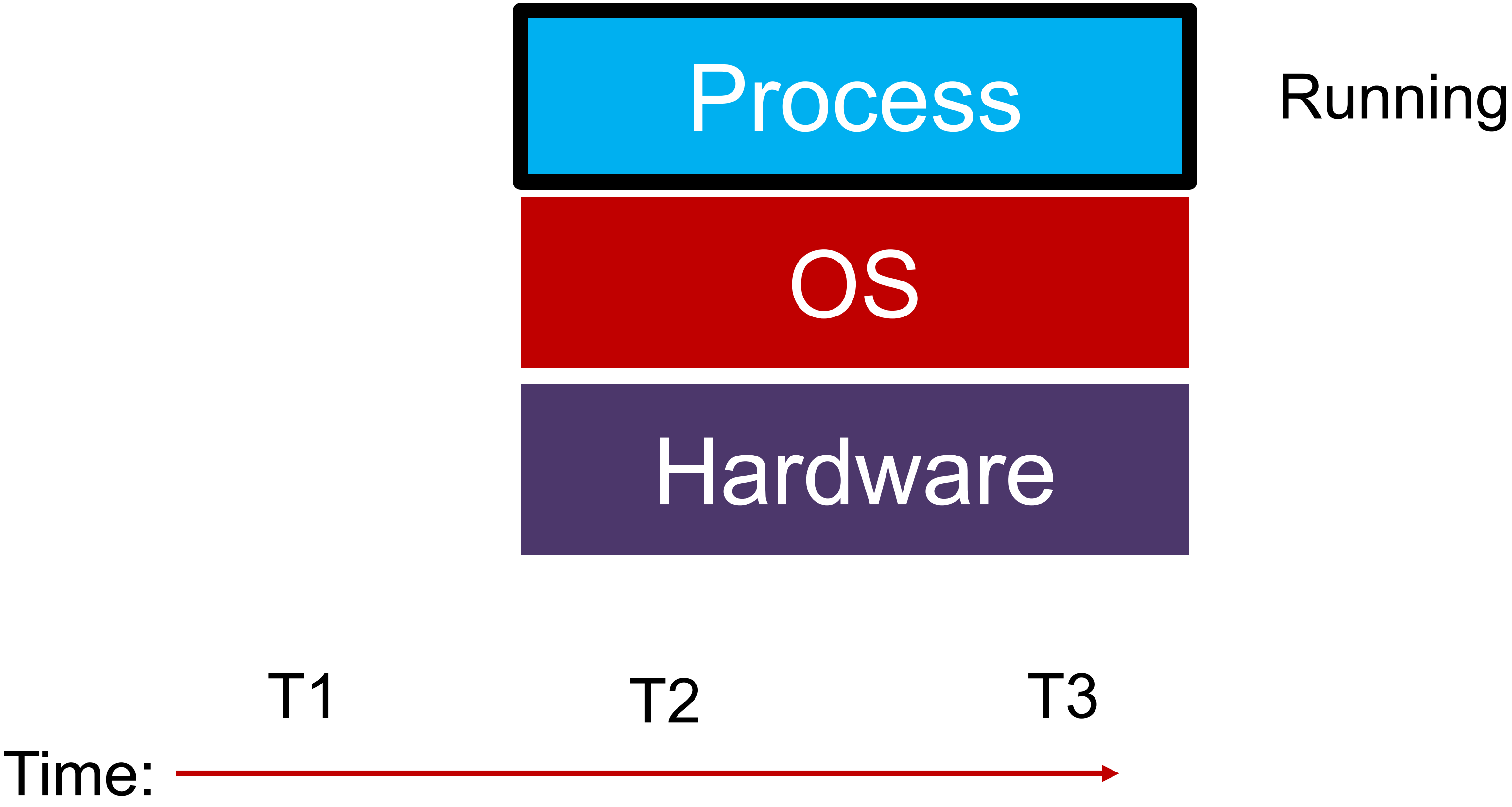
Taking Turns



Running

Time:

Taking Turns



Process

OS

Hardware

Running

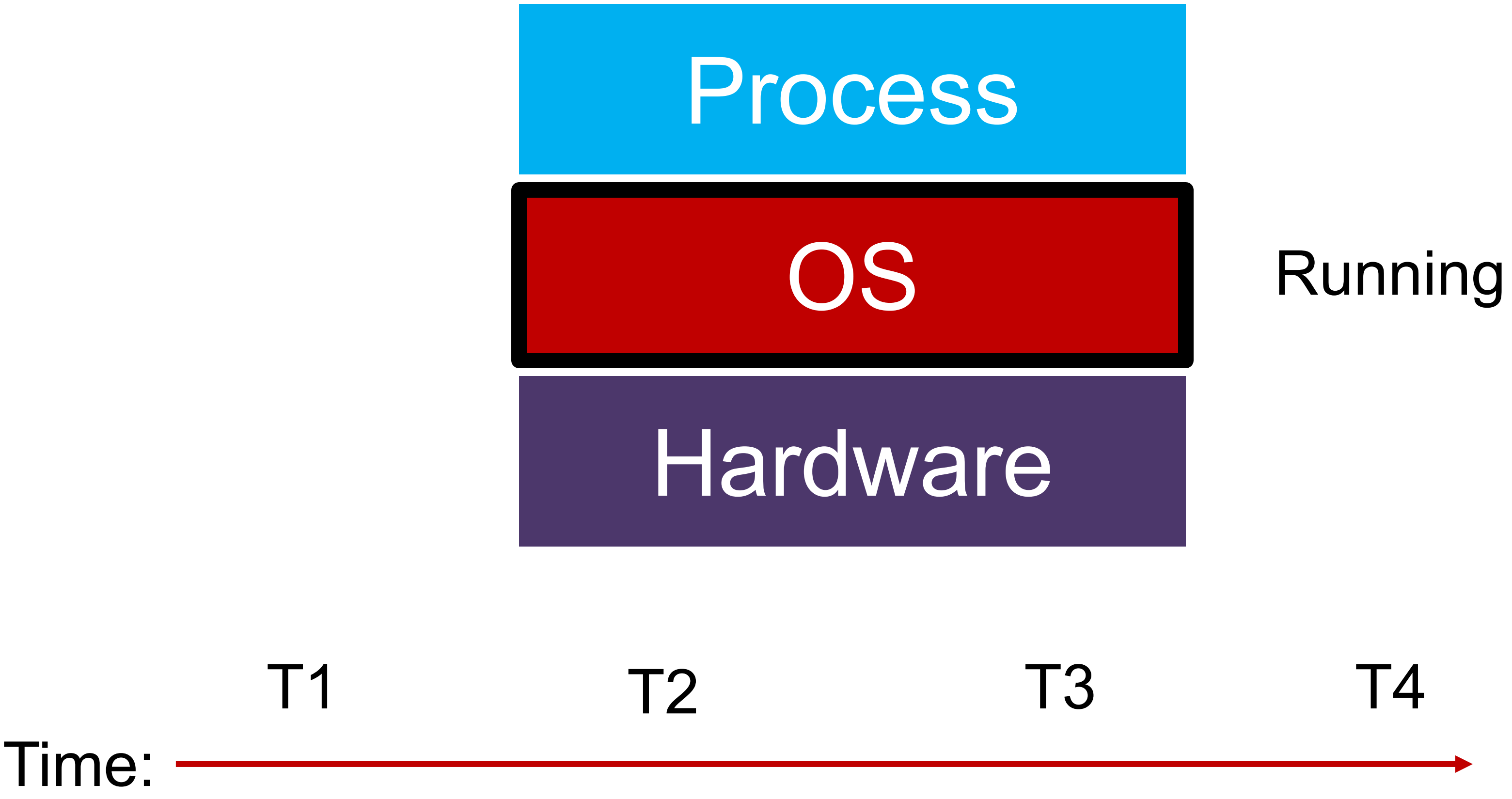
T1

T2

T3

Time:

Taking Turns



Process

OS

Hardware

Running

T1

T2

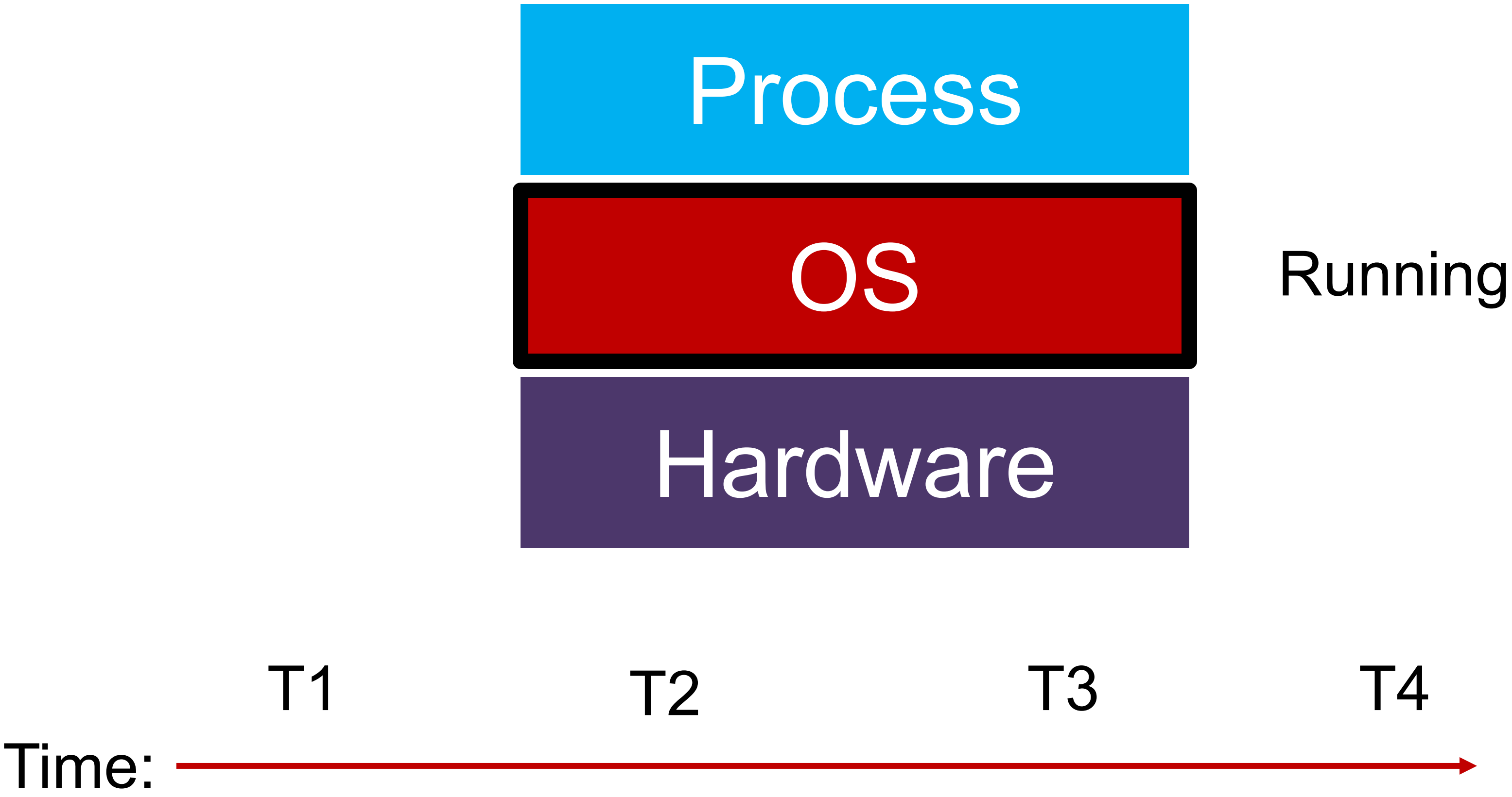
T3

T4

Time:

