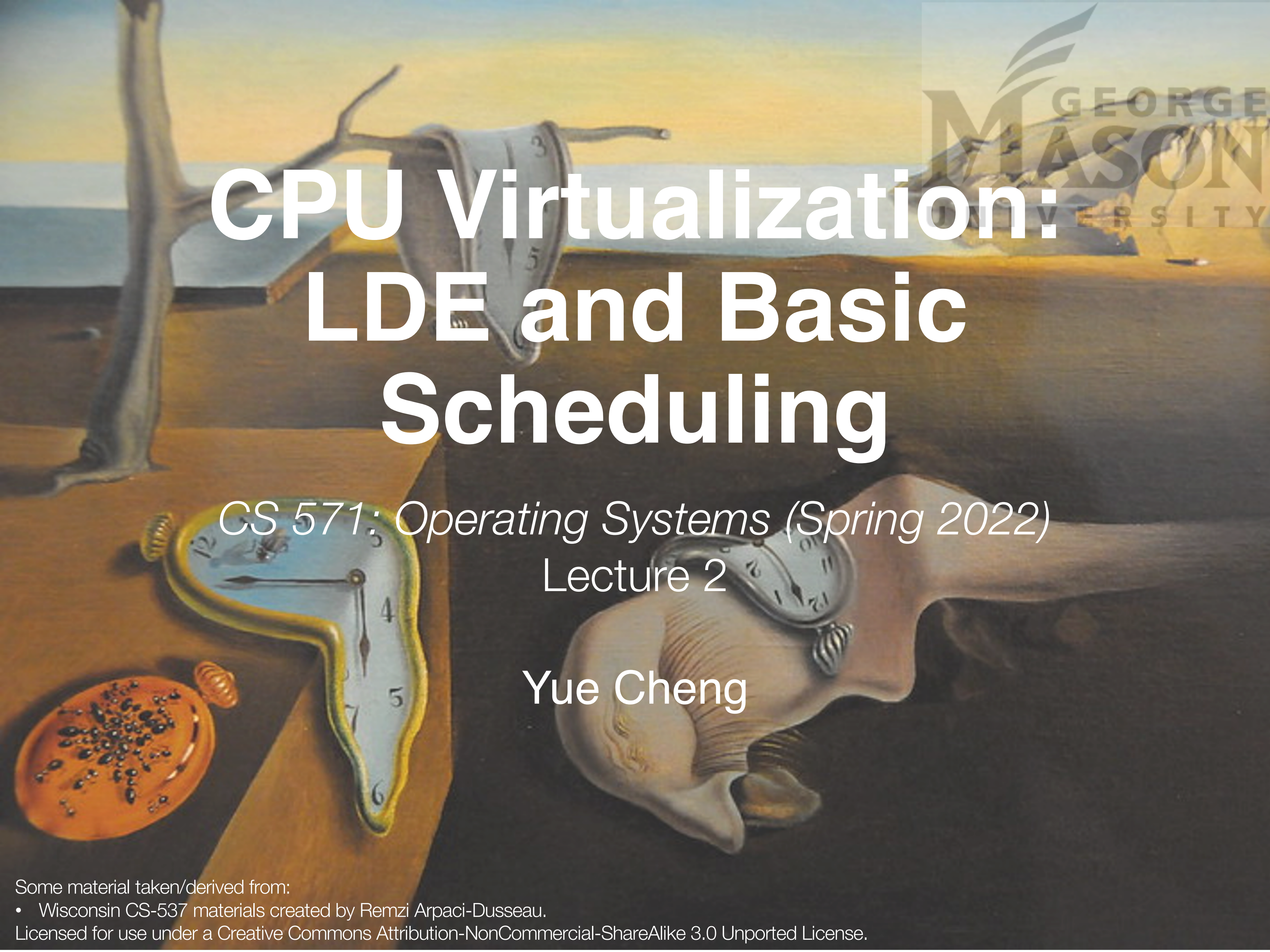


CPU Virtualization: LDE and Basic Scheduling

CS 571: Operating Systems (Spring 2022)

Lecture 2

Yue Cheng

Some material taken/derived from:

- Wisconsin CS-537 materials created by Remzi Arpaci-Dusseau.

CPU Virtualization: Outline

- Limited Direct Execution (LDE)
- Basic scheduling concept and criteria
- Basic scheduling algorithms
  - First In, First Out (FIFO)
  - Shortest Job First (SFJ)
  - Shortest Time-to-Completion First (STCF)
  - Round Robin (RR)

CPU Virtualization: Outline

- Limited Direct Execution (LDE)
- Basic scheduling concept and criteria
- Basic scheduling algorithms
  - First In, First Out (FIFO)
  - Shortest Job First (SFJ)
  - Shortest Time-to-Completion First (STCF)
  - Round Robin (RR)

Limited Direct Execution (LDE)

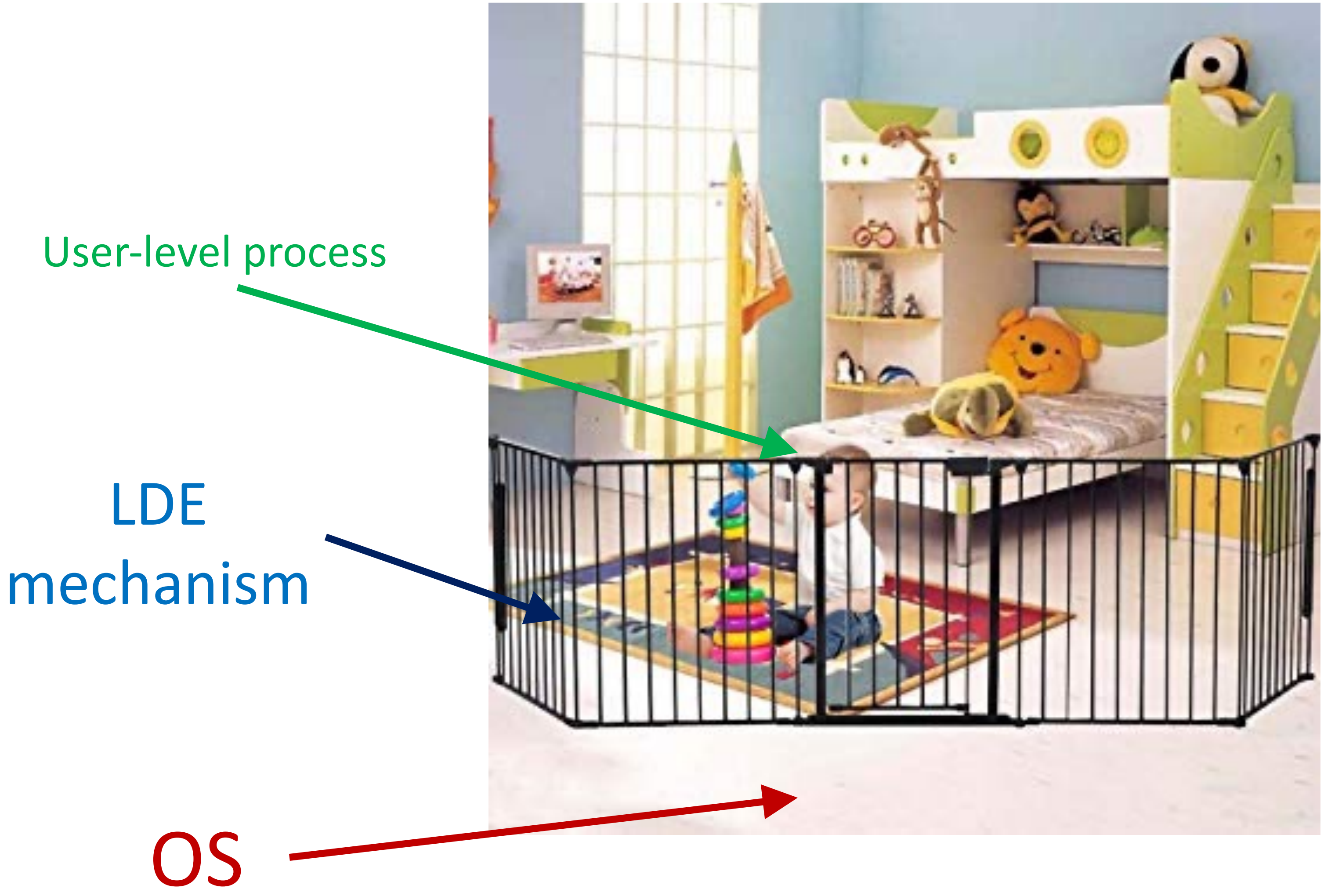
Limited Direct Execution (LDE)

- 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 allocation requests
- 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

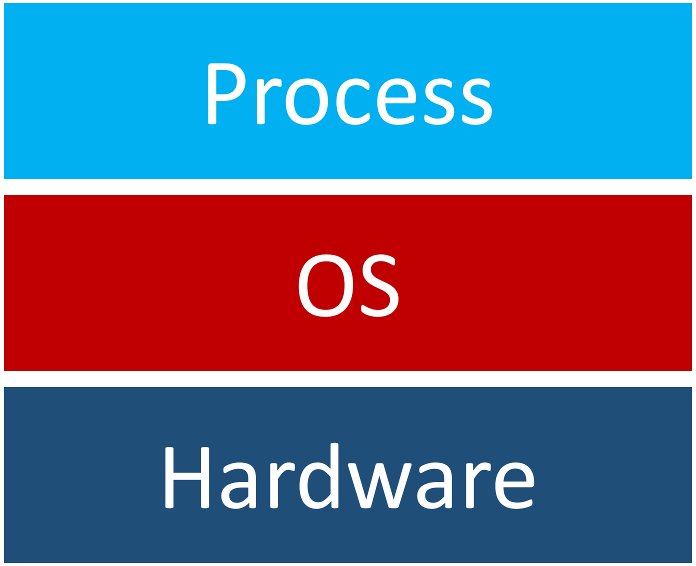
LDE: Remaining Challenges

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

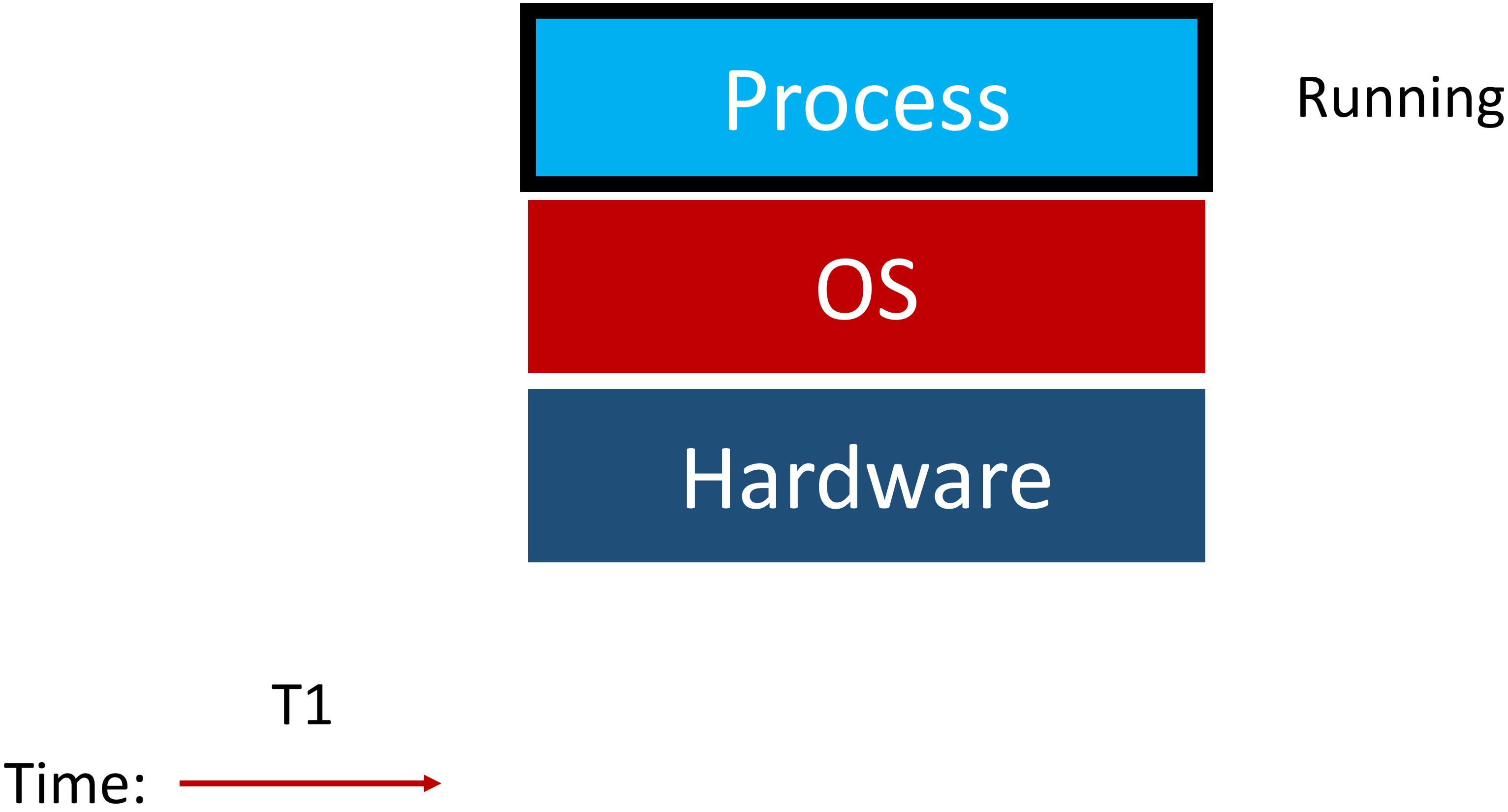


Process

OS

Hardware

Taking Turns



Process

OS

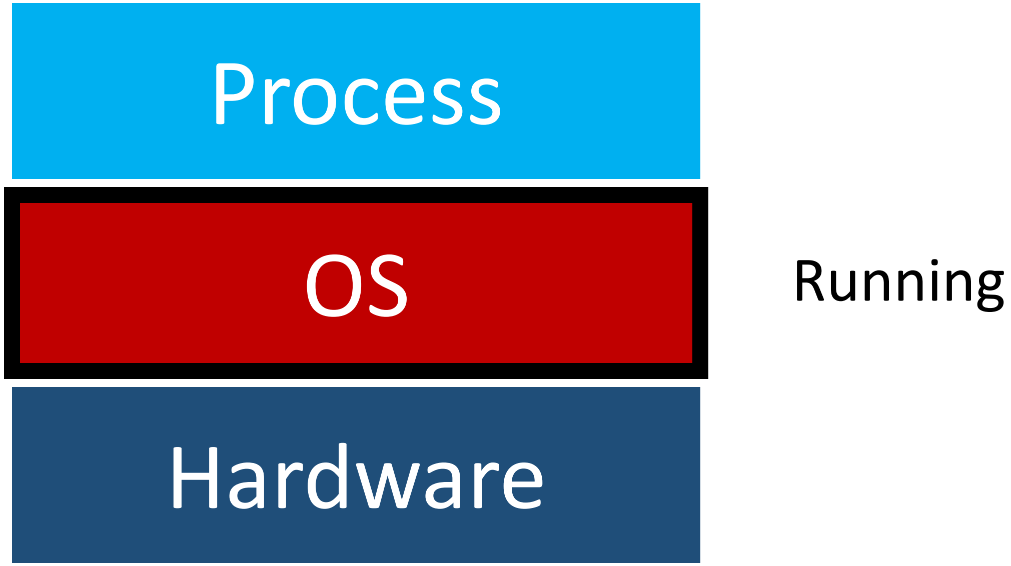
Hardware

Running

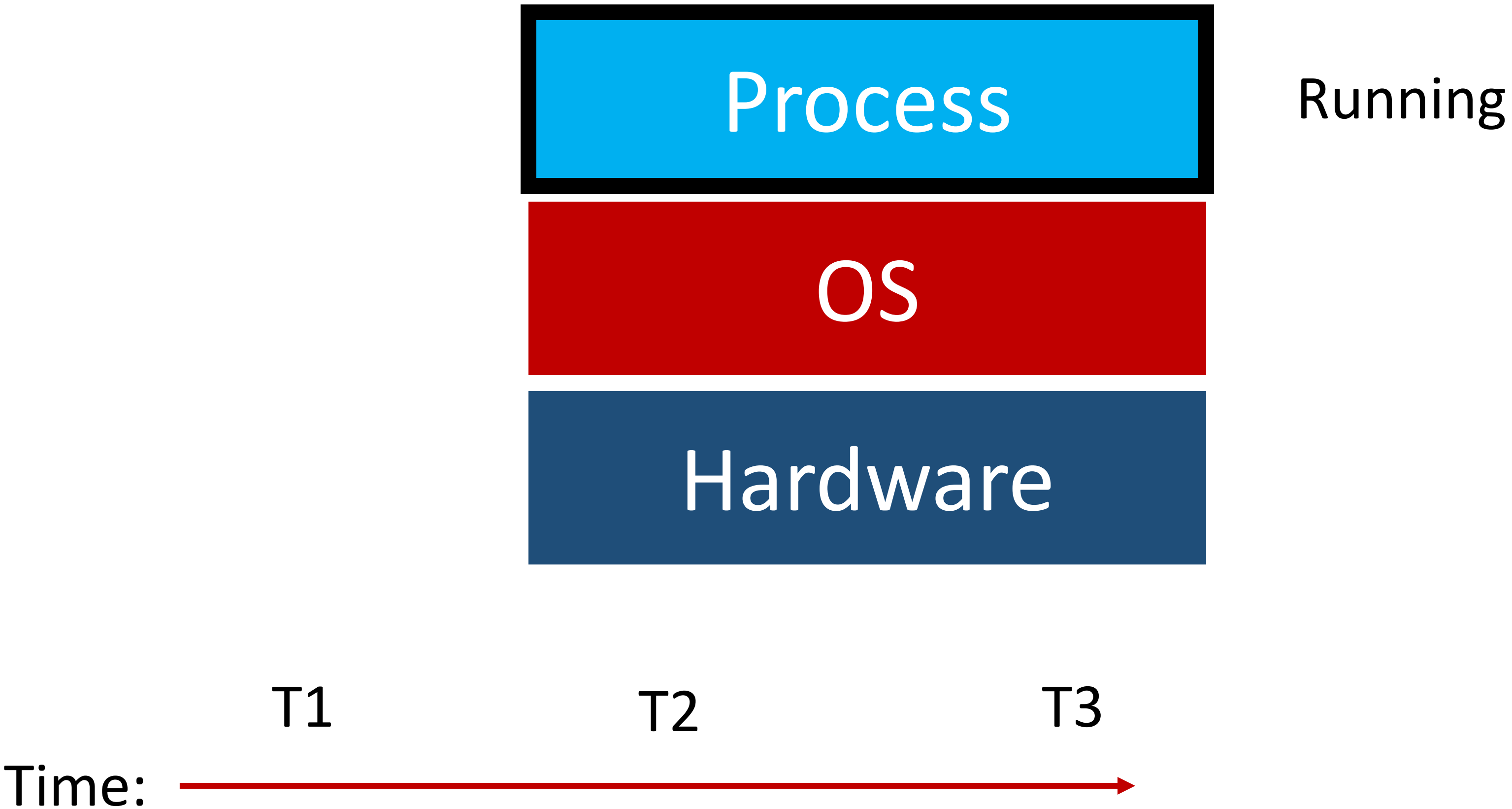
T1

Time:

