

Systems I – CS 6013

Computer Architecture and Operating Systems

Lecture 10: Context Switches

MASTER OF SOFTWARE DEVELOPMENT (MSD) PROGRAM

J. DAVISON DE ST. GERMAIN

SPRING 2024

Lecture 10 (Week 4 / Wed.) – Topics

2

- Virtual Memory Primer
 - Memory Pages, Paging, Page Table
- Anatomy of a Syscall
- Context Switching

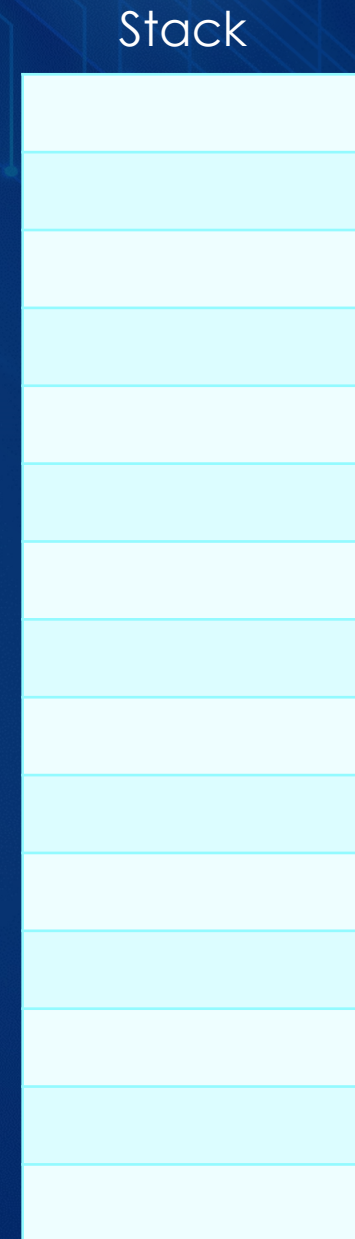
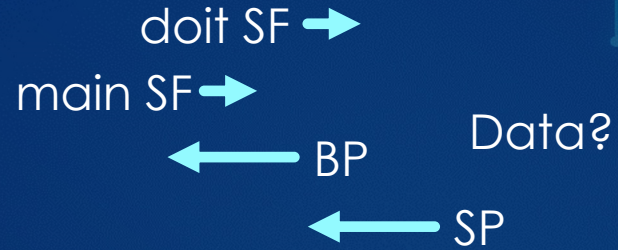
Print Integer Assembly

3

- Assembly*

dividing:

```
cmp rax, 0
je print
xor rdx, rdx      ; Must initialize dx to 0...
mov rcx, 10       ; Divide by 10 (stored in CX)
div rcx           ; AX / CX -> AX, remainder in DX
add rdx, 48       ; digit to char by adding 48 (ASCII '0')
mov [rsp+rbx], rdx ; Save digit/char on stack
inc rbx
jmp dividing
```



- What does stack look like when we are done parsing 1234?

