[537] Processes

Tyler Harter
9/8/14

0

1

2

3

4

5

6

7

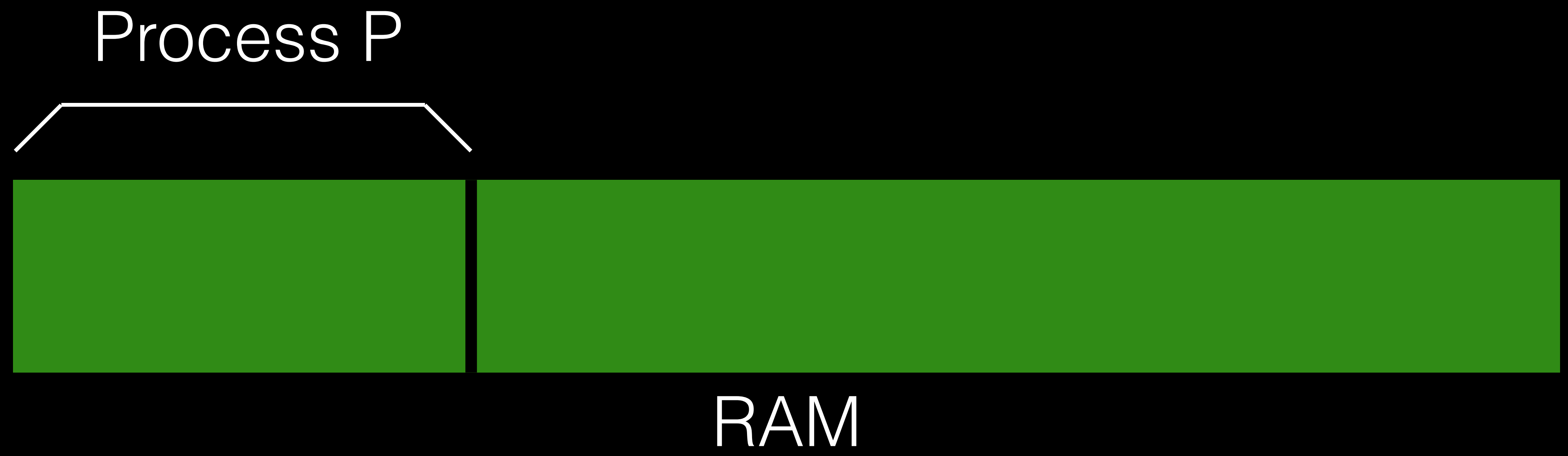
8

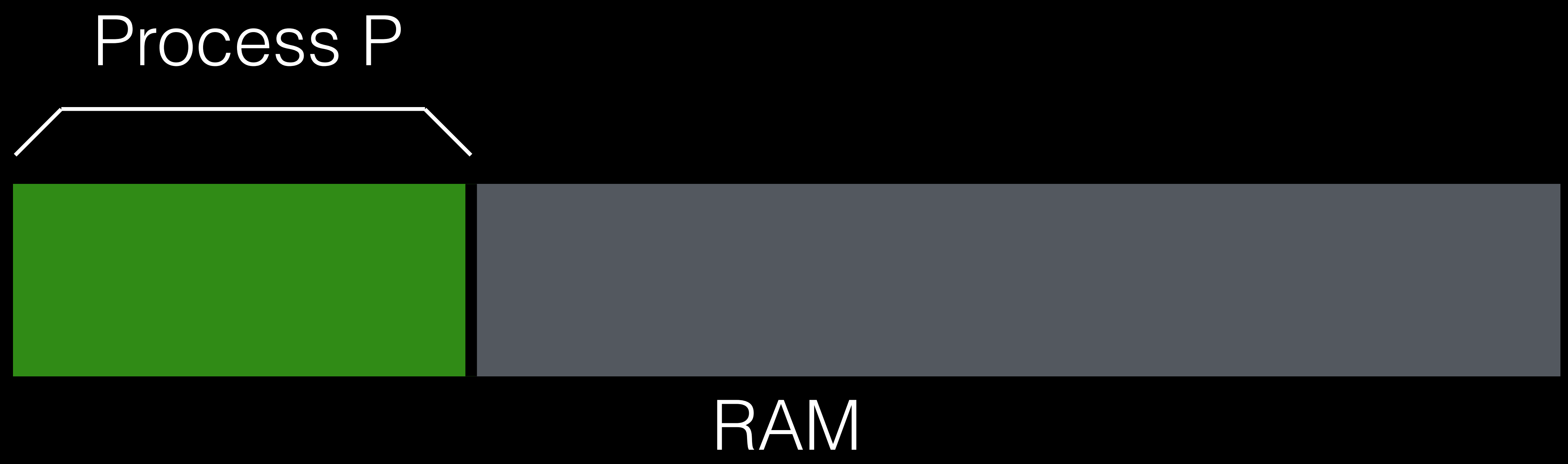
9

10

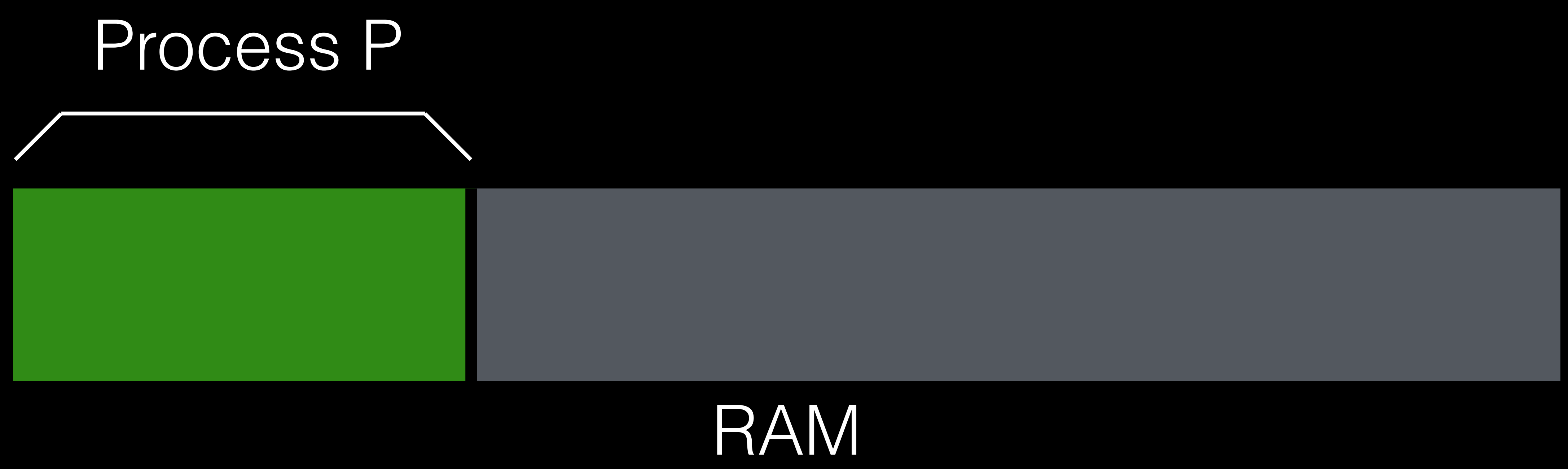
A B C D E F G H I J K L

Review: System Calls

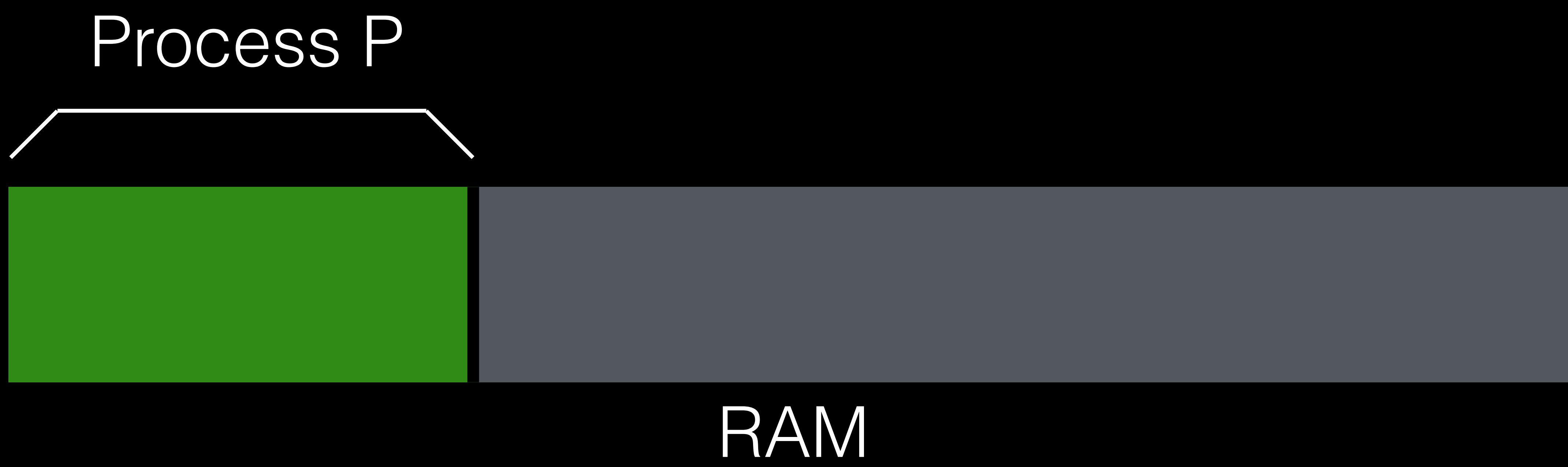

Process P
RAM



P can only see its own memory because of **user mode**
