CS202 – Advanced Operating Systems

Processes

January 13, 2025

What do modern operating systems support?

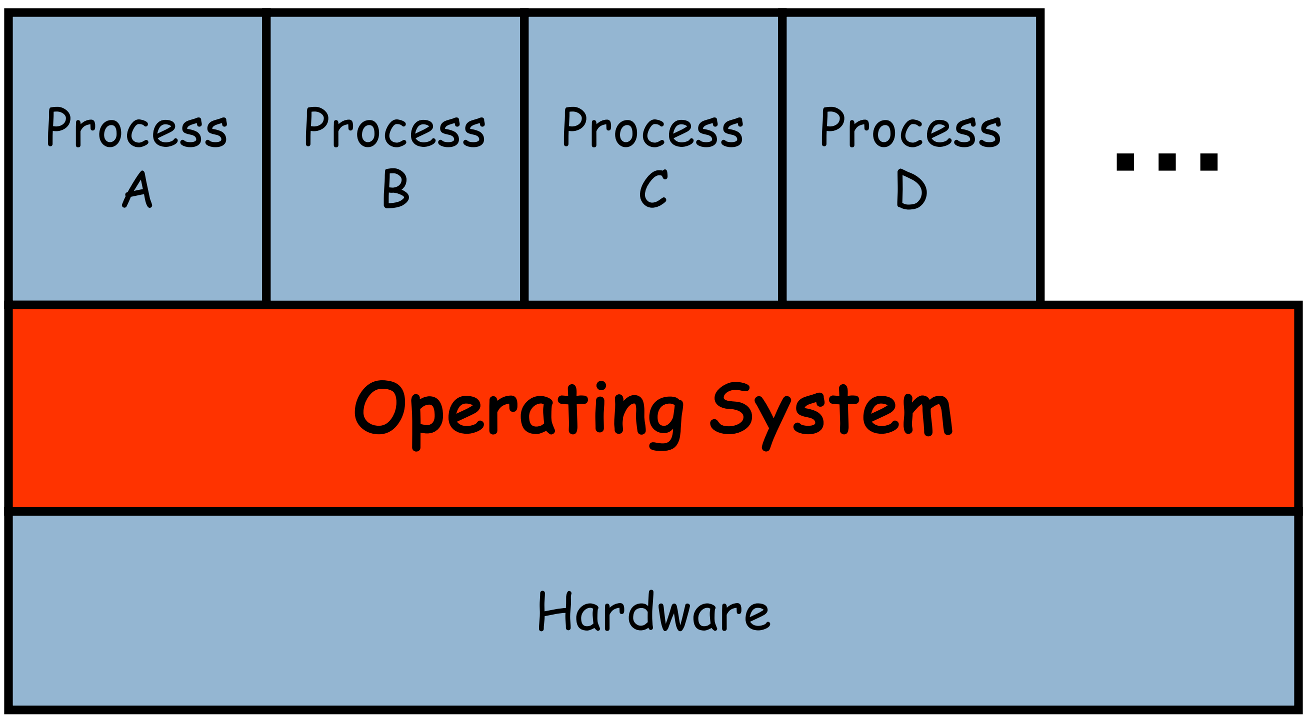
- **Virtualize** hardware/architectural resources
  - Easy for programs to interact with hardware resources
  - Share hardware resource among programs
  - Protect programs from each other (security)
- Execute multithreaded programs **concurrently**
  - Support multithreaded programming model
  - Execute multithreaded programs efficiently
- Store data **persistently**
  - Store data safely
  - Secure

The Process

- The process is the OS abstraction for CPU execution
  - It is the unit of execution
  - It is the unit of scheduling
- A process is a program in execution
  - Programs are static entities with the potential for execution
  - Process is the animated/active program
    - Starts from the program, but also includes dynamic state
    - As the representative of the program, it is the “owner” of other resources (memory, files, sockets, …)
- How does the OS implement this abstraction?
  - How does it share the CPU?

Process and OS

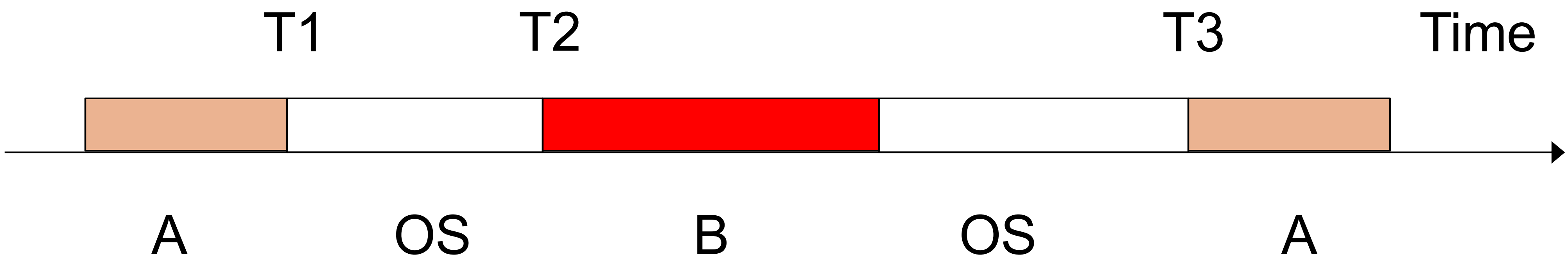
- Multiple programs (or the same program multiple times) may be loaded as processes
  - Use of system resources (hardware) is managed by the OS



