Processes

PID = 0 , child

Doing Well in This Course

- **What do you need to concentrate on to do well in this course?**

What Is Hard?

- **There are a key expectations we have for the students in this course**

- **Terminology: There is a lot of vocabulary in this course that you need to learn and use correctly – winging it won't work**
- **Greedy Algorithms: OSes solve a lot of computationally complex problems with approximate solutions – know them & tradeoffs**
- **C Programming: OSes use C language to manage memory extensively in ways that can create subtle errors – know pointers and debuggers**

Learning Terminology

- Challenge: Hard to know what is *really important* given a lot of potentially intimidating stuff - new to many
- What helps? Readings. Really!
- Problem: Reading about various OS details can be boring as it may be hard to know where this all goes or how it comes together
- What do you suggest?

Learning Terminology

- Challenge: Hard to know what is really important given a lot of potentially intimidating stuff - new to many
- What helps? Readings. Really!
- Problem: Reading about various OS details can be boring as it may be hard to know where this all goes or how it comes together
- Suggestion: read/skim to pick out key concepts from the many pages, then read/study again in with those concepts in mind
- Suggestion: Take notes of things I emphasize from lectures (may not be on the

Learning Greedy

- Challenge: To solve a hard problem efficiently, people try lots of things and you need to know these options and their trade-offs
- What helps? Focus on the main problem
- Problem: There can be a lot of noise as we discuss this and that option – i.e., lose the forest for the trees
- What do you suggest?

Learning Greedy

- Challenge: To solve a hard problem efficiently, people try lots of things and you need to know these options and their trade-offs
- What helps? Focus on the main problem
- Problem: There can be a lot of noise as we discuss this and that option – lose the forest for the trees
- Suggestion: Focus on how these solve the same problem differently
- Suggestion: And the resultant effects of these algorithm choices on operation

Learning C

- **Challenge: We are really going to use C in ways that leverage its power and danger in managing memory – beyond 311**
- **What helps? Mental model**
- **Problem: This may be the first language you have experiences with data objects and memory objects**
- **What do you suggest?**

Learning C

- Challenge: We are really going to use C in ways that leverage its power and danger in managing memory – beyond 311
- What helps? Mental model
- Problem: This may be the first language you have experiences with data objects and memory objects (pointers)
- Suggestion: Use the debugger to see how memory is represented and used (threads!)
- Suggestion: Learn safe programming techniques to avoid creating errors

Topic for Today: Processes

Program vs. Process

- What we looked at until now was a program (and its executable)
- It is not yet a process!
- A process is a program in execution.
- Think of a program as the recipe (instructions) for making a cake.
- The process is the "activity" of making the cake.
- A process has an associated program that it is executing and a state at any point of time.

Program to Process

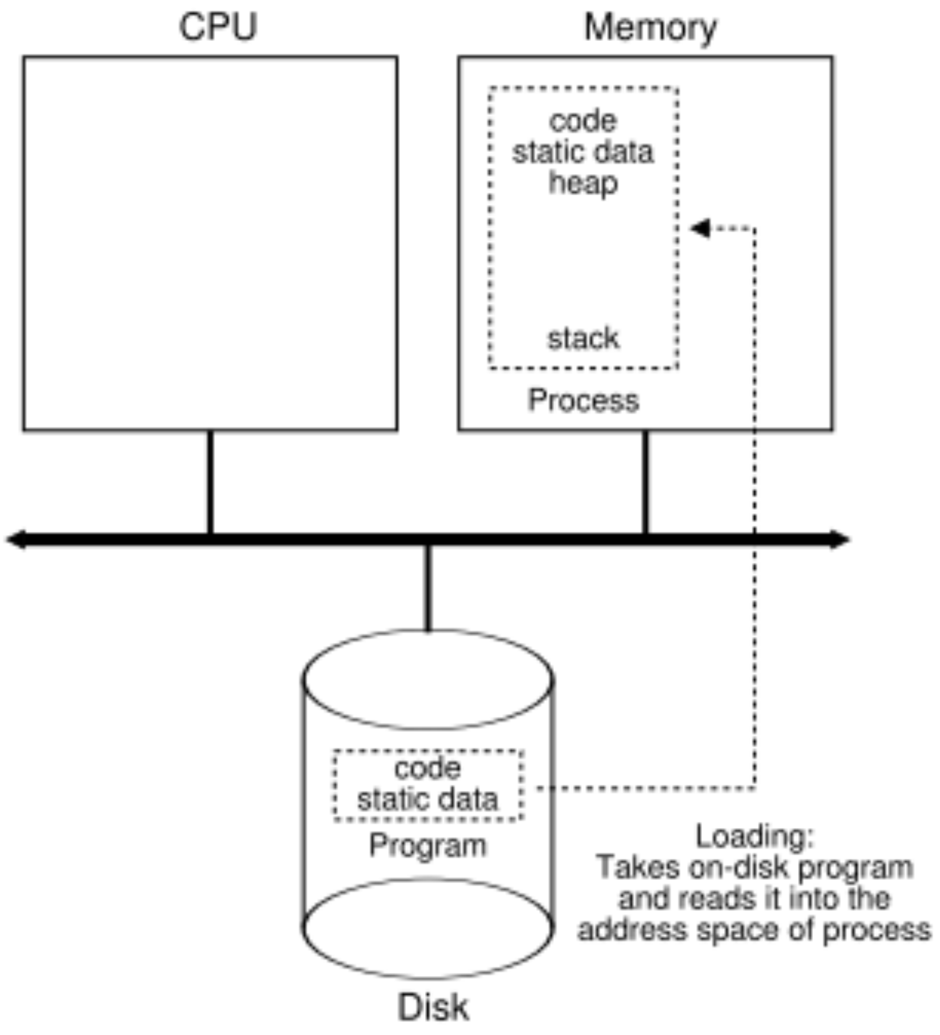


Figure 4.1: **Loading: From Program To Process**

Creating a process in UNIX

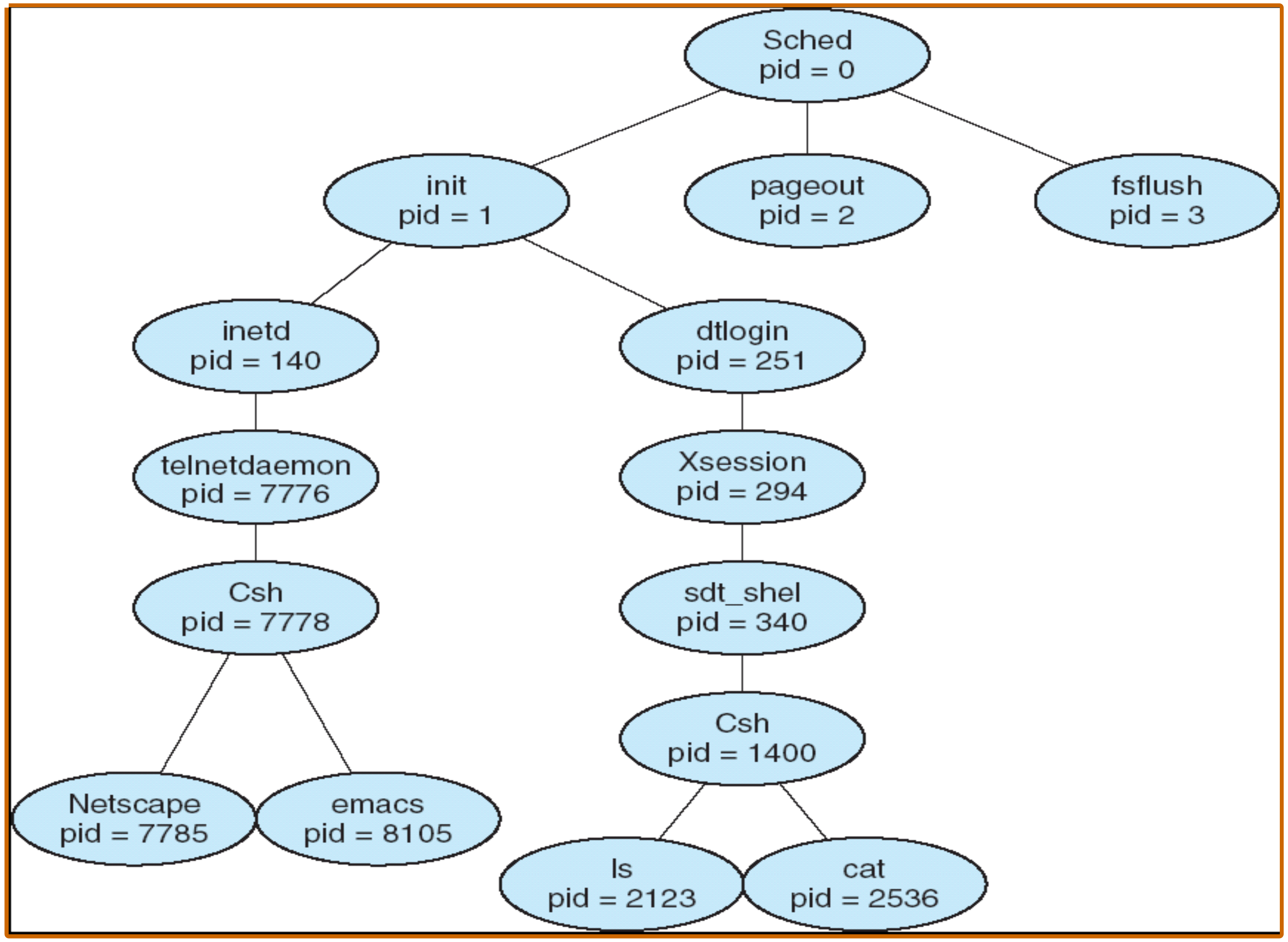
- 2 system calls: fork() and exec()
  - When a process calls a fork() during execution, a duplicate of the calling process is created and both processes execute the next instruction after the fork.
  - When a process calls exec() with an executable as parameter, the calling process is overwritten by the process created to run this executable.
- Do a “man” on these syscalls to find out more.