(other areas, including kernel, are hidden)

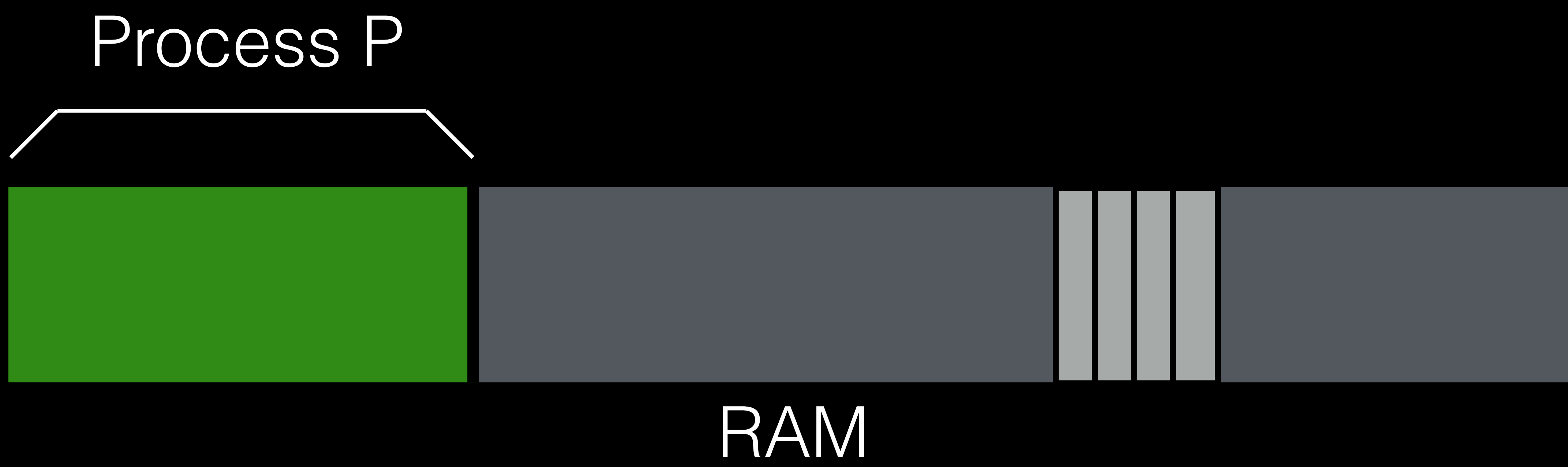


P wants to call read()




```
movl $6, %eax;   int $64
```

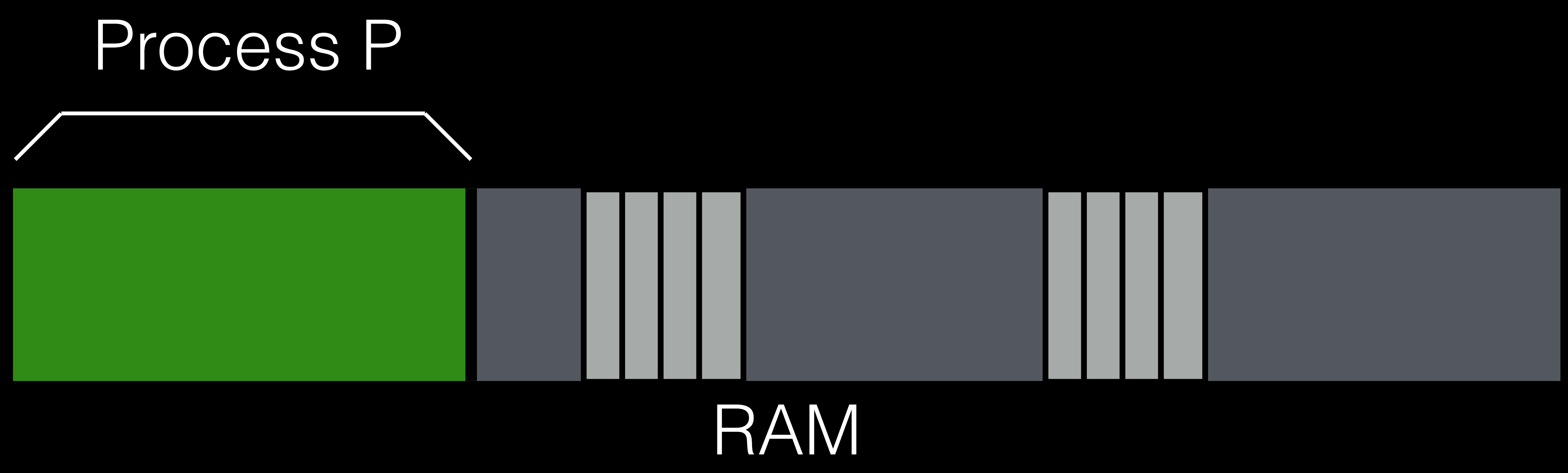
```
static int (*syscalls[])(void)    (syscall.c)
```




```
movl $6, %eax;   int $64
```

syscall-table index

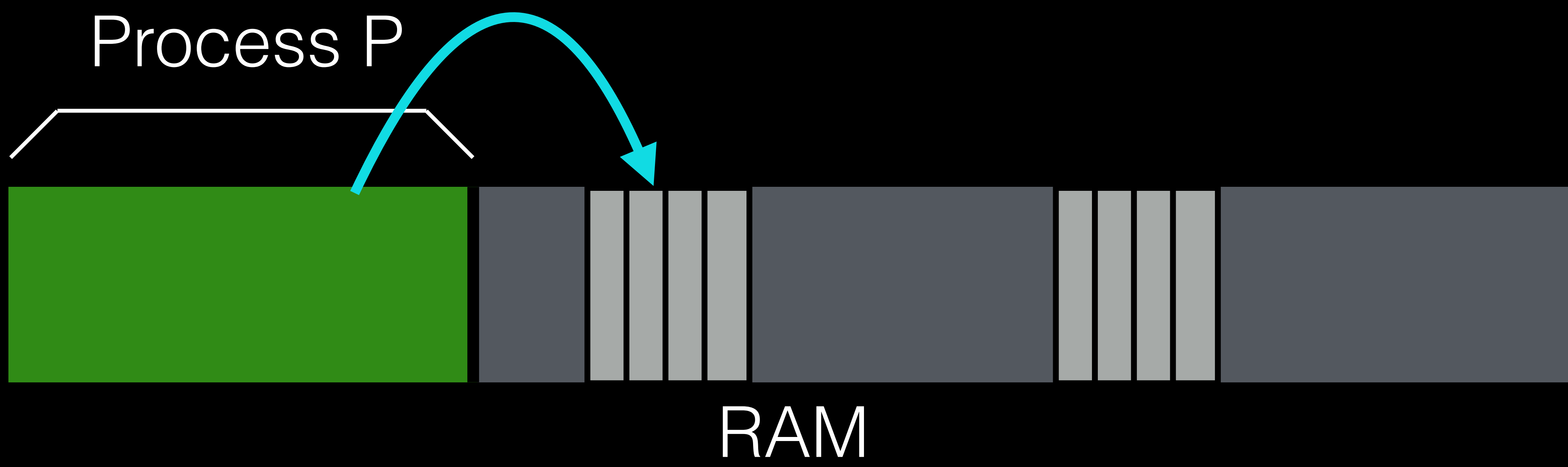
`struct gatedesc idt[256]`   (trap.c)