Taking Turns



Taking Turns



Process

OS

Hardware

Running

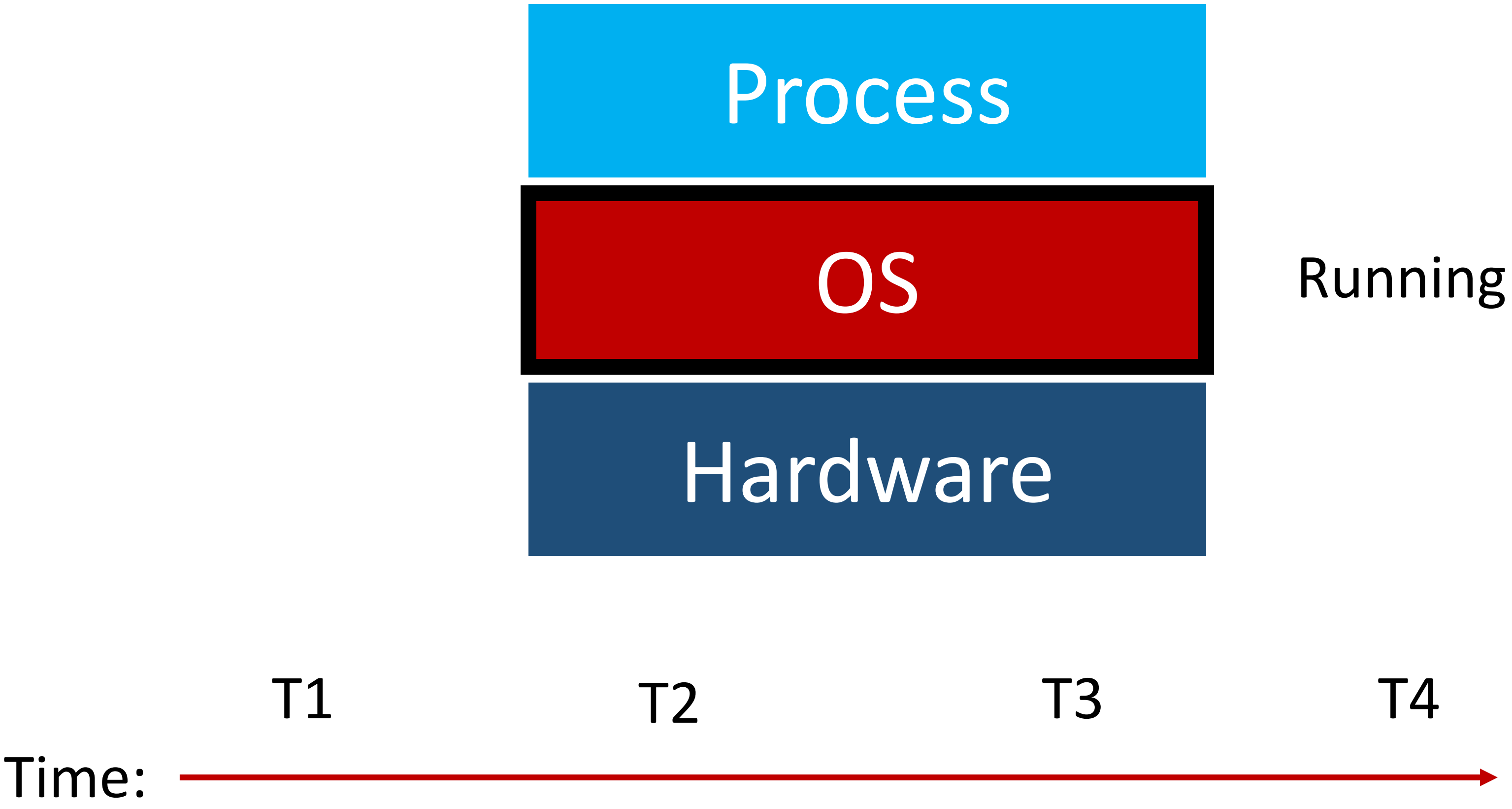
T1

T2

T3

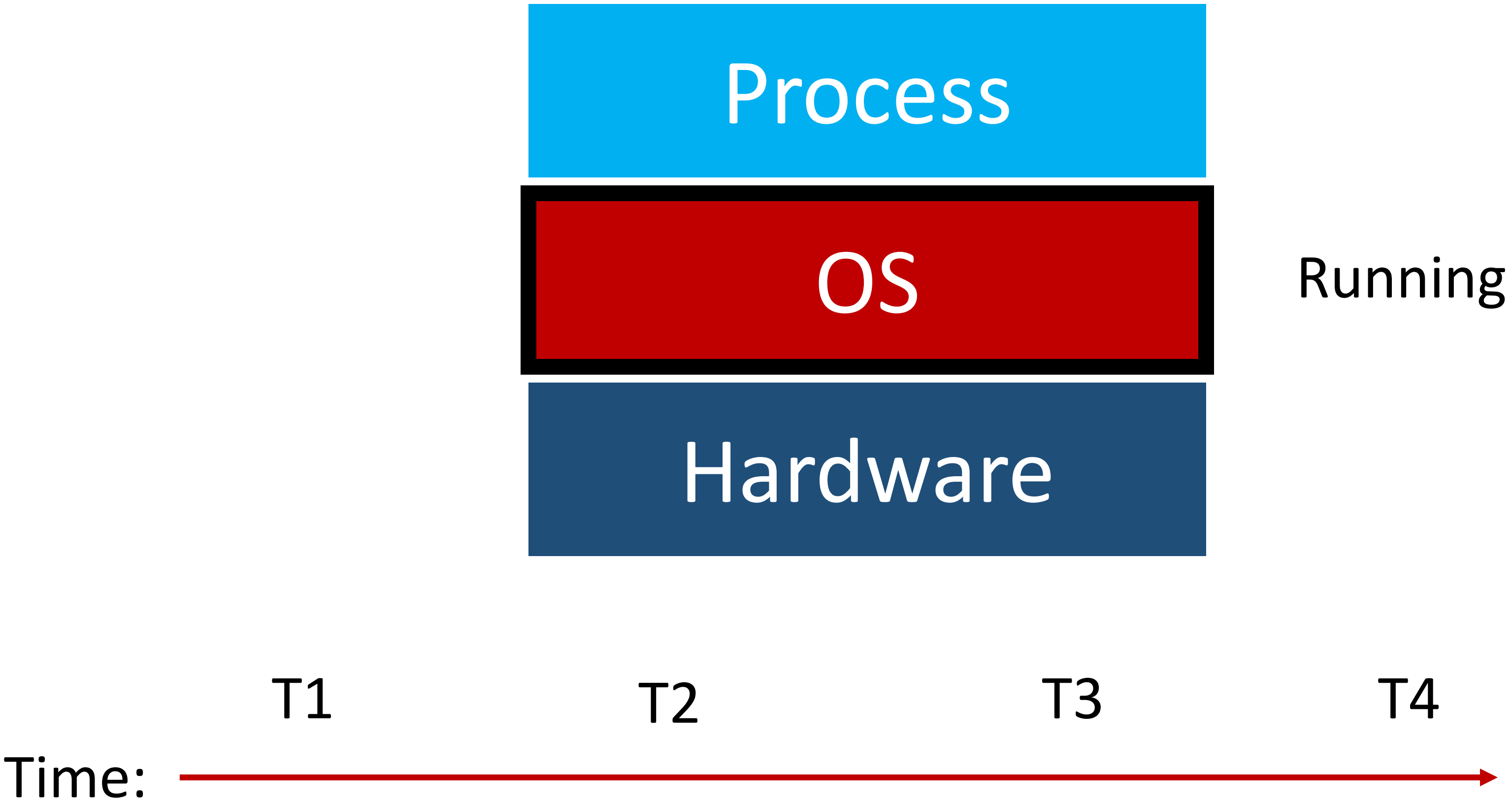
Time:

Taking Turns



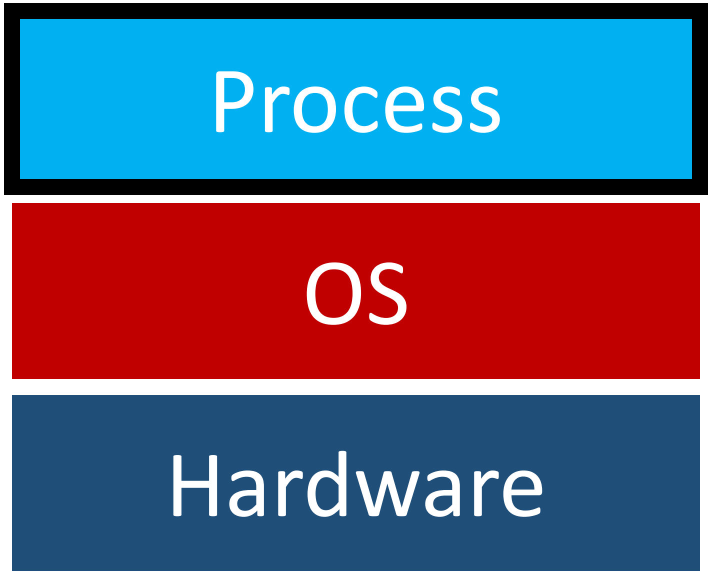
Taking Turns

Question: when/how do we switch to OS?

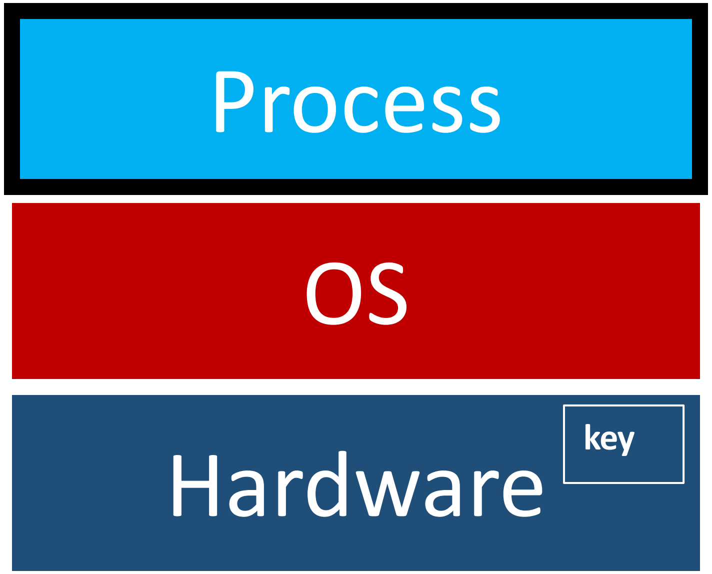


Exceptions

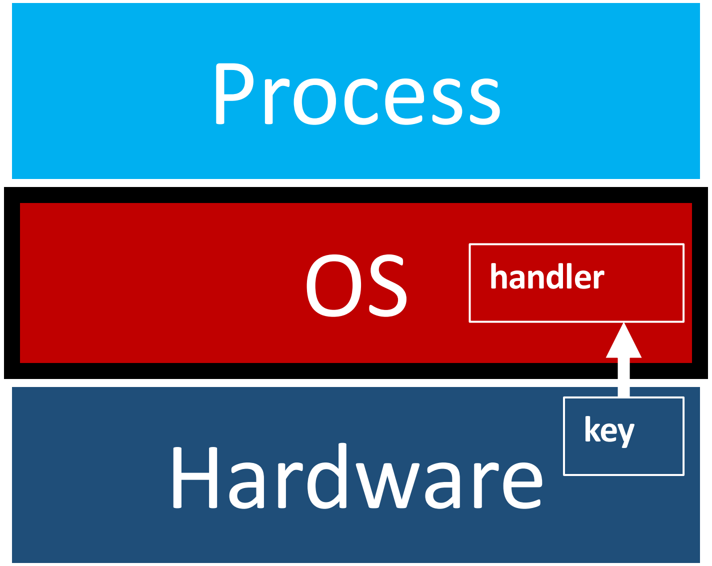
Interrupt



Interrupt

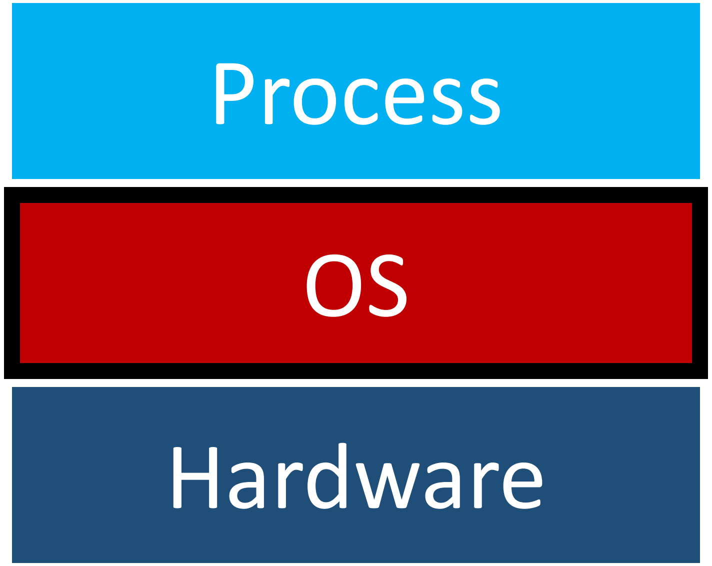


Interrupt

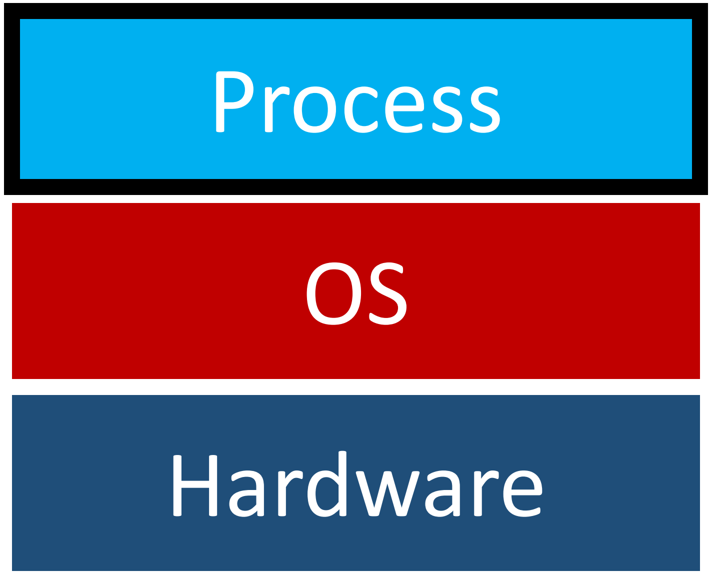


Hardware interrupt

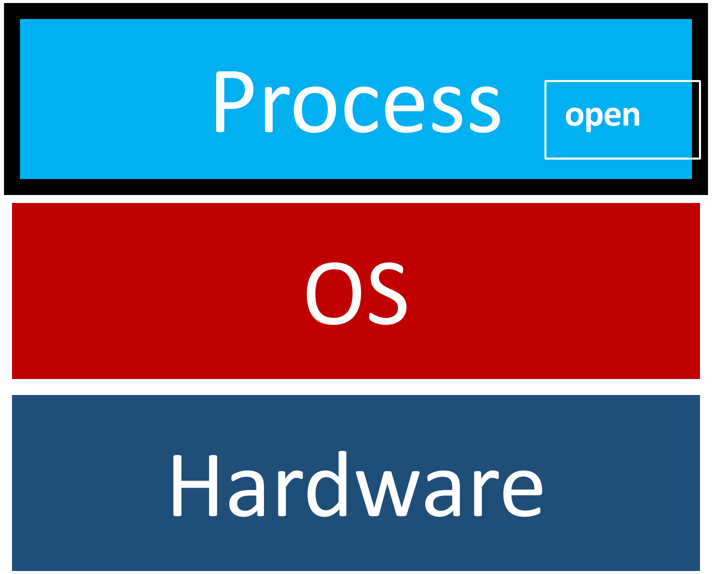
Interrupt



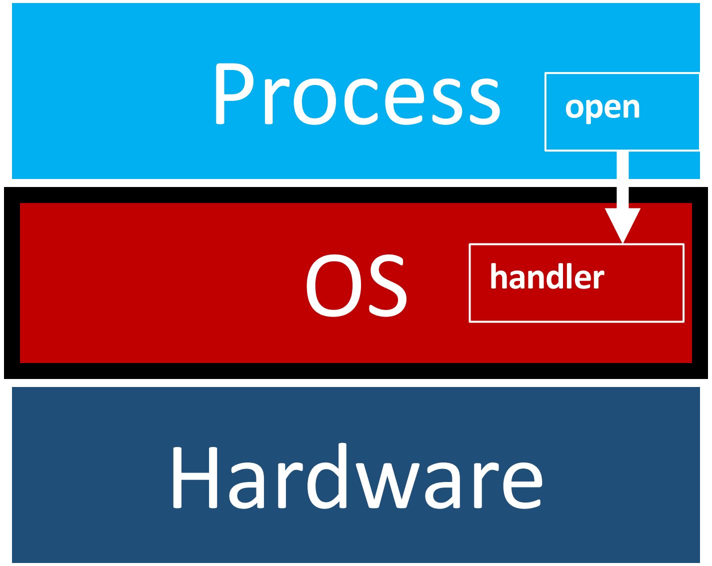
System Call



System Call

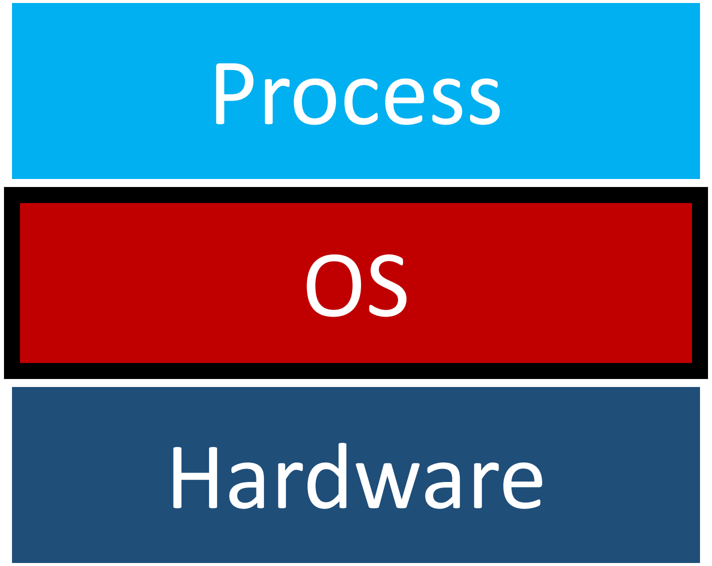


System Call



System call "trap"