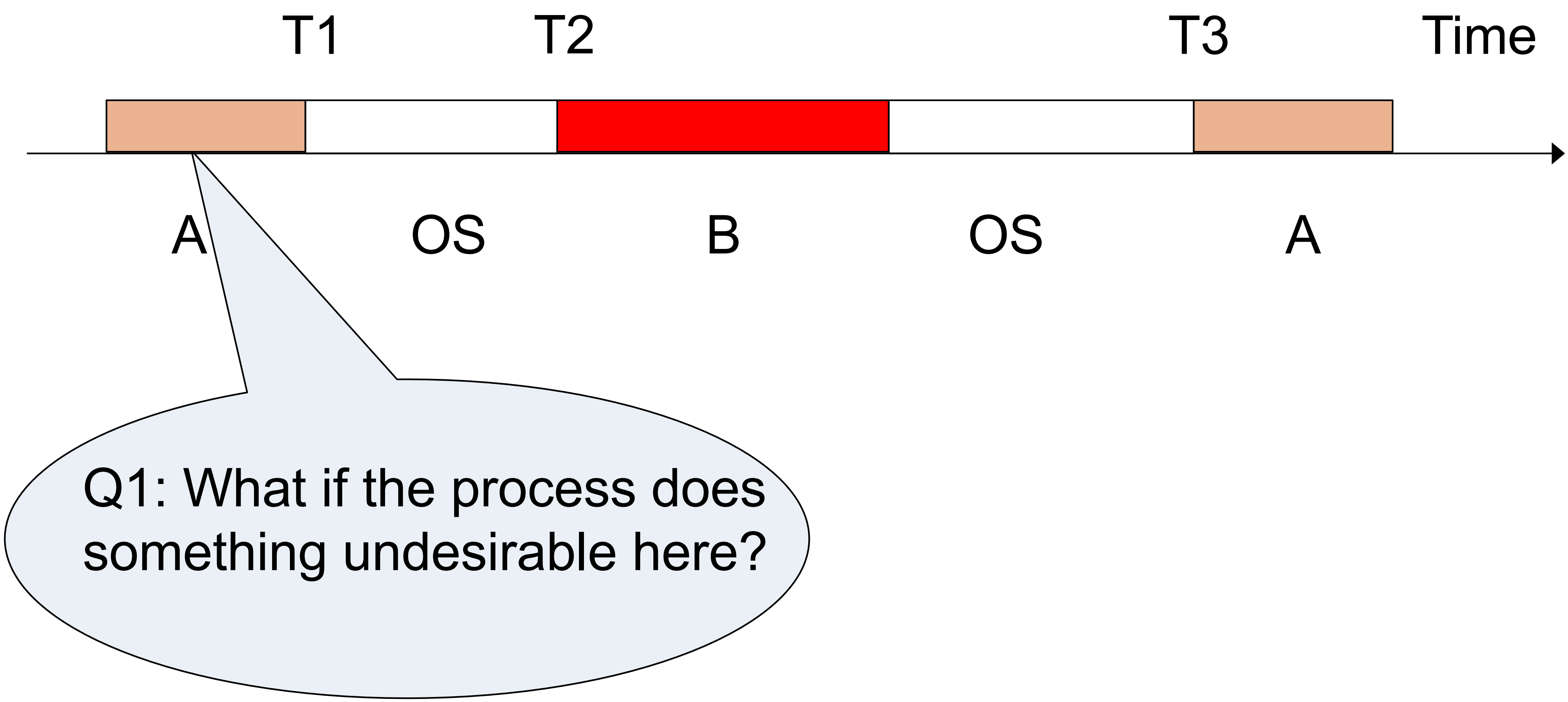
How Does the OS Help Multiple Processes Share the Same CPU?

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:



- What “undesirable” things might a process do?

Undesirable #1: Privileged Instructions

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

Privileged Instructions

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

Undesirable #2: Error Conditions

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

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**

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**
- **Or exceptions or software interrupts**

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?

Traps

- On detecting trap, CPU must:
  - Save process state
  - Transfer control to trap handler (in OS)
    - CPU indexes *trap vector* by trap number
    - Jumps to address
  - 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!