`movl $6, %eax;    int $64`

syscall-table index

trap-table index

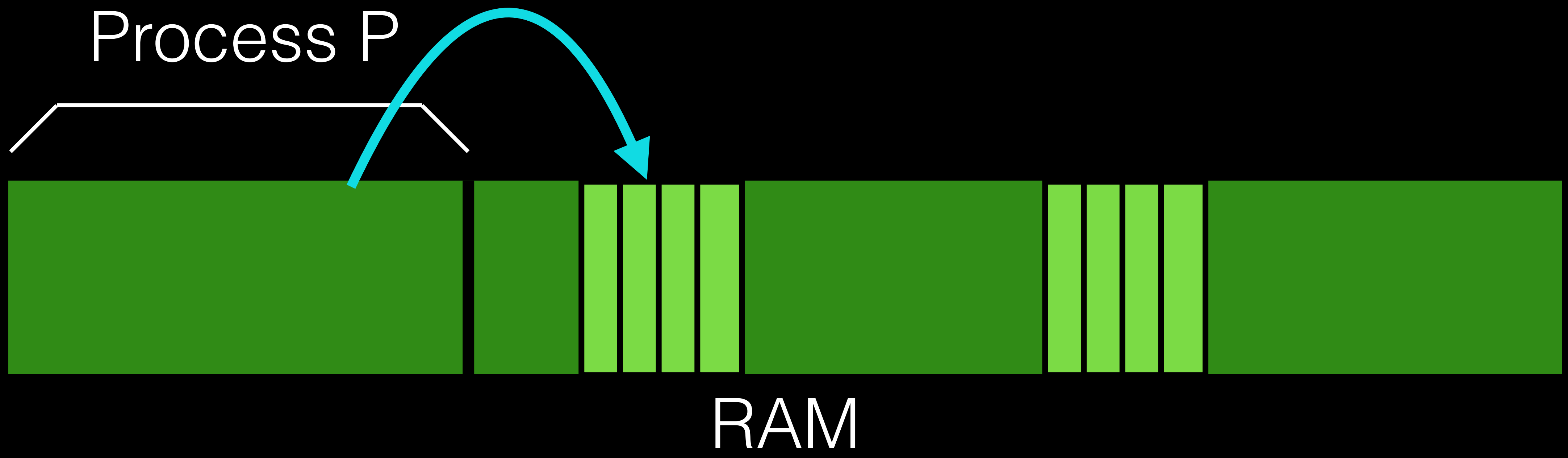



```
movl $6, %eax;    int $64
```

syscall-table index

trap-table index

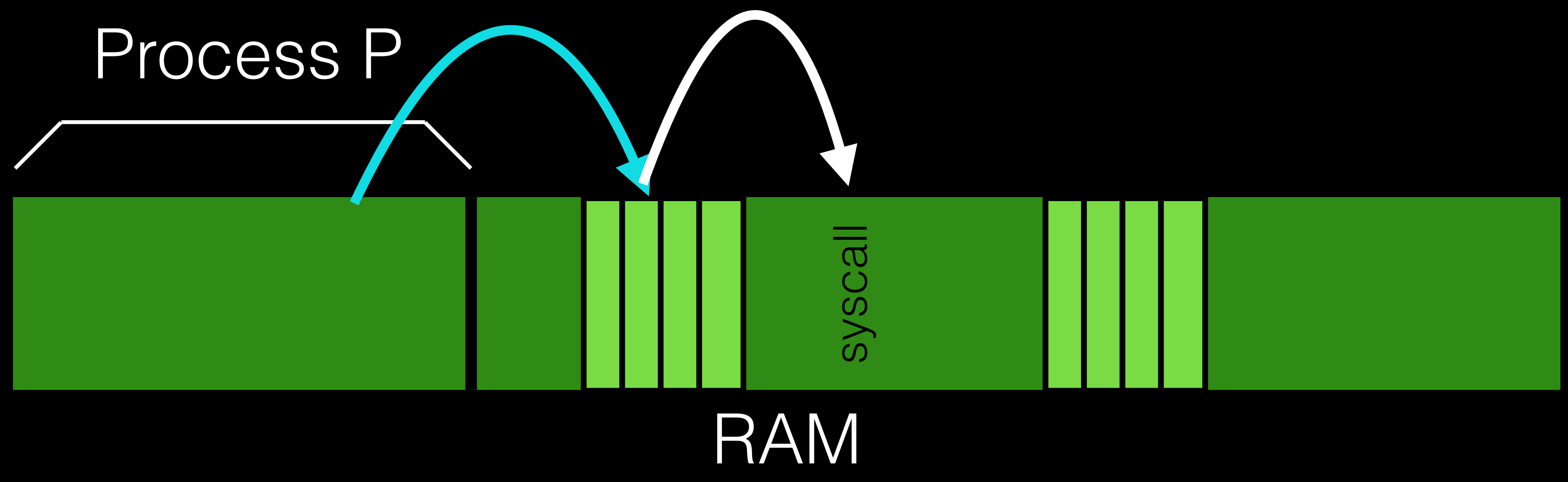
Kernel mode: we can do anything!




```
movl $6, %eax;    int $64
```