- What really happens when you type "a.out" in the shell?
  - The shell is itself a process.
  - Upon receiving this command to run a.out, the shell does a fork() to create a duplicate of itself.
  - The duplicate then does an exec() with the file a.out as a parameter, which results in running this program.

OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute **call** main()	
	Run main()
	Execute **return** from main
Free memory of process	
Remove from process list	

Figure 6.1: **Direct Execution Protocol (Without Limits)**

A tree of processes on a typical system



Multiple processes

- Typically, a computer system needs to manage multiple activities.
- E.g., if you are throwing a party, you not only may make a cake, but you may also bake a pizza, …., in addition.
- Each of these activities is again a process.
- But, let us say there is no one else to help you (a single CPU).

Options

- Perform the activities one after another (batching)
- Time-multiplex the CPU amongst the processes (multiprogramming/time-sharing), i.e., even if one activity is not fully done, you may still want to move on to processing another activity.

Which is better?

- Say we are "batching", i.e., finish making the cake before starting on the pizza.
- Recipe for cake: mix the flour, add sugar, place in oven, wait for 30 minutes, add icing.
- Do you want to keep staring at the oven for 30 minutes while the cake is baking?
- The oven is like an I/O device. Batching can result in a gross misuse of CPU resources when there is I/O.
- You want to move on to starting on the pizza while the cake is baking.

- While early mainframes employed “batching,” nearly all systems today (both desktops and servers) use time-sharing/multiprogramming.

PENNSTATE

How do we implement time-sharing?

- We need a way of pre-empting (taking away) the CPU from the currently executing process.
- We then give the CPU to another process that can potentially use the CPU.
- This re-assigning of the CPU from one process to another is called context-switching.

PENNSTATE

When should OS perform context switching?

- When an application process cannot proceed (say waiting for I/O).
- Perhaps even periodically (using timer interrupts).
- Regardless, we do NOT want applications to be aware (and require appropriate code) that they are being context-switched.
- Context-switching is transparent to an application (when you are writing code, you never care that there are other processes that may also execute in the middle).

Implementing Context Switch

- To provide transparent context switch:
  - You need to save the "context" of the current process.
  - You need to restore the "context" of another process
- Context is the state of the process that is necessary for its execution
- Such saving and restoring of state ensures that the process is itself unaware that someone else executed in the middle.

Process Control Block (PCB)

- data structure in the operating system

- Each process has a data structure called PCB

pid, register value, pc, sp, hp,

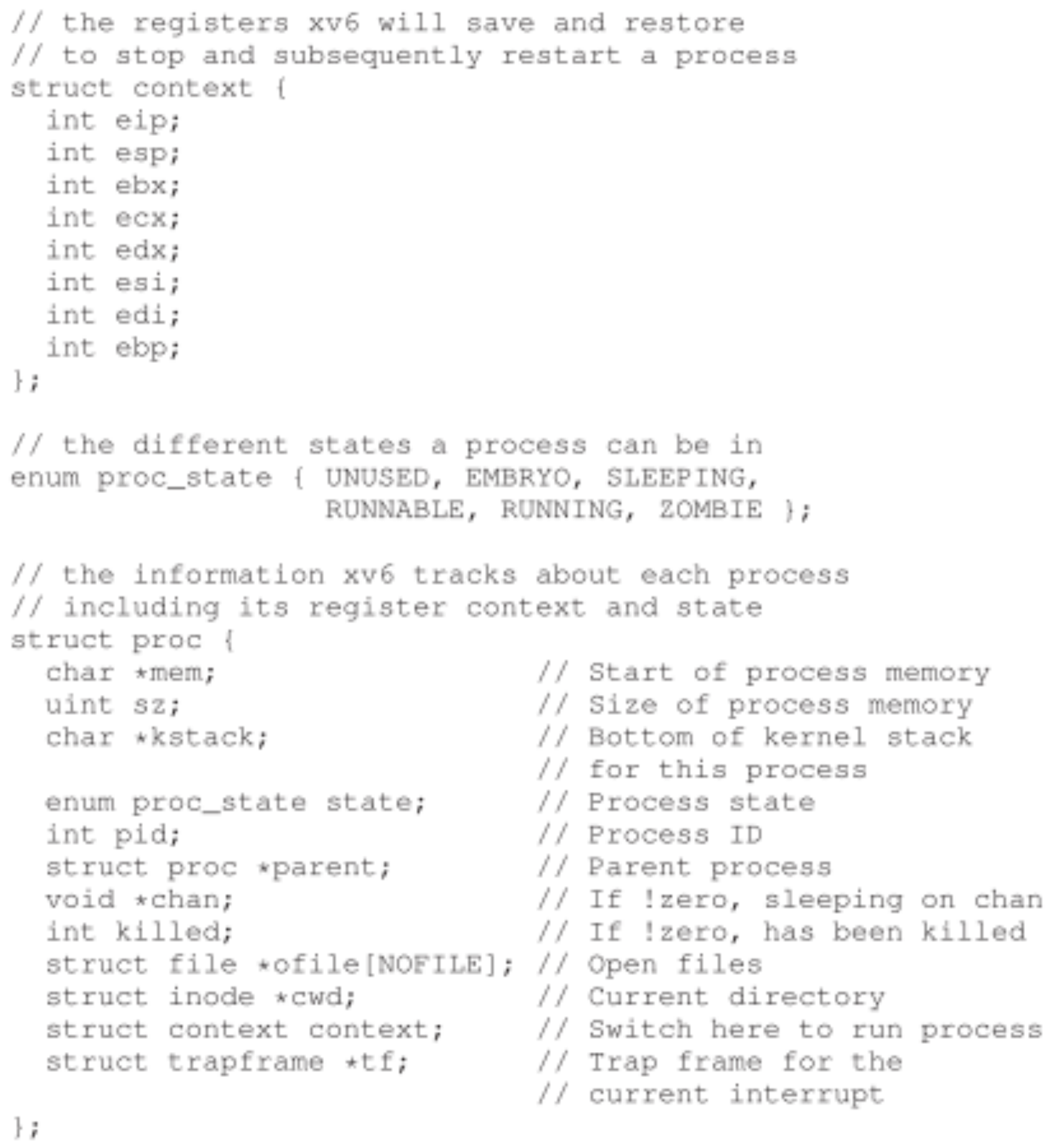
Data

- Stores process state overall
- Including the context that is saved and restored on a context switch

Process Data Structure

- Code segment + Data Segment + Stack Segment + Heap Segment – Together they are typically referred to as address space.
- Process ID
- Parent Process
- Registers (context)– note process assumes that all CPU registers are available to it.
- Other state info maintained by OS (e.g., open files, scheduling state, etc.)