System Call

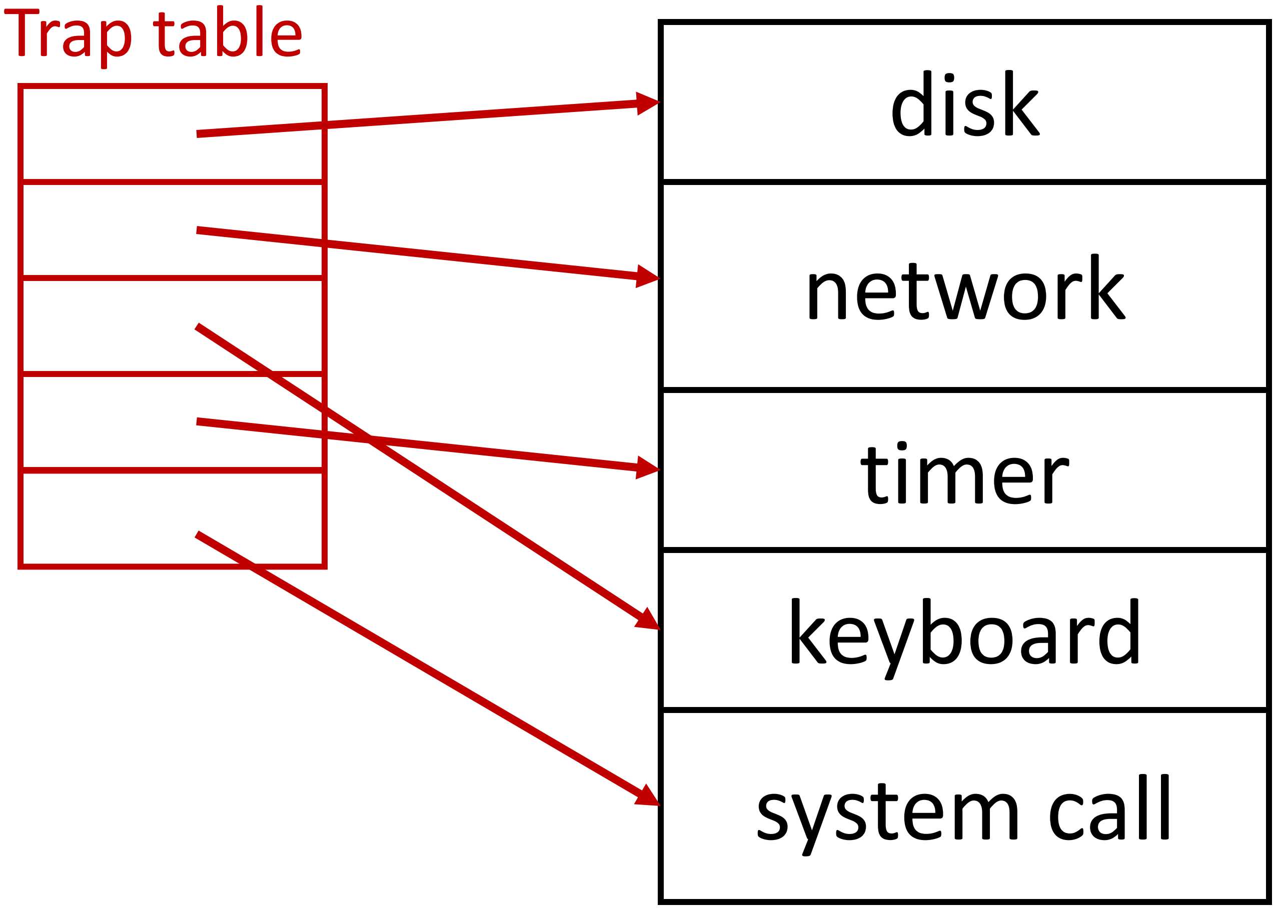


Exception Handling

Exception Handling: Implementation

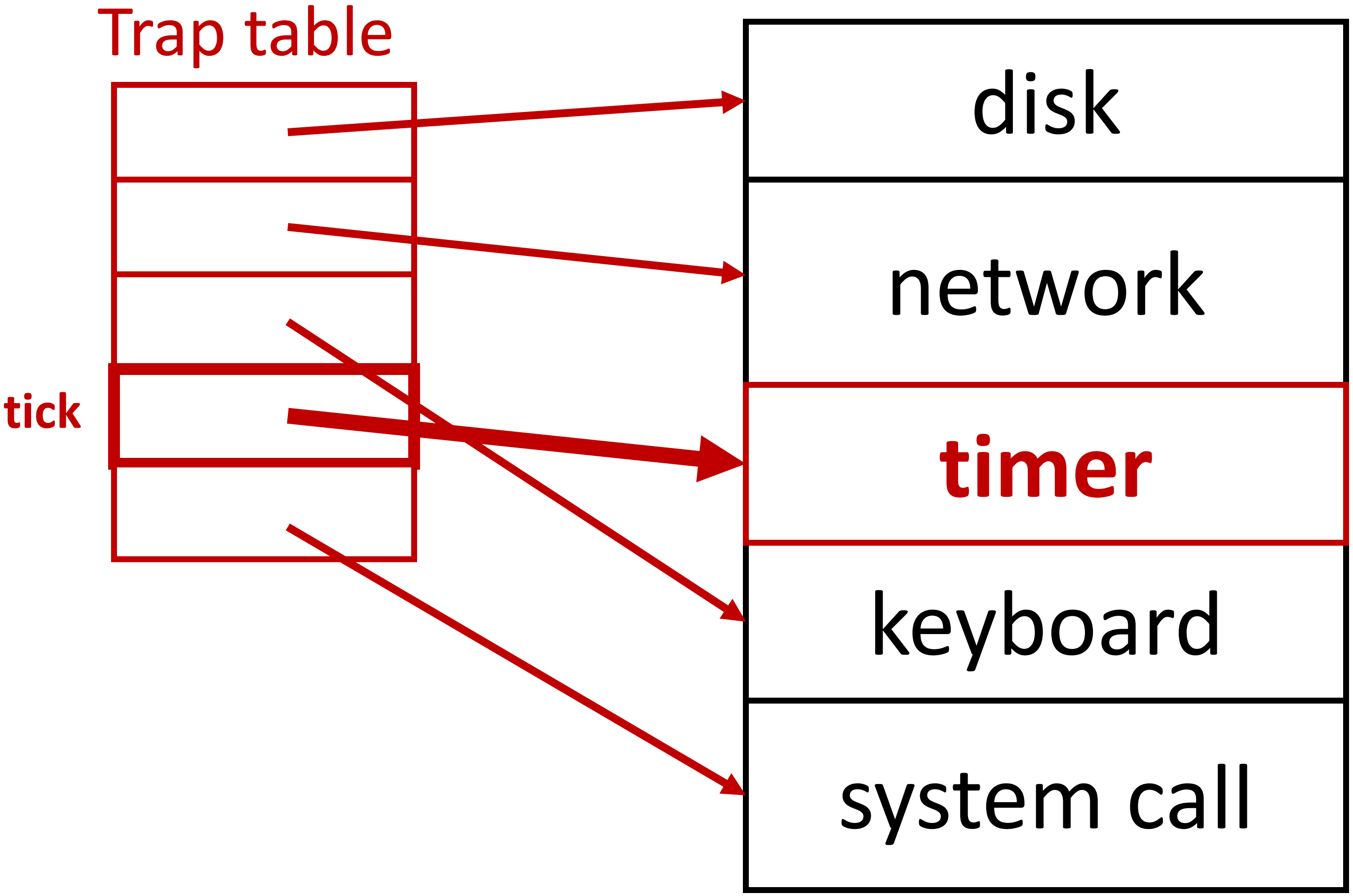
- Goal: Processes and hardware should be able to call functions in the OS

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

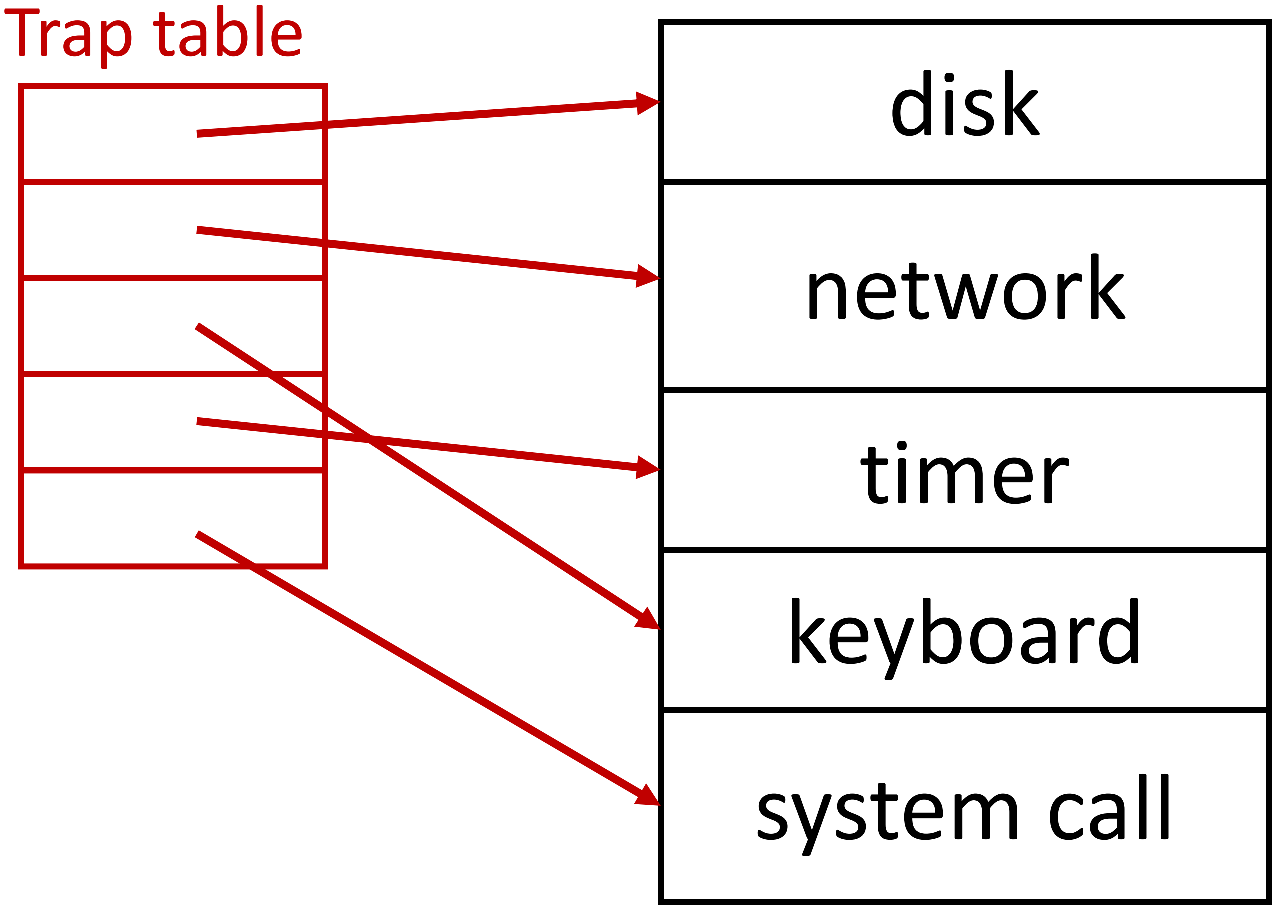


Use array of function pointers to locate OS functions

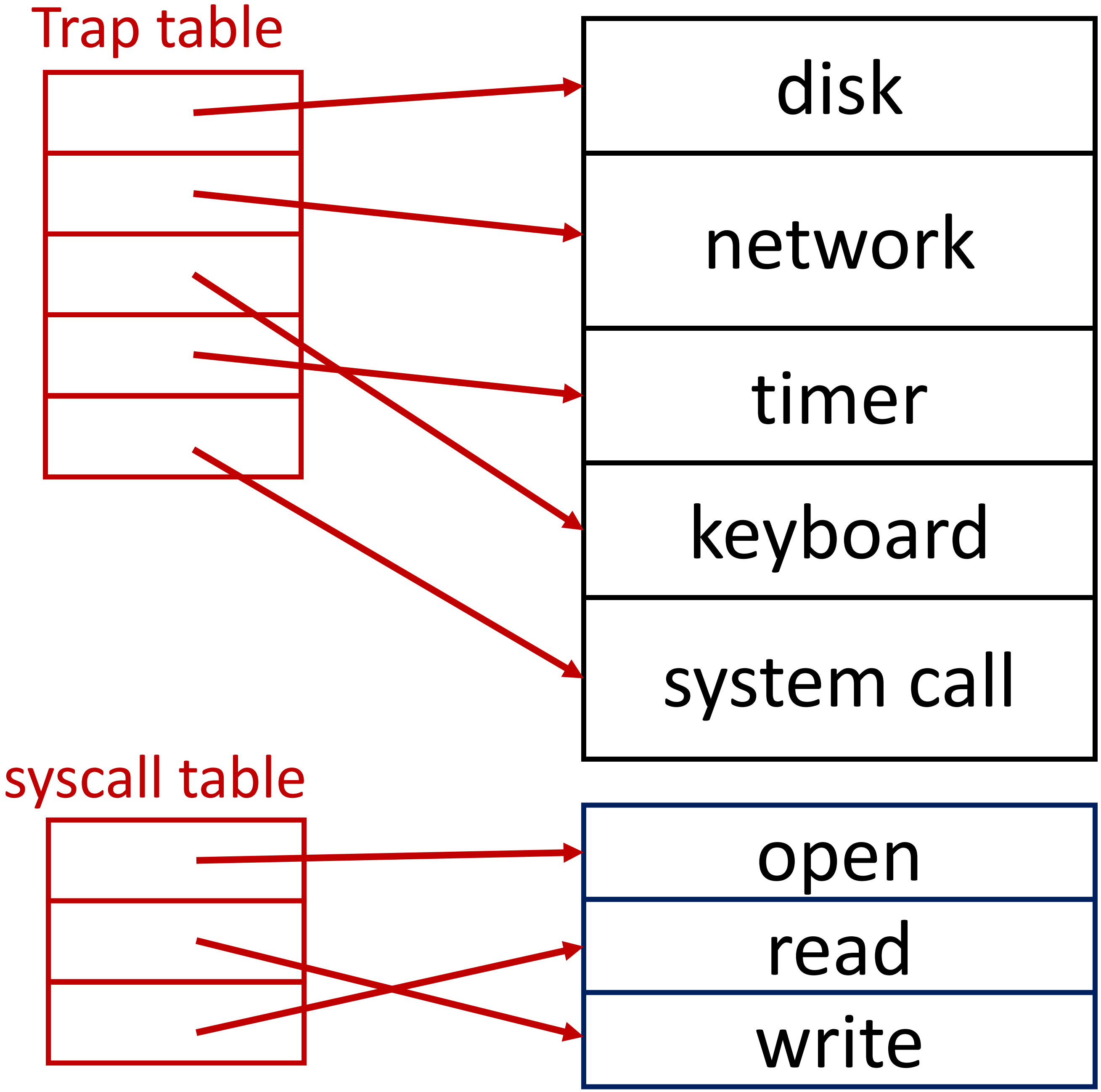
(**Hardware knows where this is**)