syscall-table index

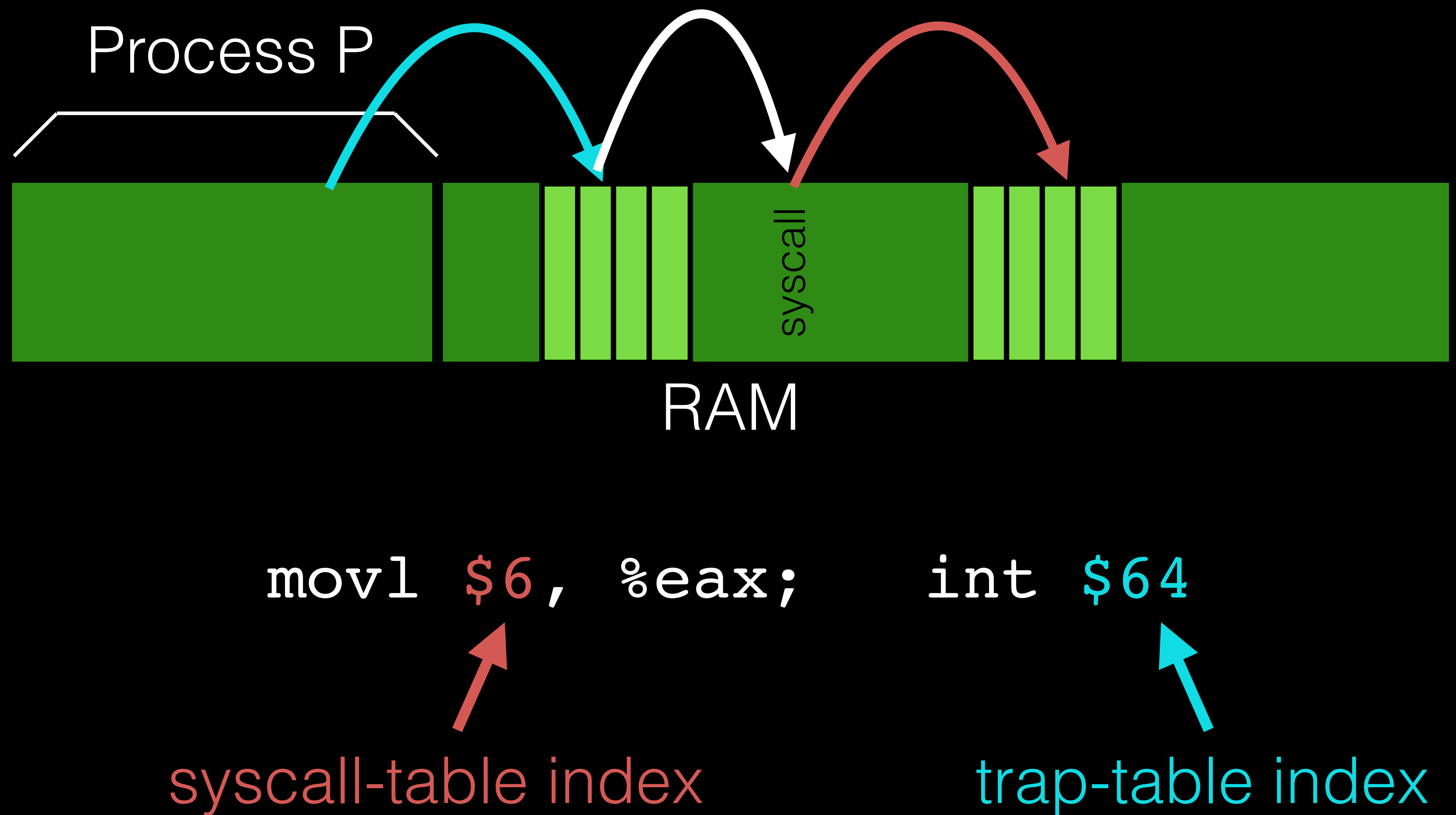
trap-table index

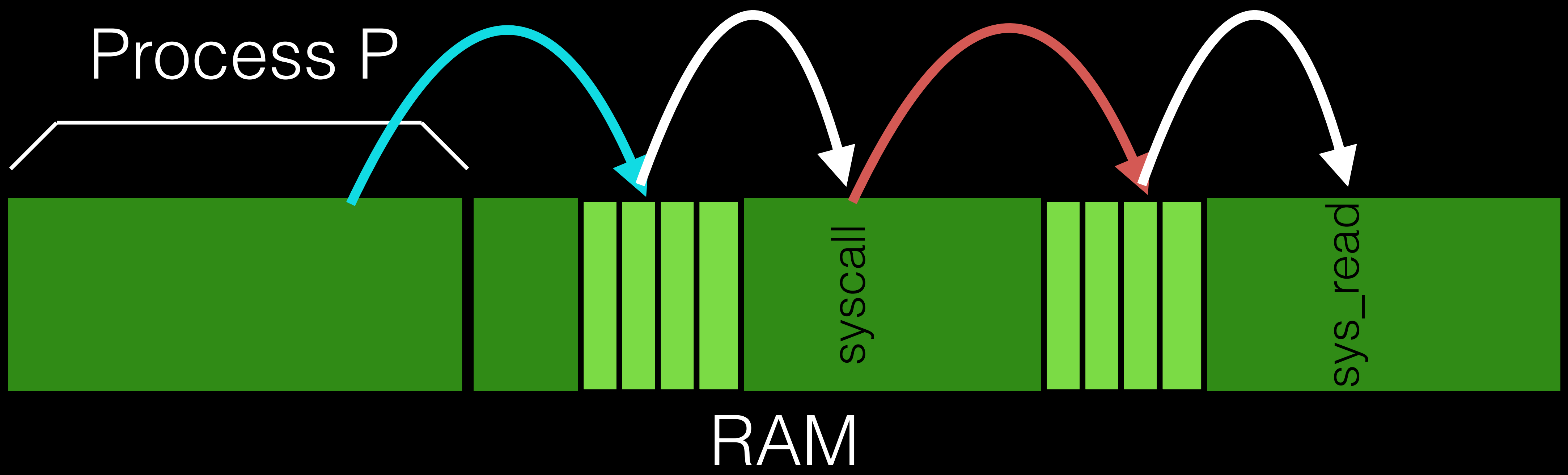



```
movl $6, %eax;    int $64
```

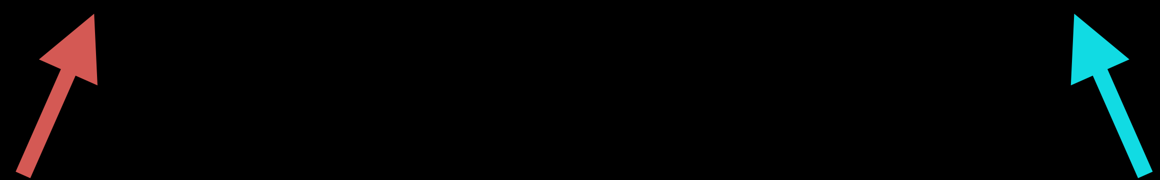
syscall-table index

trap-table index


Process P
syscall
RAM
movl $6, %eax;   int $64
syscall-table index
trap-table index

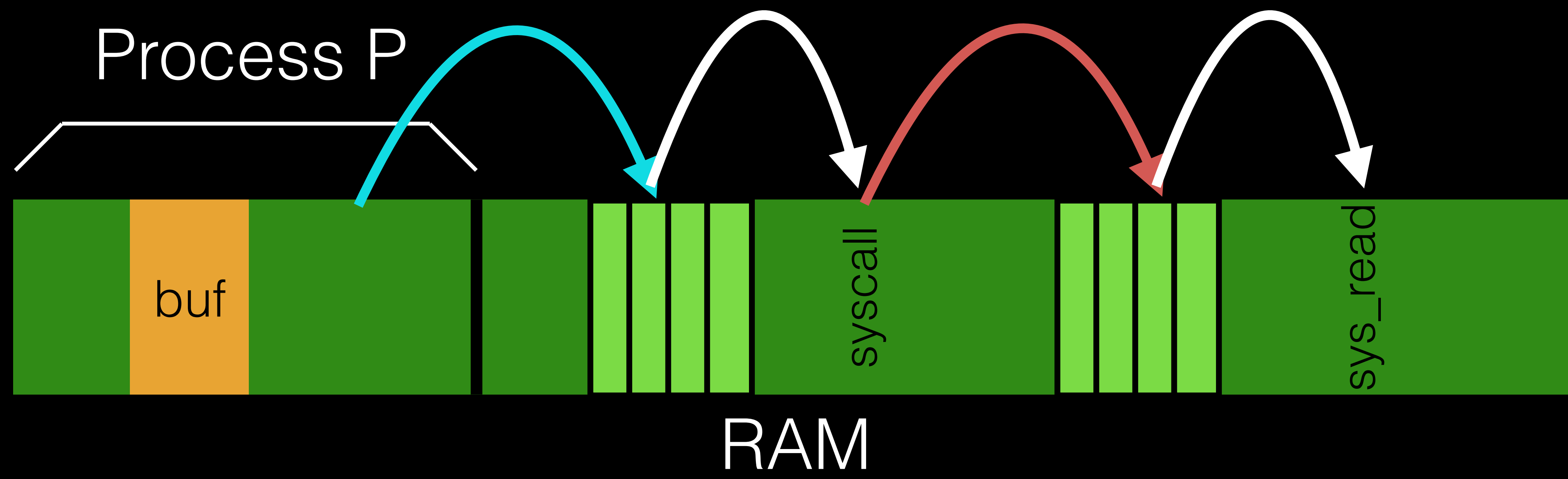



```
movl $6, %eax;    int $64
```



syscall-table index

trap-table index




```
movl $6, %eax;    int $64
```

syscall-table index

trap-table index

Processes

What’s a Process?