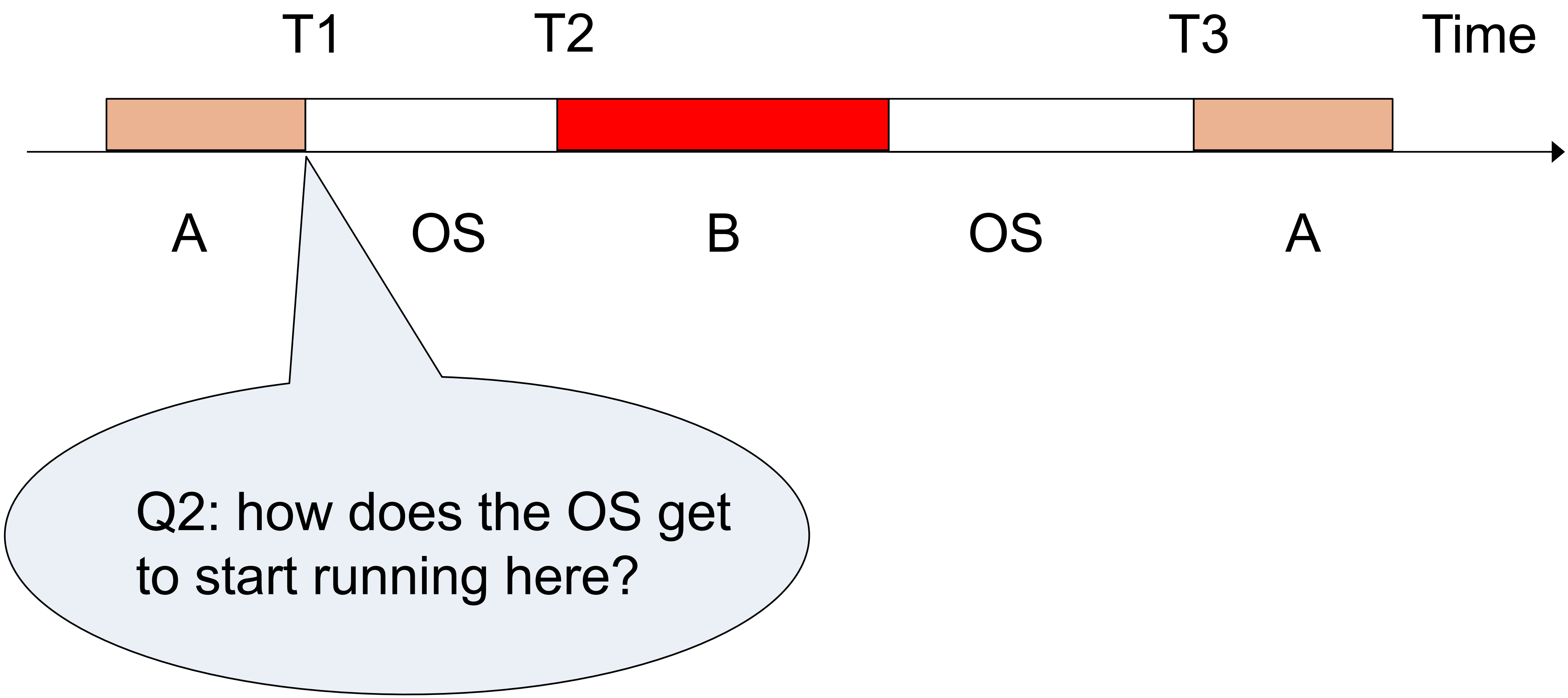
- We would like the CPU to raise a trap when a process executes a privileged instruction
- But how would the CPU know the difference between a process and the OS?
  - An instruction is an instruction!

Dual CPU Mode

- CPUs offer at least two “modes” of operation
  - User mode and Kernel mode (OS, Supervisor)
  - 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

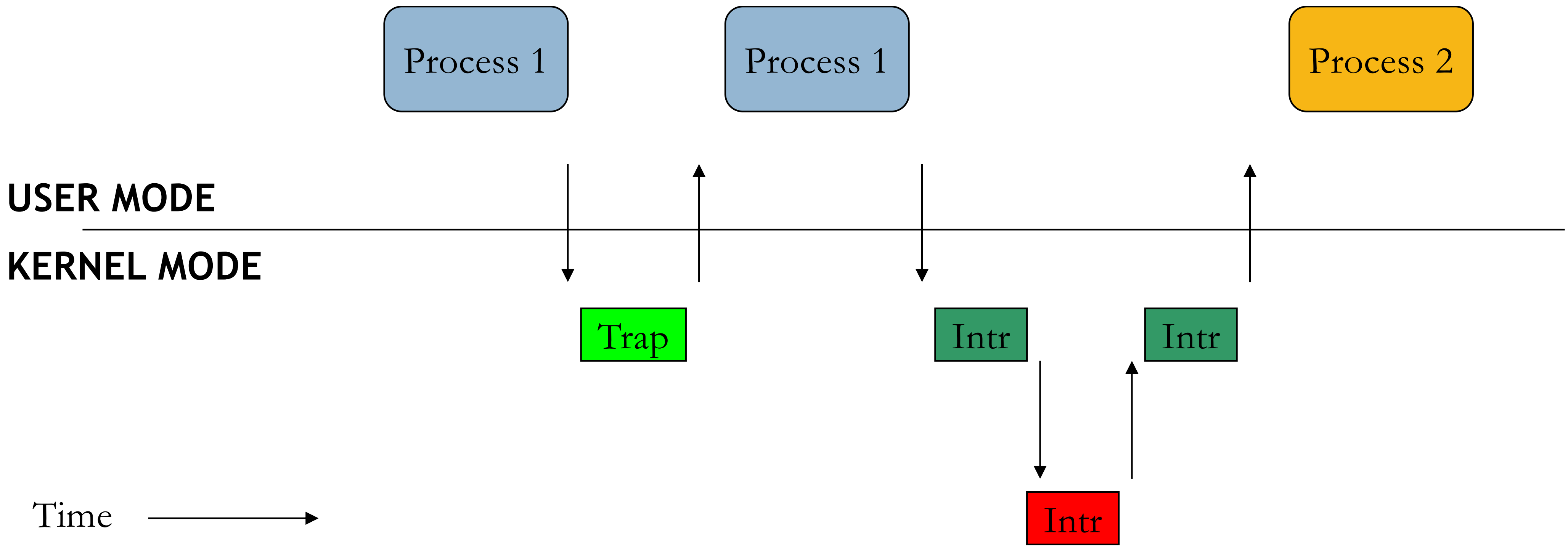
- There must be a mechanism via which the OS gets a chance to run on the CPU every so often
  - E.g., A timer interrupt that periodically lets the OS run, typically, once every few milliseconds

Interrupts

More generally:

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

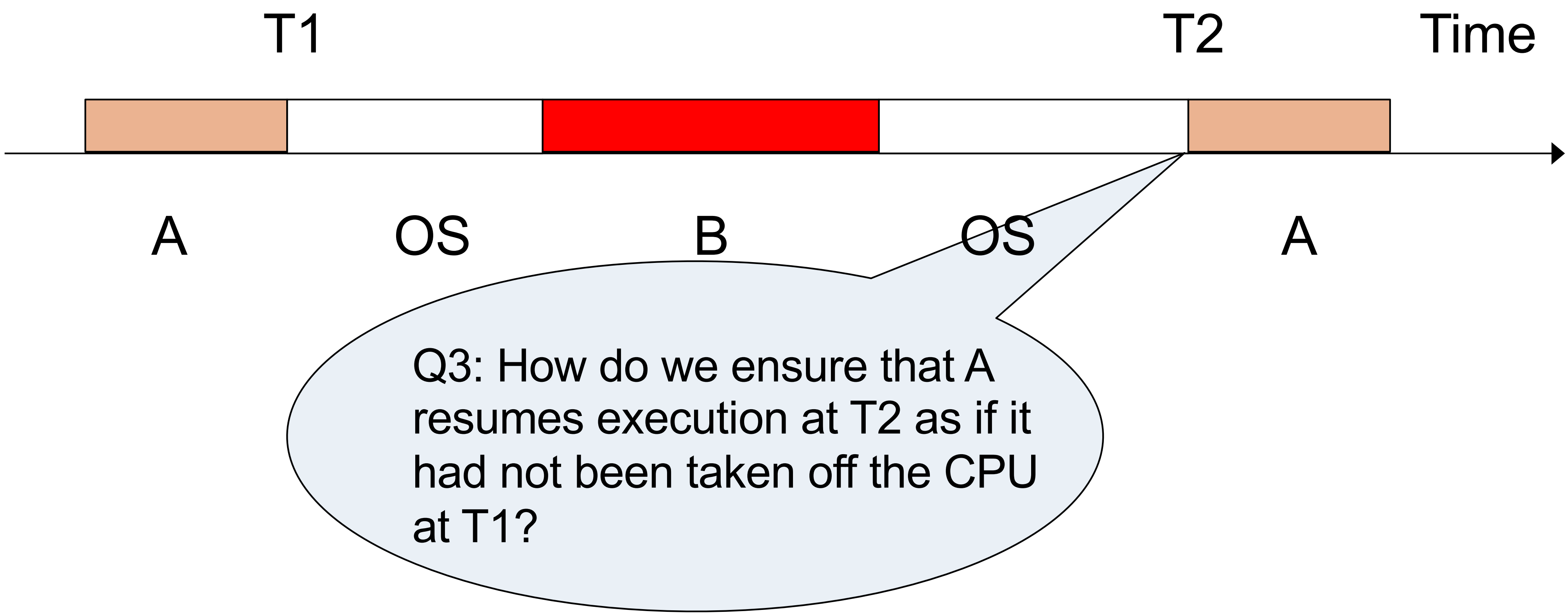
Interrupts and Traps



- Only two ways to enter supervisor mode from user mode

Interrupts

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

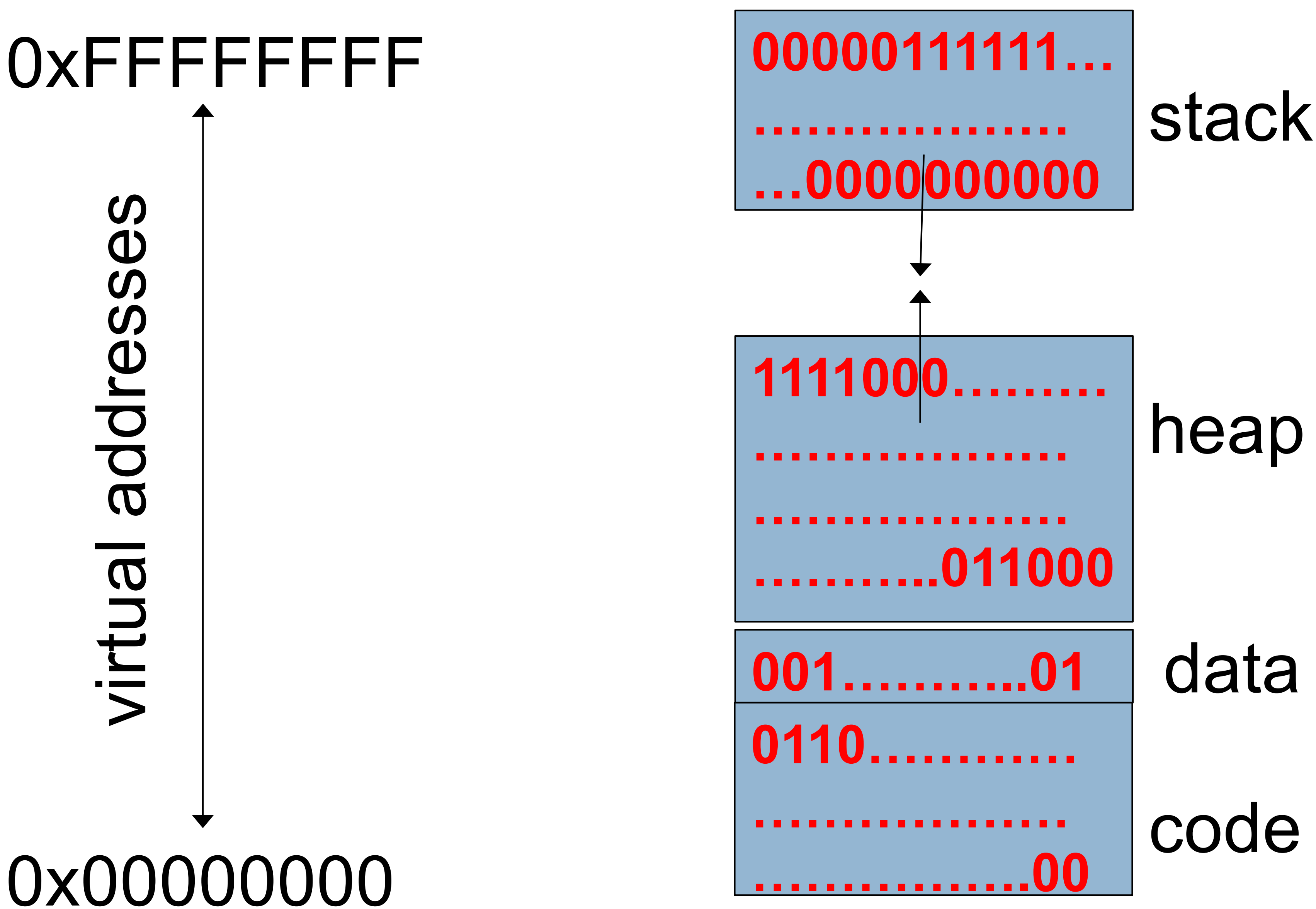