Process Data Structure



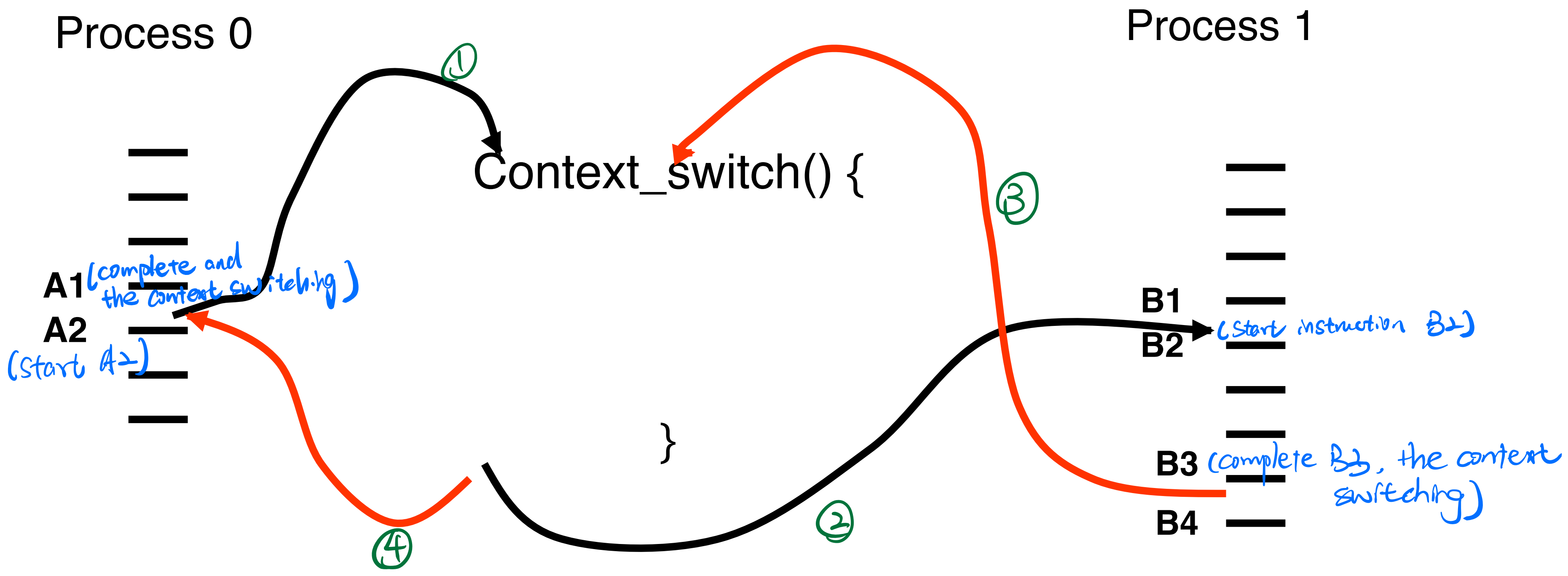
```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                   // Start of process memory
  uint sz;                     // Size of process memory
  char *kstack;                // Bottom of kernel stack
                               // for this process
  enum proc_state state;       // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  void *chan;                  // If !zero, sleeping on chan
  int killed;                  // If !zero, has been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  struct context context;      // Switch here to run process
  struct trapframe *tf;        // Trap frame for the
                               // current interrupt
};
```

Figure 4.5: **The xv6 Proc Structure**

Context Switch



Think about what it takes to create a process!

What is the advantage of a “process” view?

- Implement concurrency (between users, between activities of a user, …)
- Insulate one activity from another

Drawbacks of a process view

- They are heavy weight – higher scheduling (context switch) costs
  - Direct costs of switching address spaces.
  - Indirect costs (e.g., cache flushes)

- State Sharing is a problem

What are the overheads?

address space

- Switching code, data, stack and heap
- Saving and restoring registers
- Saving and restoring other state maintained by OS

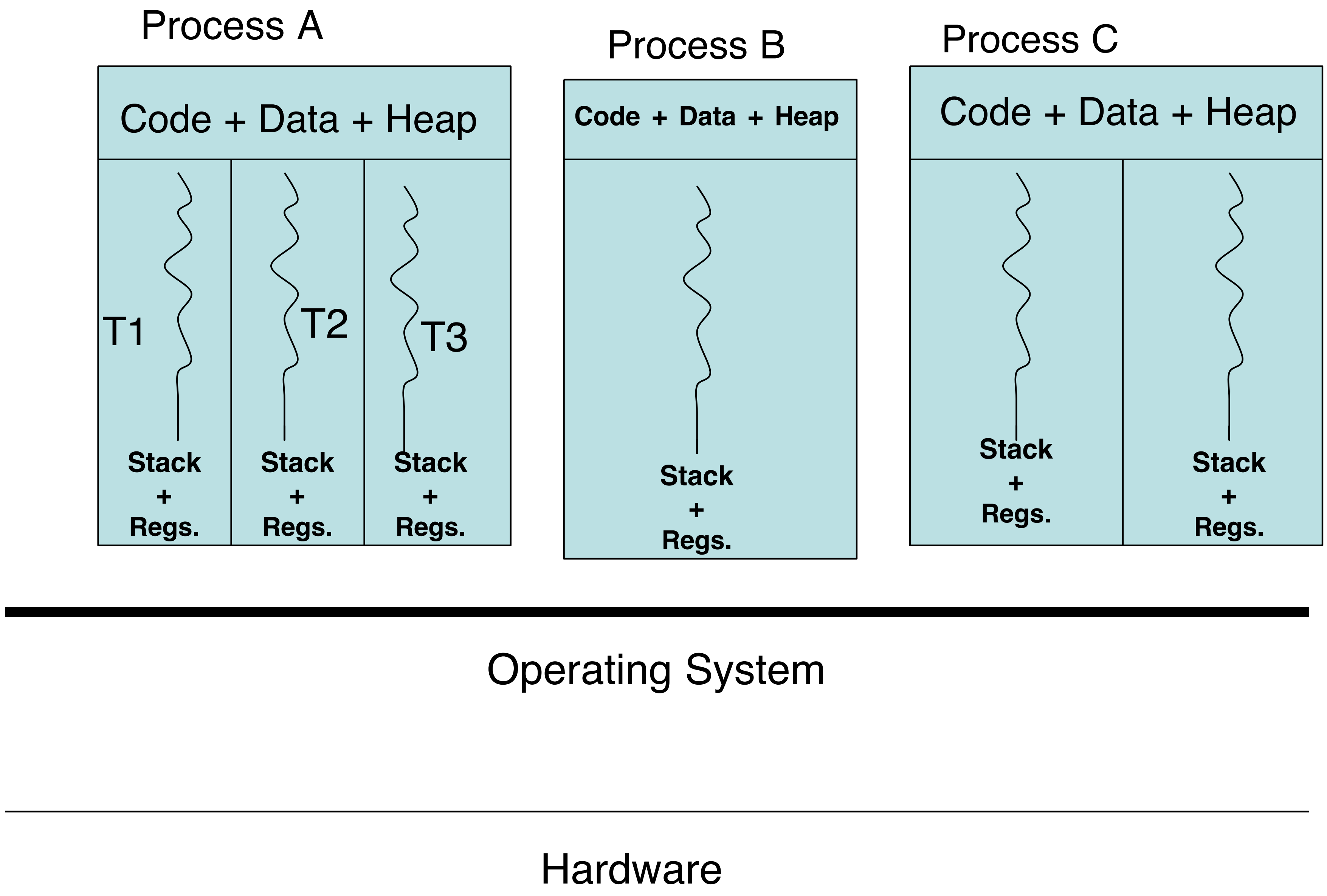
- Can we do better?

Solution: Threads

- Activities within the same address space.
- Threads within a process share (code, data, heap).
- Only stacks are disjoint.
- Switching between threads only involves switching stacks.
- Sharing is implicit
- No protection between threads of a process – but this is OK since they are meant to be cooperative.

stack
register are not shared.