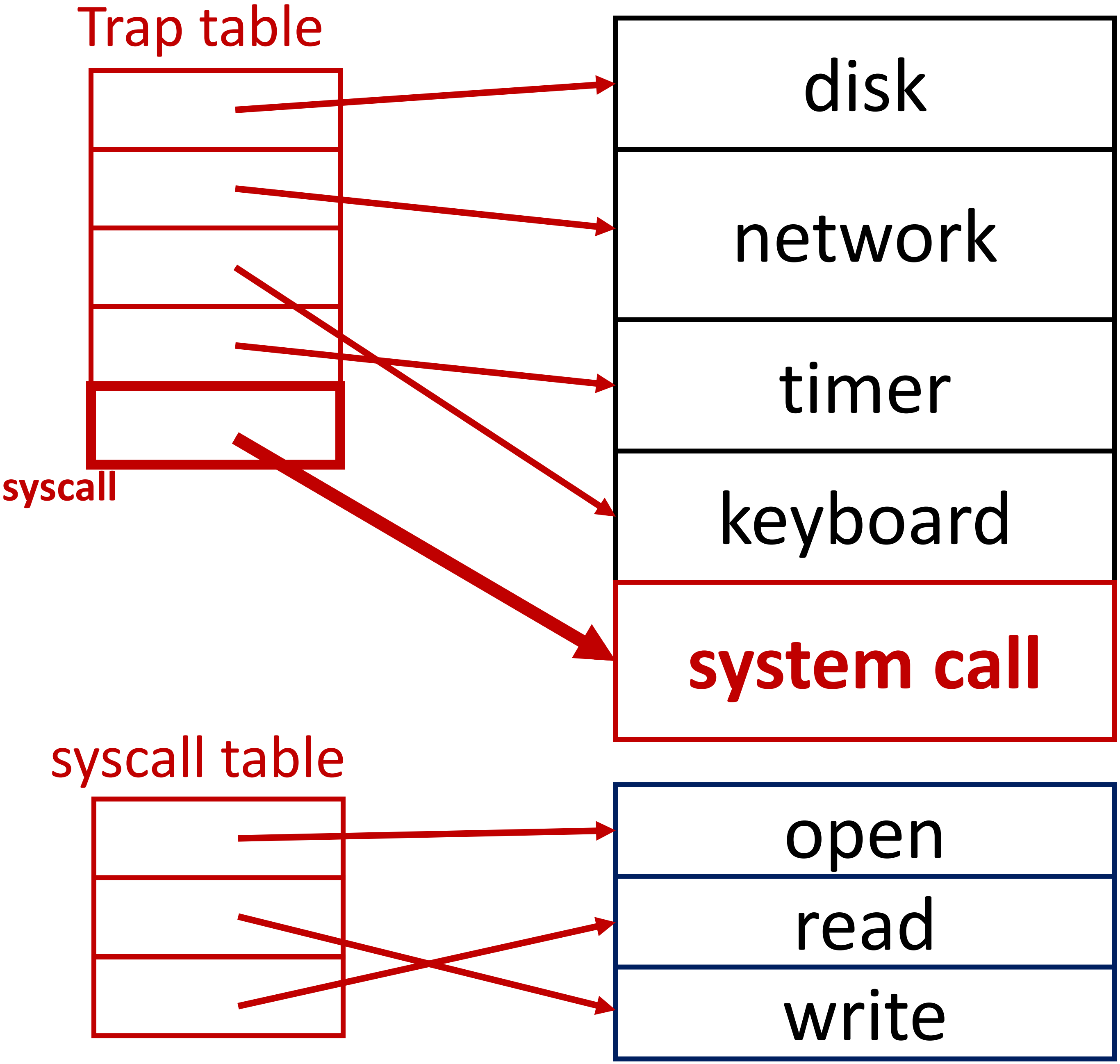


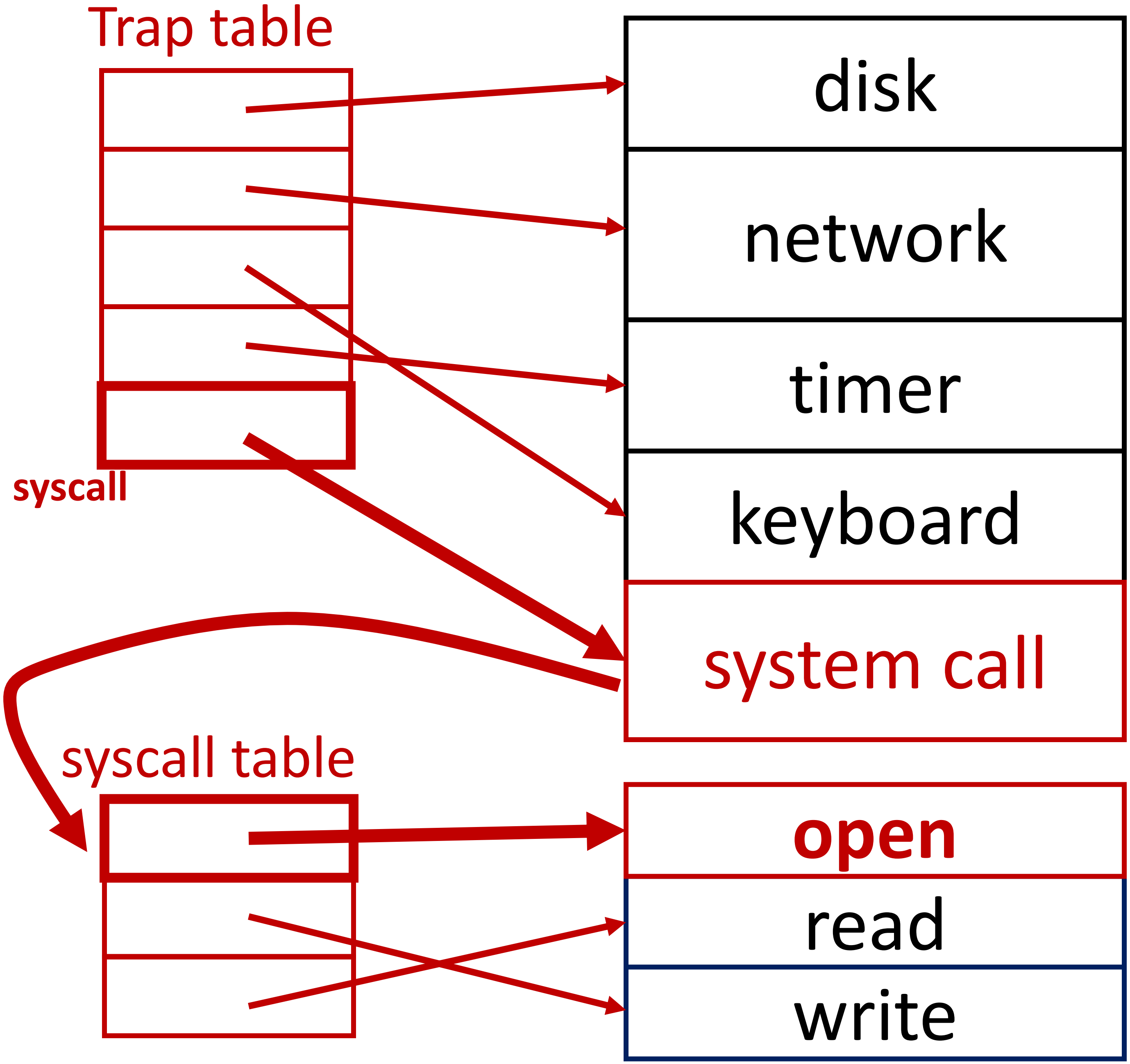
Use array of function pointers to locate OS functions
(Hardware knows this through `lidt` instruction)



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

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

What to do with processes that are not running?

- A: Store context in OS struct

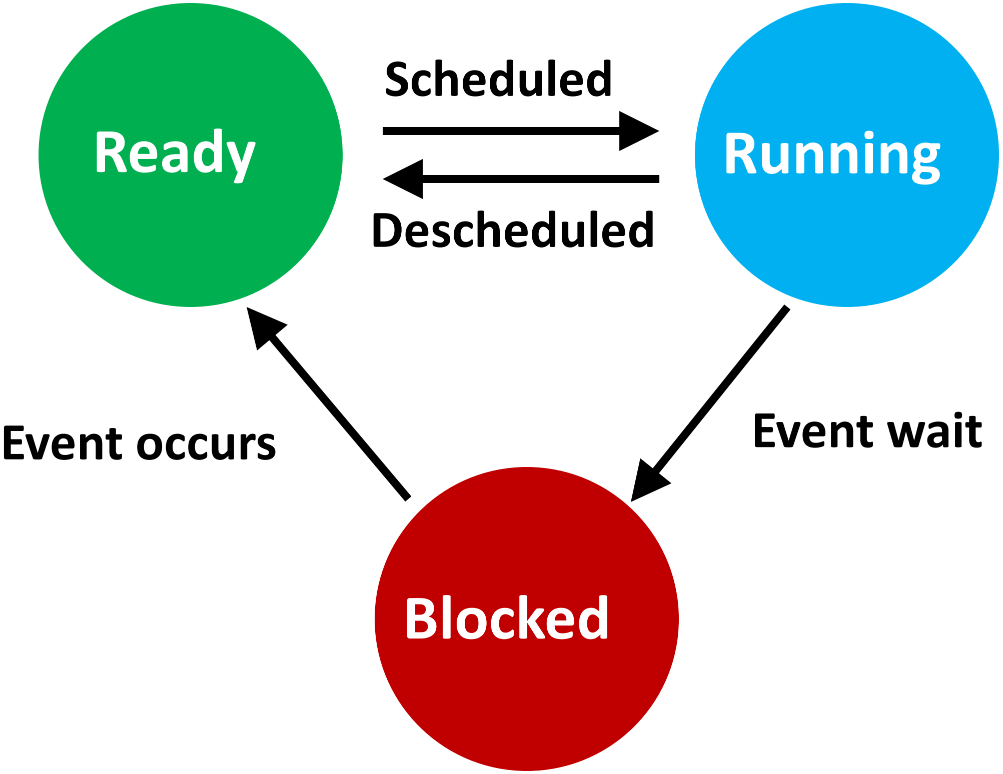
- Context:
  - CPU registers
  - Open file descriptors
  - State

What to do with processes that are not running?

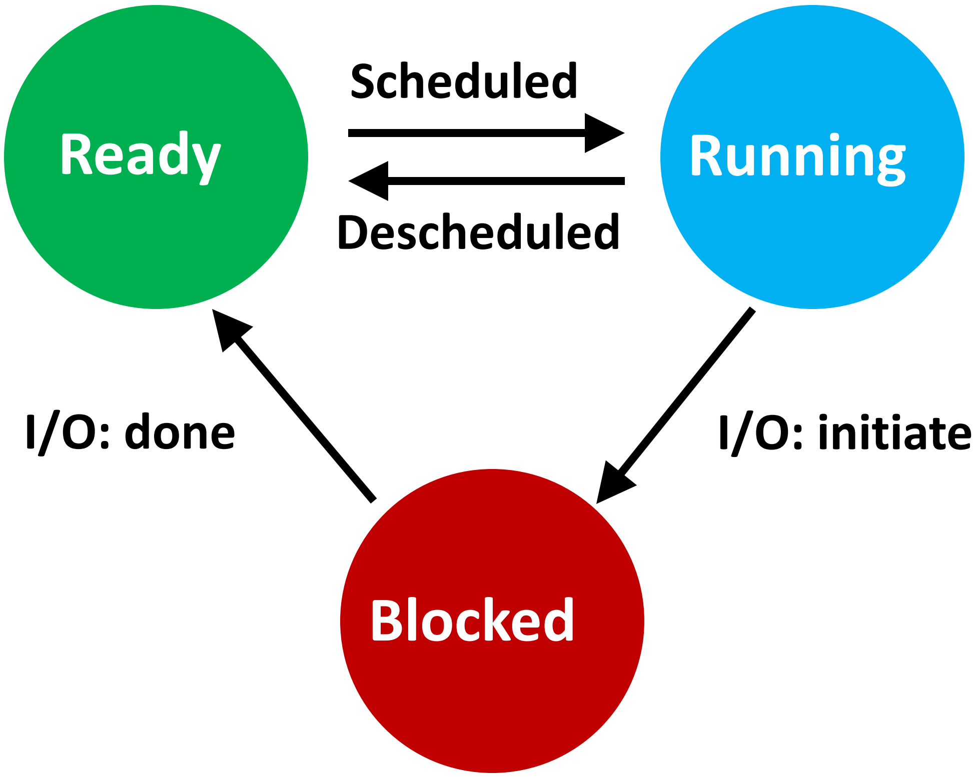
- A: Store context in OS struct

- Context:
  - CPU registers
  - Open file descriptors
  - State

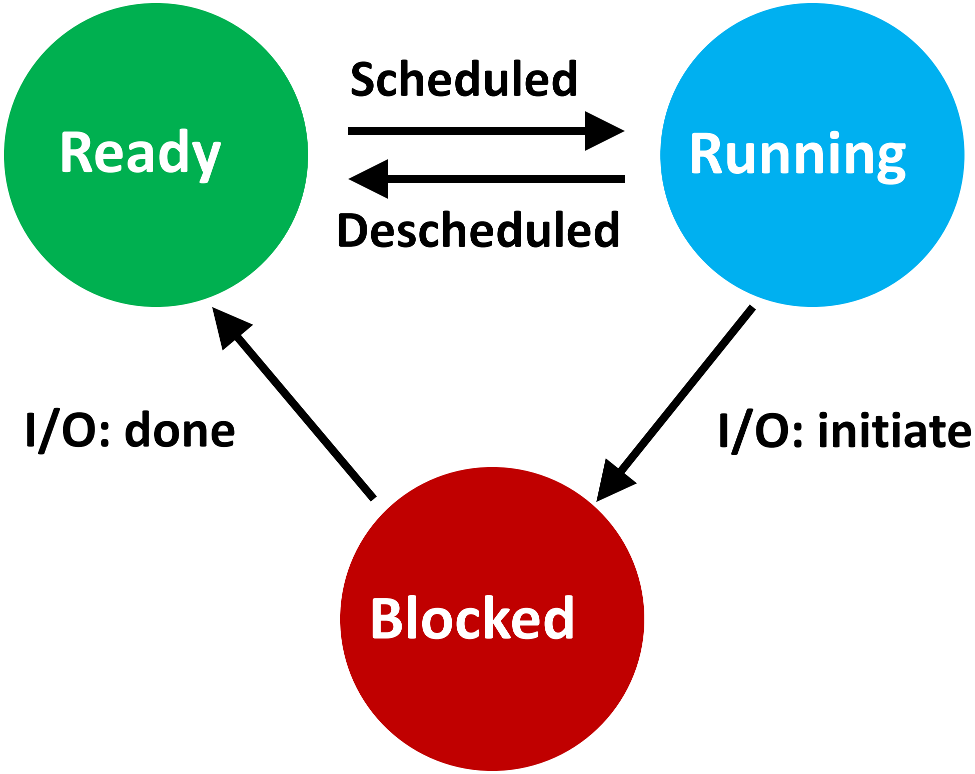
Process State Transitions



Process State Transitions

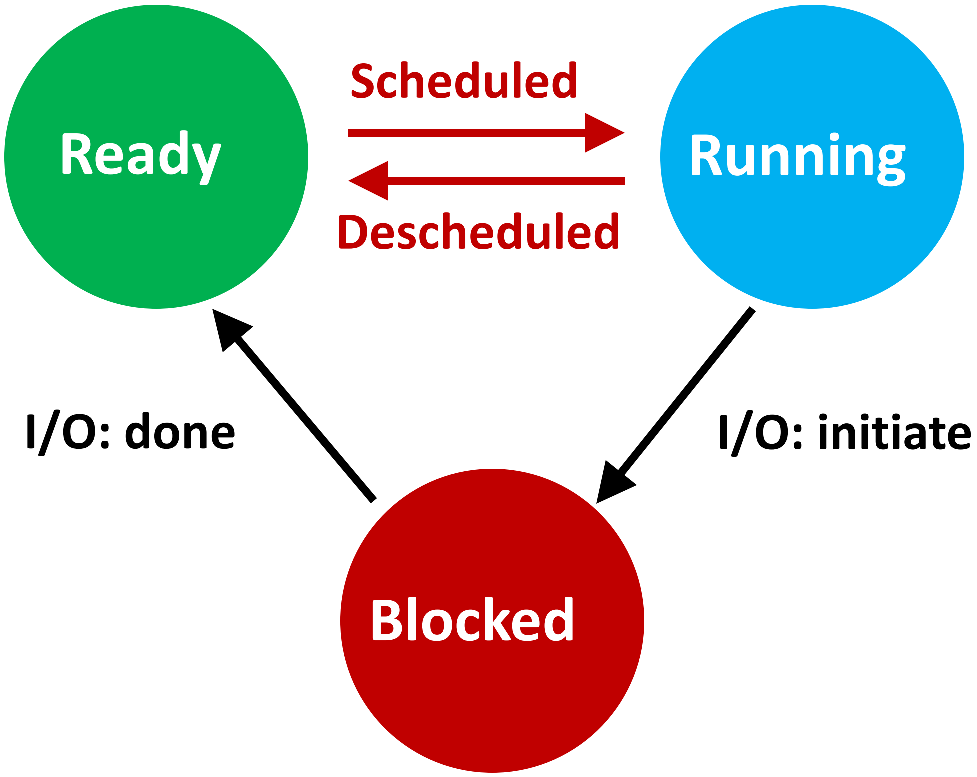


Process State Transitions



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

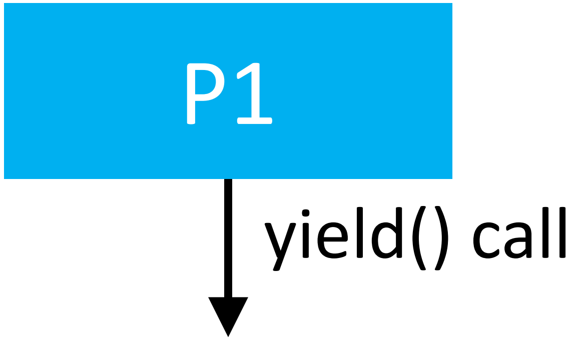
Cooperative Approach

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

P1

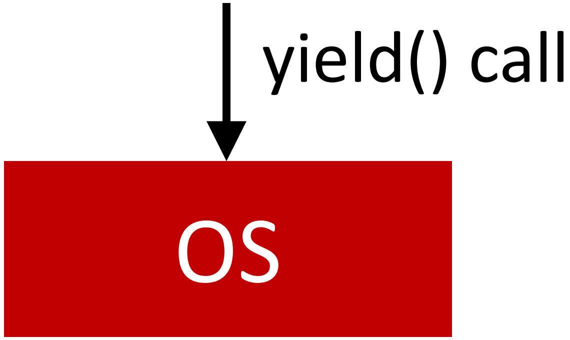
Cooperative Approach

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



Cooperative Approach

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



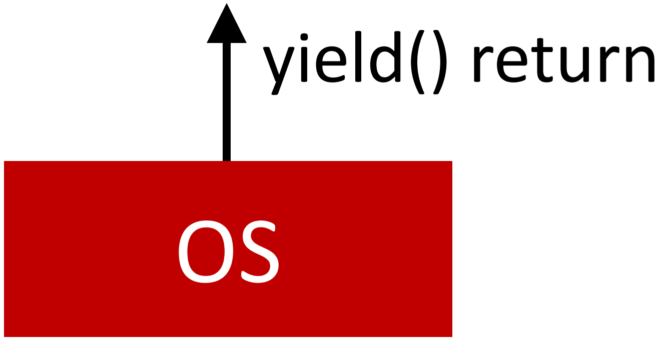
Cooperative Approach

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

OS

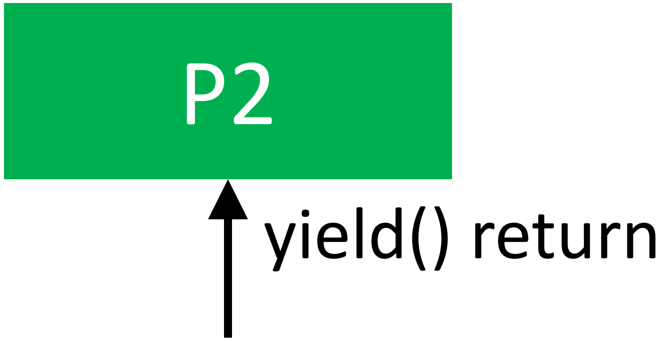
Cooperative Approach

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



Cooperative Approach

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



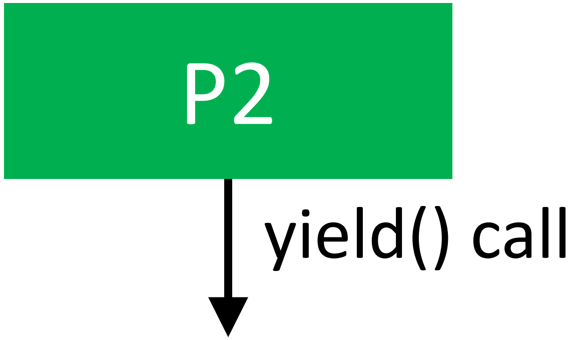
Cooperative Approach

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

P2

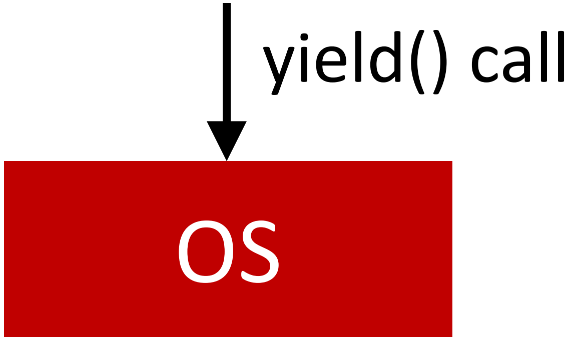
Cooperative Approach

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



Cooperative Approach

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



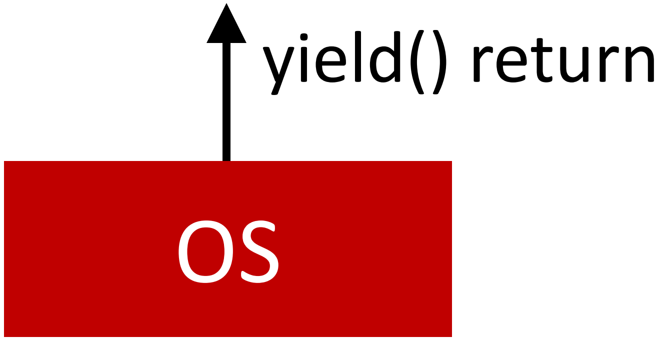
Cooperative Approach

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

OS

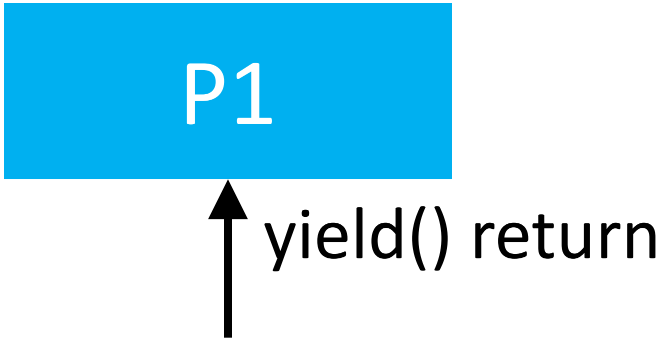
Cooperative Approach

- 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

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()`?