- By ensuring that we save the entire “state” of A at T1 and can resume it from this state at T2
- state(A, T1) == state(A, T2)
- What is the state of A at T1?

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



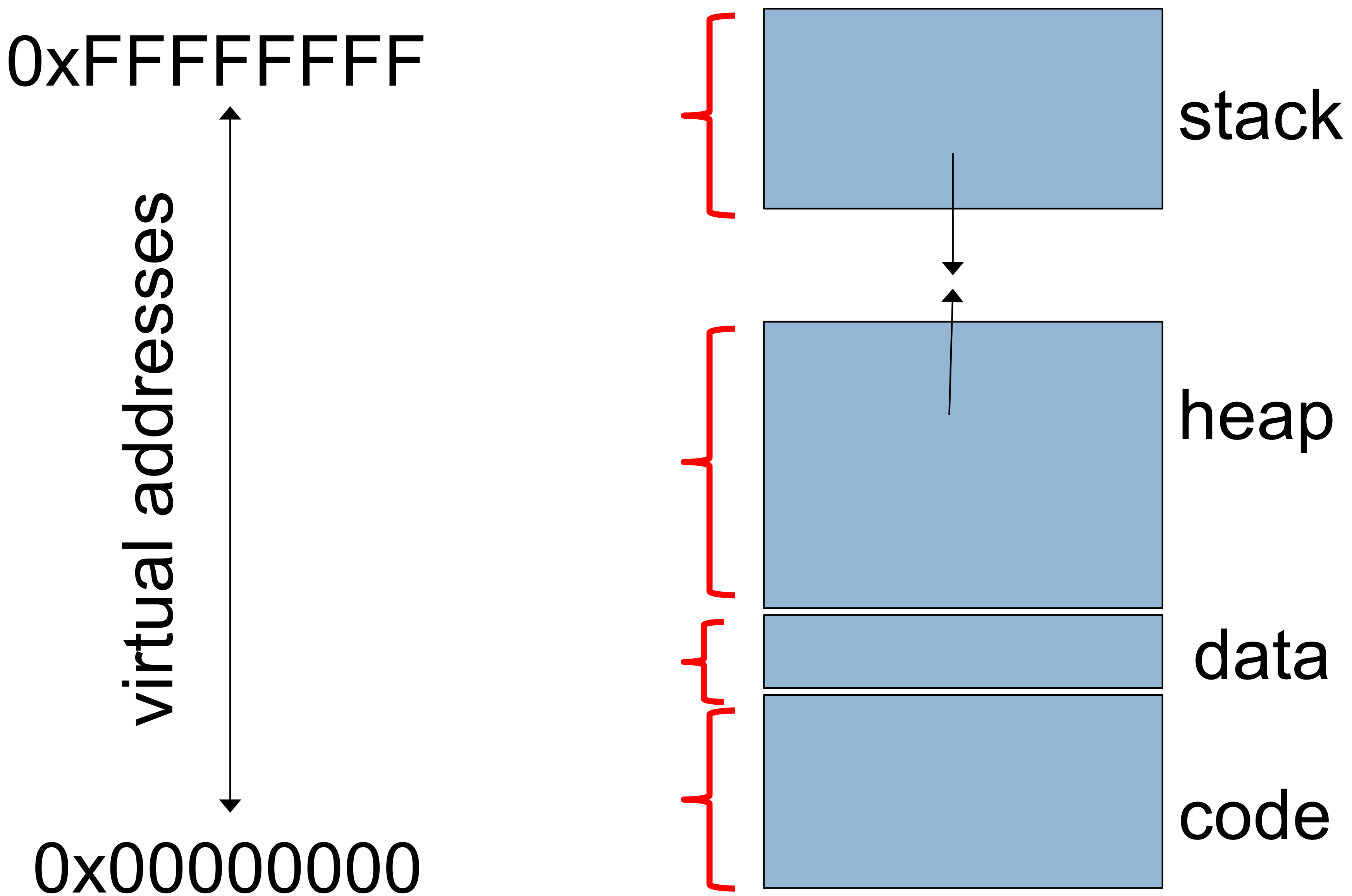
State of A at time T1 (1)

- #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)