Threads

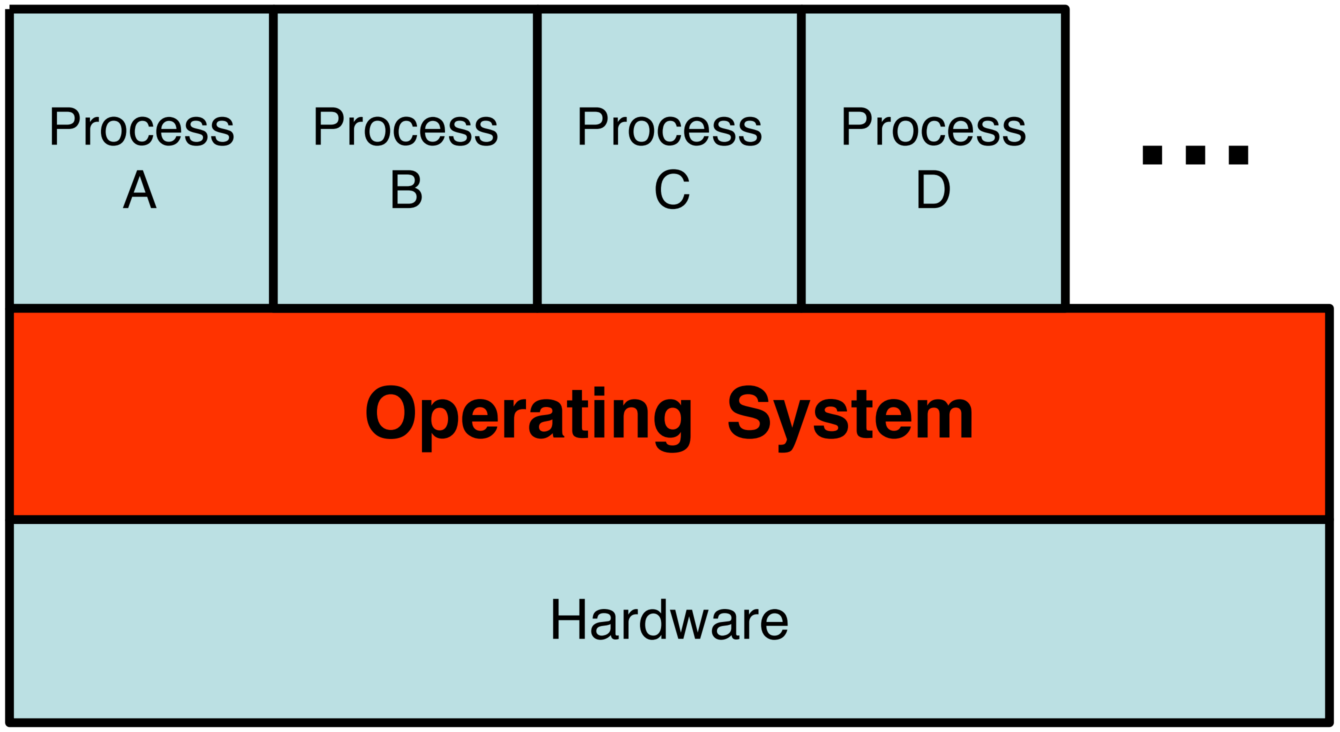

Process A
Code + Data + Heap
T1
T2
T3
Stack + Regs.
Stack + Regs.
Stack + Regs.
Process B
Code + Data + Heap
Stack + Regs.
Process C
Code + Data + Heap
Stack + Regs.
Stack + Regs.
Operating System
Hardware

Solution: Threads

- More later

Process and OS

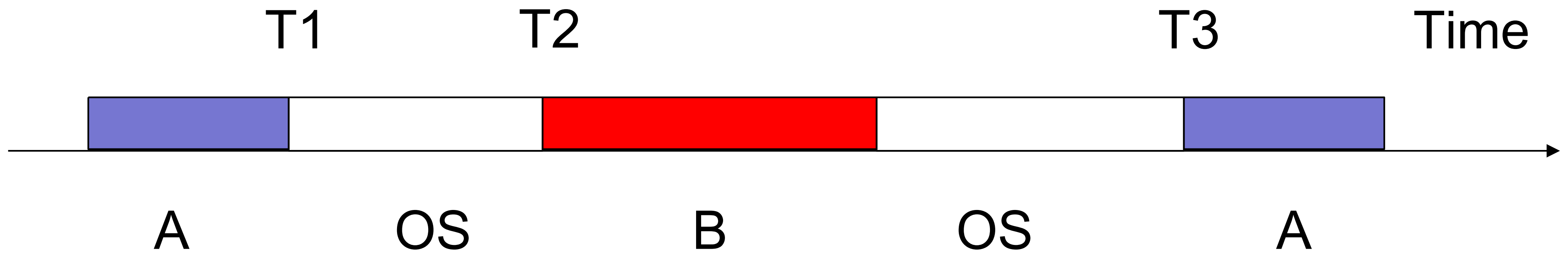
- Focused mainly on process-to-process interactions up until now
  - but enabled by processes use of OS



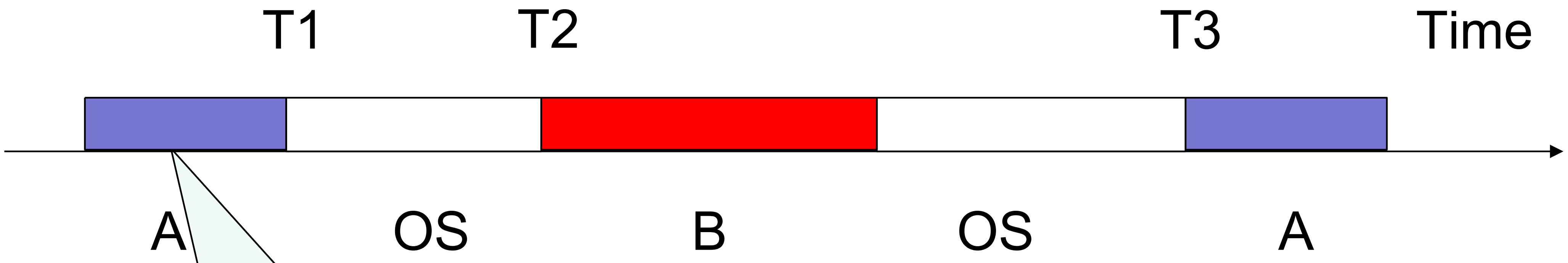
How Does the OS Help Multiple Processes Share the Same CPU?

E.g., Two Processes on a CPU

- Let’s consider (only) two processes A and B that are running on the same CPU (along with the OS)
- Let us look closely at some illuminating events in such a system



We identify four basic questions to consider



Q1: What if the process does something undesirable here?

infinite loop

try to reach outside of memory space

math : divide by 0

- What "undesirable" things might a process do?

PENNSTATE

Undesirable #1: Executing Privileged Instructions

Instructions that only execute in kernal core

- Question: Should a process be allowed to execute all instructions in the ISA?
- Answer: No
- E.g., what could go wrong if a program were allowed to execute the “halt” instruction?

[illegible]

syscall more like unprivileged instruction, but trigger traps on purpose

Privileged Instructions

- Instructions that are "security-sensitive" must be "privileged"
  - Security-sensitive: affect the operation of other process (integrity)
    - E.g., shut down computer, modify address space, modify IO
  - Security-sensitive: snoop data from other process (secrecy)
    - E.g., read address space, leak IO
  - Privileged: Run by trusted code – i.e., by the OS
  - More later…

Undesirable #2: Certain Error Conditions

- Consider the following errors our programs often run into:
  - Segmentation fault
  - Division by zero
  - More to come

Solution: Traps

- Let the CPU be designed s.t. upon the occurrence of the following, it enters a special error-like state and control jumps to OS
  - A process executes a “privileged” instruction
  - A process or the OS encounters one of these error conditions
- Such events are called **traps**

Traps for system calls

- Programs are offered a special instruction via which they can raise a trap
  - E.g., "syscall" on x86
  - Is this a privileged instruction?