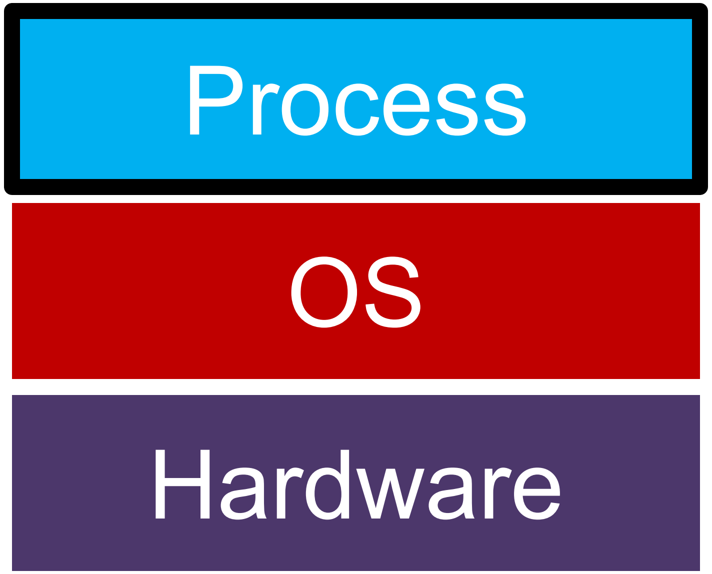
Taking Turns

Question: when/how do we switch to OS?