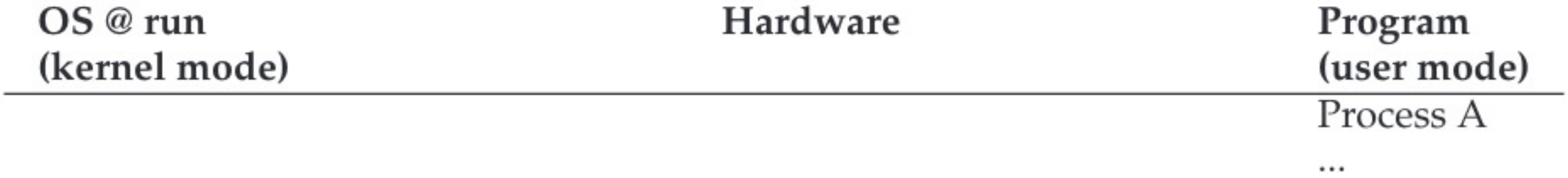
Non-Cooperative Approach

- Switch contexts on timer (hardware) interrupt

- Set up before running any processes

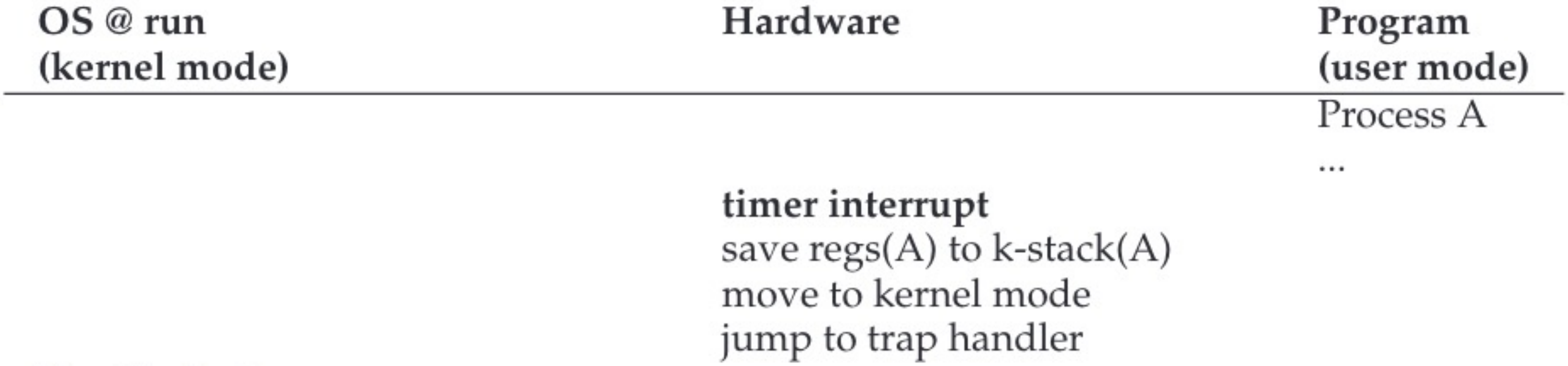
- Hardware does not let processes prevent this
  - Hardware/OS enforces process preemption

Non-Cooperative Approach



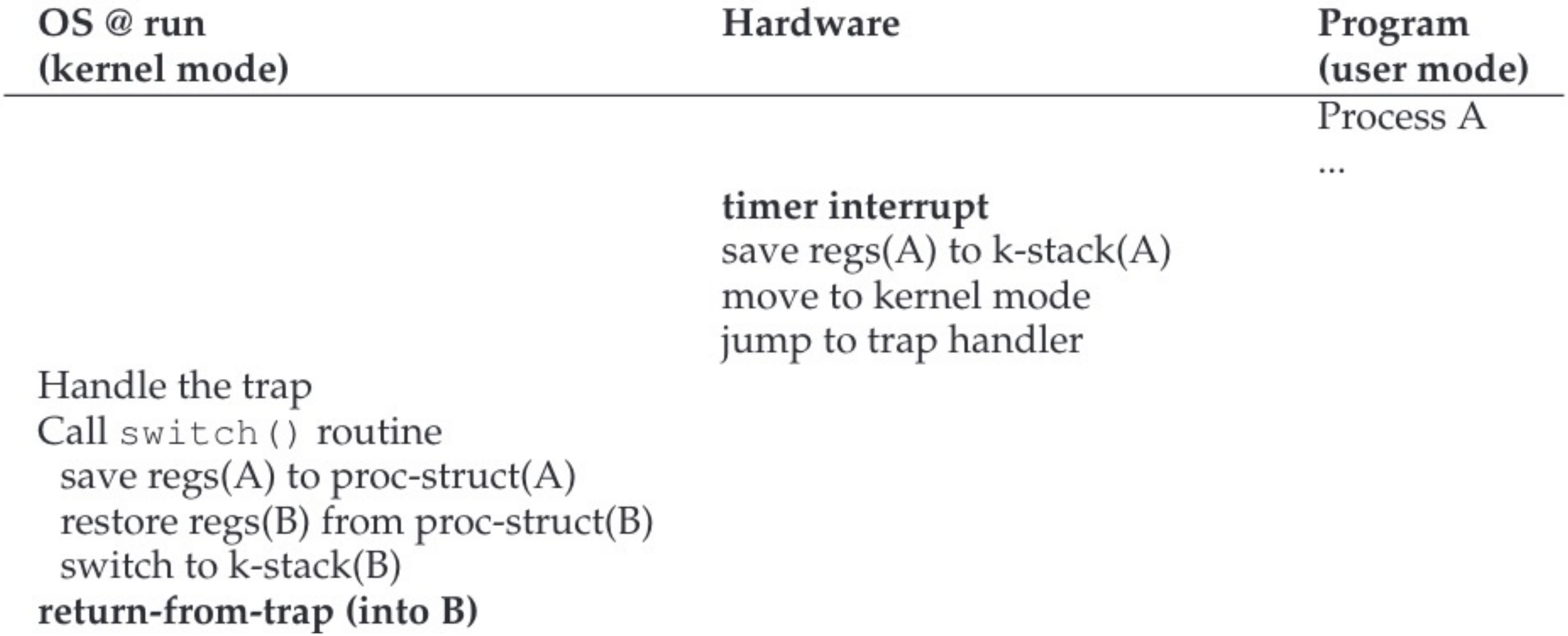
OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A ...

Non-Cooperative Approach



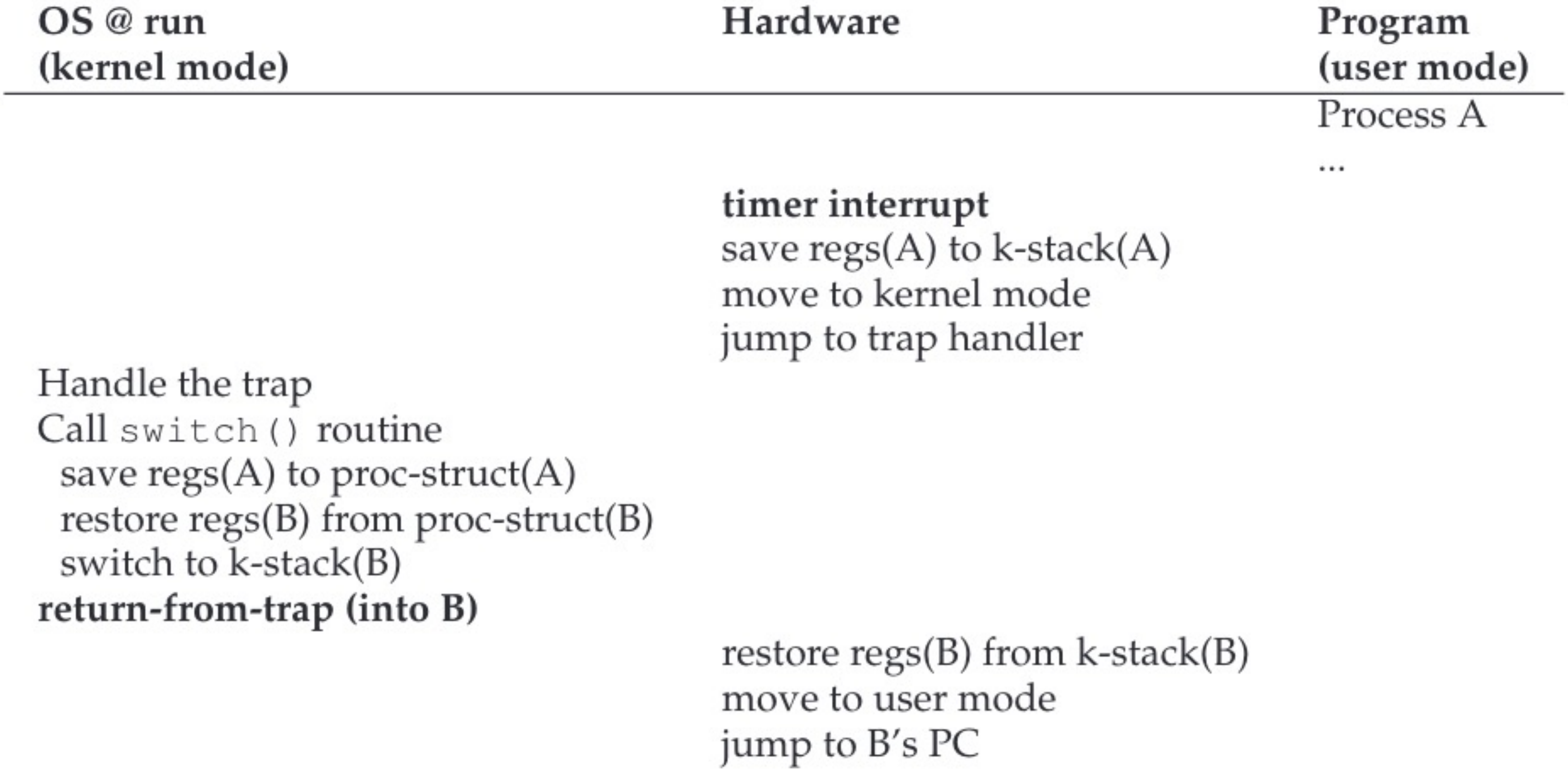
OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A ...
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	

Non-Cooperative Approach



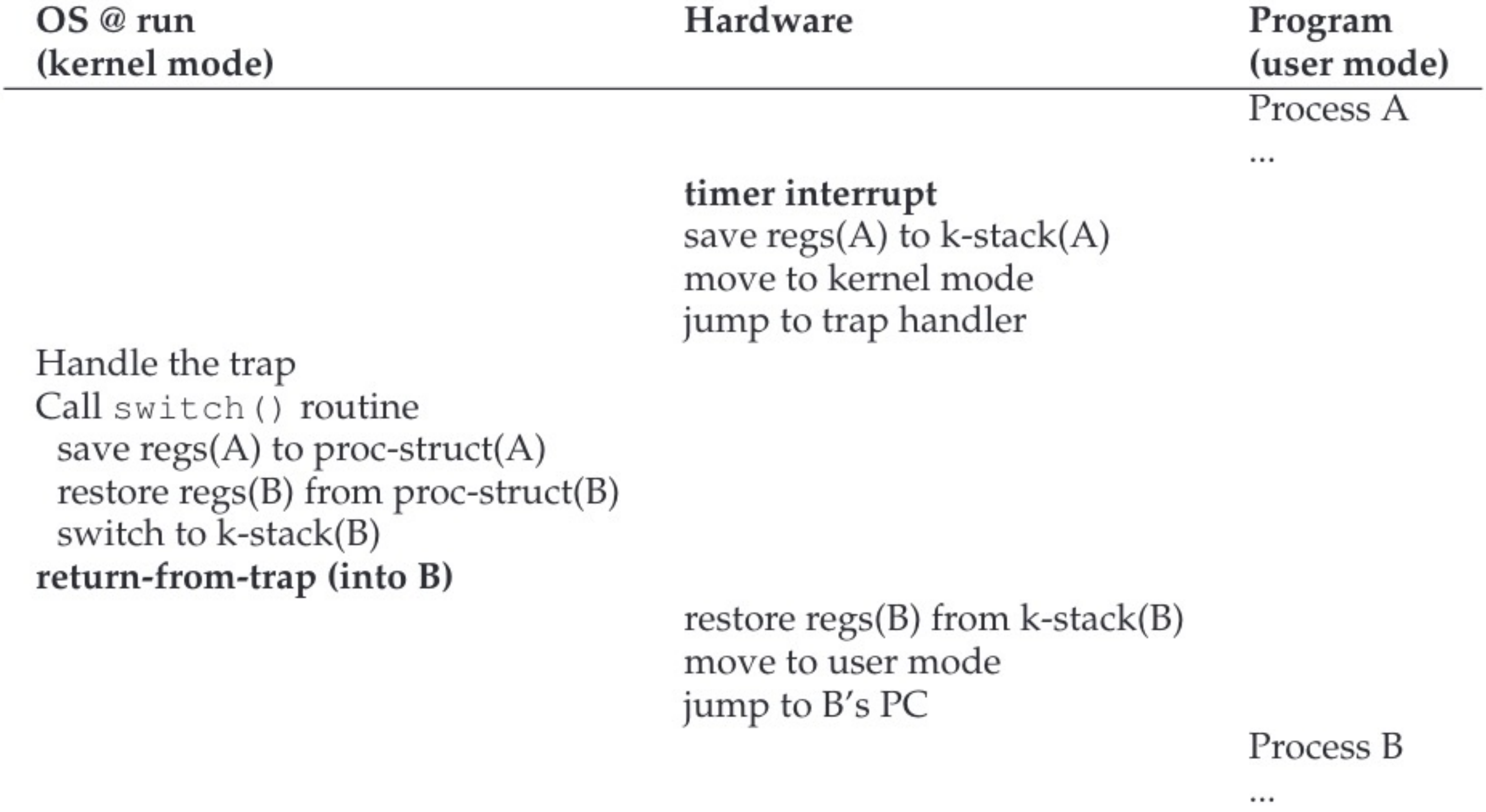
OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A
		…
	timer interrupt	
	save regs(A) to k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) to proc-struct(A)		
restore regs(B) from proc-struct(B)		
switch to k-stack(B)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

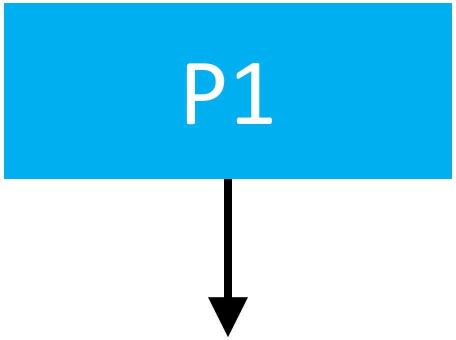


OS @ run	**Hardware**	**Program**
(kernel mode)		**(user mode)**
		Process A
		...
	timer interrupt	
	save regs(A) to k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) to proc-struct(A)		
restore regs(B) from proc-struct(B)		
switch to k-stack(B)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

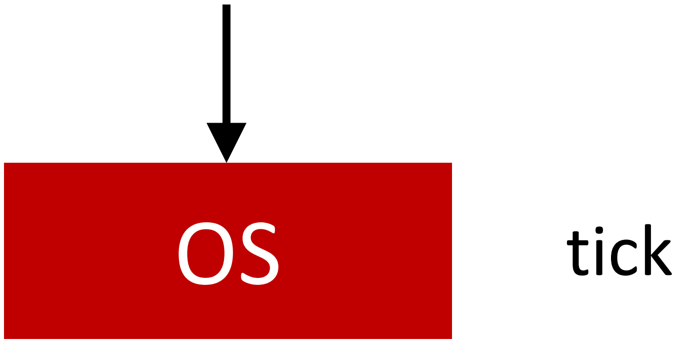
P1

Preemptive Approach



tick

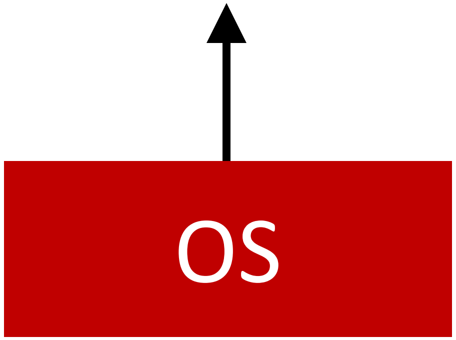
Preemptive Approach



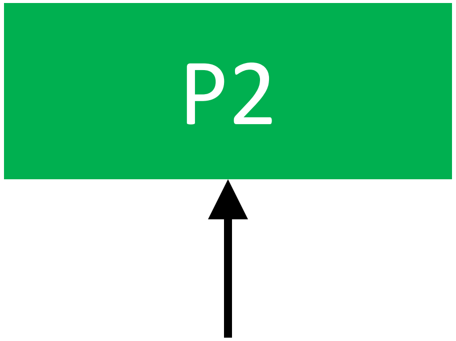
Preemptive Approach

OS

Preemptive Approach



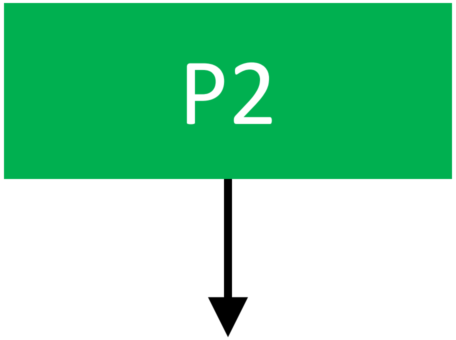
Preemptive Approach



Preemptive Approach

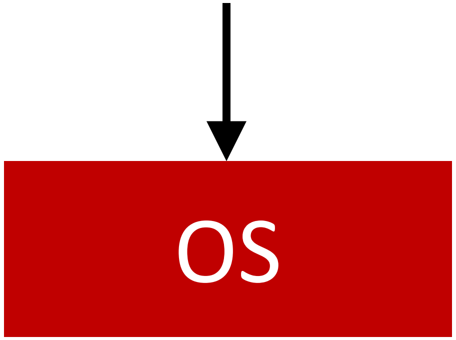
P2

Preemptive Approach



tick

Preemptive Approach

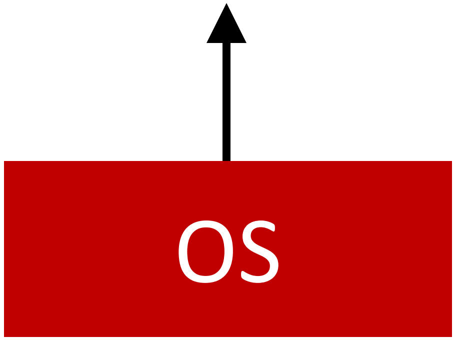


tick

Preemptive Approach

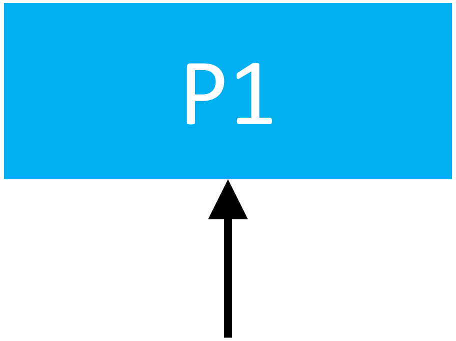
OS

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

CPU Virtualization: Outline

- Limited Direct Execution (LDE)
- Basic scheduling concept and criteria
- Basic scheduling algorithms
  - First In, First Out (FIFO)
  - Shortest Job First (SFJ)
  - Shortest Time-to-Completion First (STCF)
  - Round Robin (RR)