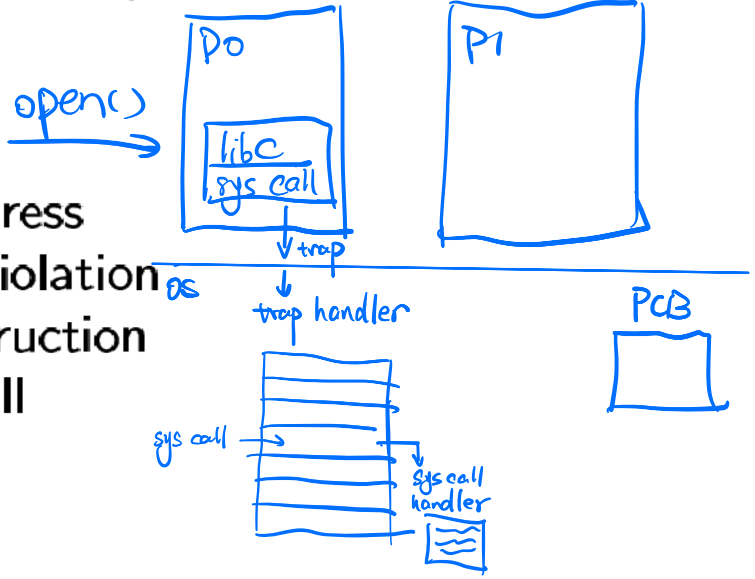
?

syscall more like unprivileged Instruction, but trigger traps on purpose

Traps

- On detecting trap, CPU must:
  - ① Save process state
  - ② Transfer control to trap handler (in OS)
    - CPU indexes *trap vector* by trap number
    - Jumps to address
  - ③ Restore process state and resume

0:	0x00080000	Illegal Address
1:	0x00100000	Memory Violation
2:	0x00100480	Illegal Instruction
3:	0x00123010	System Call
. . .	. . .	



A Final Missing Piece!

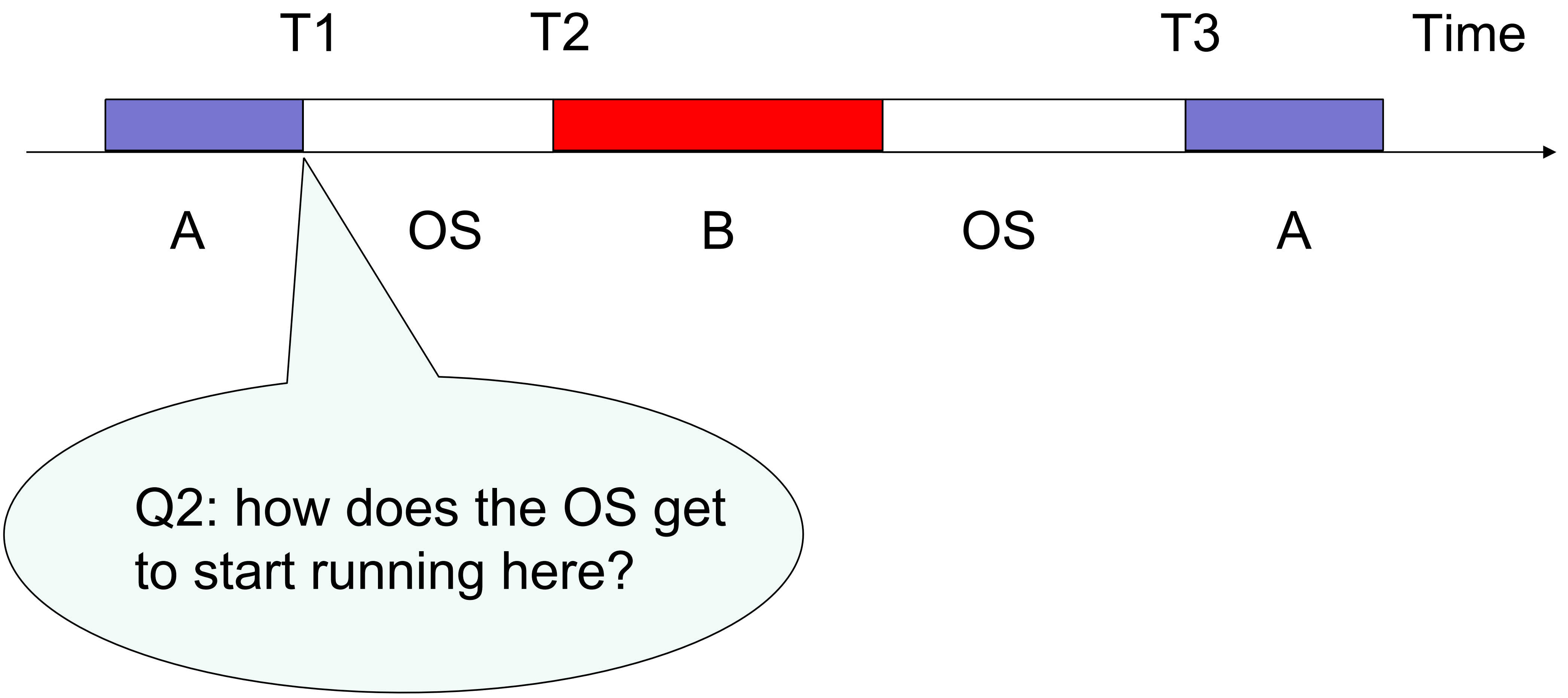
- We would like the CPU to raise a trap when a process executes a privileged instruction
- But how would the CPU know the difference between a process and the OS?
  - An instruction is an instruction!

Dual CPU Mode

- CPUs offer at least two “modes” of operation
  - User mode and Kernel (OS, Supervisor) mode
  - Execute privileged instruction in user mode ⍰ trap
  - E.g., Mode bit provided by hardware
    - Provides ability to distinguish when CPU is running process or OS
  - E.g., x86 offers four modes called “rings” with ring 0 for OS and ring 3 for processes

Dual CPU Mode

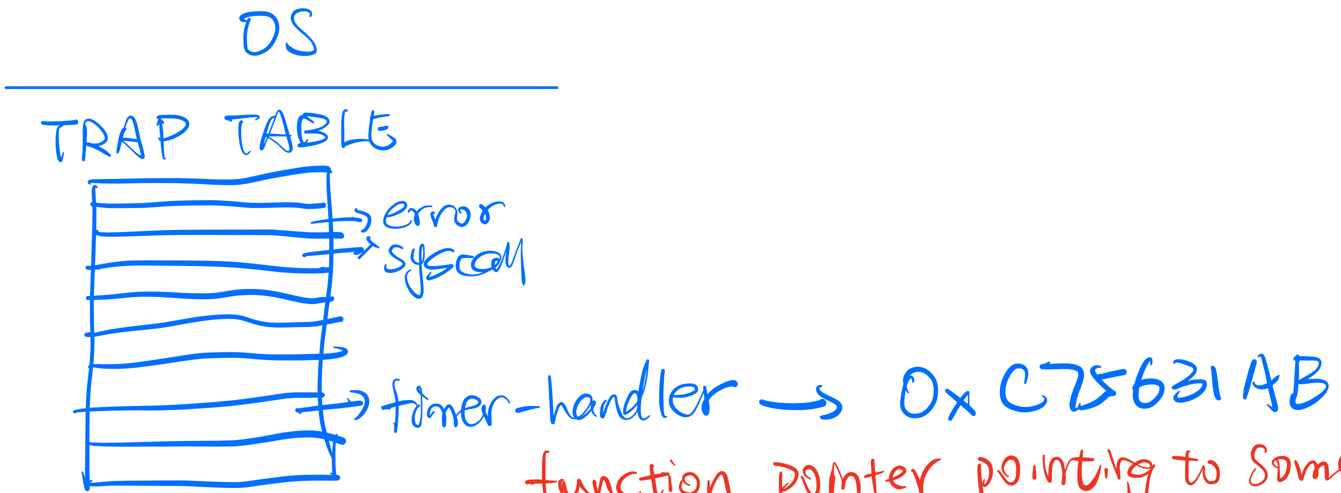
- OS runs with CPU in kernel mode
- Is responsible to ensure programs run with CPU in user mode
- What is required to realize the above?
  - OS is the first software to run!
    - The booting up of the OS
  - OS has the ability to change CPU mode from kernel to user
  - Programs have the ability to change CPU mode from user to kernel