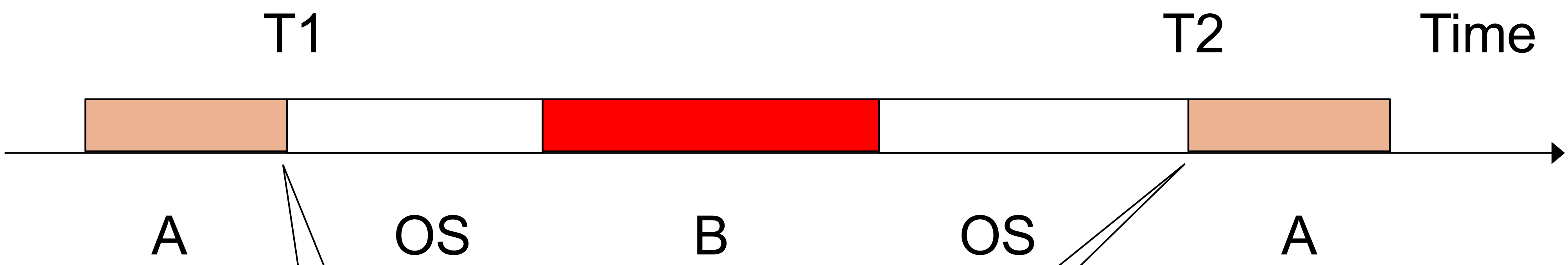
- #3: All the register values at time T1 need to be saved in main memory and restored at time T2

- 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?

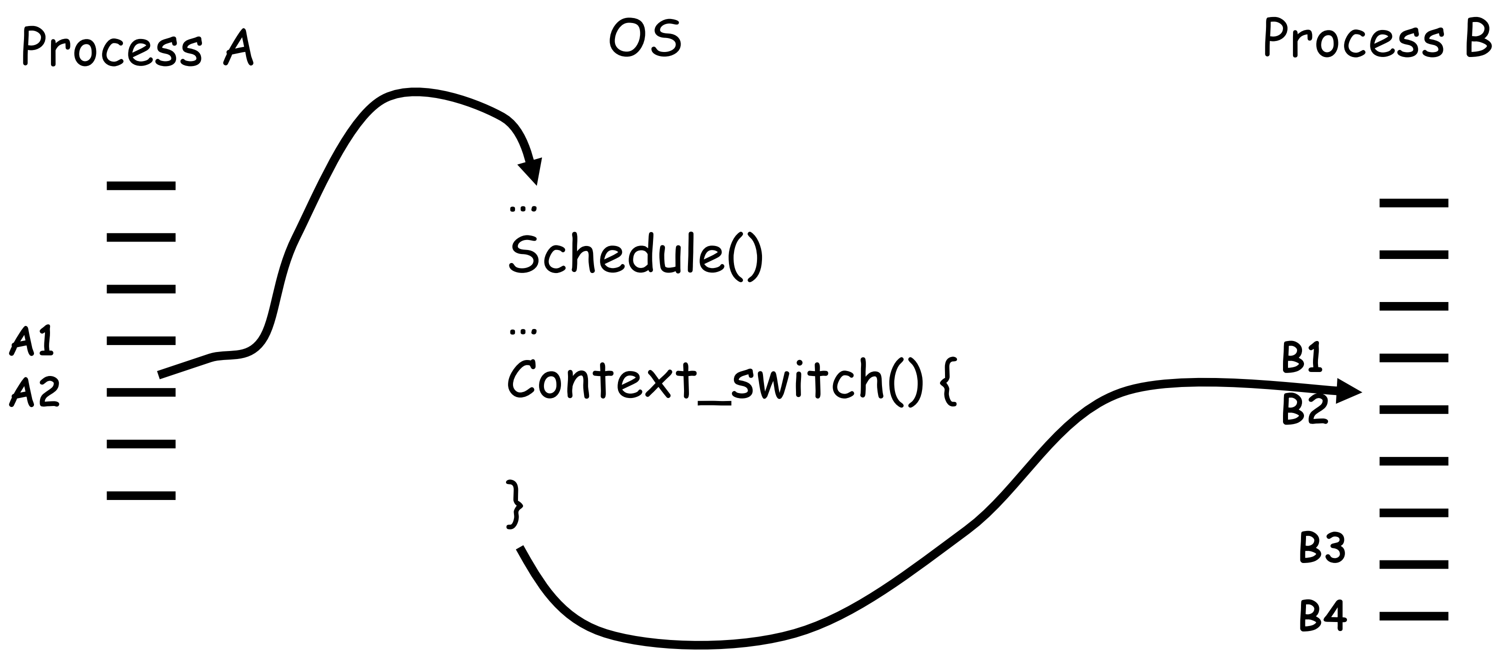


- Hint: Suppose A's instructions executed just before and after T1 are:

Load R1, 100

Add R1, R2, R3

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)**
		Process A ...
	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

The idea: virtualization

- The operating system presents an illusion of a virtual machine to each running program and maintains architectural states of a von Neumann machine
  - Processor
  - Memory
  - I/O
- Each virtualized environment accesses architectural facilities through some sort of application programming interface (API)
- Dynamically map those virtualized resources into physical resources