Java analogy:

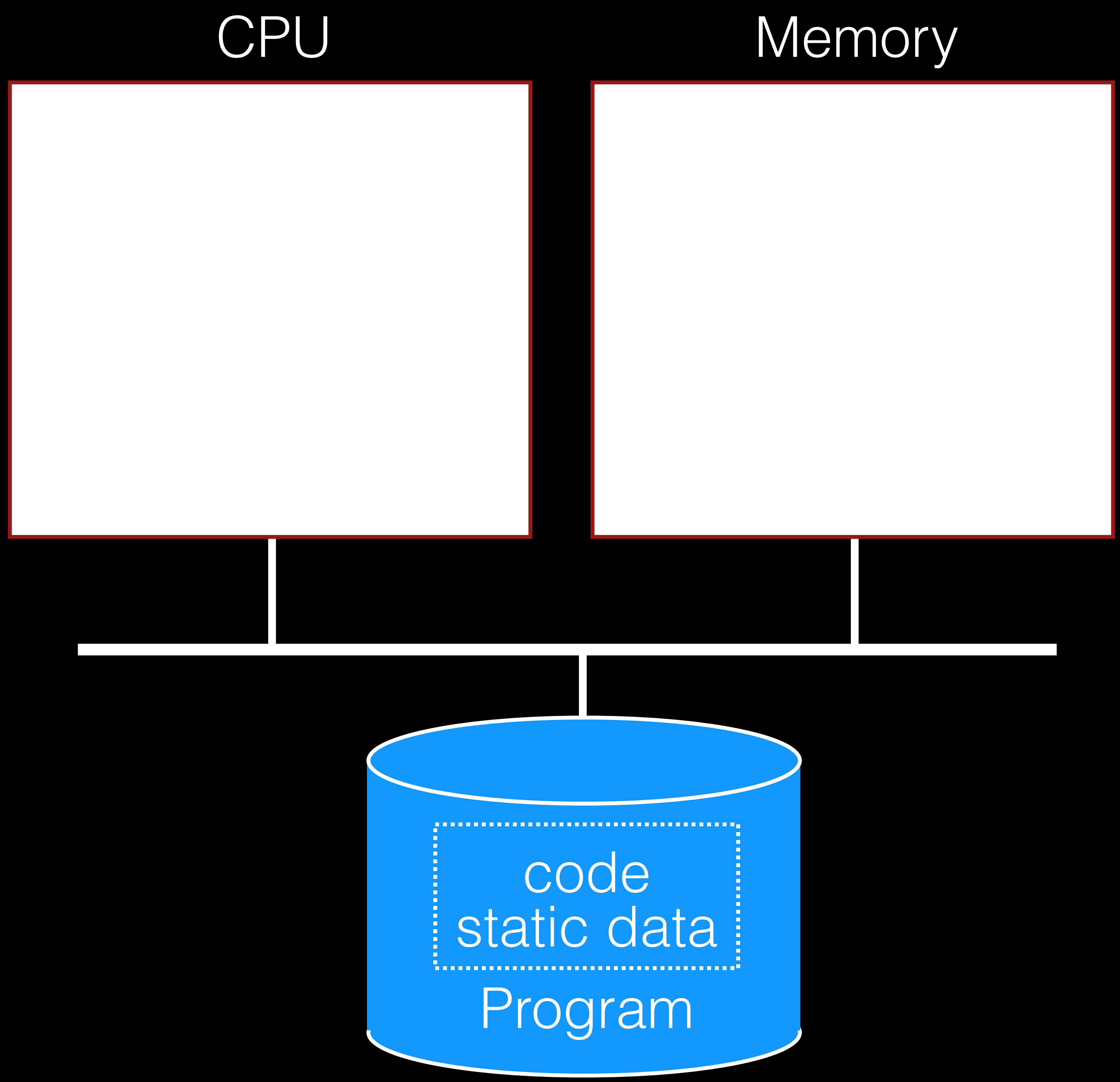
  class => “program”

  object => “process”

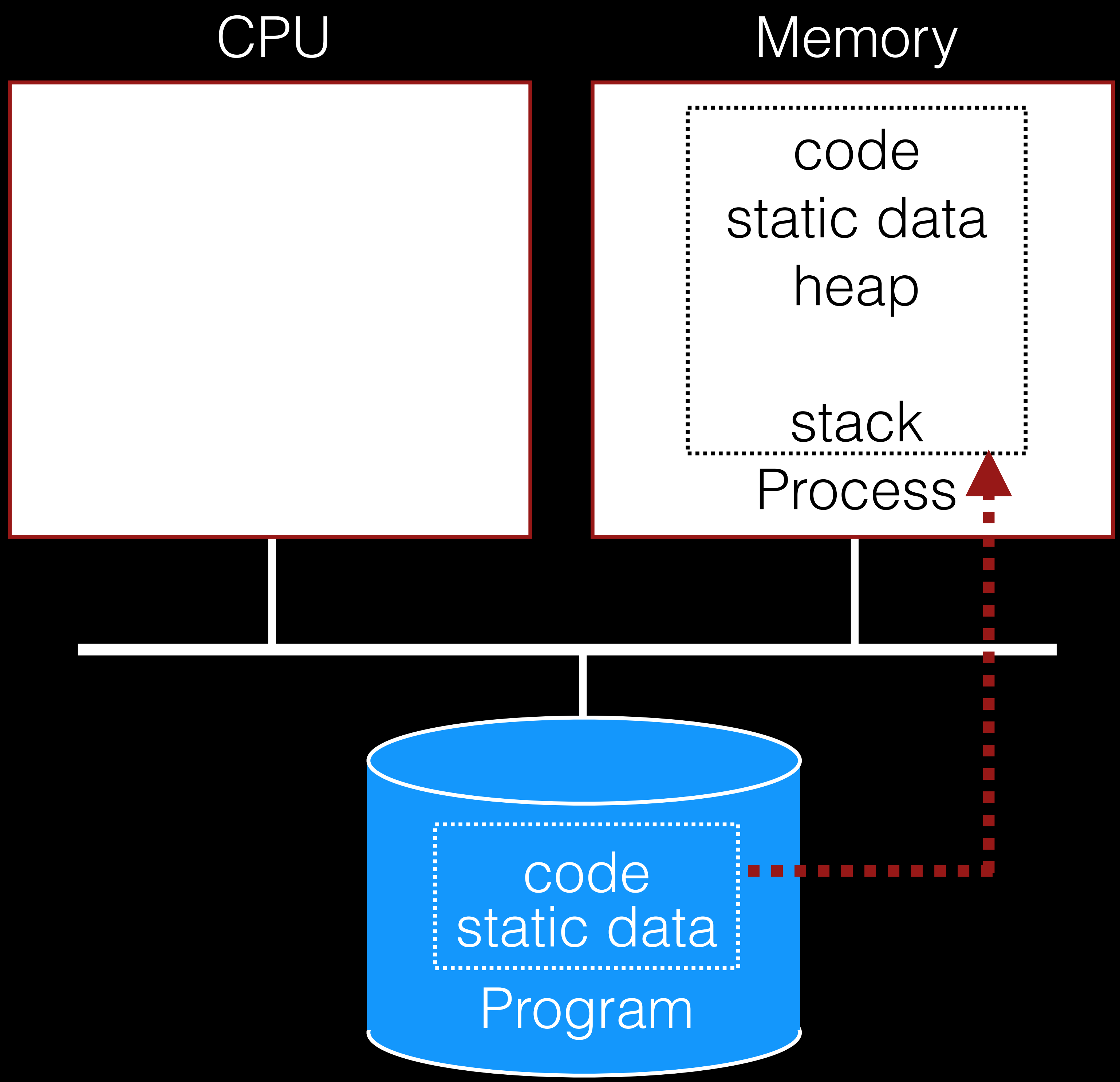
Programs are just code.
Processes are running programs.

A process is an instance of a program.
There may be 0 or more processes per program.

Process Creation



Process Creation



What’s in a Process?

Processes share code, but each has its own “context”

CPU

Instruction Pointer (aka Program Counter)

Stack Pointer

Memory

set of memory addresses (“address space”)

`cat /proc/<PID>/maps`

Disk

set of file despritors

`cat /proc/9506/fdinfo/*`

Do we enough CPUs?

Linux commands:

```
ps ax | wc

top

cat /proc/cpuinfo | grep 'model name'
```

How do we share?

CPU?

Memory?

Disk?

How do we share?

CPU? (a: time sharing)

Memory? (a: space sharing)

Disk? (a: space sharing)

How do we share?

CPU? (a: time sharing)

TODAY

Memory? (a: space sharing)

Disk? (a: space sharing)

How do we share?