- Need some way outside the process's control to force control back to the OS

Interrupts

- There must be a mechanism via which the OS gets a chance to run on the CPU every so often
  - E.g., A timer interrupt that periodically lets the OS run, typically, once every few milliseconds



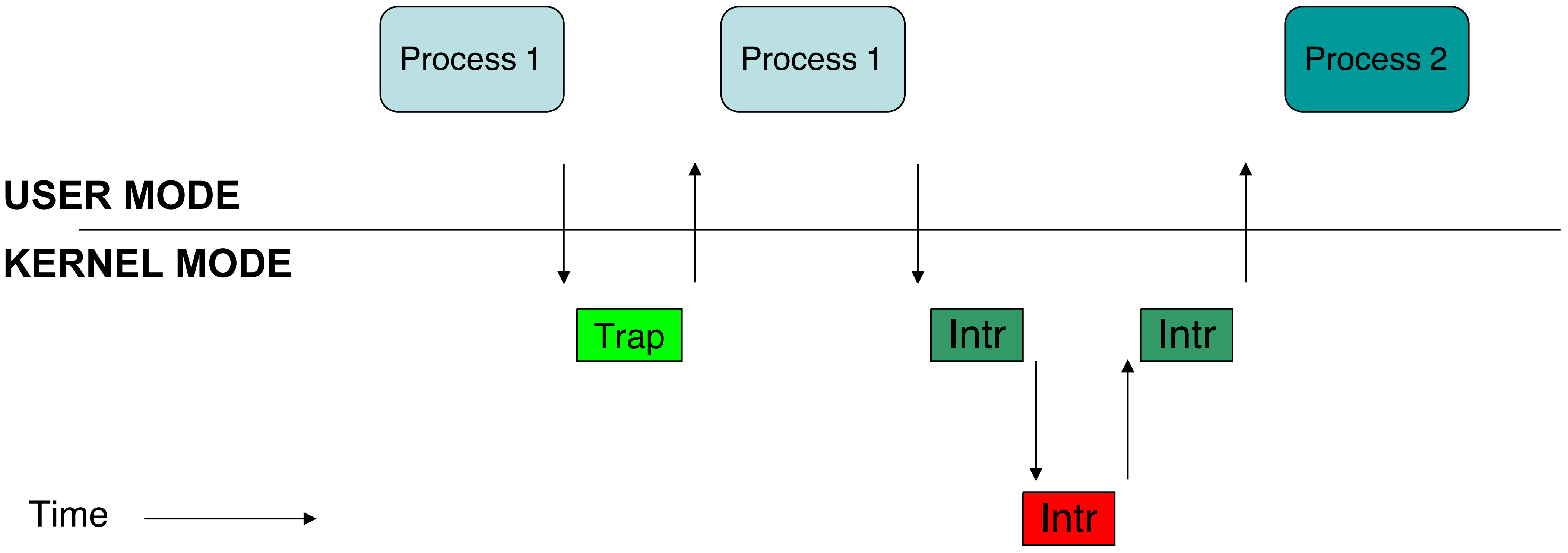
function pointer pointing to some function that will run to handle the timer interrupt

Interrupts

More generally:

- Interrupts are special conditions **external to the CPU** that require OS attention
  - Note difference from traps
- CPU designed to switch from user space to kernel mode upon detecting an interrupt
  - Example: A keystroke raises an interrupt

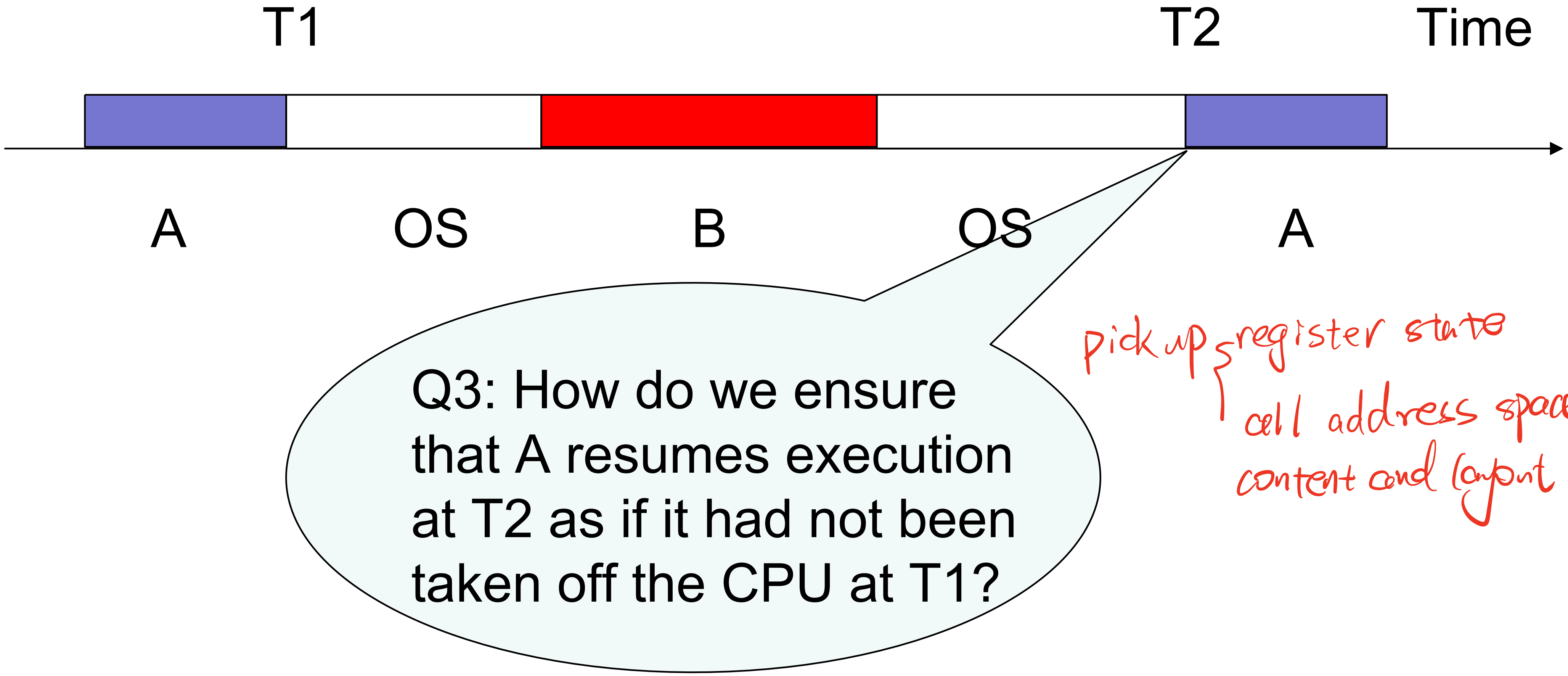
Interrupts and Traps



- Only two ways to enter supervisor mode from user mode

Interrupts

- Are fundamental to I/O processing
  - Which we will discuss in detail later…



Q3: How do we ensure that A resumes execution at T2 as if it had not been taken off the CPU at T1?

pick up { register state
all address space content and layout

- By ensuring that we save the entire “state” of A at T1 and can resume it from this state at T2
- state(A, T1) == state(A, T2)
- What is the state of A at T1?