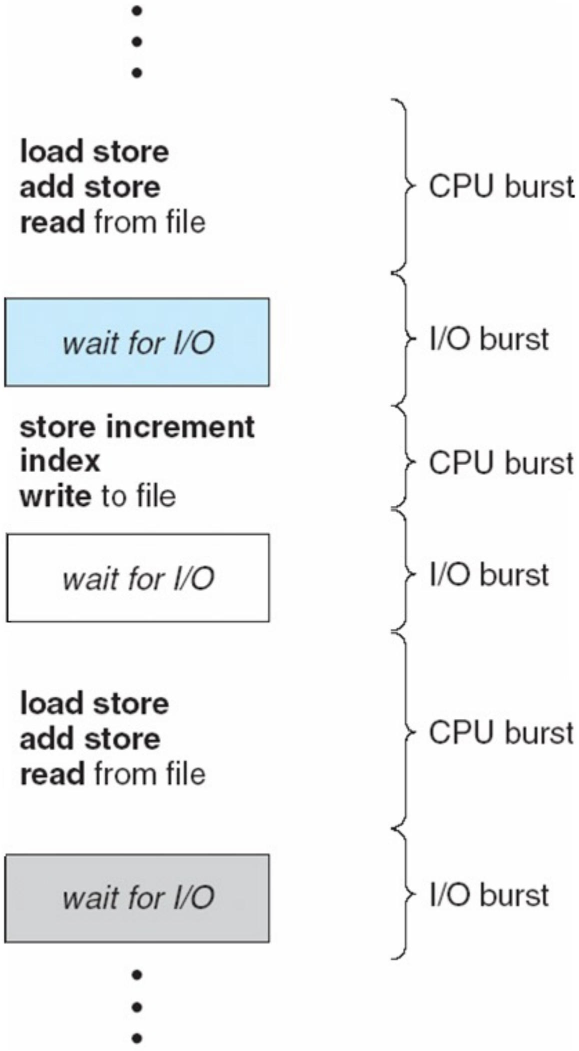
Basic Concepts

- During its lifetime, a process goes through a sequence of CPU and I/O bursts

- The CPU scheduler will select one of the processes in the ready queue for execution

- The CPU scheduler algorithm may have tremendous effects on the system performance
    - Interactive systems: Responsiveness
    - Real-time systems: Not missing the deadlines

Alternating Sequence of CPU and I/O Bursts



Scheduling Metrics

- To compare the performance of scheduling algorithms
  - CPU utilization – percentage of time CPU is busy executing jobs
  - Throughput – # of processes that complete their execution per time unit
  - Turnaround time – amount of time to execute a particular process
  - Waiting time – amount of time a process has been waiting in the ready queue or waiting for some event
  - Response time – amount of time it takes from when a request was submitted until the first response is produced, not the complete output

Optimization Goals

- To **maximize**:
  - Maximize the CPU utilization
  - Maximize the throughput

- To **minimize**:
  - Minimize the (average) turnaround time
  - Minimize the (average) waiting time
  - Minimize the (average) response time

First In, First Out (FIFO)

Workload Assumptions

1. Each job runs for the same amount of time

2. All jobs arrive at the same time

3. All jobs only use the CPU (no I/O)

4. The run-time of each job is known

FIFO

- First-In, First-Out: Run jobs in arrival (time) order

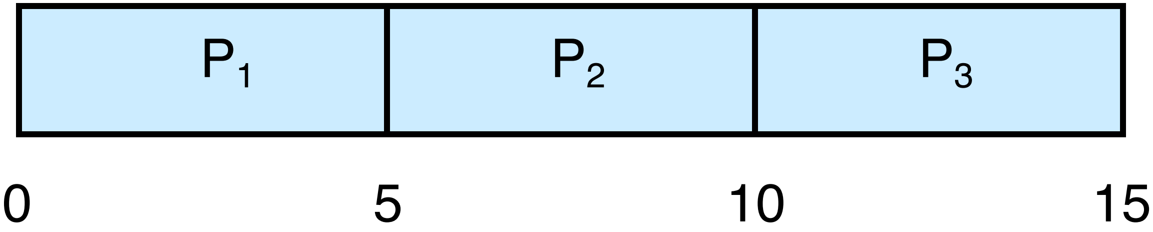
FIFO

First-In, First-Out: Run jobs in arrival (time) order

Def: waiting_time = start_time – arrival_time

Process	Burst Time
P_1	5
P_2	5
P_3	5

- ○ Suppose that the processes arrive in order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule:



- ○ Waiting time for P_1 = 0; P_2 = 5; P_3 = 10
- ○ Average waiting time: 5

FIFO

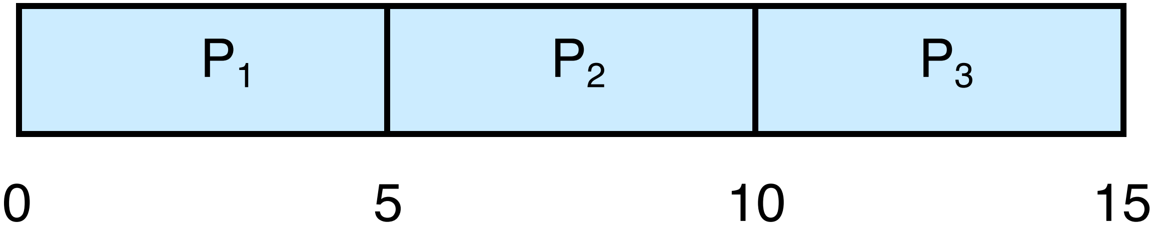
First-In, First-Out: Run jobs in arrival (time) order

What is the average turnaround time?

Def: turnaround_time = completion_time – arrival_time

Process	Burst Time
P_1	5
P_2	5
P_3	5

- Suppose that the processes arrive in order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule:



- Waiting time for P_1 = 0; P_2 = 5; P_3 = 10
- Average waiting time: 5

FIFO

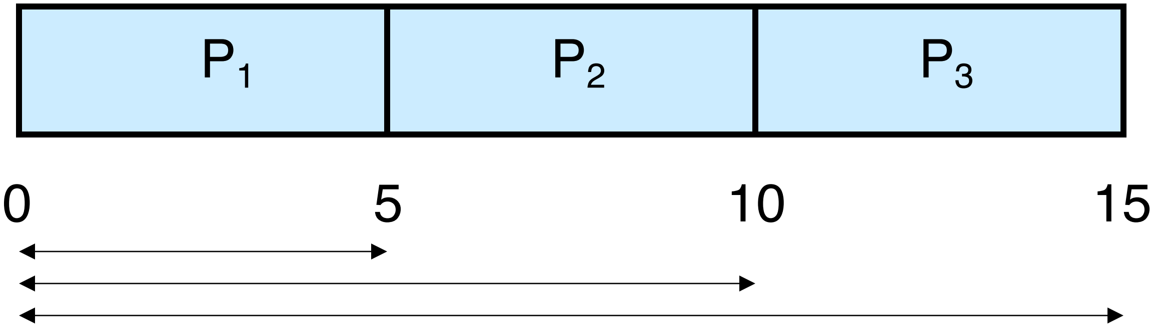
First-In, First-Out: Run jobs in arrival (time) order

What is the average turnaround time?

Def: turnaround_time = completion_time – arrival_time

Process	Burst Time
P_1	5
P_2	5
P_3	5

- Suppose that the processes arrive in order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule:



FIFO

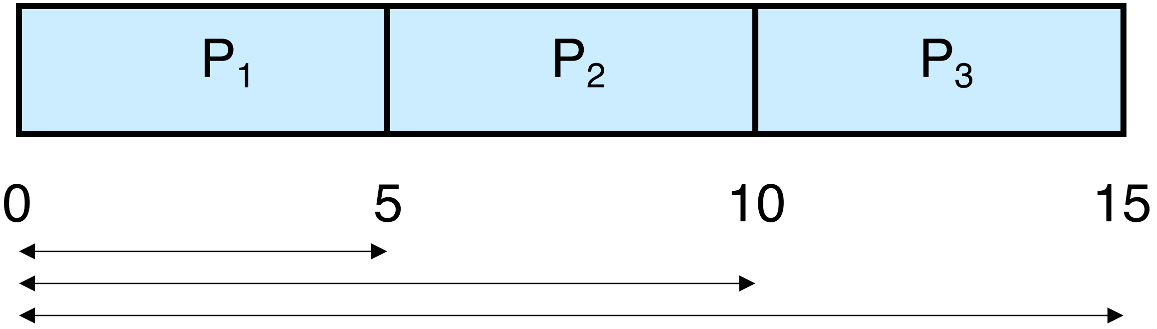
First-In, First-Out: Run jobs in arrival (time) order

What is the average turnaround time?

Def: turnaround_time = completion_time – arrival_time

Process	Burst Time
P_1	5
P_2	5
P_3	5

- Suppose that the processes arrive in order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule:



Average turnaround time: (5+10+15)/3 = 10

Workload Assumptions

1. Each job runs for the same amount of time

2. All jobs arrive at the same time

3. All jobs only use the CPU (no I/O)

4. The run-time of each job is known

Workload Assumptions

1. ~~Each job runs for the same amount of time~~

2. All jobs arrive at the same time

3. All jobs only use the CPU (no I/O)

4. The run-time of each job is known

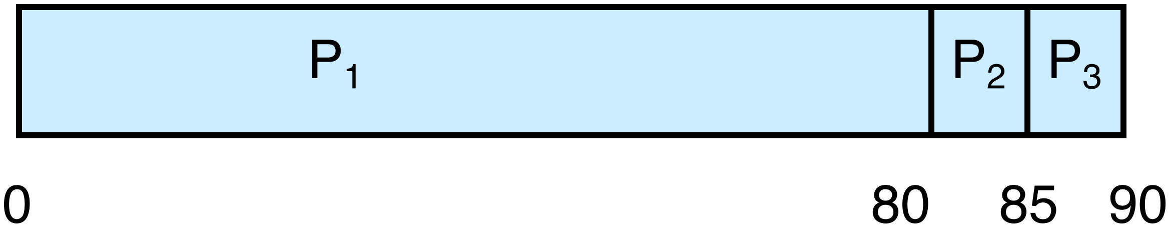
Example: Big First Job

JOB	arrival_time	run_time
P1	~0	80
P2	~0	5
P3	~0	5

What is the average turnaround time?

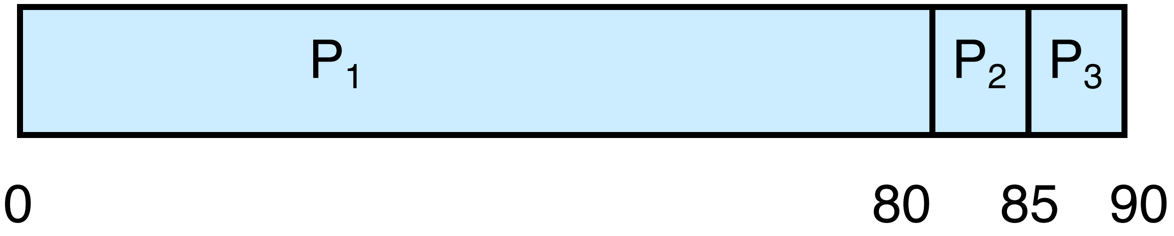
Example: Big First Job

JOB	arrival_time	run_time
P1	~0	80
P2	~0	5
P3	~0	5



Example: Big First Job

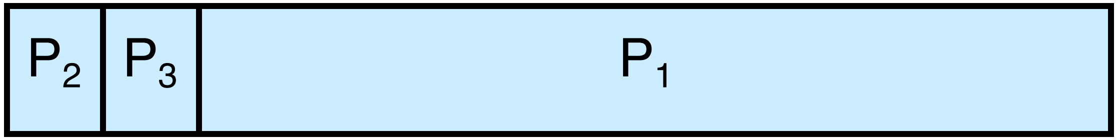
JOB	arrival_time	run_time
P1	~0	80
P2	~0	5
P3	~0	5



Average turnaround time: (80+85+90) / 3 = 85

Convoy Effect

Better Schedule?



Shortest Job First (SJF)

Passing the Tractor

- New scheduler: SJF (Shortest Job First)

- Policy: When deciding which job to run, choose the one with the smallest run_time

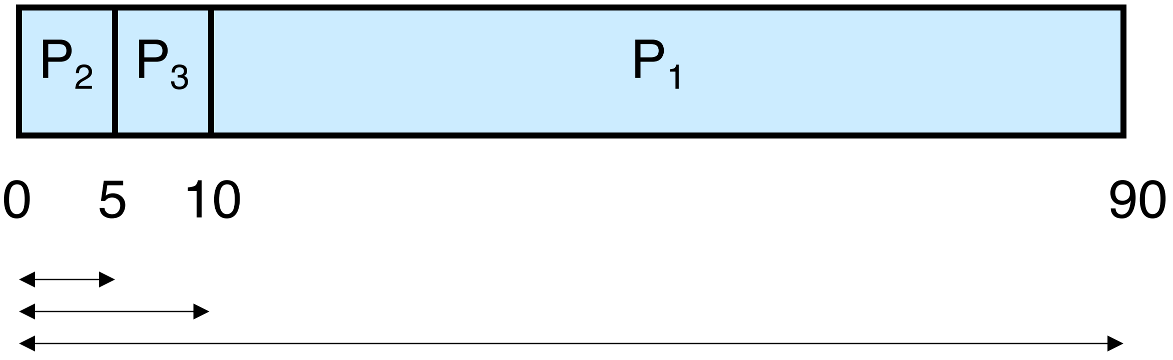
Example: SJF

JOB	arrival_time	run_time
P1	~0	80
P2	~0	5
P3	~0	5

What is the average turnaround time with SJF?

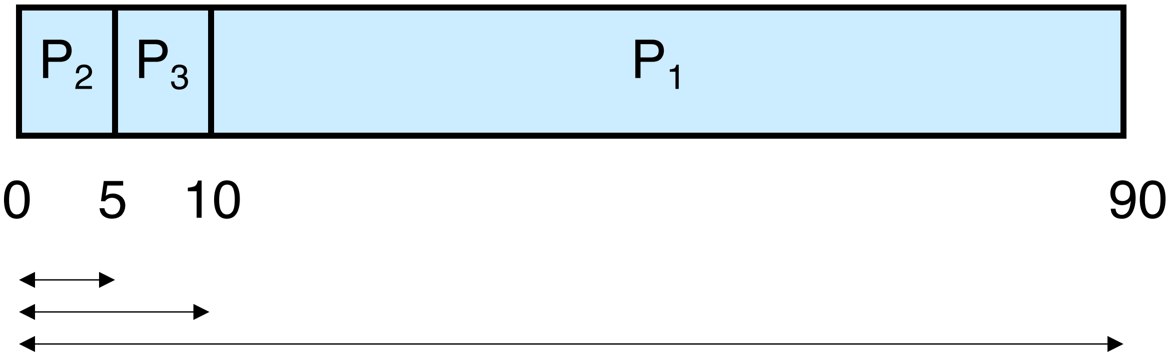
Example: SJF

JOB	arrival_time	run_time
P1	~0	80
P2	~0	5
P3	~0	5



Example: SJF

JOB	arrival_time	run_time
P1	~0	80
P2	~0	5
P3	~0	5



Average turnaround time: (5+10+90) / 3 = 35

Workload Assumptions

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. All jobs only use the CPU (no I/O)
4. The run-time of each job is known

Workload Assumptions

1. ~~Each job runs for the same amount of time~~

2. ~~All jobs arrive at the same time~~

3. All jobs only use the CPU (no I/O)

4. The run-time of each job is known

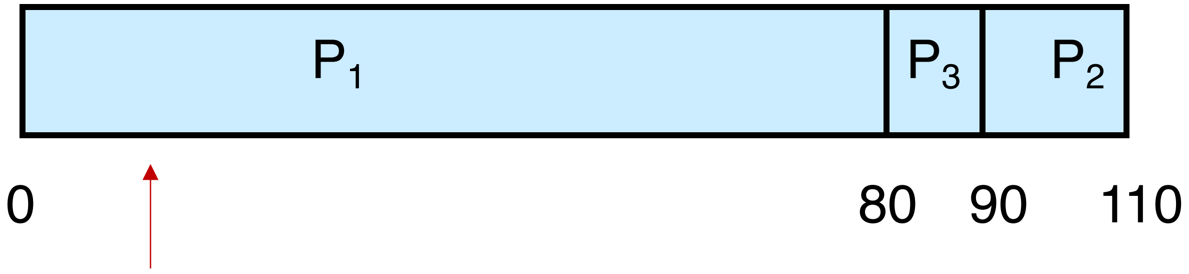
Shortest Job First (Arrival Time)

JOB	arrival_time	run_time
P1	~0	80
P2	**~15**	20
P3	**~15**	10

What is the average turnaround time with SJF?