Print Integer Assembly

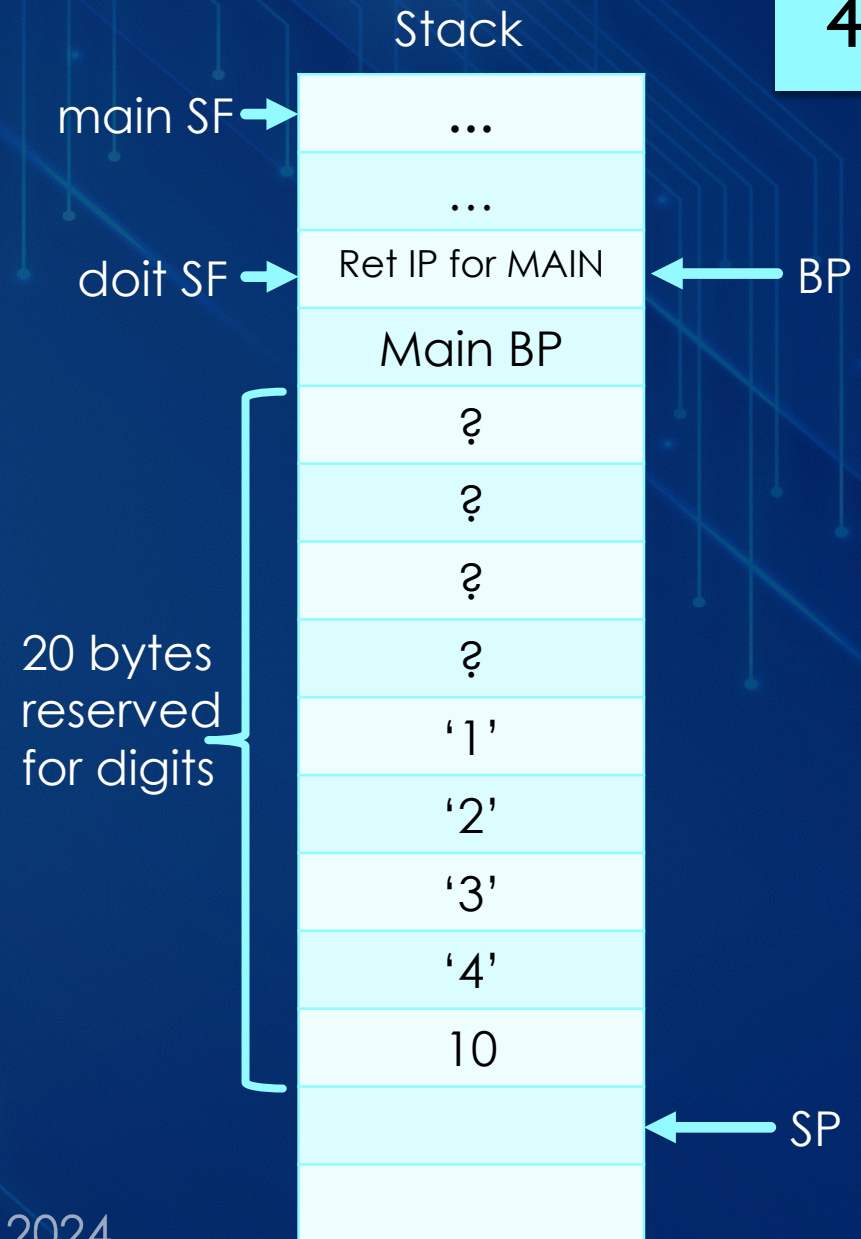
4

- Assembly*

dividing:

```
cmp rax, 0
je print
xor rdx, rdx      ; Must initialize dx to 0...
mov rcx, 10       ; Divide by 10 (stored in CX)
div rcx           ; AX / CX -> AX, remainder in DX
add rdx, 48       ; digit to char by adding 48 (ASCII '0')
mov [rsp+rbx], rdx ; Save digit/char on stack
inc rbx
jmp dividing
```

- *Note: there is a subtle bug in this code... What is it?