CPU? (a: time sharing)

TODAY

Memory? (a: space sharing)

Disk? (a: space sharing)

Goal: processes should NOT even 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` (file descriptors)
- `state` (sleeping, running, etc)

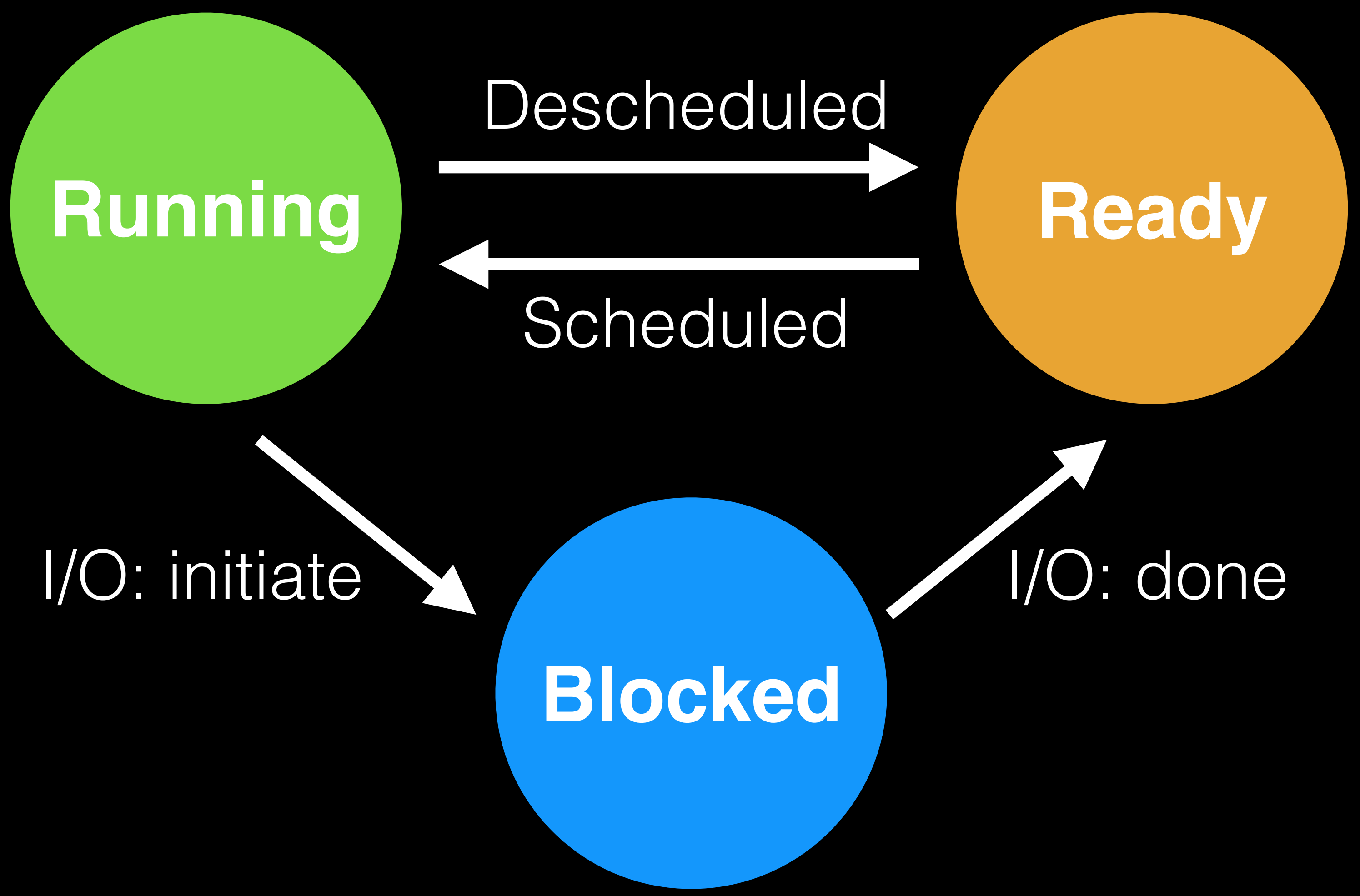
What to Do with Processes That Are Not Running?

A: store context in OS struct

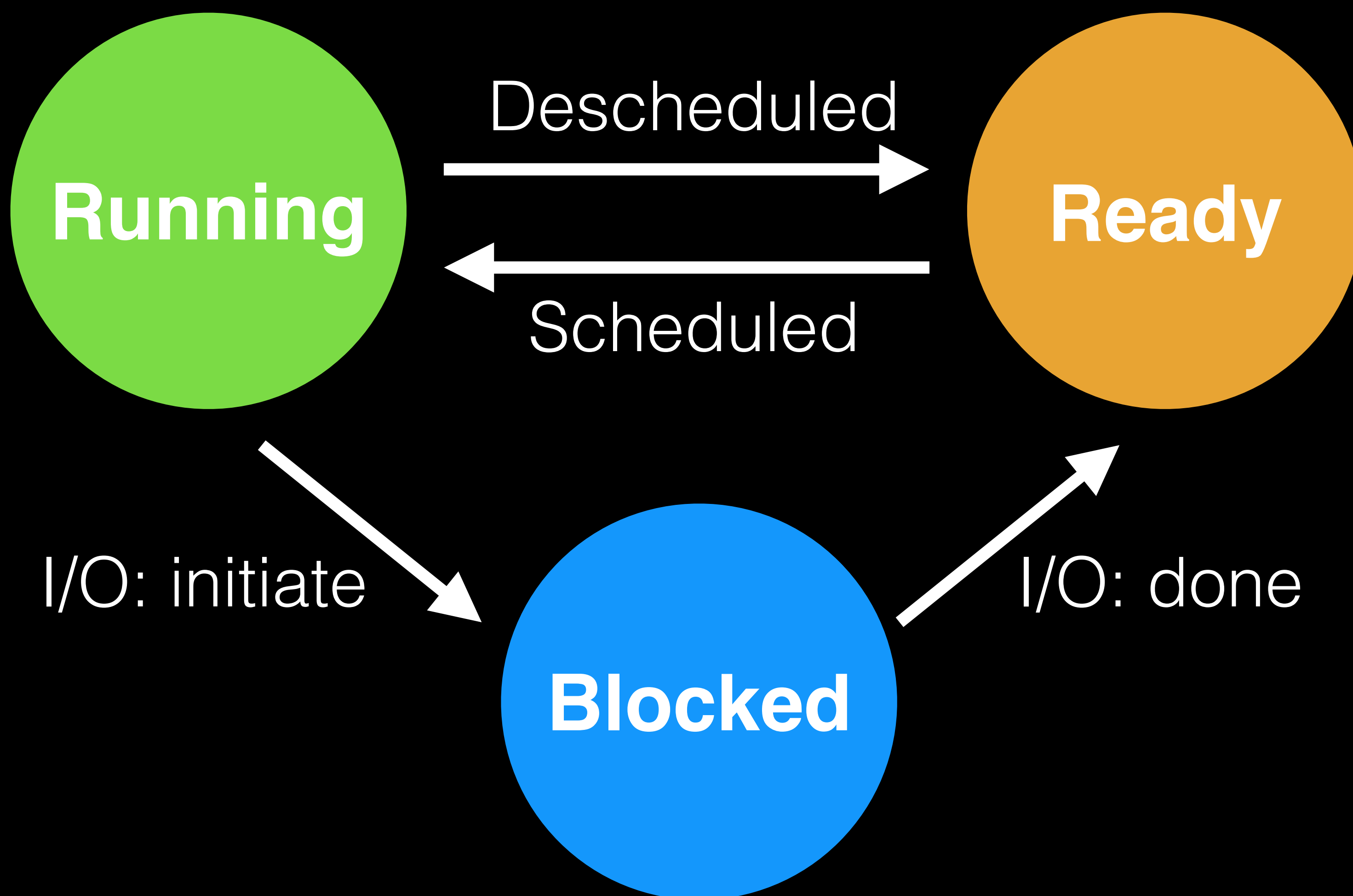
Look in kernel/proc.h

- `context` (CPU registers)
- `ofile` (file descriptors)
- `state` (sleeping, running, etc)