Demo, Virtualization


```
double a;

int main(int argc, char *argv[])
{
    int cpu, status, i;
    int *address_from_malloc;
    cpu_set_t my_set;       // Define your cpu_set bit mask.
    CPU_ZERO(&my_set);      // Initialize it all to 0, i.e. no CPUs selected.
    CPU_SET(4, &my_set);    // set the bit that represents core 7.
    sched_setaffinity(0, sizeof(cpu_set_t), &my_set); // Set affinity of this process to the defined mask, i.e. only 7.
    status = syscall(SYS_getcpu, &cpu, NULL, NULL);
```

getcpu system call to retrieve the executing CPU ID

```
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s process_nickname\n",argv[0]);
        exit(1);
    }

    srand((int)time(NULL)+(int)getpid());
    a = rand();
```

create a random number

```
    fprintf(stderr, "\nProcess %s is using CPU: %d. Value of a is %lf and address of a is %p\n",argv[1], cpu, a, &a);
    sleep(1);
```

print the value of a and address of a

```
    fprintf(stderr, "\nProcess %s is using CPU: %d. Value of a is %lf and address of a is %p\n",argv[1], cpu, a, &a);
    sleep(3);
```

print the value of a and address of a again after sleep

```
    return 0;
}
```

Virtualization Demo

```
Process C is using CPU: 4. Value of a is 685161796.000000 and address of a is 0x6010b0
Process A is using CPU: 4. Value of a is 217757257.000000 and address of a is 0x6010b0
Process B is using CPU: 4. Value of a is 2057721479.000000 and address of a is 0x6010b0
Process D is using CPU: 4. Value of a is 1457934803.000000 and address of a is 0x6010b0
Process C is using CPU: 4. Value of a is 685161796.000000 and address of a is 0x6010b0
Process A is using CPU: 4. Value of a is 217757257.000000 and address of a is 0x6010b0
Process B is using CPU: 4. Value of a is 2057721479.000000 and address of a is 0x6010b0
Process D is using CPU: 4. Value of a is 1457934803.000000 and address of a is 0x6010b0
```

Different values

The same processor!

Different values are preserved

The same memory address!

Why virtualization

- How many of the following statements are true about why operating systems virtualize running programs?
  ① Virtualization can help improve the utilization and the throughput of the underlying hardware
  ② Virtualization may allow the system to execute more programs than the number of physical processors installed in the machine
  ③ Virtualization may allow a running program or running programs to use more than the installed physical memory
  ④ Virtualization can improve the latency of executing each program
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

Why virtualization

- How many of the following statement is true about why operating systems virtualize running programs?
  1. Virtualization can help improve the utilization and the throughput of the underlying hardware
  2. Virtualization may allow the system to execute more programs than the number of physical processors installed in the machine
  3. Virtualization may allow a running program or running programs to use more than install physical memory

     Make programs less machine-dependent
  4. Virtualization can improve the latency of executing each program

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

What the OS must track for a process?

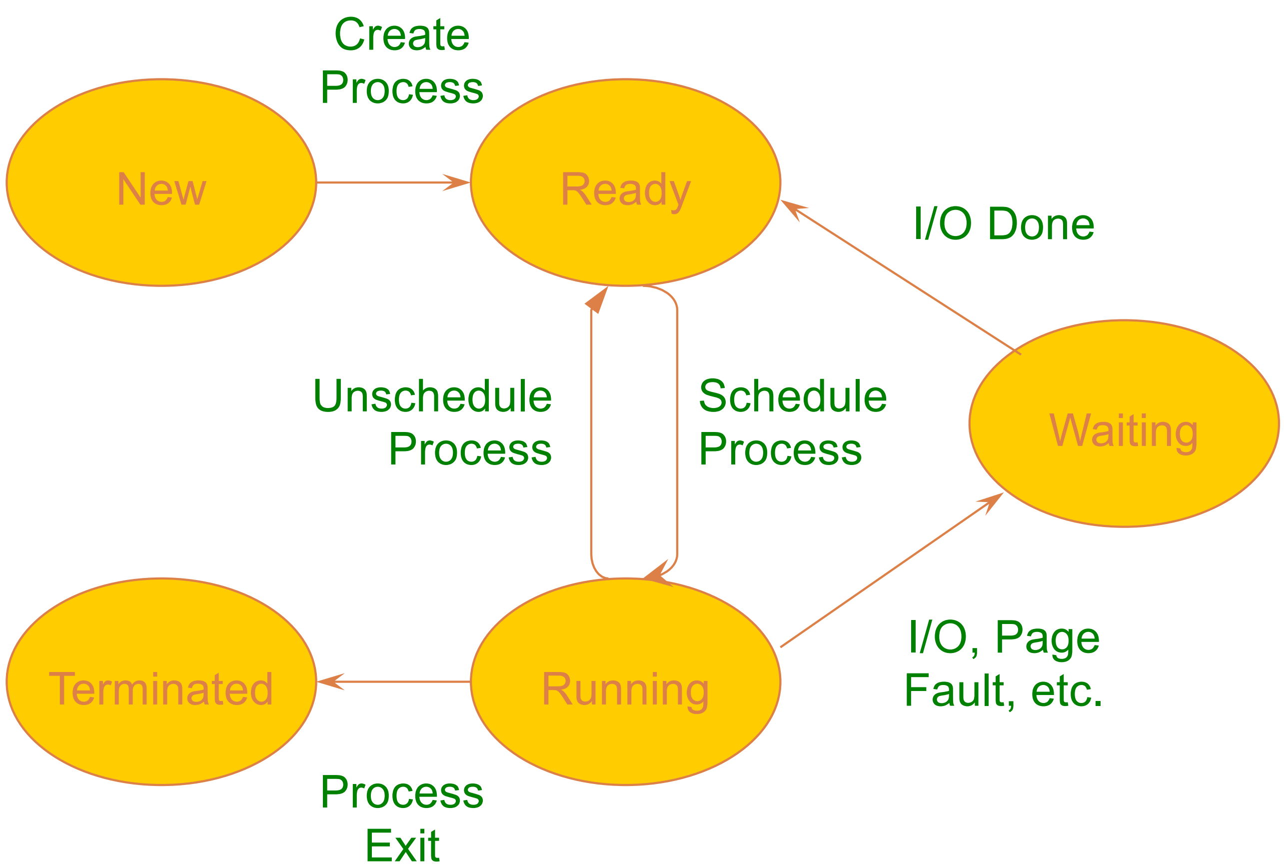
- Which of the following information does the OS need to track for each process?
  A. Stack pointer
  B. Program counter
  C. Process state
  D. Registers
  E. All of the above