Exceptions

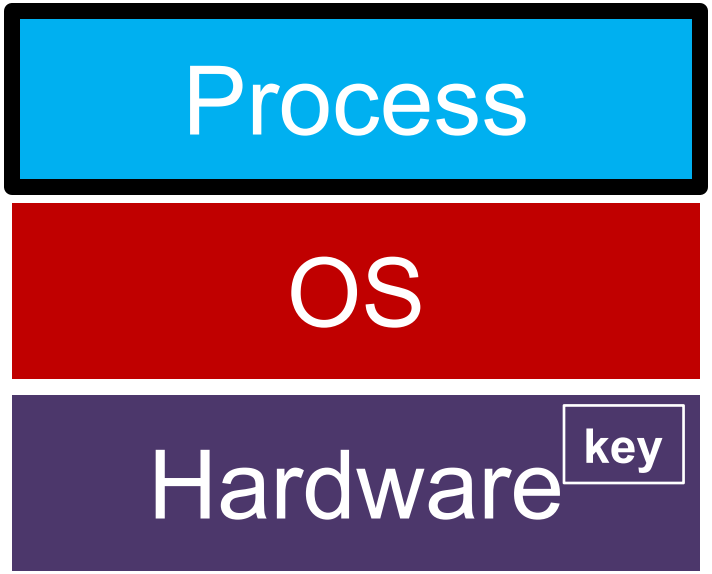
Interrupt


Process

OS

Hardware

Interrupt

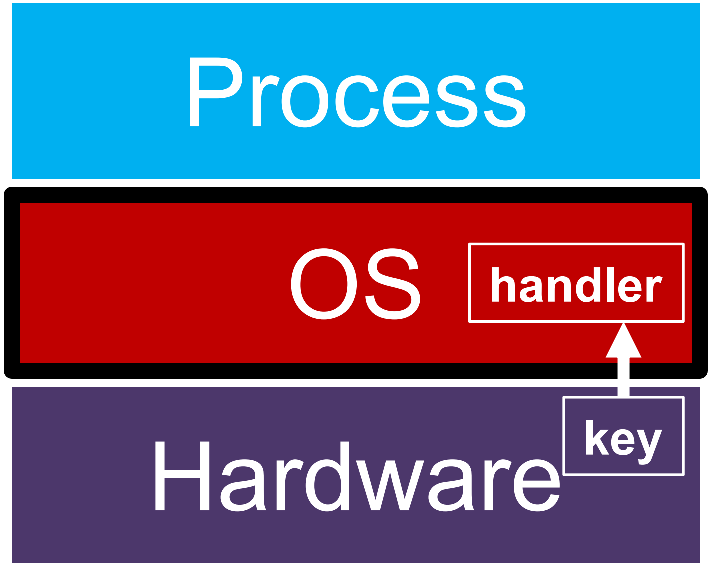

Process

OS

Hardware

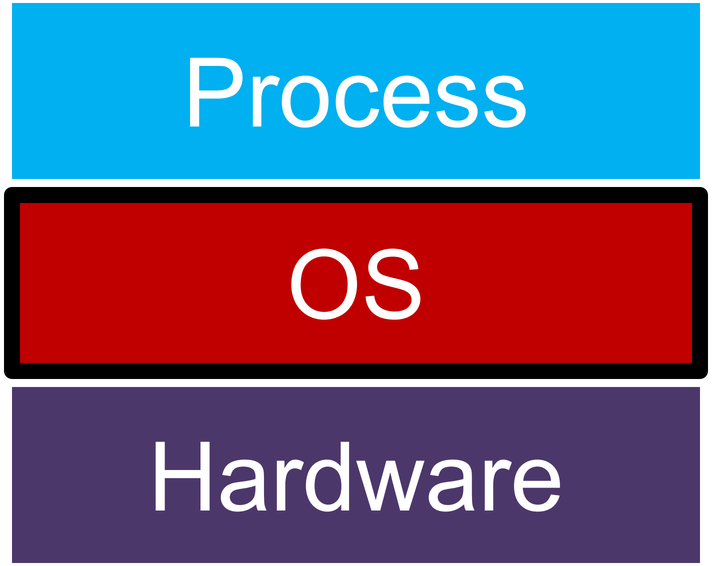
key

Interrupt



Hardware interrupt

Interrupt

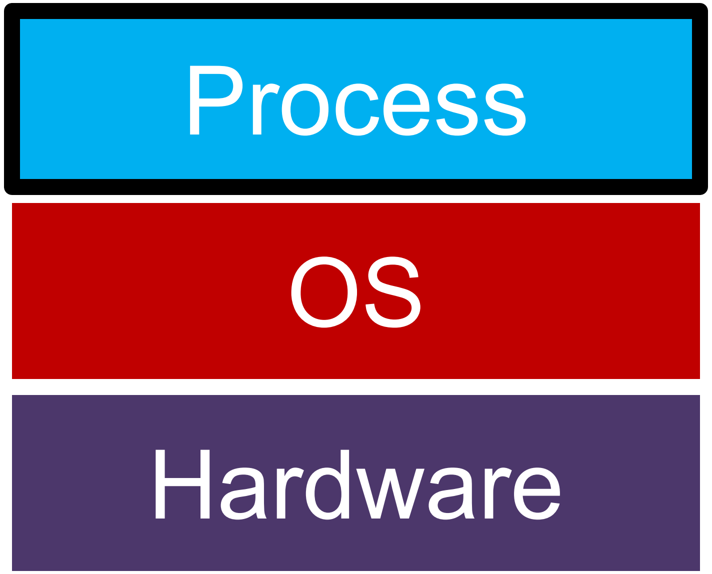


Process

OS

Hardware

System Call

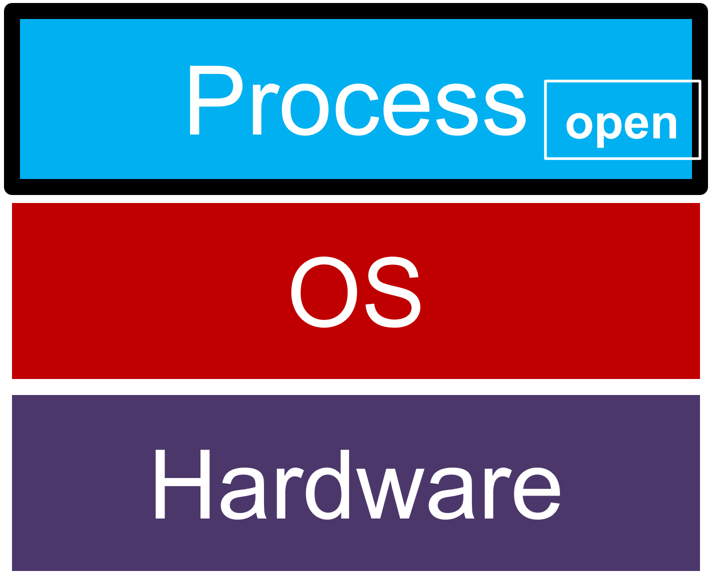


Process

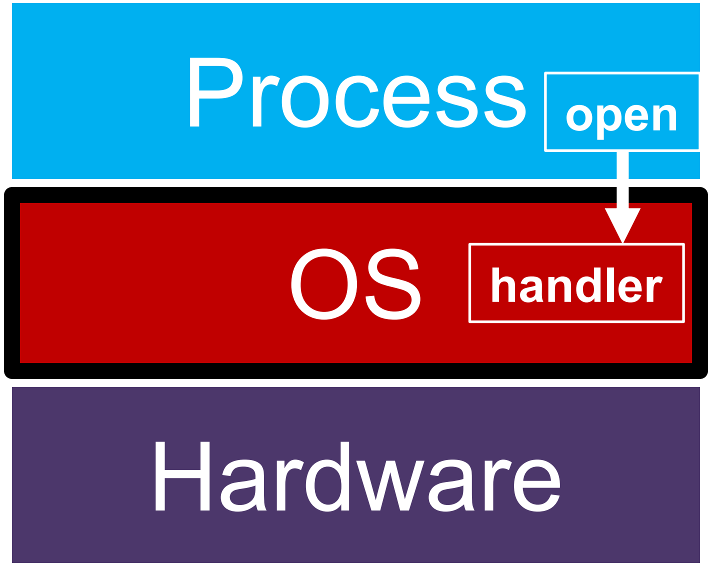
OS

Hardware

System Call

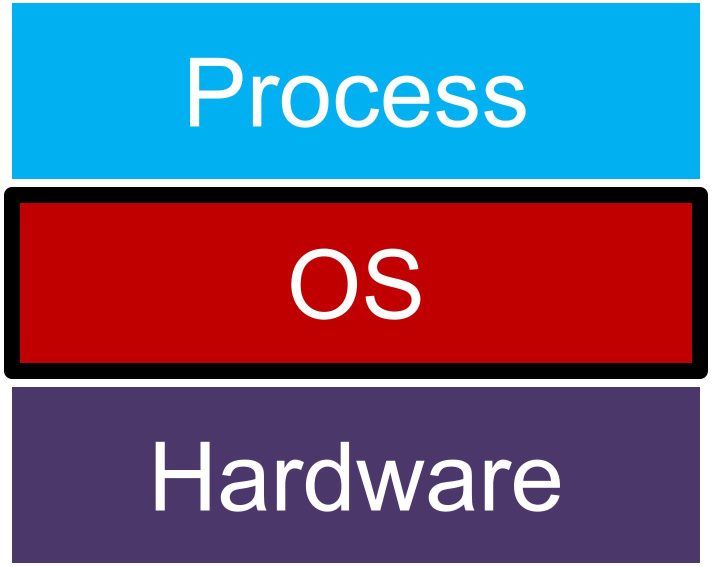


System Call



System call “trap”

System Call



Process

OS

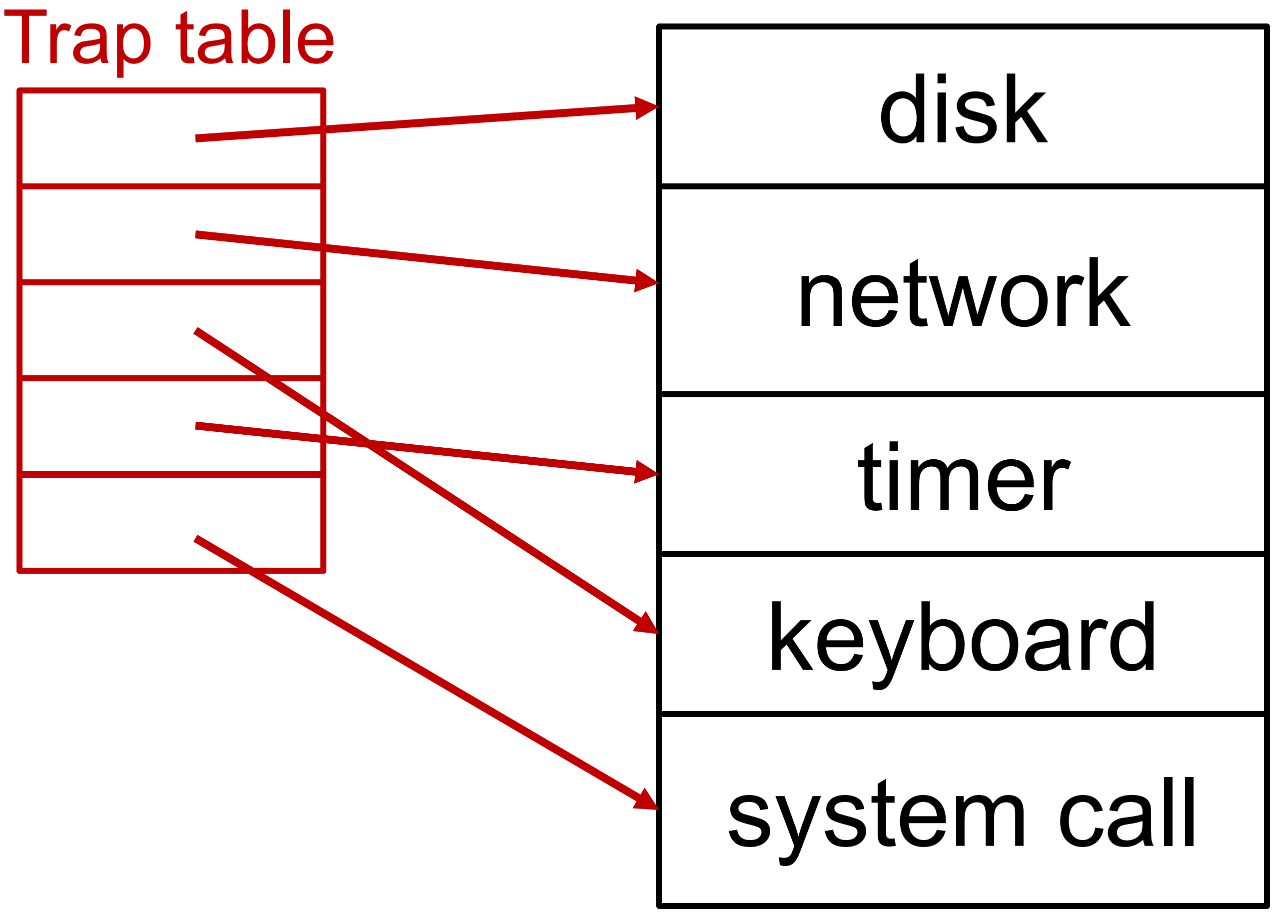
Hardware

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

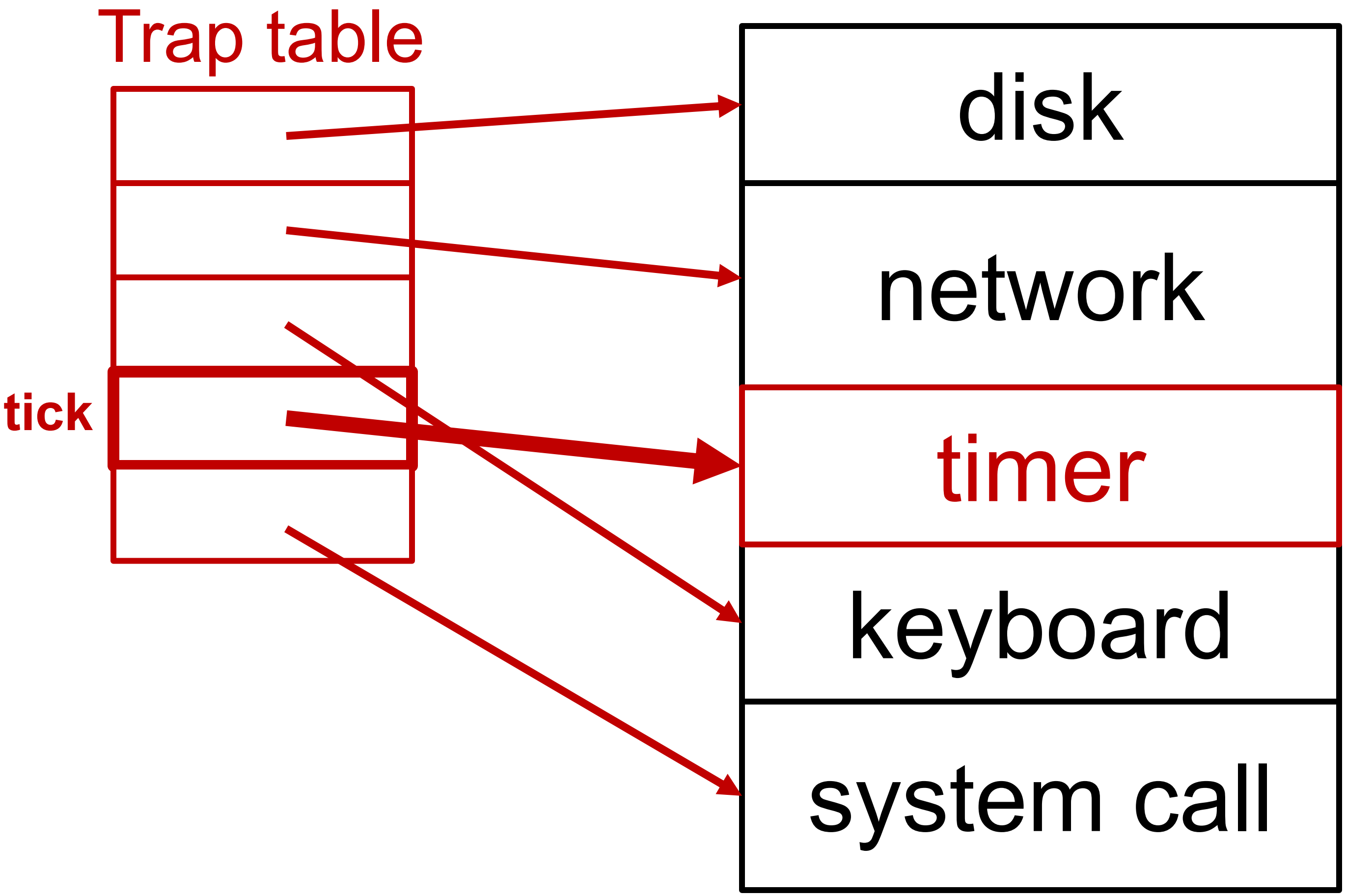
network

timer

keyboard

system call

Use array of function pointers to locate OS functions
(**Hardware knows where this is**)