State Transitions

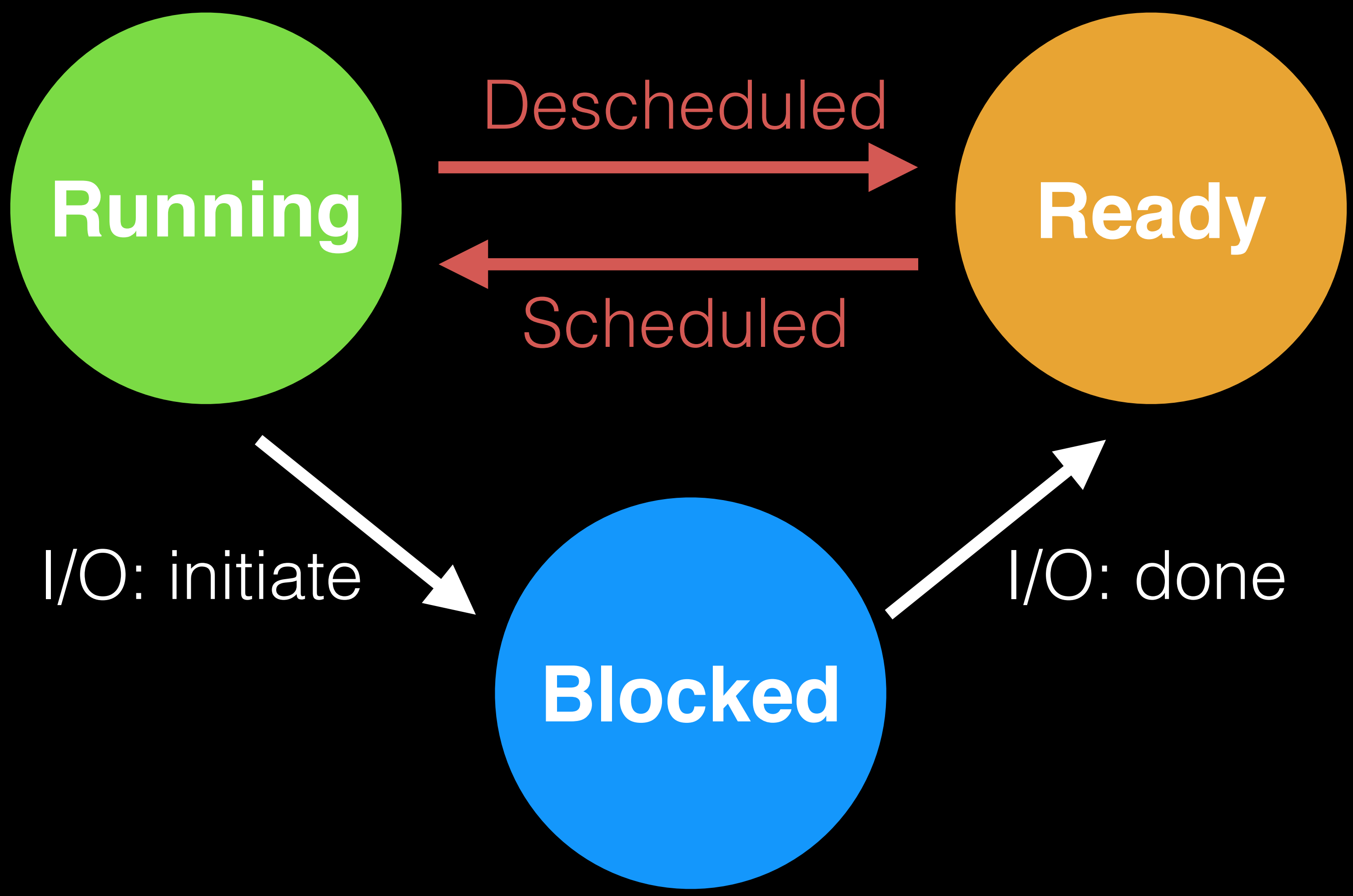


State Transitions



View process state with “ps xa”

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



Administrative Stuff

- P1 due on 9/16 (eight days left!)
- Office hours: today after class (in lab), Wed 2-3pm
- Exam prep: understand book and exams
- Reading: chapters 1-2 (last time) and 3-6 (today)
- Learning names
- Wait list: good news!

CPU Time Sharing

Goal 1: efficiency

OS should have minimal overheard

Goal 2: control

Processes shouldn't do anything bad

OS should decide when processes run

Solution: limited direct execution

Limited Direct Execution

What to limit?

General memory access

Disk I/O

Special x86 instructions like `lidt`

How? Get HW help, put processes in “user mode”

What to limit?