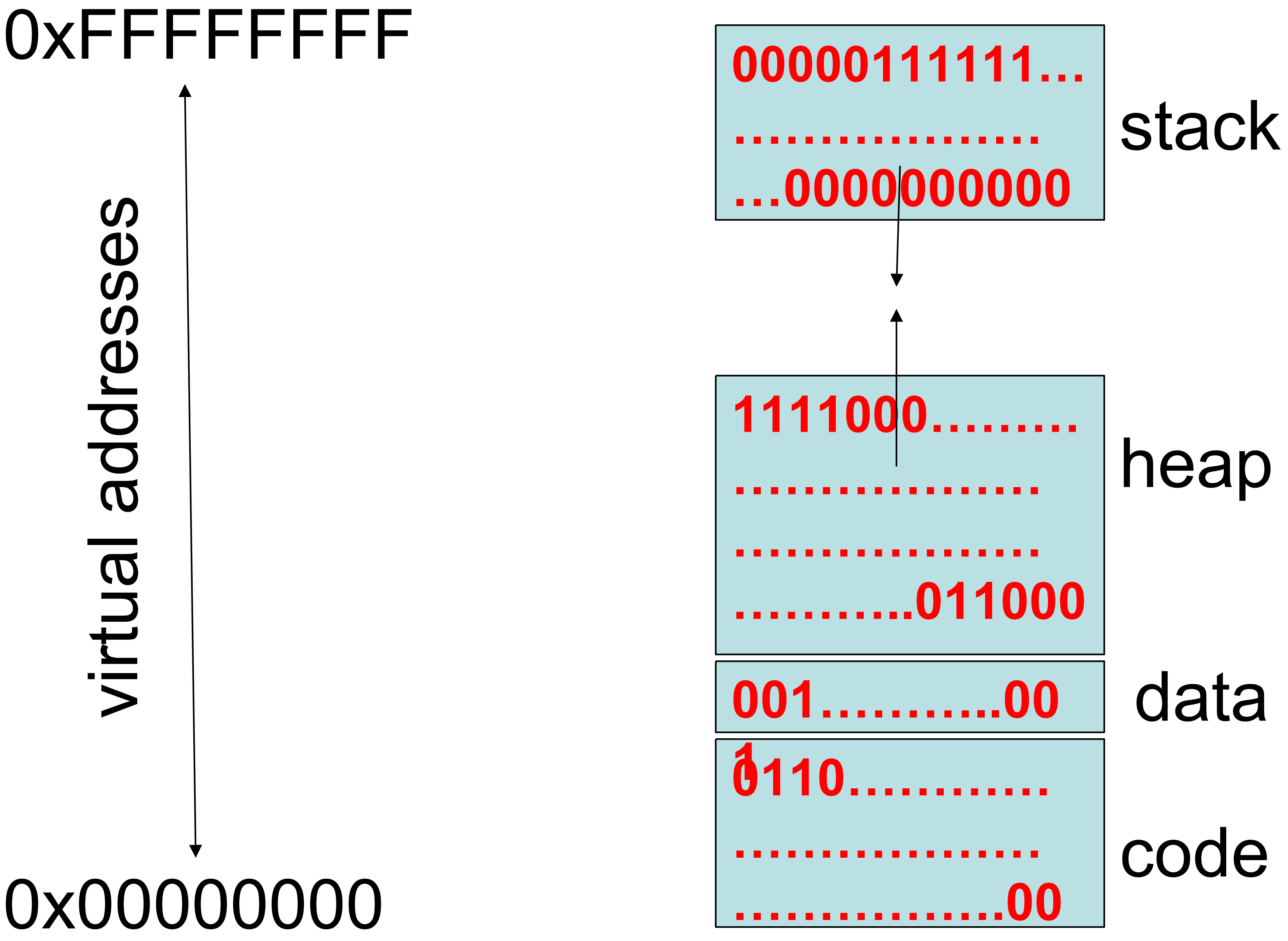
[illegible] respect to all the hardware

↳ state of process respect to [illegible]

State of A at time T1 (1)

- #1: Contents of A’s address space
  - What are the code, data, heap, and stack values of the process at T1?

A's Address Space



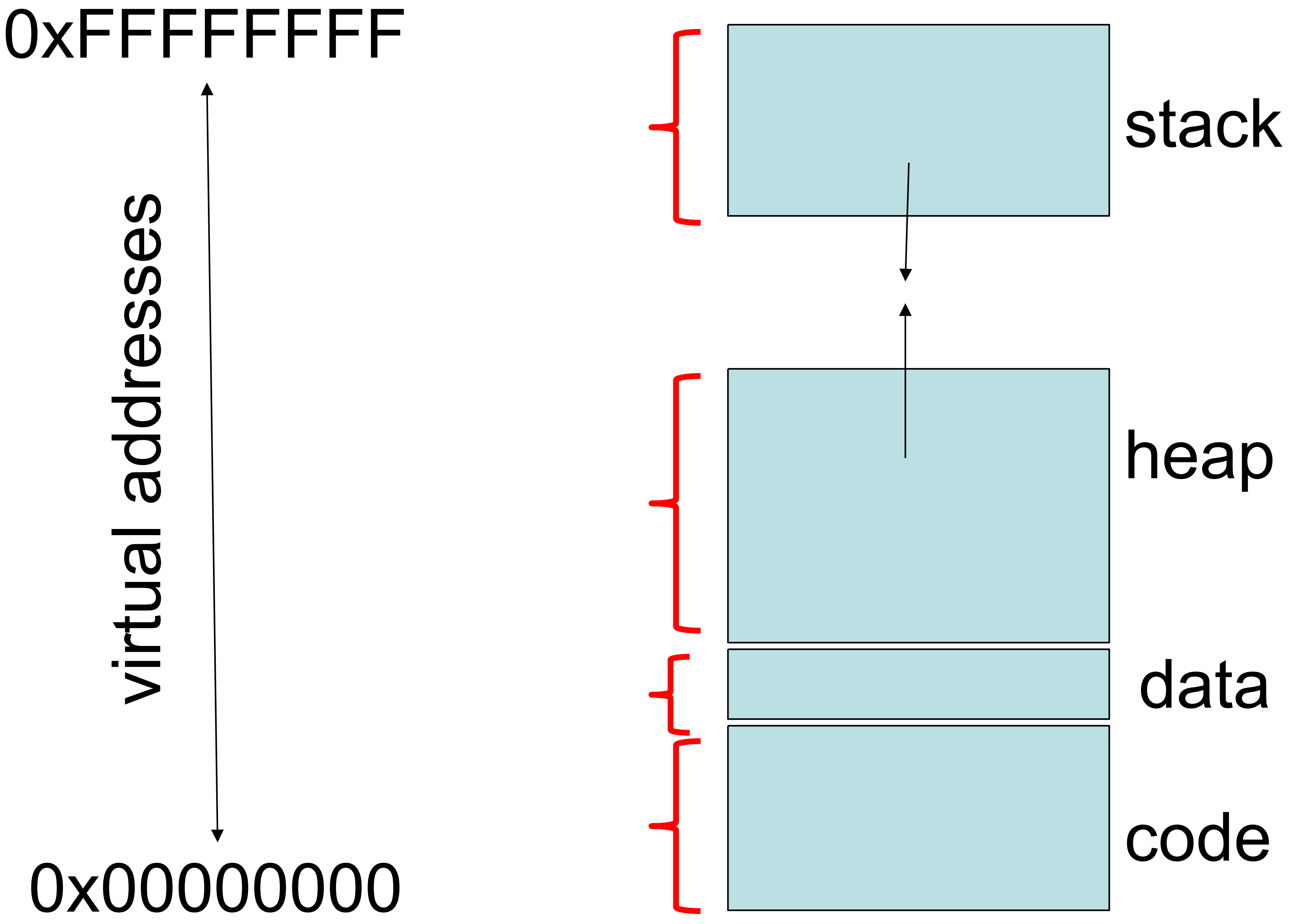
State of A at time T1 (1)

- #1: Contents of A’s address space
  - What are the code, data, heap, and stack values of the process at T1?
- Q: Where do these reside at time T1?
  - In a portion of main memory set aside for A
  - We rely on memory manager to ensure they remain unchanged by other processes during [T1, T2]
    - More details when we study virtual memory management

State of A at time T1 (2)

- #2: Layout of A's address space
  - The address ranges the code, data, heap, stack span

Layout of Address Space



State of A at time T1 (2)