Trap table

tick

disk

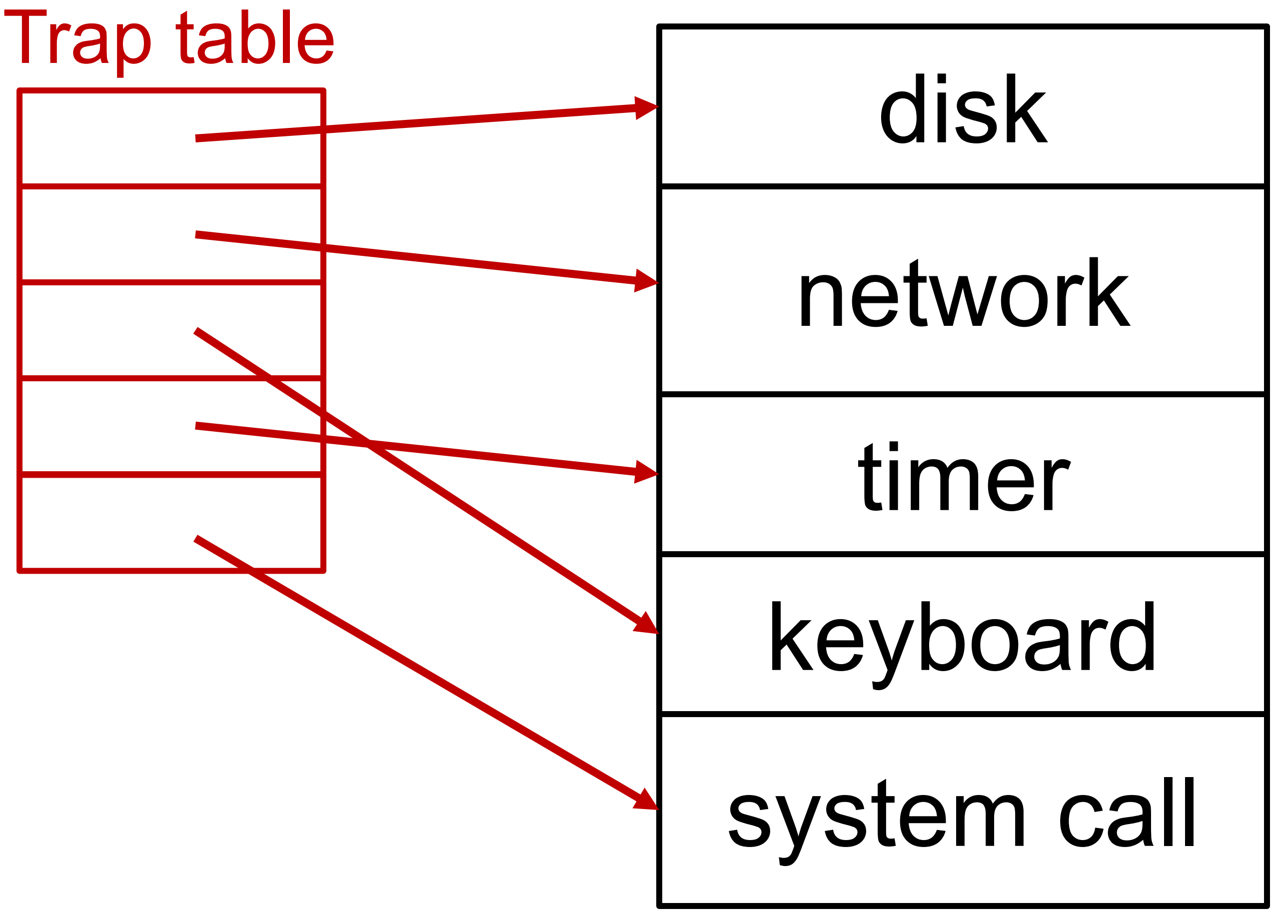
network

timer

keyboard

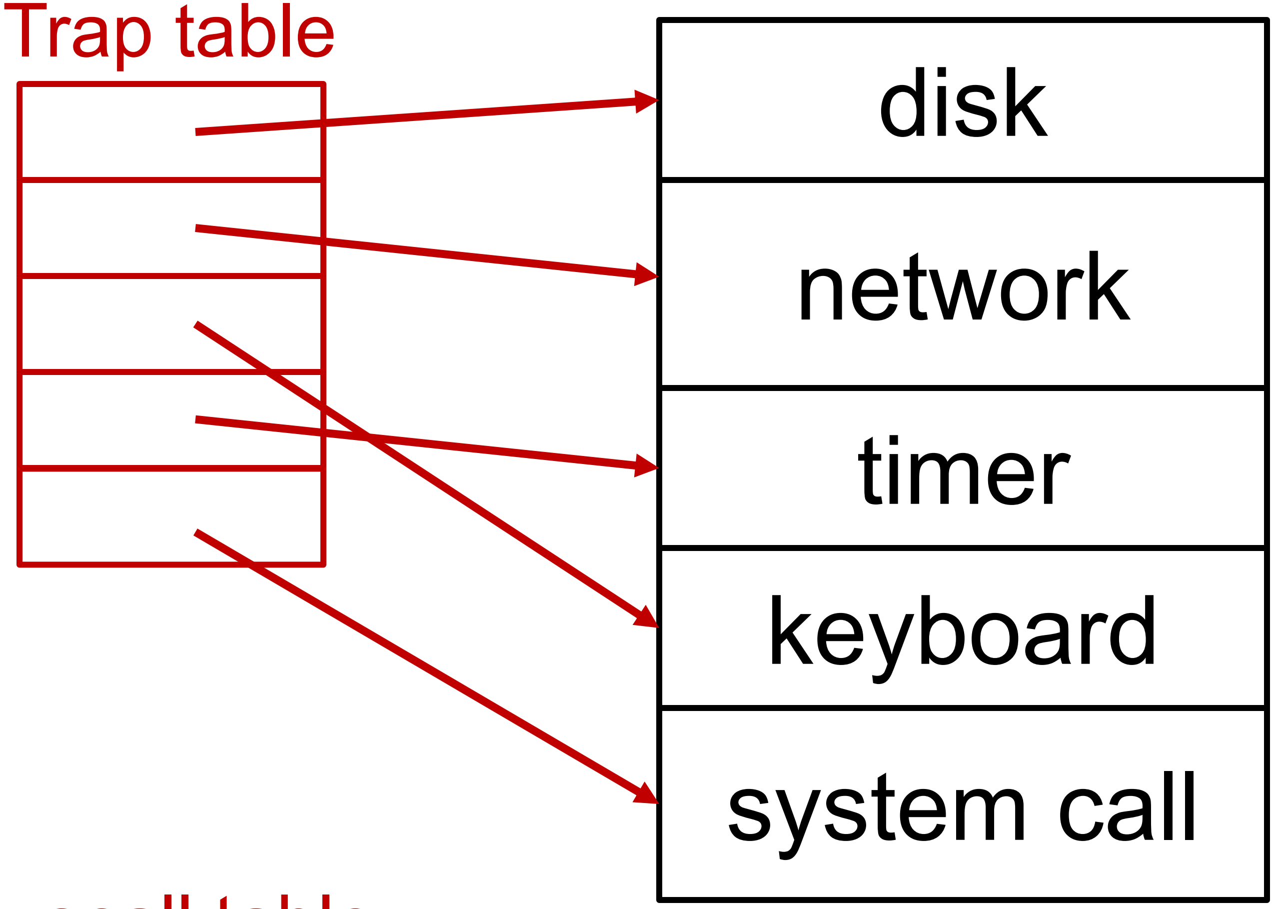
system call

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



How to handle variable number of system calls?