Shortest Job First (Arrival Time)

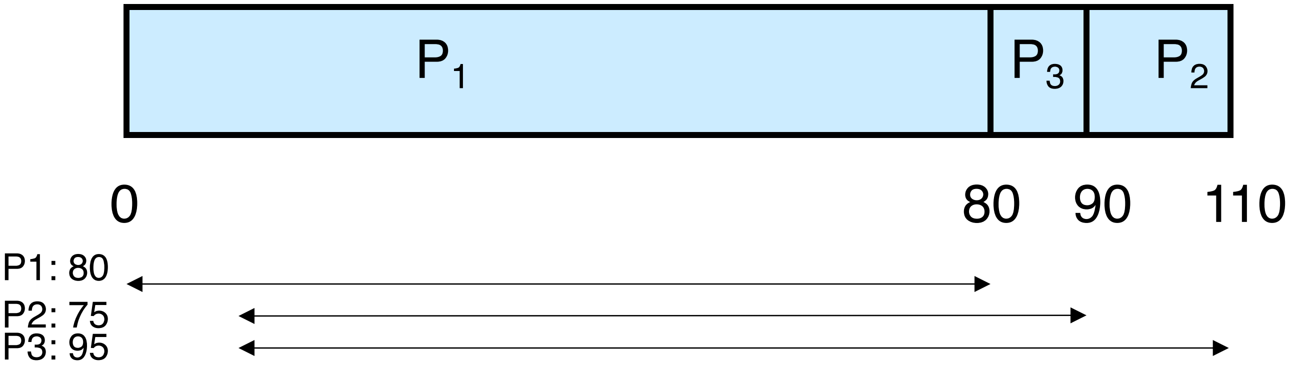
JOB	arrival_time	run_time
P1	~0	80
P2	**~15**	20
P3	**~15**	10



[P2, P3 arrive at 15]

Shortest Job First (Arrival Time)

JOB	arrival_time	run_time
P1	~0	80
P2	**~15**	20
P3	**~15**	10



Average turnaround time: (80+75+95) / 3 = ~83.3

A Preemptive Scheduler

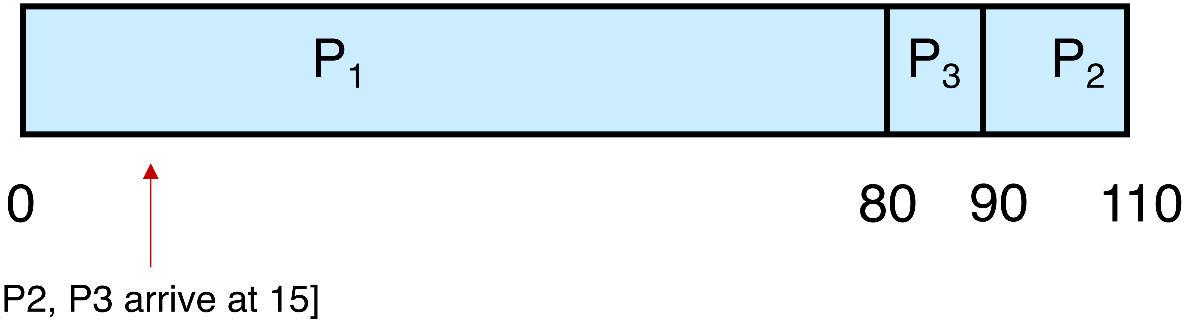
- Previous schedulers: FIFO and SJF are non-preemptive

- New scheduler: STCF (Shortest Time-to-Completion First)

- Policy: Switch jobs so we always run the one that will complete the quickest

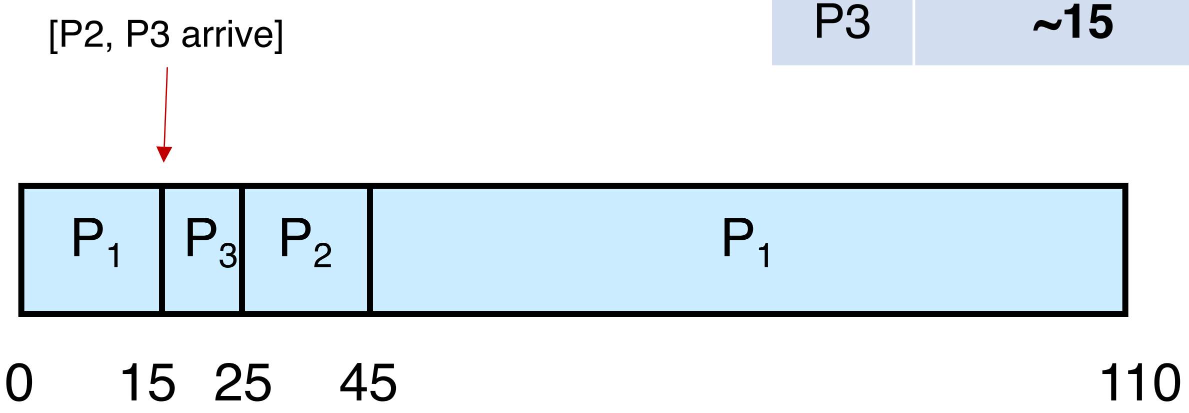
SJF

JOB	arrival_time	run_time
P1	~0	80
P2	**~15**	20
P3	**~15**	10



STCF

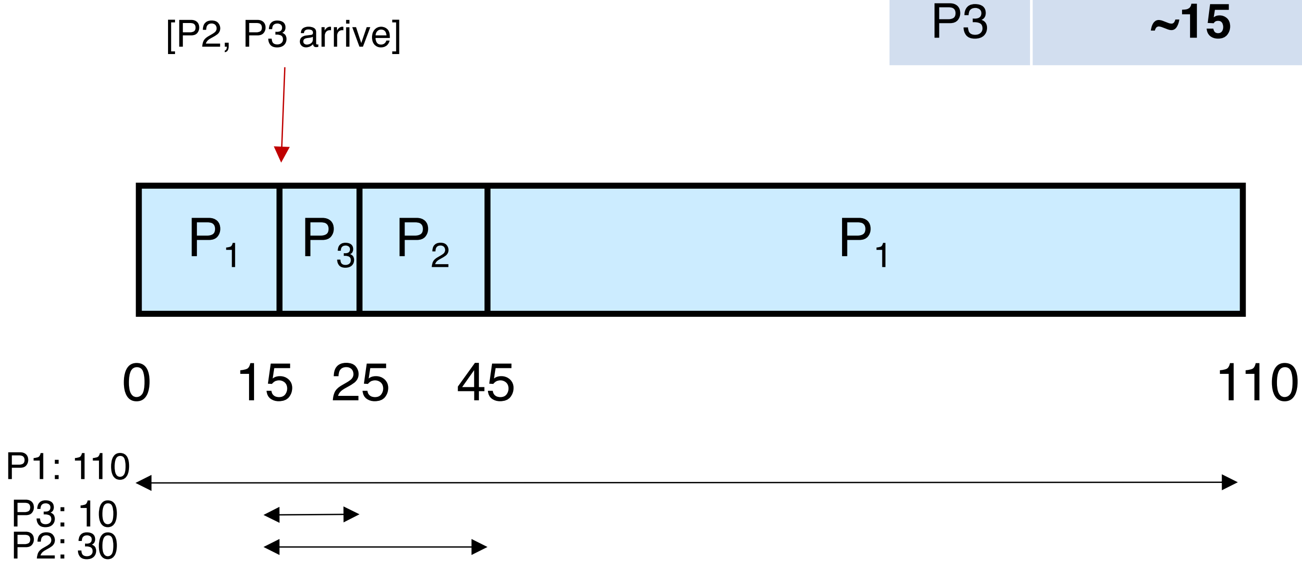
JOB	arrival_time	run_time
P1	~0	80
P2	**~15**	20
P3	**~15**	10



What is the average turnaround time with STCF?

STCF

JOB	arrival_time	run_time
P1	~0	80
P2	**~15**	20
P3	**~15**	10



[P2, P3 arrive]

P_1 P_3 P_2 P_1

0 15 25 45 110

P1: 110

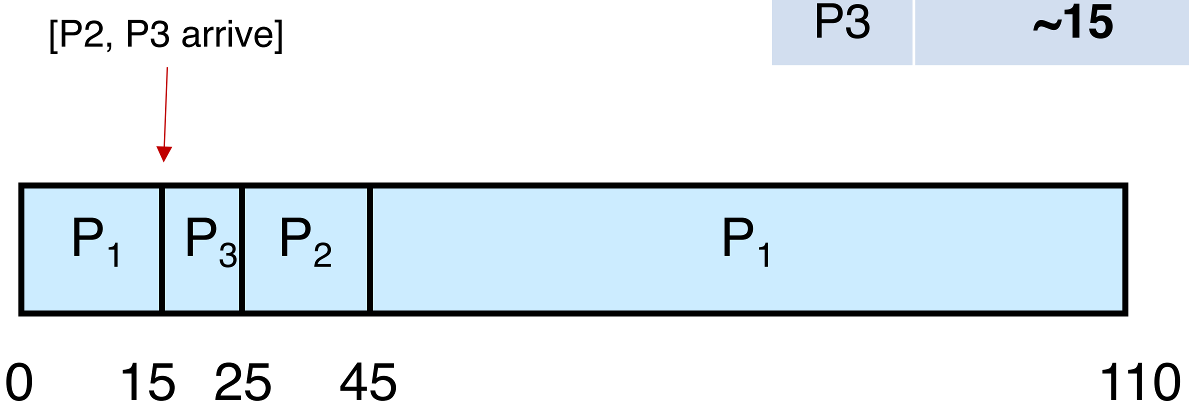
P3: 10

P2: 30

Average turnaround time: (110+30+10) / 3 = 50

STCF

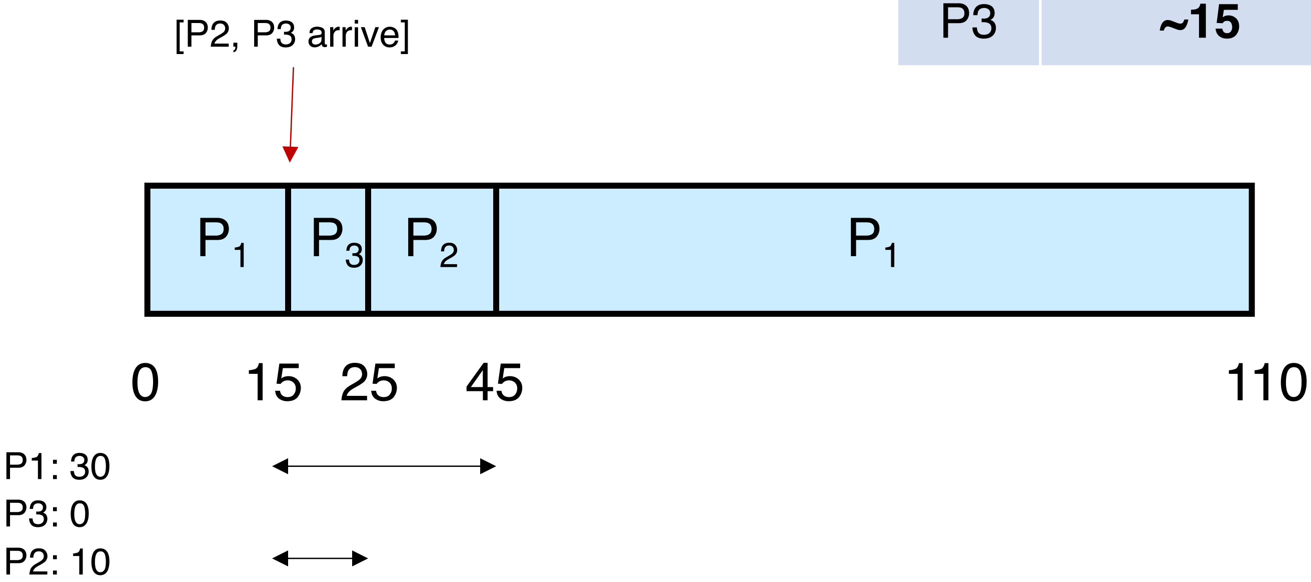
JOB	arrival_time	run_time
P1	~0	80
P2	**~15**	20
P3	**~15**	10



What is the average waiting time with STCF?

STCF

JOB	arrival_time	run_time
P1	~0	80
P2	**~15**	20
P3	**~15**	10



Average waiting time: (30+10+0) / 3 = ~13.3

Optimality of SJF and STCF

- Non-preemptive SJF is optimal if all the processes are ready simultaneously
  - Gives minimum average waiting time for a given set of processes

Optimality of SJF and STCF

- Non-preemptive SJF is optimal if all the processes are ready simultaneously
  - Gives minimum average waiting time for a given set of processes

- What is the rationale behind the optimality of STCF?

Optimality of SJF and STCF

- Non-preemptive SJF is optimal if all the processes are ready simultaneously
  - Gives minimum average waiting time for a given set of processes

- What is the rationale behind the optimality of STCF?
  - A: STCF is optimal, considering a more realistic scenario where all the processes may be arriving at different times

Optimality of SJF and STCF

- Non-preemptive SJF is optimal if all the processes are ready simultaneously
  - Gives minimum average waiting time for a given set of processes

Q: What's the problem then?

We don't know how long a job would run!

- What is the intuition behind the optimality of SRTF?
  - A: SRTF is optimal, considering a more realistic scenario where all the processes may be arriving at different times

Estimating the Length of Next CPU Burst

- Idea: Based on the observations in the recent past, we can try to predict

- Techniques such as exponential averaging are based on combining the observations in the past and our predictions using different weights

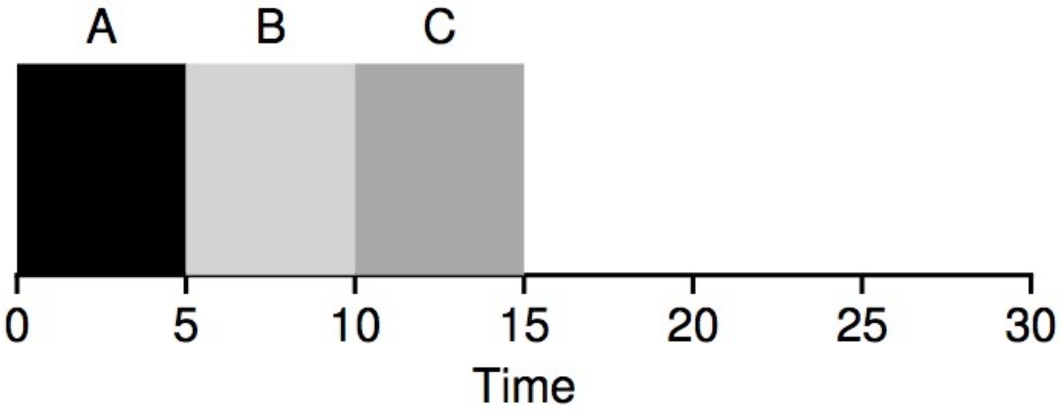
- Exponential averaging
  - t_n: actual length of the *nth* CPU burst
  - z_{n+1}: predicted value for the next CPU burst
  - $z_{n+1} = k.t_n + (1-k).z_n$
  - Commonly, k is set to ½

Response Time

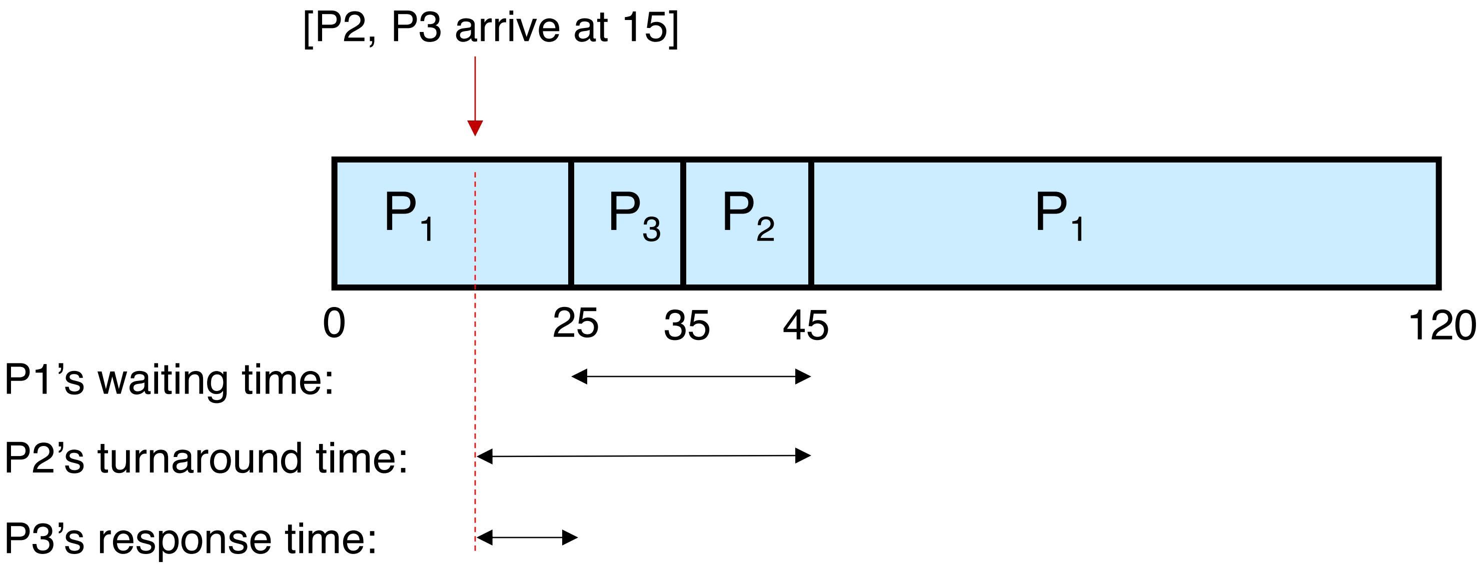
- Response time definition

$$T_{response} = T_{first\_run} - T_{arrival}$$

- SJF's average response time (all 3 jobs arrive at same time)
  - (0 + 5 + 10)/3 = 5



Waiting, Turnaround, Response



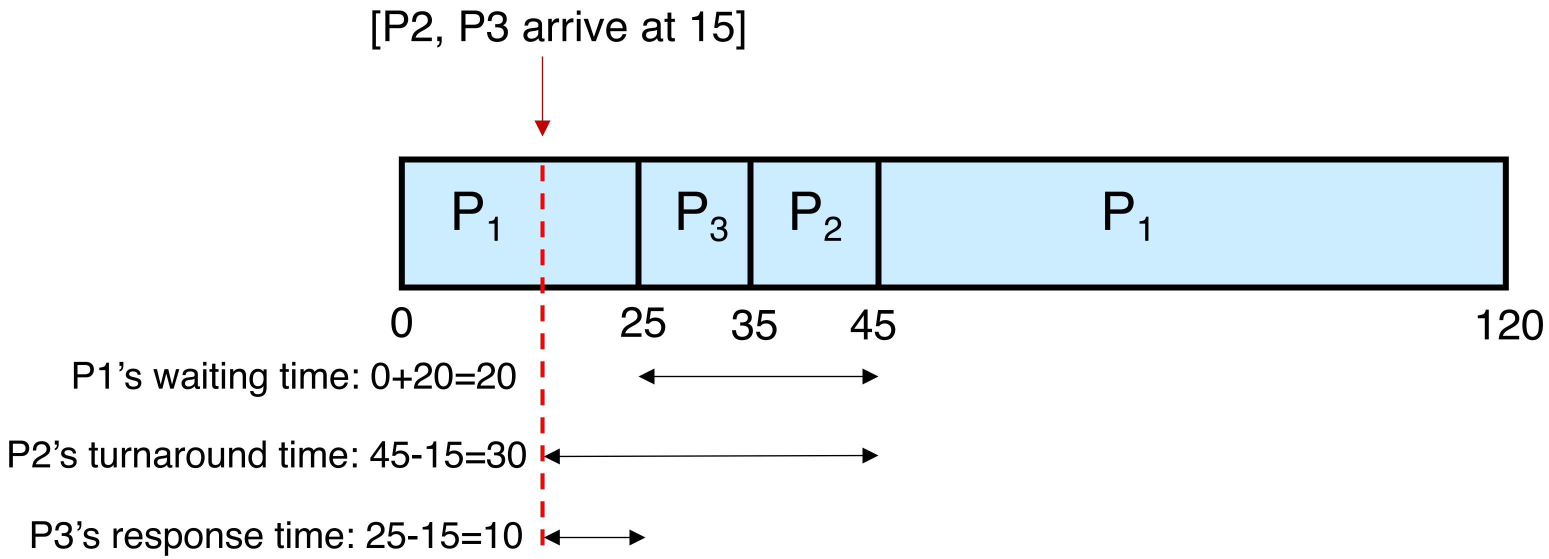
[P2, P3 arrive at 15]

P1's waiting time:

P2's turnaround time:

P3's response time:

Waiting, Turnaround, Response



[P2, P3 arrive at 15]

P1's waiting time: 0+20=20

P2's turnaround time: 45-15=30

P3's response time: 25-15=10

Q: What is P1's response time?