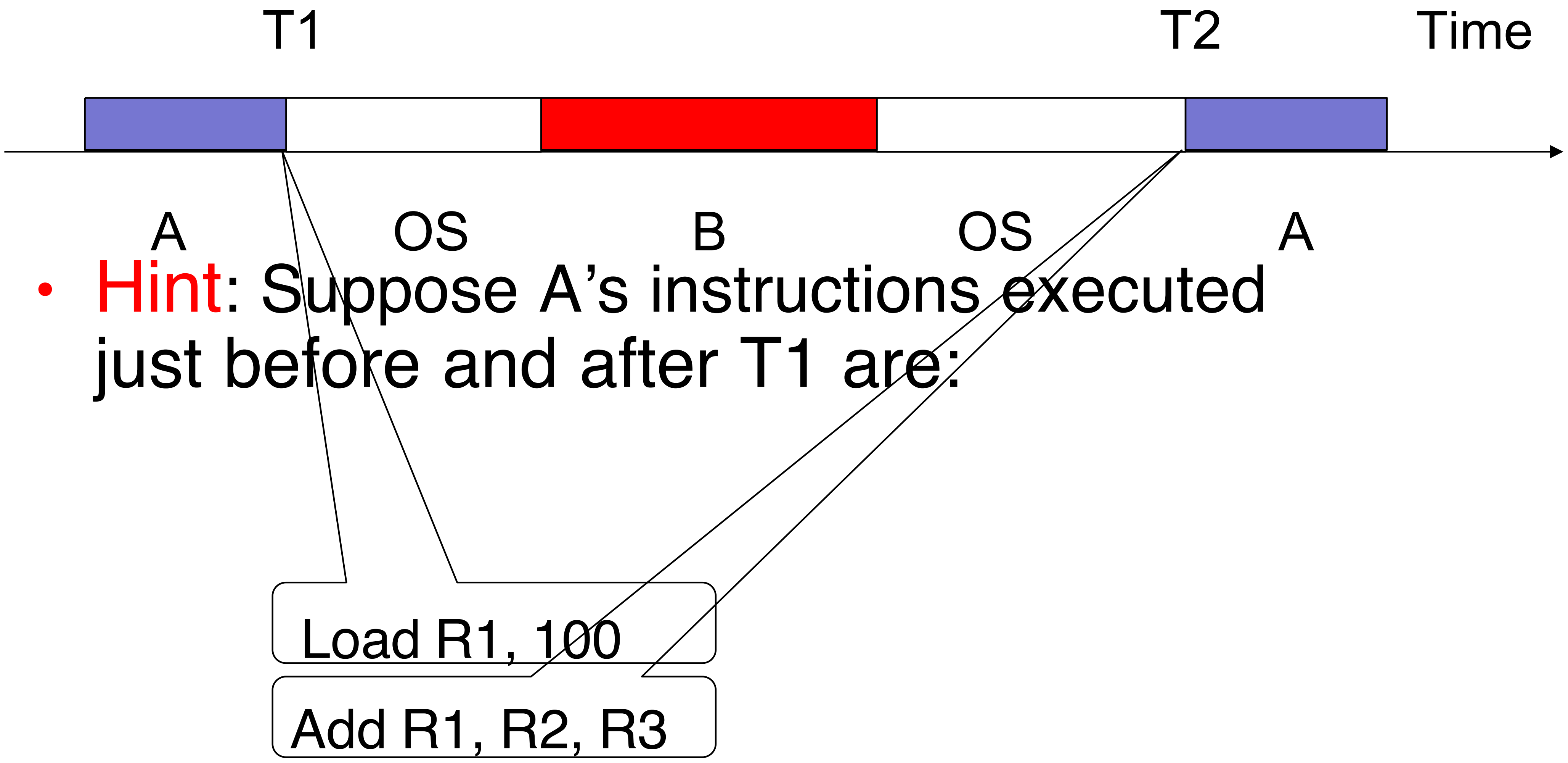
- Layout of A’s address space
  - The address ranges the code, data, heap, stack span
- Q: Where are these address ranges stored?
  - Somewhere in memory
  - In whose address space? Again, A’s address space is a valid choice

State of A at time T1 (3)

- #3: All the register values at time T1 need to be saved in main memory and restored at time T2
- Called the **hardware context** of process A
- Typically, the hardware context specifies the runtime state of the process
  - E.g., Stack Pointer Register (SP)
  - E.g., Program Counter (PC)

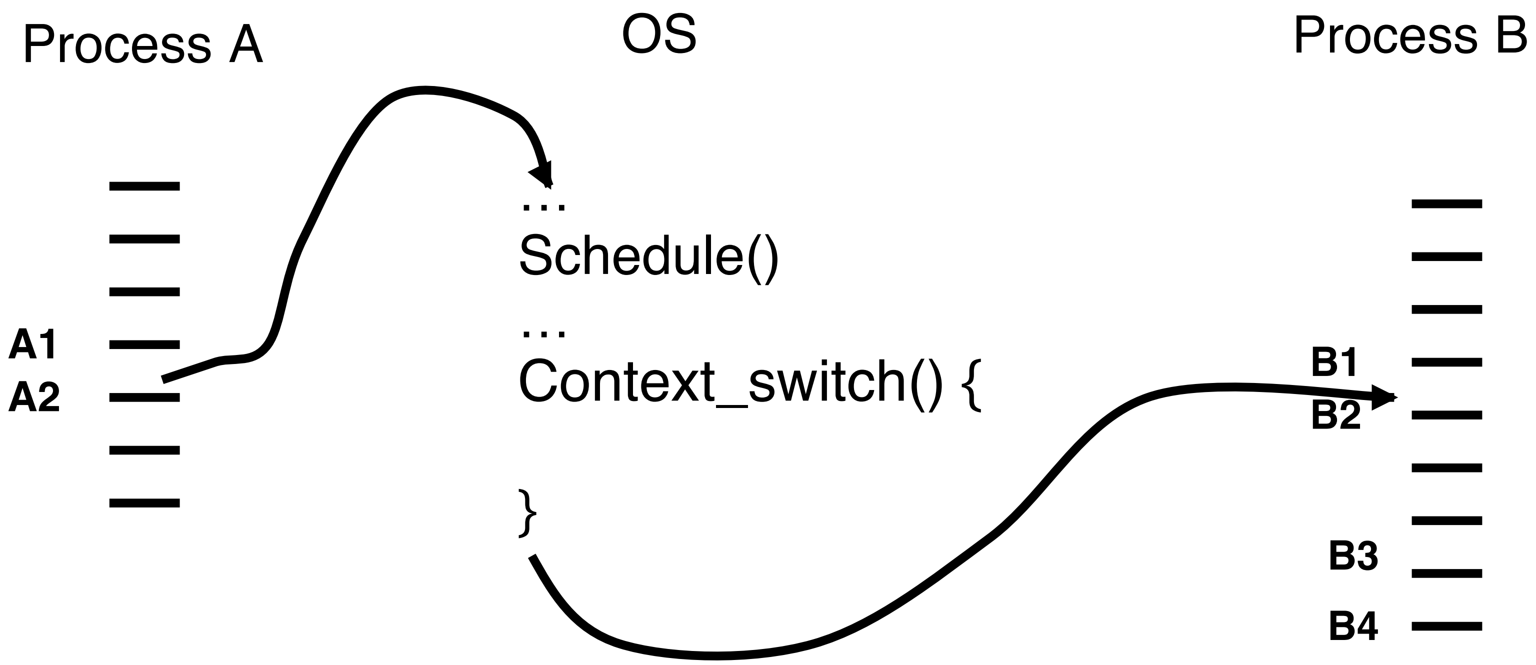
State of A at time T1 (3)

- Anything else?



- Hint: Suppose A's instructions executed just before and after T1 are:

Context Switch



Context Switch: More Detail

OS @ boot (kernel mode)	**Hardware**
initialize trap table	
	remember addresses of... syscall handler timer handler
start interrupt timer	
	start timer interrupt CPU in X ms

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
	store in Process control block which in kernal	Process A ... core heap segment
kernel $\rightarrow$	**timer interrupt** save regs(A) → k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call `switch()` routine save regs(A) → proc_t(A) restore regs(B) ← proc_t(B) switch to k-stack(B) **return-from-trap (into B)**		
	restore regs(B) ← k-stack(B) move to user mode jump to B's PC	
		Process B ...

State of A at time T1 (4)

- #4: I/O resources being used by the process
  - E.g., open files, network sockets, etc.

- How does your process reference an open file?
  - E.g., via the *open* syscall

State of A at time T1 (4)

- #4: I/O resources being used by the process
  - E.g., open files, network sockets, etc.
- Information held by the OS in its own address space
  - More when we discuss I/O

PENNSTATE

1855

Questions?