Trap table



disk

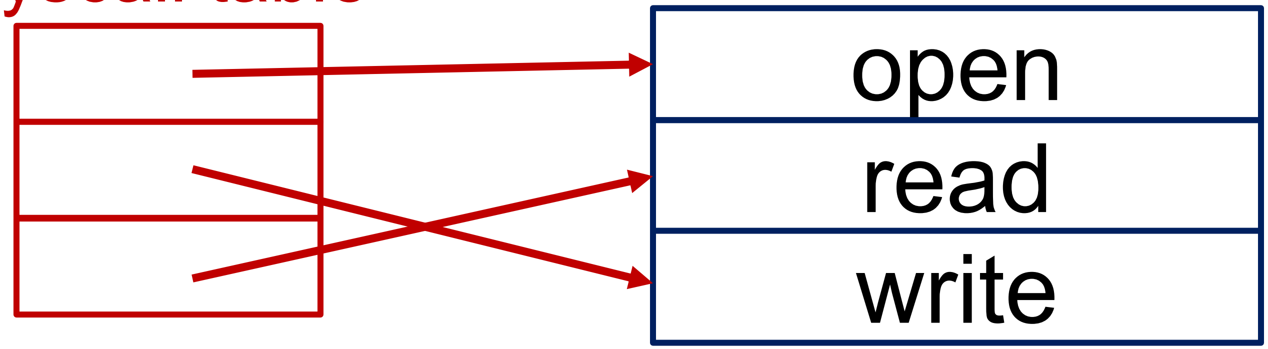
network

timer

keyboard

system call

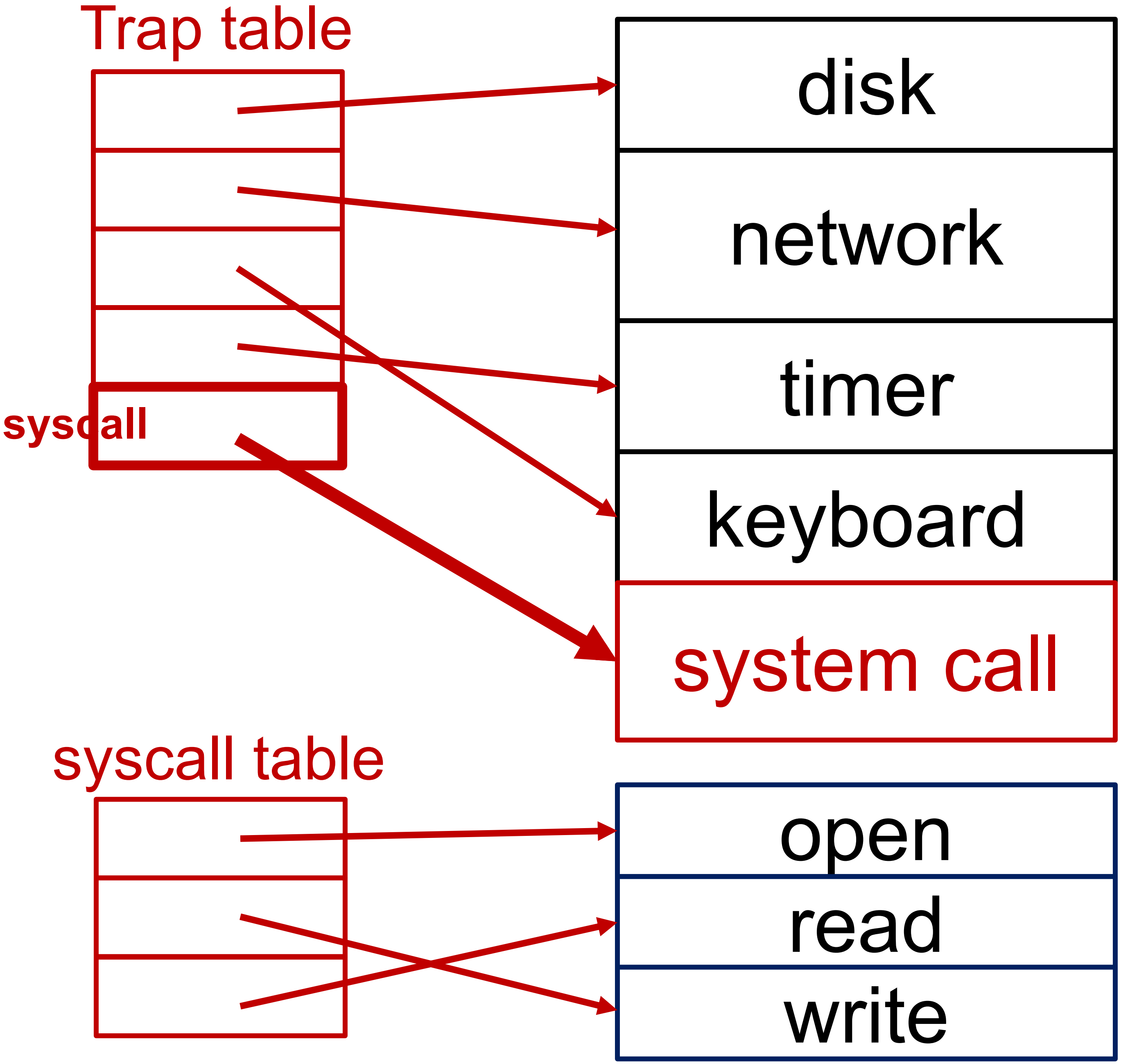
syscall table



open

read

write



Trap table

syscall

disk

network

timer

keyboard

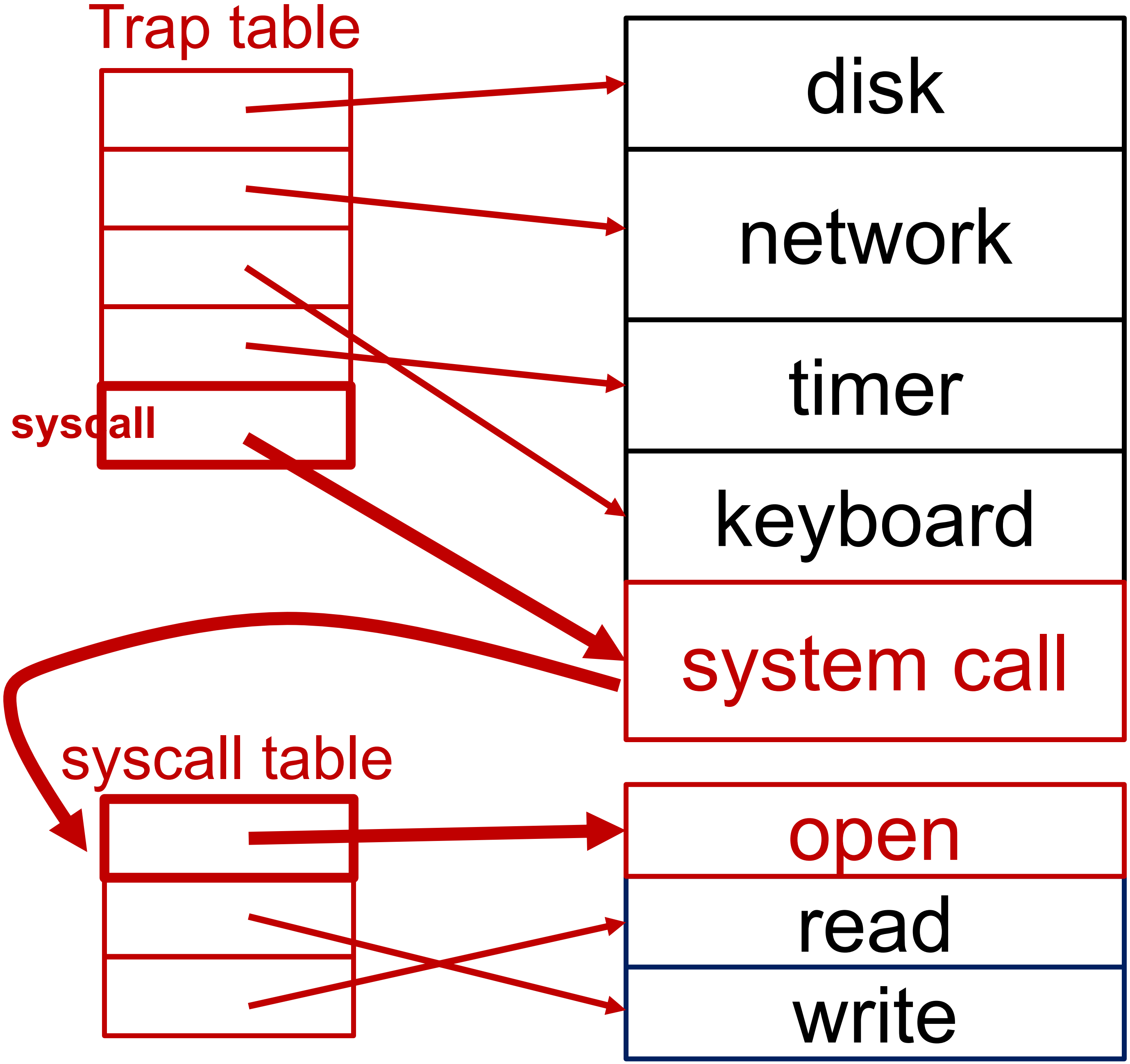
system call

syscall table

open

read

write



Trap table

syscall

disk

network

timer

keyboard

system call

syscall table

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

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

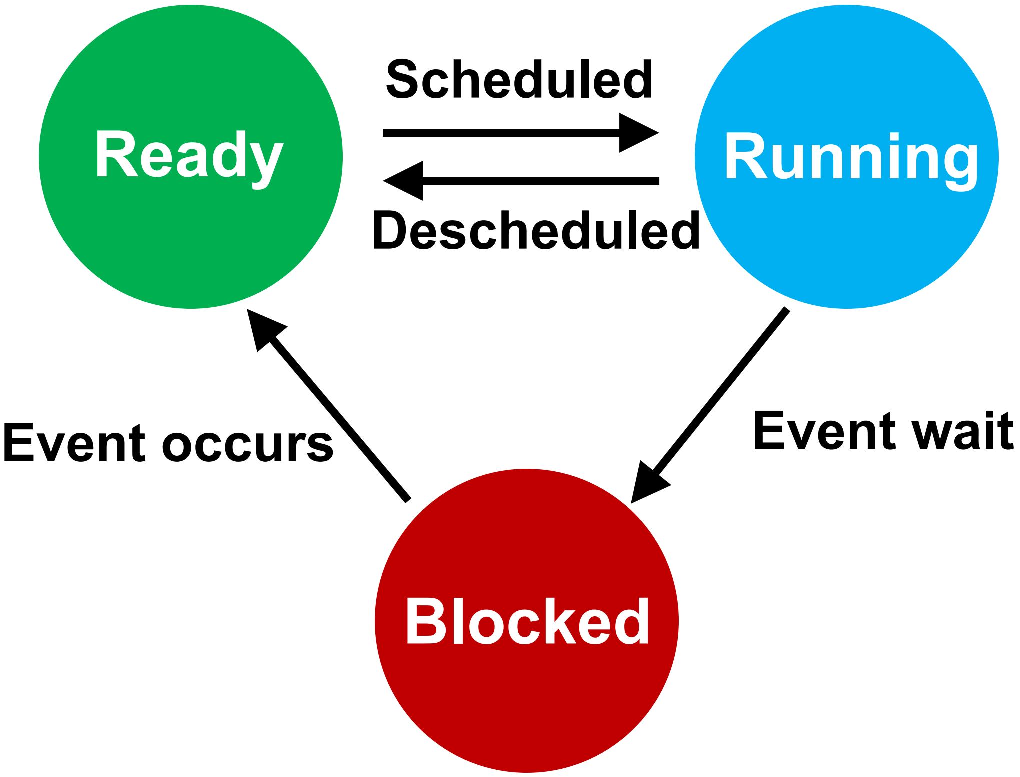
- Look in `kernel/proc.h`
  - `context` (CPU registers)
  - `ofile` (open file descriptors)
  - `state` (sleeping, running, etc.)

What to do with processes that are not running?

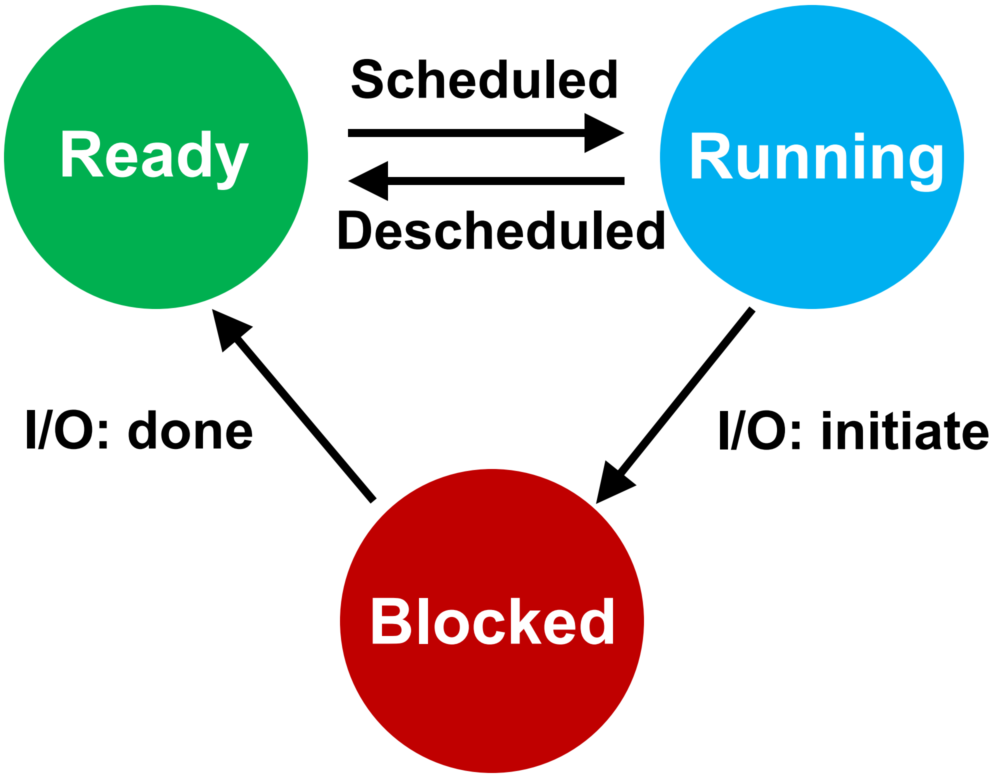
- A: Store context in OS struct

- Look in `kernel/proc.h`
  - `context` (CPU registers)
  - `ofile` (open file descriptors)
  - **`state`** (sleeping, running, etc.)