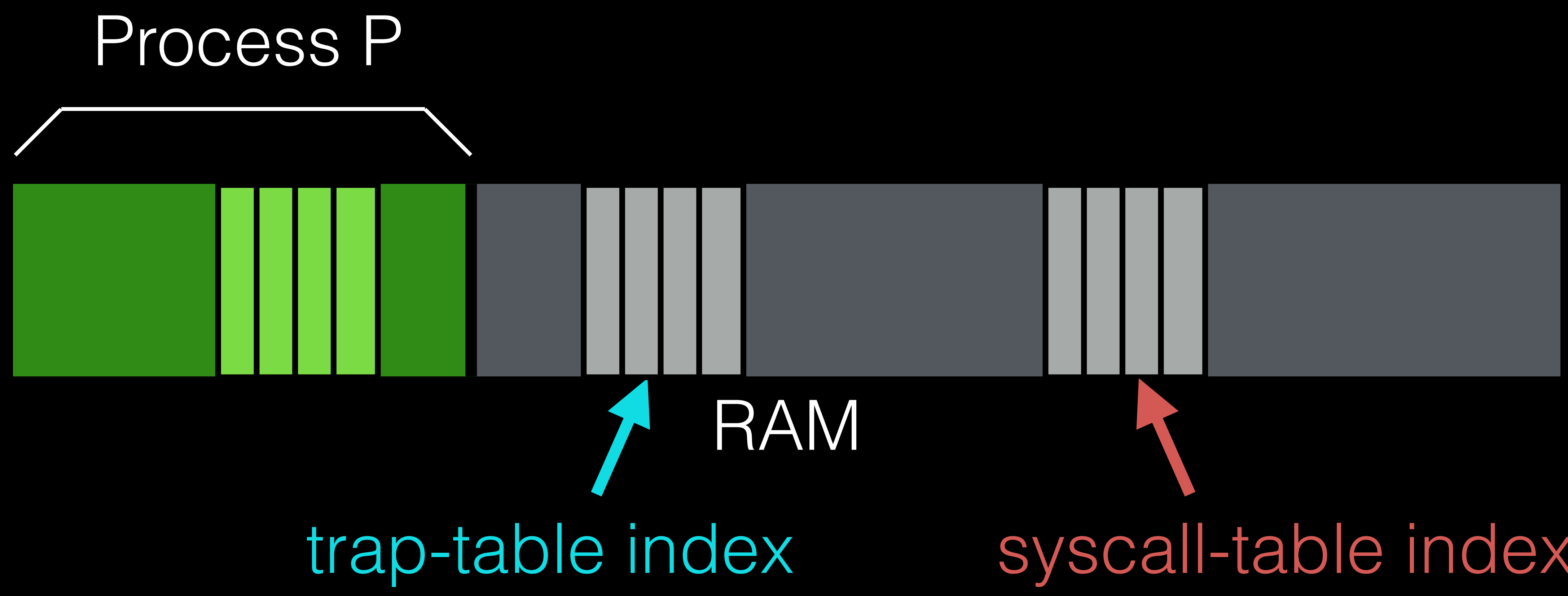
General memory access

Disk I/O

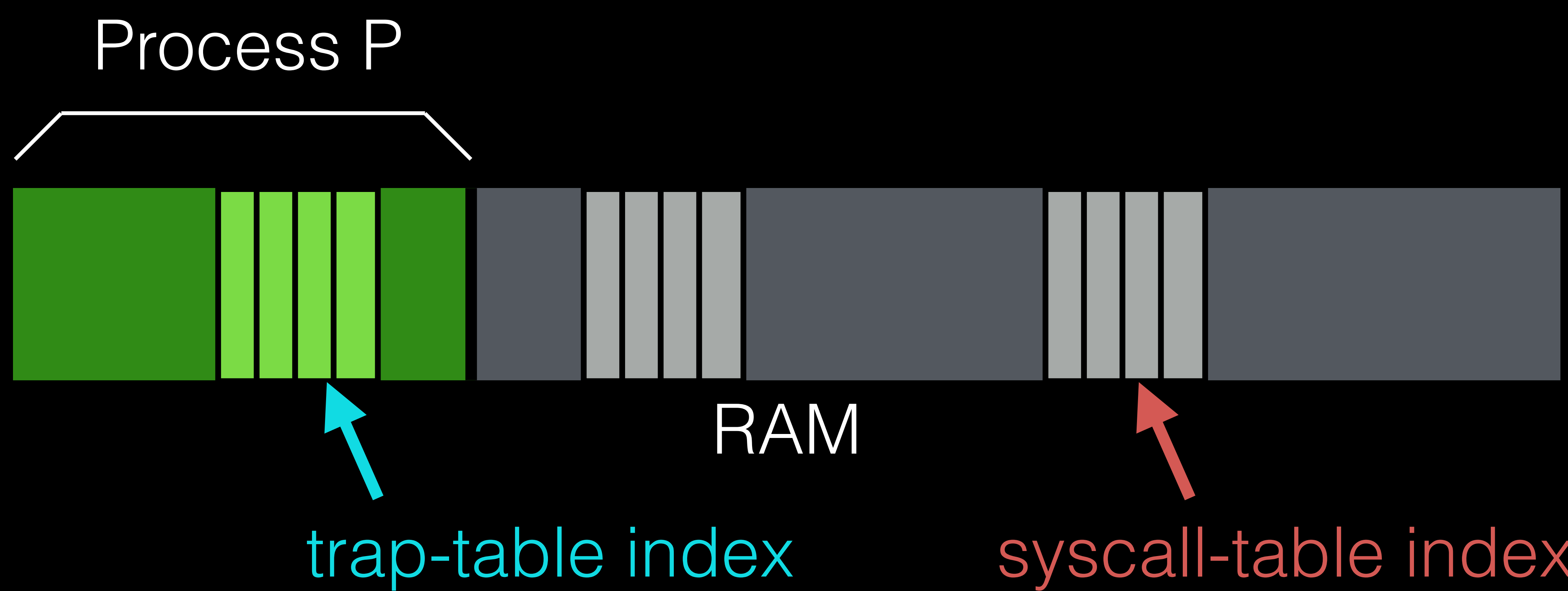
Special x86 instructions like `lidt`

How? Get HW help, put processes in “user mode”

`lidt` example



`lidt` example



P tries to call `lidt`!

`lidt` example

CPU warns OS, OS kills P

Context Switch

Problem: when to switch process contexts?

Direct execution => OS can't run while process runs

How can the 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

How can the 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

How can the OS do anything while it's not running?
A: it can't

Solution: **switch on interrupts**. But which interrupt?

Cooperative Approach

Switch contexts for syscall interrupt.

Provide special `yield()` system call.

Cooperative Approach

Switch contexts for syscall interrupt.

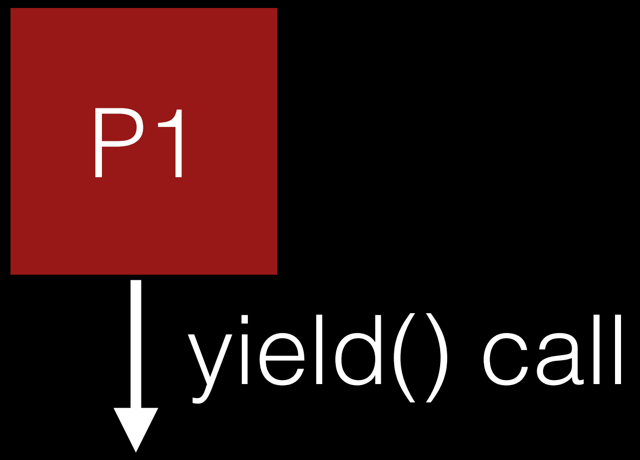
Provide special **`yield()`** system call.

P1

Cooperative Approach

Switch contexts for syscall interrupt.

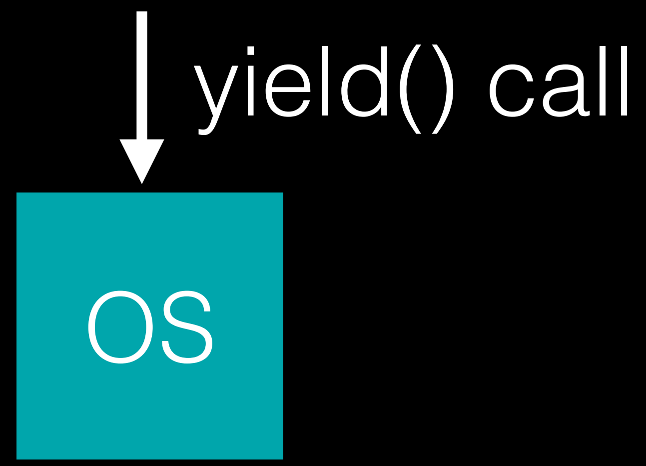
Provide special **`yield()`** system call.



Cooperative Approach

Switch contexts for syscall interrupt.

Provide special **`yield()`** system call.



Cooperative Approach

Switch contexts for syscall interrupt.

Provide special **`yield()`** system call.

OS

Cooperative Approach

Switch contexts for syscall interrupt.

Provide special **`yield()`** system call.

Cooperative Approach

Switch contexts for syscall interrupt.

Provide special **`yield()`** system call.

Cooperative Approach

Switch contexts for syscall interrupt.

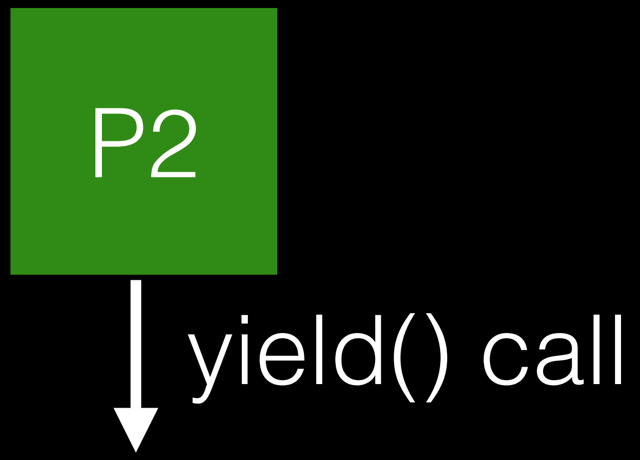
Provide special **`yield()`** system call.

P2

Cooperative Approach

Switch contexts for syscall interrupt.

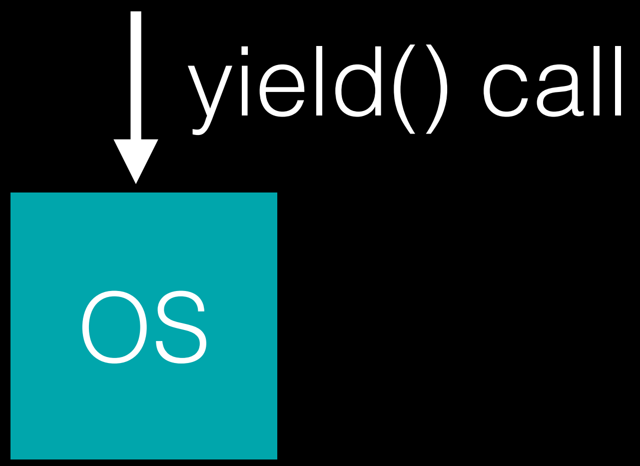
Provide special **`yield()`** system call.



Cooperative Approach

Switch contexts for syscall interrupt.

Provide special **`yield()`** system call.



Cooperative Approach

Switch contexts for syscall interrupt.

Provide special `yield()` system call.

OS

Cooperative Approach

Switch contexts for syscall interrupt.

Provide special **`yield()`** system call.

Cooperative Approach

Switch contexts for syscall interrupt.

Provide special **`yield()`** system call.

Cooperative Approach

Switch contexts for syscall interrupt.

Provide special **`yield()`** system call.

Non-Cooperative Approach

Switch contexts on timer interrupt.

Set up before running any processes.

HW does not let processes prevent this.

Is it better to be cooperative or non-cooperative?

Operating System	Hardware	Program
		Process A …

Operating System	Hardware	Program
		Process A …
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	

Operating System	Hardware	Program
		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 return-from-trap (into B)		

Operating System	Hardware	Program
		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 return-from-trap (into B)		
	restore regs(B) from k-stack(B) move to user mode jump to B's IP	

Operating System	Hardware	Program
		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 return-from-trap (into B)		
	restore regs(B) from k-stack(B) move to user mode jump to B’s IP	
		Process B …

Summary

- Smooth context switching makes each process think it has its own CPU (virtualization!)
- Direct execution makes processes fast
- Hardware provides a lot of OS support
  - limited direct execution
  - timer interrupts
  - automatic register saving

Things to Look Forward to

- CPU-sharing policy (Wed lecture)
- Process APIs (Thu discussion)
  Also: syscall timing and more C review
- Memory virtualization (next Mon lecture)