Round Robin (RR)

Workload Assumptions

1. Each job runs for the same amount of time

2. All jobs arrive at the same time

3. All jobs only use the CPU (no I/O)

4. The run-time of each job is known

Workload Assumptions

1. ~~Each job runs for the same amount of time~~

2. ~~All jobs arrive at the same time~~

3. ~~All jobs only use the CPU (no I/O)~~

4. The run-time of each job is known

Extension to Multiple CPU & I/O Bursts

- When the process arrives, it will try to execute its first CPU burst
  - It will join the ready queue
  - The priority will be determined according to the underlying scheduling algorithm and considering only that specific (i.e. first) burst

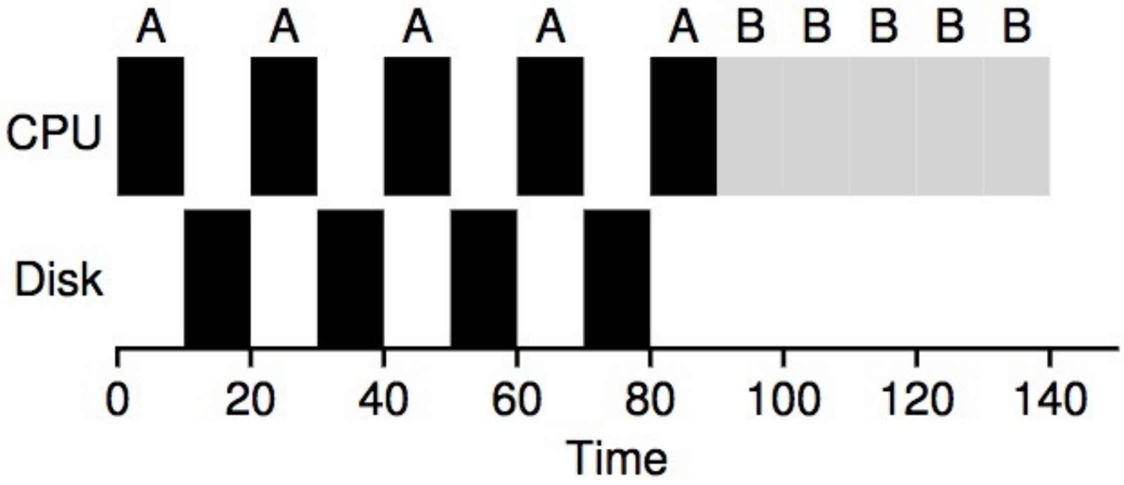
- When it completes its first CPU burst, it will try to perform its first I/O operation (burst)
  - It will join the device queue
  - When that device is available, it will use the device for a time period indicated by the length of the first I/O burst.

- Then, it will re-join the ready queue and try to execute its second CPU burst
  - Its new priority may now change (as defined by its second CPU burst)!

Round Robin (RR)

- Each process gets a small unit of CPU time (time quantum). After this time has elapsed, the process is preempted and added to the end of the ready queue
- Newly-arriving processes (and processes that complete their I/O bursts) are added to the end of the ready queue
- If there are n processes in the ready queue and the time quantum is q, then no process waits more than (n-1)q time units
- Performance
  - q large ⇒ FIFO
  - q small ⇒ Processor Sharing (The system appears to the users as though each of the n processes has its own processor running at the (1/n)th of the speed of the real processor)

Not I/O Aware

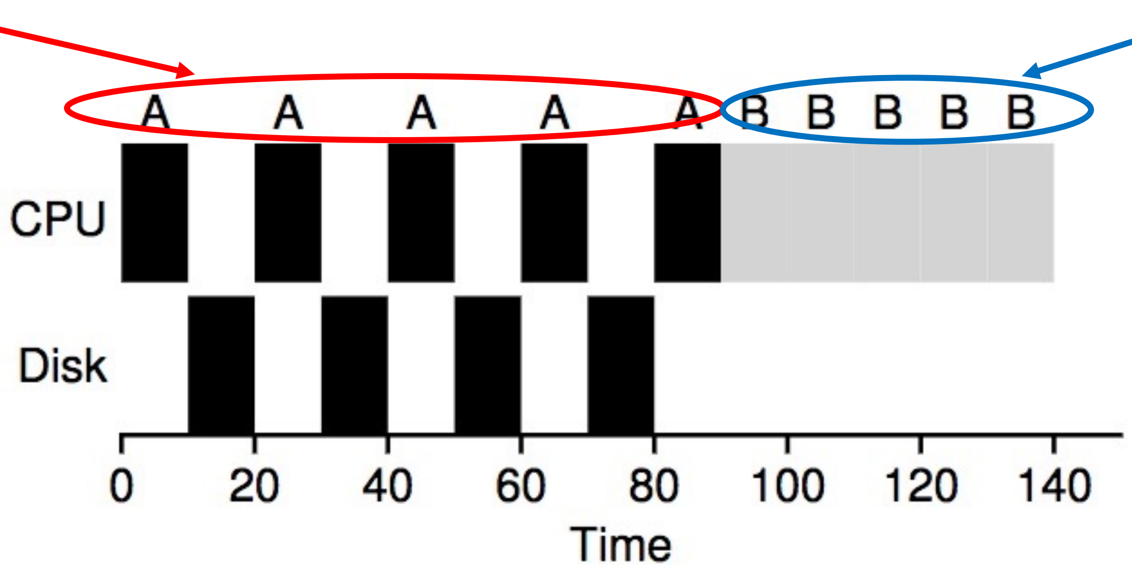


Poor use of resources

Not I/O Aware

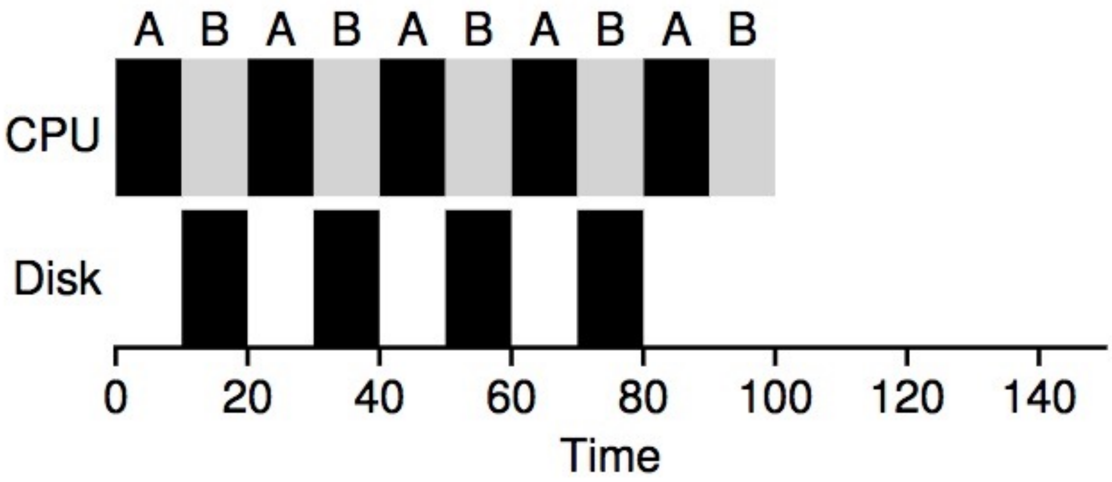
I/O-intensive

CPU-intensive



Poor use of resources

I/O Aware (Overlap)

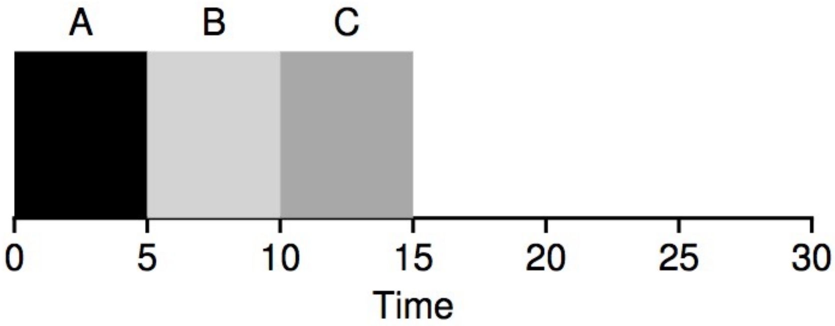


Overlap allows better use of resources!

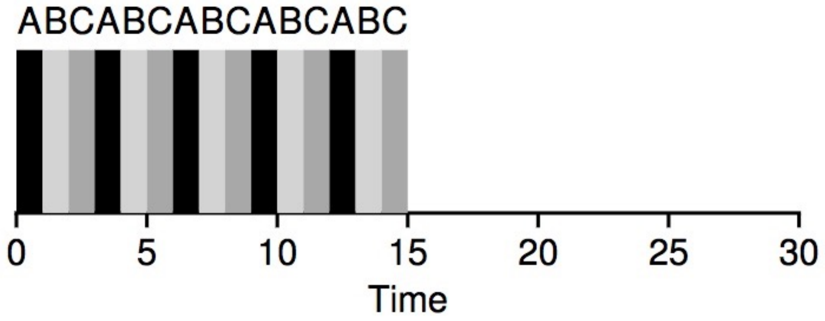
RR

Process	Burst Time
A	5
B	5
C	5

- SJF's average response time
    - (0 + 5 + 10) / 3 = 5



- RR's average response time (time quantum = 1)
    - (0 + 1 + 2) / 3 = 1



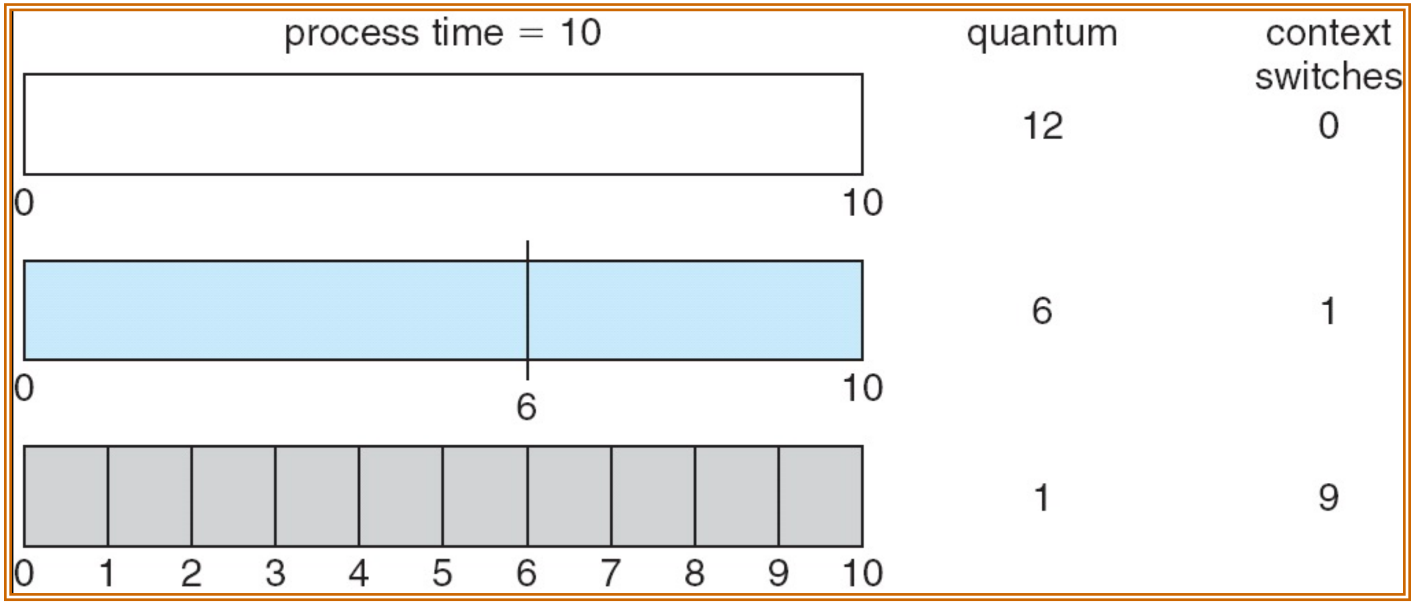
Tradeoff Consideration

- Typically, RR achieves higher average turnaround time than SJF, but better response time
  - Turnaround time only cares about when processes finish

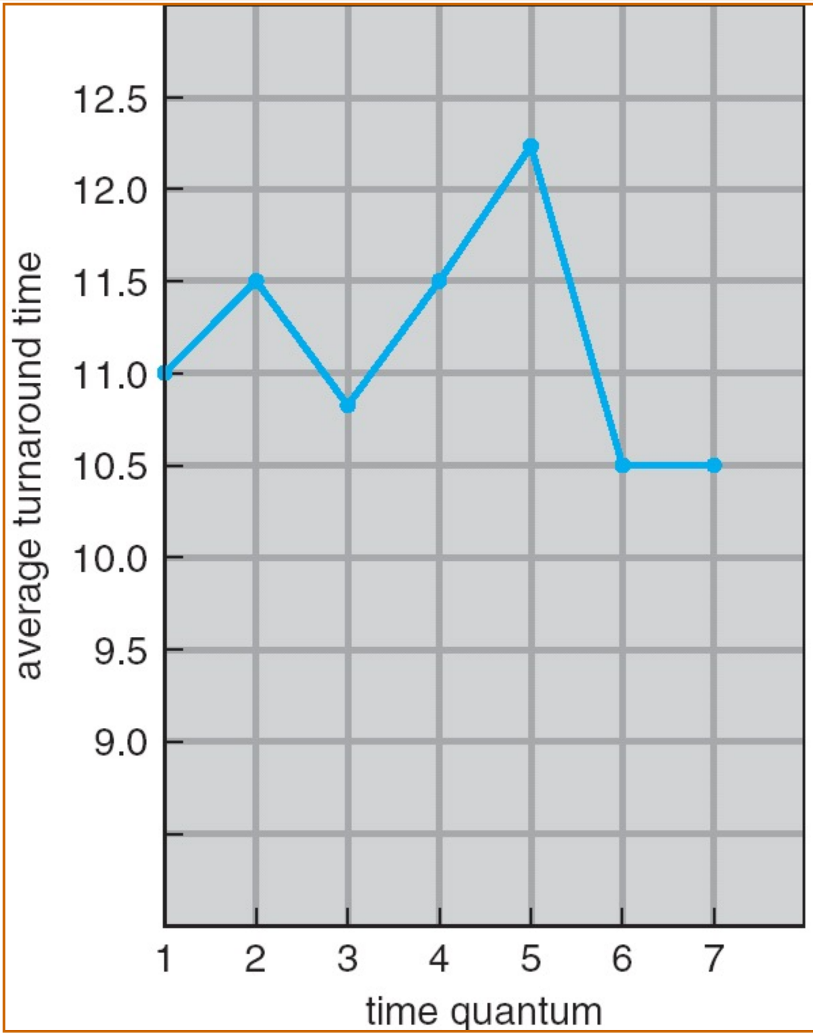
- RR is one of the worst policies
  - if turnaround time is the metric

Choosing a Time Quantum

- The effect of quantum size on context-switching time must be carefully considered
- The time quantum must be large with respect to the context-switch time
- Turnaround time also depends on the size of the time quantum

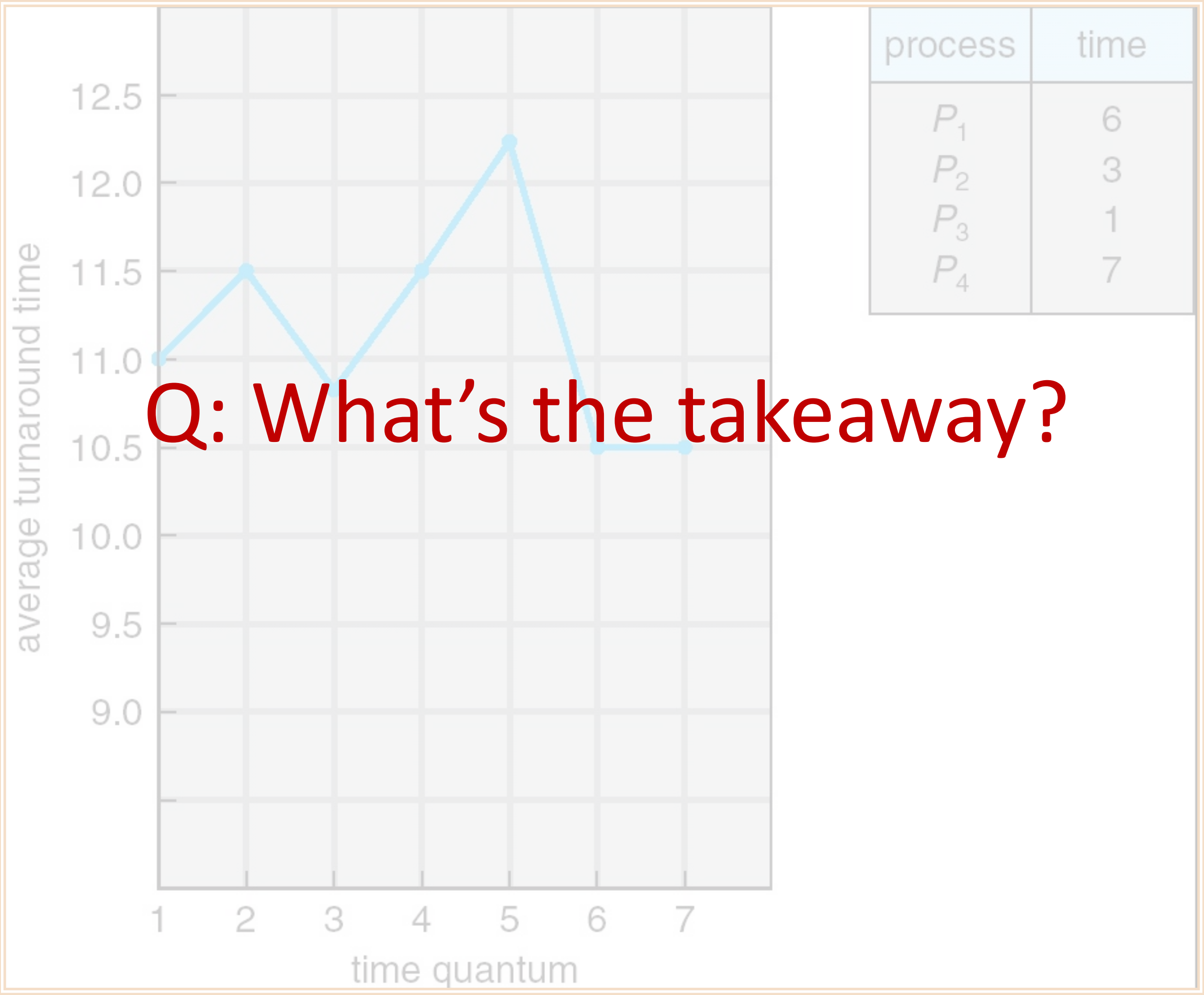


Time Quantum vs. Turnaround Time



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Time Quantum vs. Turnaround Time



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Q: What's the takeaway?

Some adjustments about the format

- Received feedback from students
- Balance between fundamentals and advanced topics
  - More weight on fundamentals and slightly less on research papers
  - Less discussion but more engagement thru:
    - Three mini exams: virtualization/concurrency, persistence, and advanced (VM/container/distributed systems) (20%)
    - popup quizzes + in-class participations (10%)
  - We will still use paper review forms to help with better understanding

Announcements

- Project idea due by end of Week 3 (Feb 11)

- Stop by during open studio for a discussion
  - This week’s open studio rescheduled to 11-12pm

Find your teammate + explore project ideas, now

- Ed (the "Search for teammate" section)
  - I'm Yue Cheng, an Nth year grad student
  - I'm interested in operating systems
  - I pan to build a new Datacenter OS that does …
- Form a team due by this Friday
- Pick an idea due in 1 week
- Project proposals due in 3 weeks (one page)
  - What do you plan to do?
  - Why is it interesting?
  - How will you do it (feasibility)?
    - Basic idea
  - What's your plan and schedule?
  - What resources do you need?