Process State Transitions



Process State Transitions



Ready

Scheduled

Descheduled

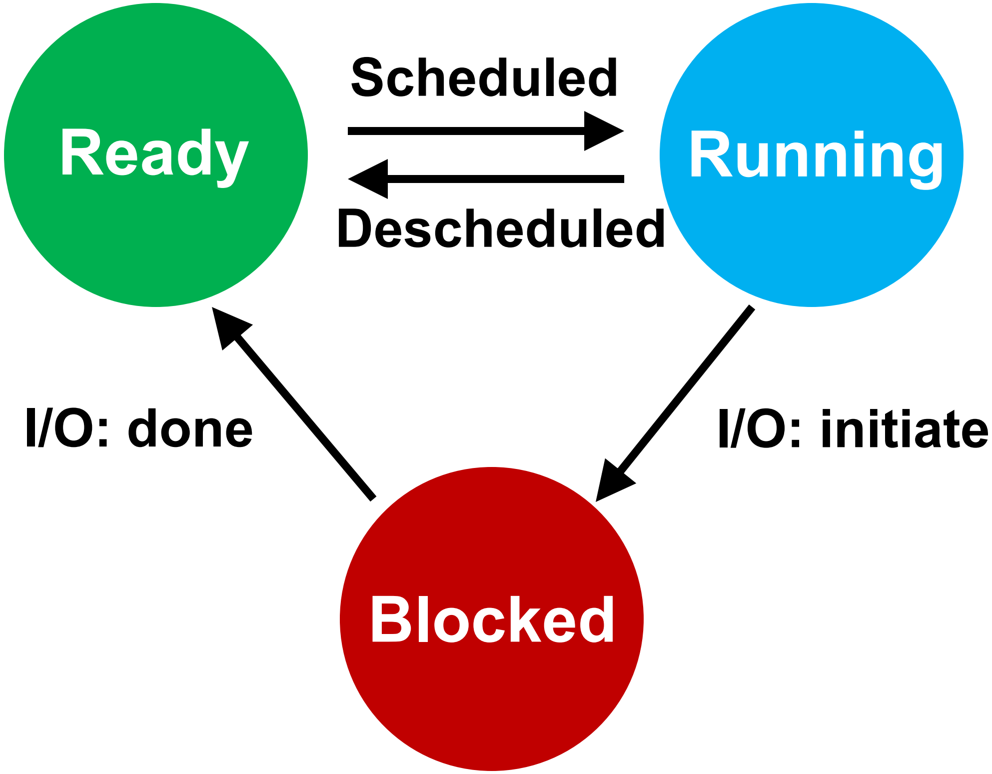
Running

I/O: done

I/O: initiate

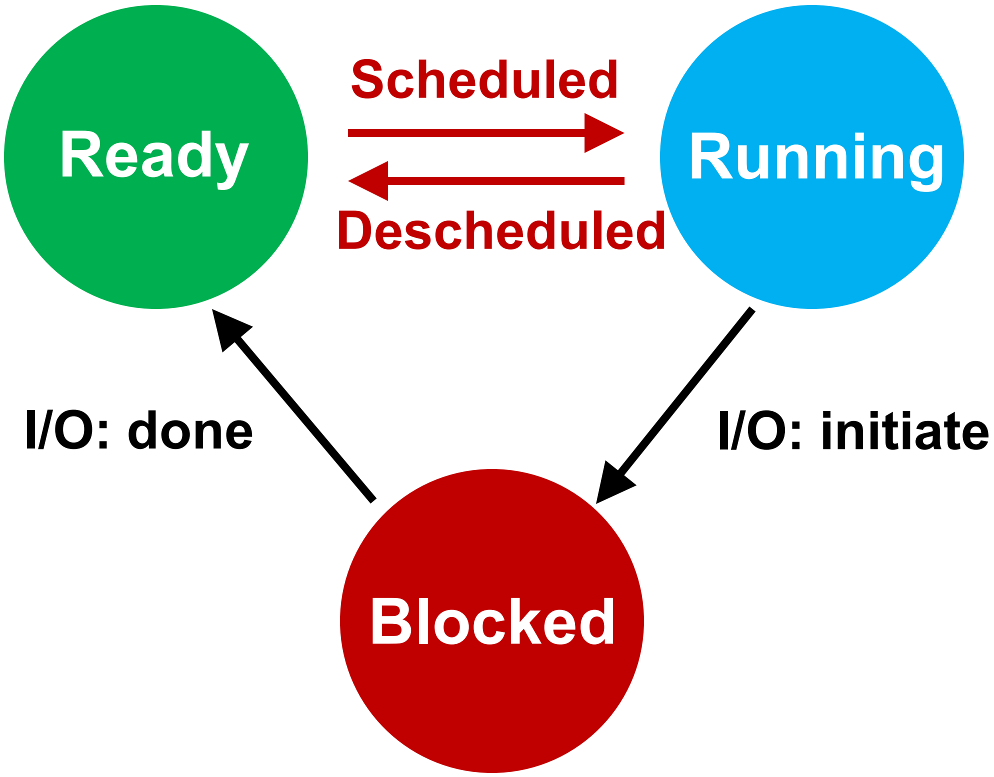
Blocked

Process State Transitions



View process state with “`ps xa`”