Process Execution State

- A process is born, executes for a while, and then dies

- The process execution state that indicates what it is currently doing
  - Running: Executing instructions on the CPU
    - It is the process that has control of the CPU
    - How many processes can be in the running state simultaneously?
  - Ready: Waiting to be assigned to the CPU
    - Ready to execute, but another process is executing on the CPU
  - Waiting: Waiting for an event, e.g., I/O completion
    - It cannot make progress until event is signaled (disk completes)

Execution State Graph



What the OS must track for a process?

- Which of the following information does the OS need to track for each process?
  A. Stack pointer
  B. Program counter
  C. Process state
  D. Registers
  E. All of the above

  - You also need to keep other process information like an unique process id, process states, I/O status, and etc…

PCB Data Structure

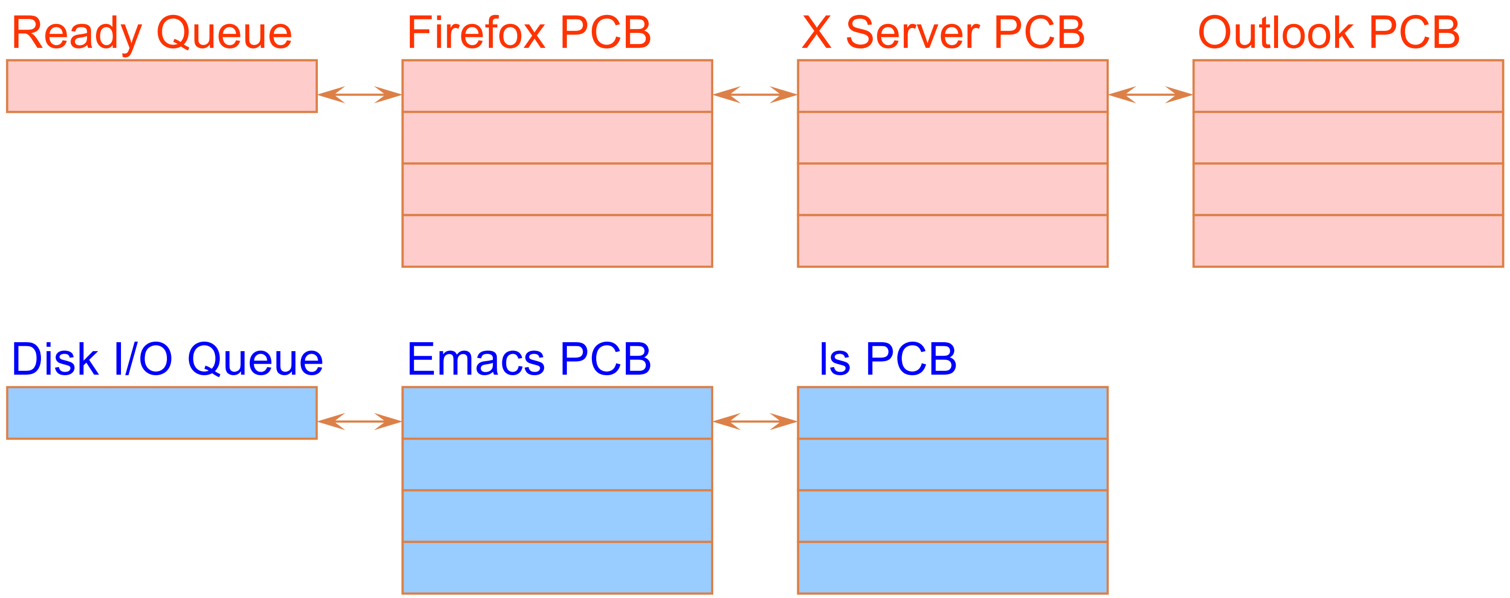
- A Process Control Block (PCB) is where OS keeps all of a process's hardware execution state when the process is not running
  - Process ID (PID)
  - Execution state
  - Hardware state: PC, SP, regs
  - Memory management
  - Scheduling
  - Accounting
  - Pointers for state queues
  - Etc.
- This state is everything that is needed to restore the hardware to the same configuration it was in when the process was switched out of the hardware

Xv6 struct proc


```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Linear address of proc's pgdir
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  volatile int pid;            // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // Switch here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

State Queues



Console Queue

Sleep Queue

.

.

.

There may be many wait queues, one for each type of wait (disk, console, timer, network, etc.)

Conclusions

- Today was a review of CPU virtualization
- The main abstraction is a process
  - Enables the OS to run multiple instances of programs at the same time while sharing the CPU among those processes
- The OS provides a number of concepts to enable seamless execution of multiple processes
  - Traps, Interrupts, Context Switching, etc.
- Overview of threads next time
  - But please read on your own if you need more
  - From the Three Easy Steps textbook

Questions