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



Ready

Scheduled

Descheduled

Running

I/O: done

I/O: initiate

Blocked

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

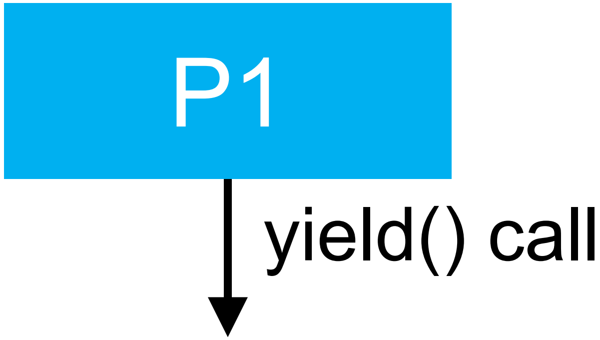
Cooperative Approach

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

P1

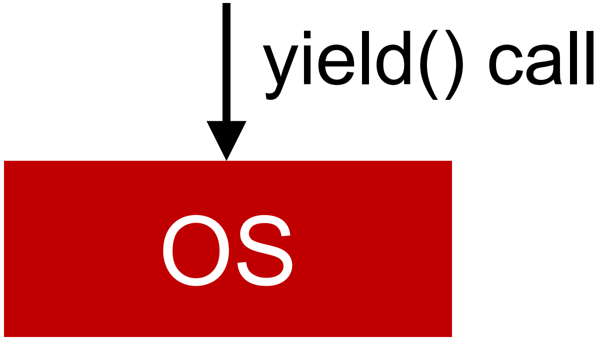
Cooperative Approach

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



Cooperative Approach

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



yield() call

OS

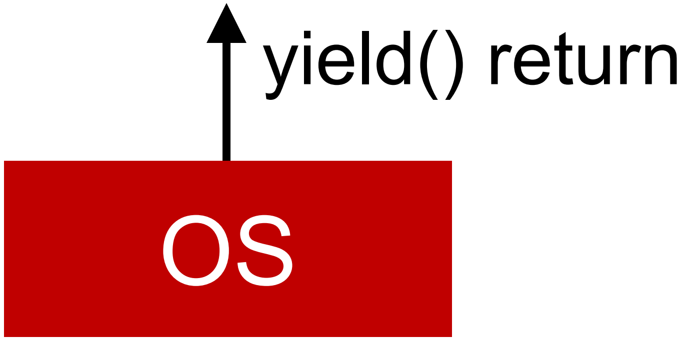
Cooperative Approach

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

OS

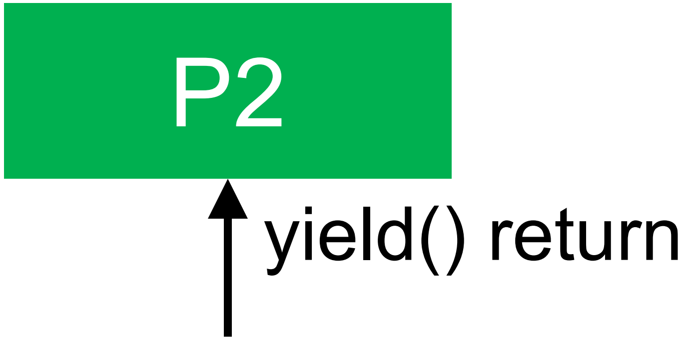
Cooperative Approach

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



Cooperative Approach

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



P2

yield() return

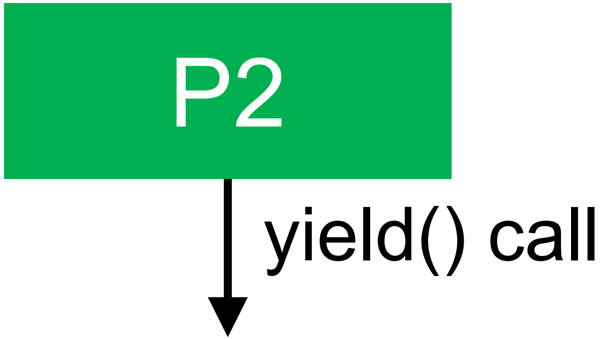
Cooperative Approach

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

P2

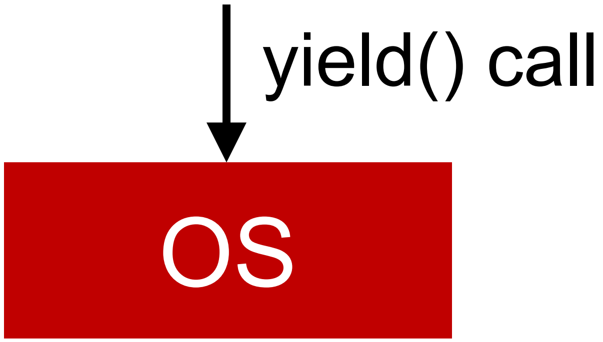
Cooperative Approach

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



Cooperative Approach

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



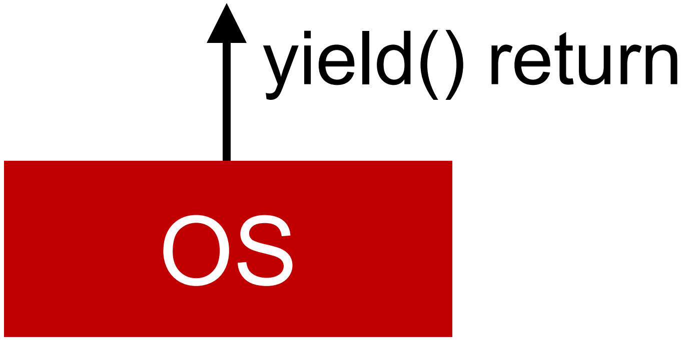
Cooperative Approach

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

OS

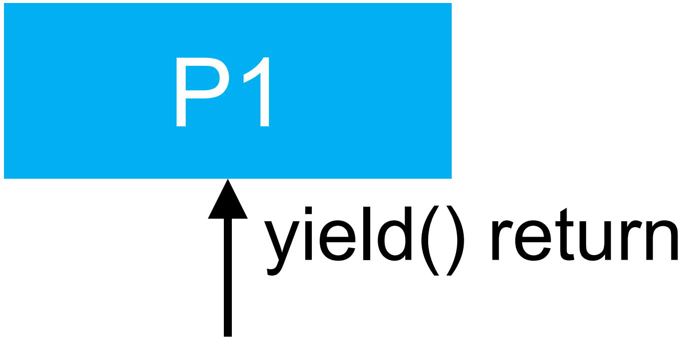
Cooperative Approach

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



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()?

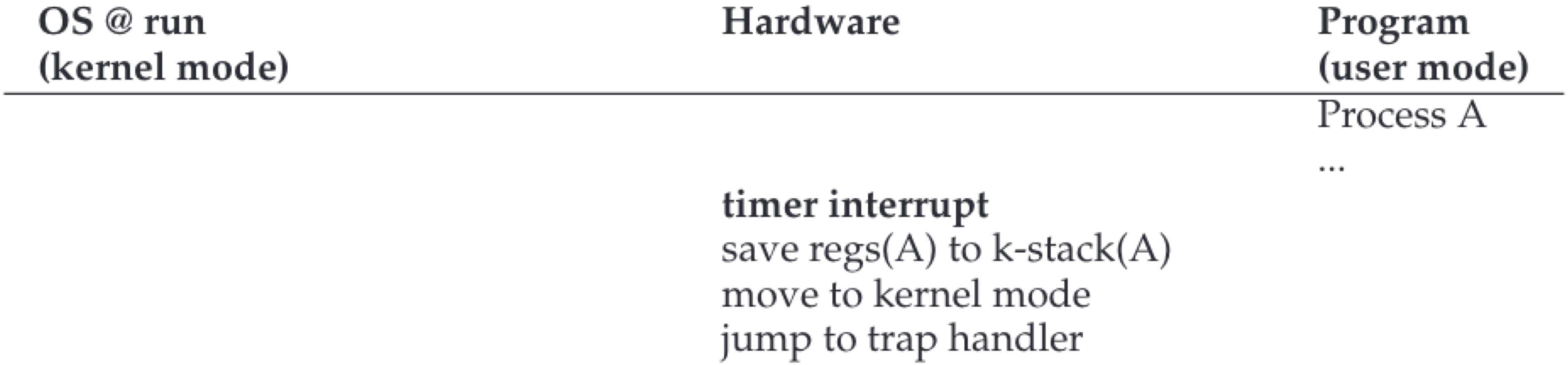
Non-Cooperative Approach

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

Non-Cooperative Approach

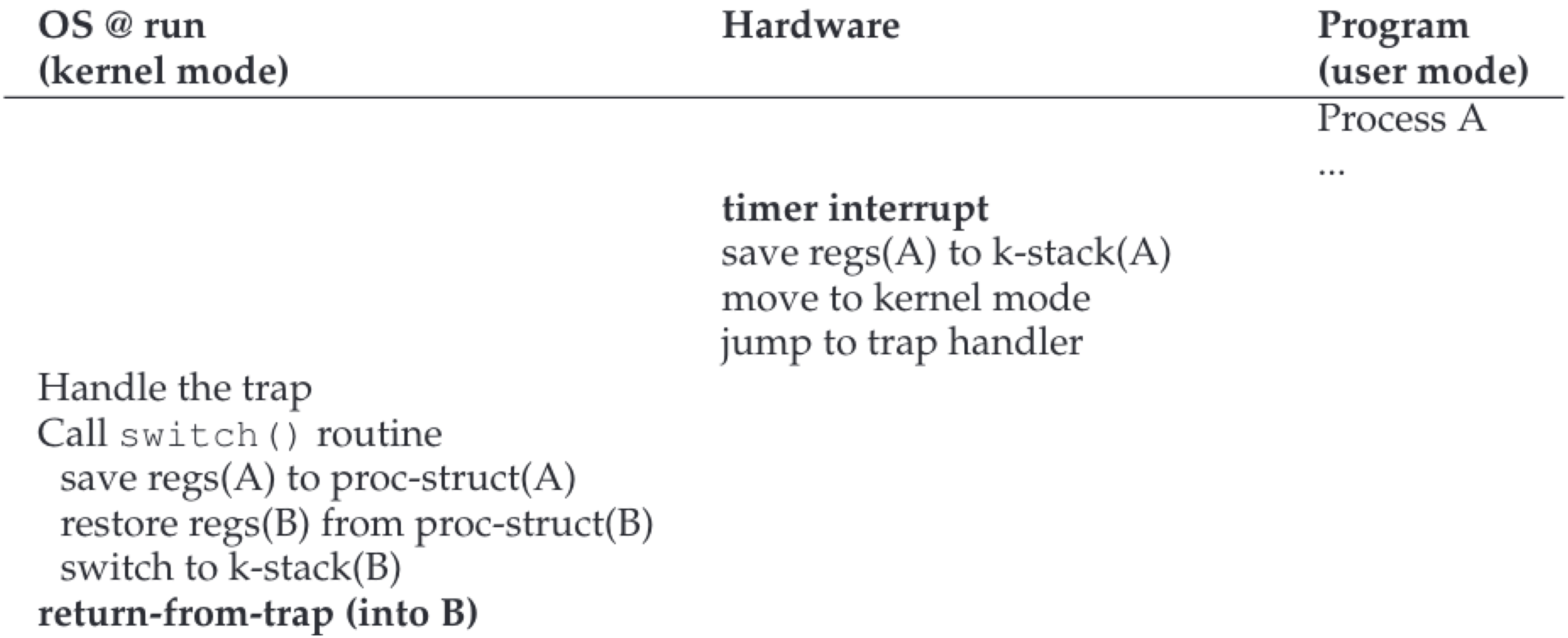
OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A ...

Non-Cooperative Approach



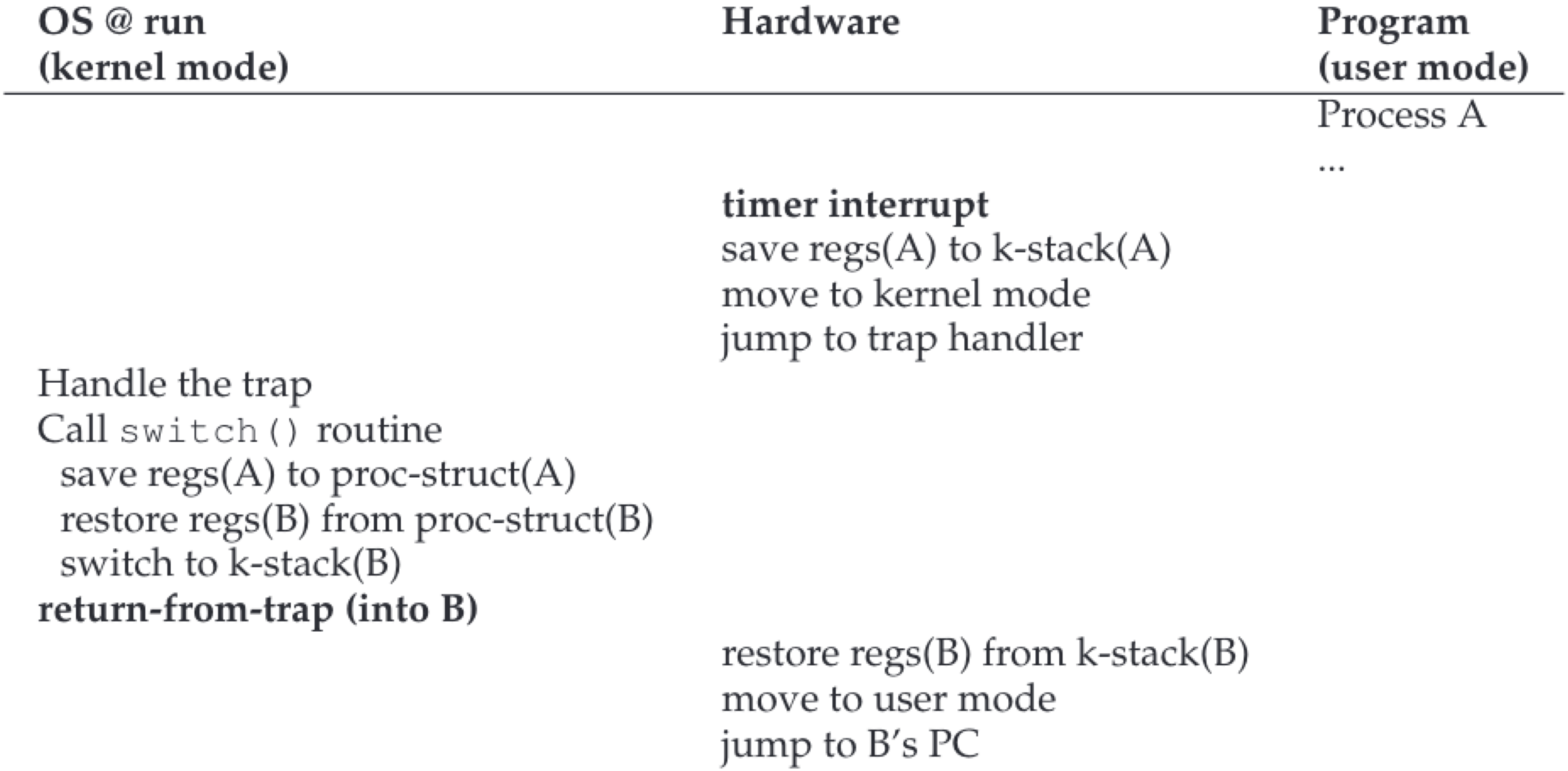
OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	

Non-Cooperative Approach



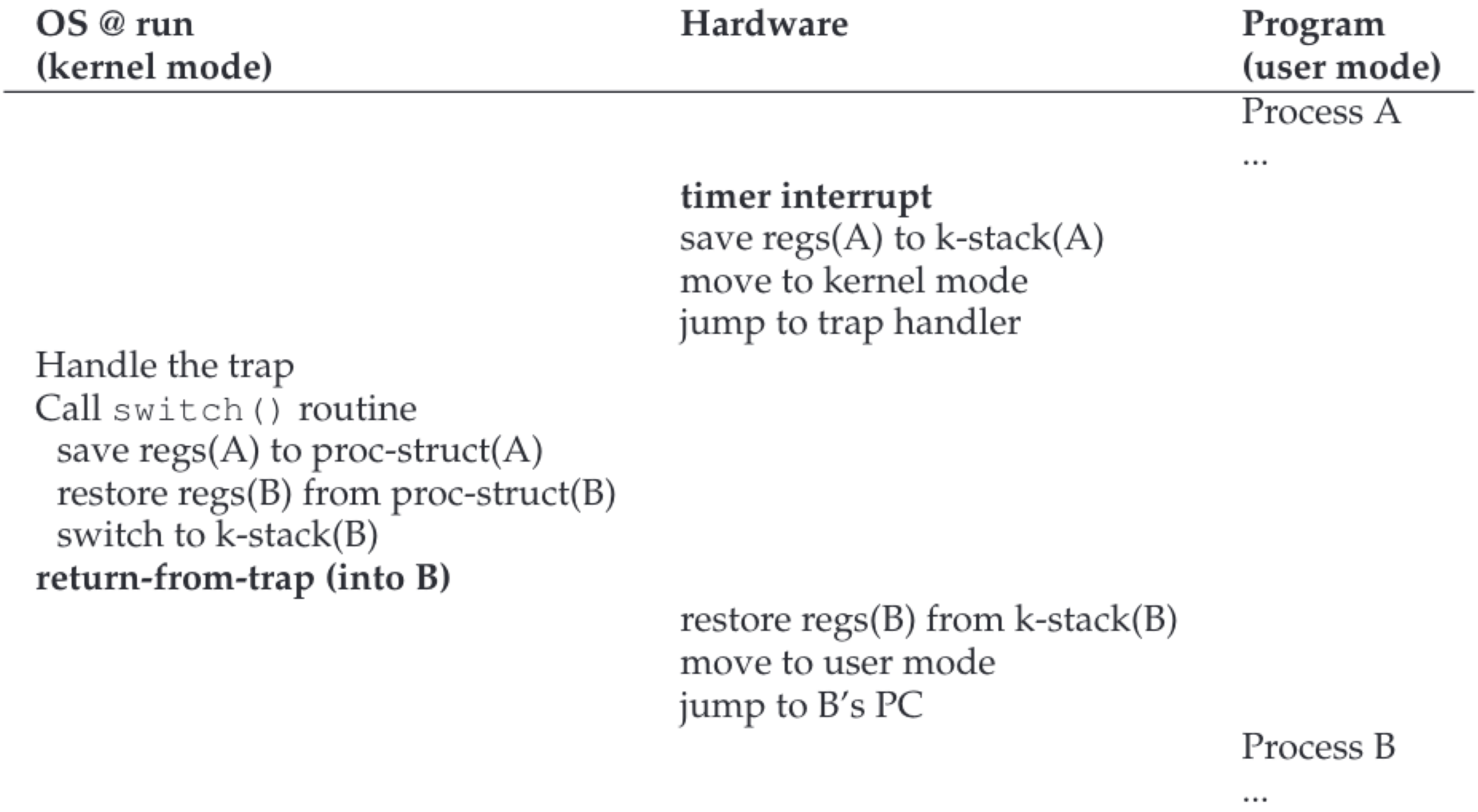
OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A ...
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call `switch()` routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) **return-from-trap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

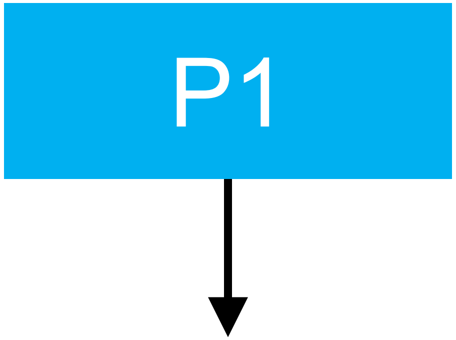
Handle the trap
Call `switch()` routine
  save regs(A) to proc-struct(A)
  restore regs(B) from proc-struct(B)
  switch to k-stack(B)
return-from-trap (into B)		

	restore regs(B) from k-stack(B) move to user mode jump to B's PC	
		Process B ...

Preemptive Approach

P1

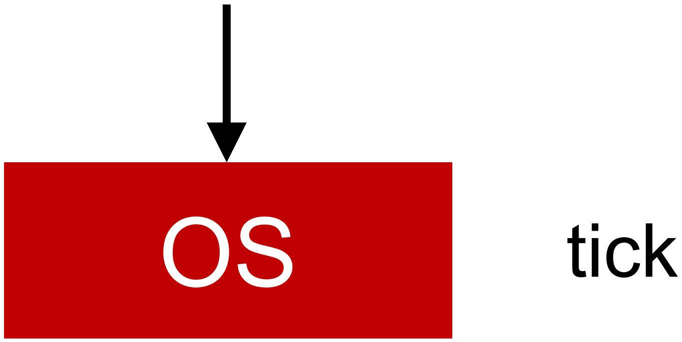
Preemptive Approach



P1

tick

Preemptive Approach



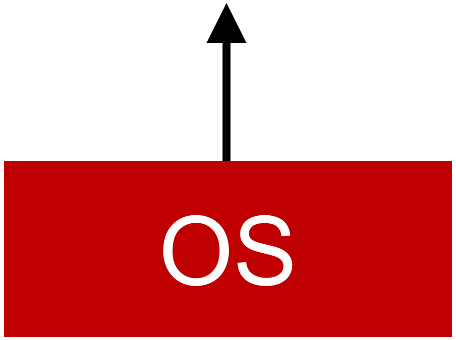
OS

tick

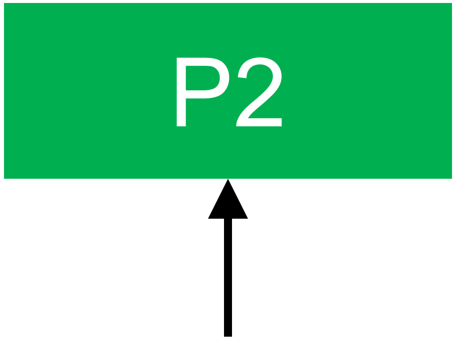
Preemptive Approach

OS

Preemptive Approach



Preemptive Approach

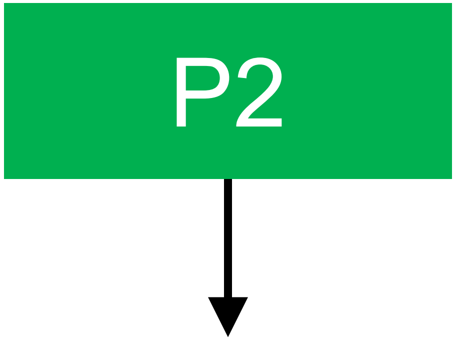


P2

Preemptive Approach

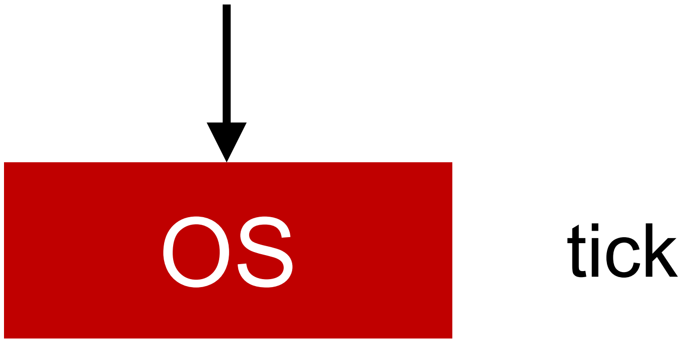
P2

Preemptive Approach



tick

Preemptive Approach



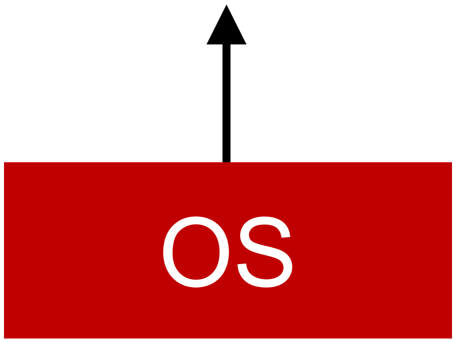
OS

tick

Preemptive Approach

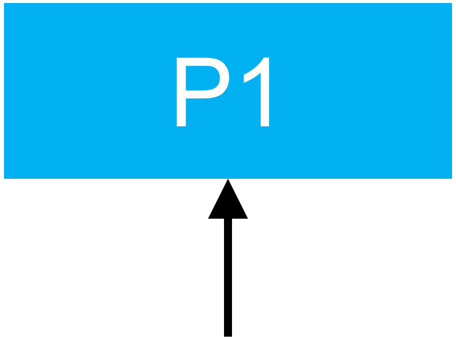
OS

Preemptive Approach



OS

Preemptive Approach



P1

Preemptive Approach

P1

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