Print Integer Assembly

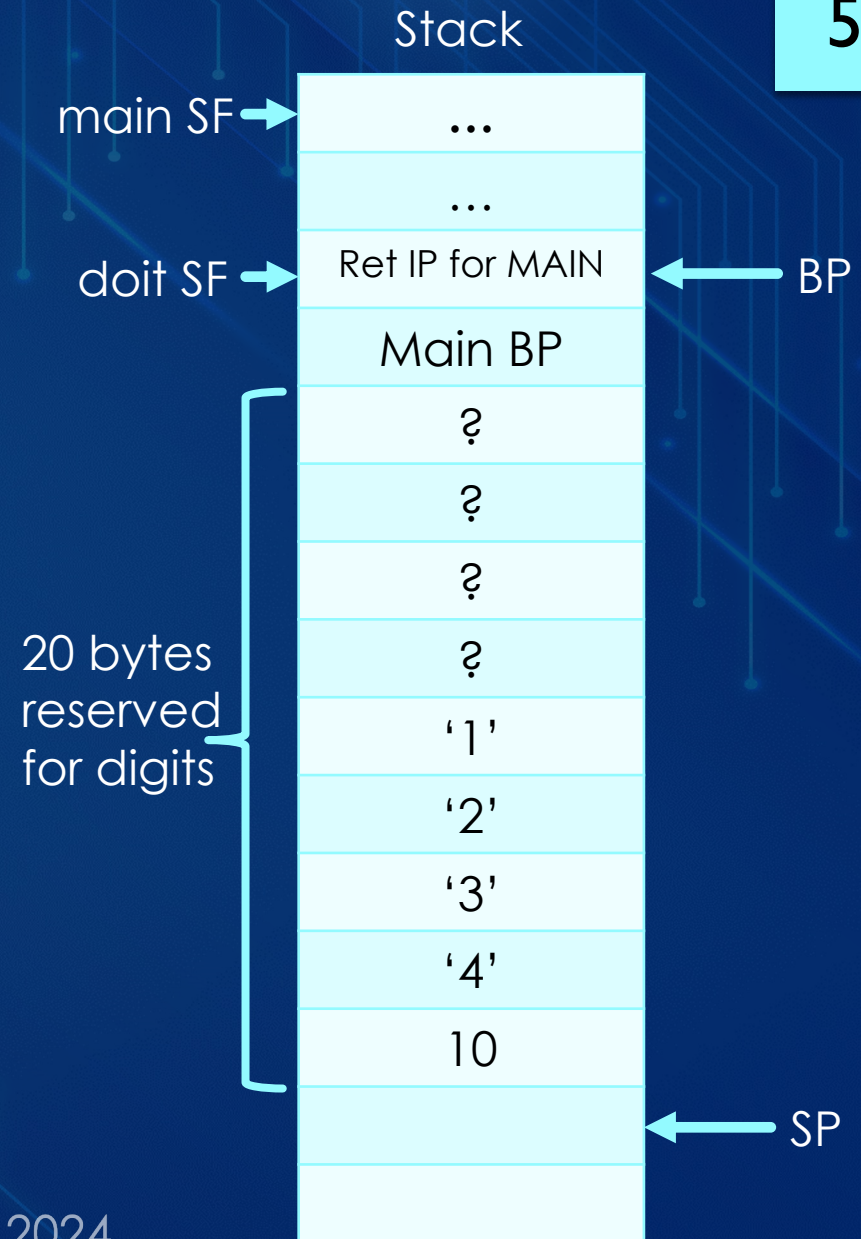
5

- Assembly*

dividing:

```
cmp rax, 0
je print
xor rdx, rdx      ; Must initialize dx to 0...
mov rcx, 10       ; Divide by 10 (stored in CX)
div rcx           ; AX / CX -> AX, remainder in DX
add rdx, 48       ; digit to char by adding 48 (ASCII '0')
mov [rsp+rbx], rdx ; Save digit/char on stack
inc rbx
jmp dividing
```

- Why is this a bug? How to fix the bug?



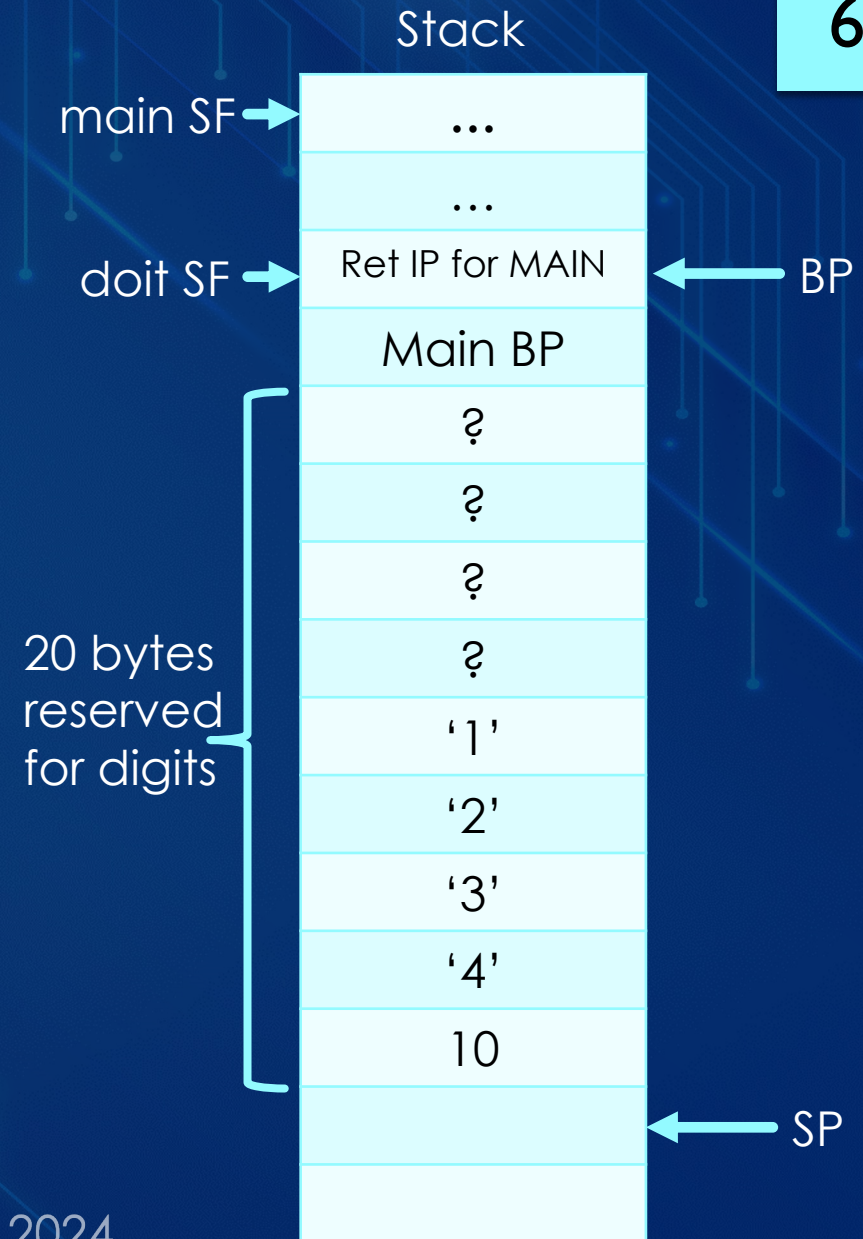
Print Integer Assembly

6

- Why is this a bug? Let's look at this code:

```
mov rdx, 'a'  
mov [rsp+1], rdx ; Save digit/char on stack
```

- What does the stack look like after this line?



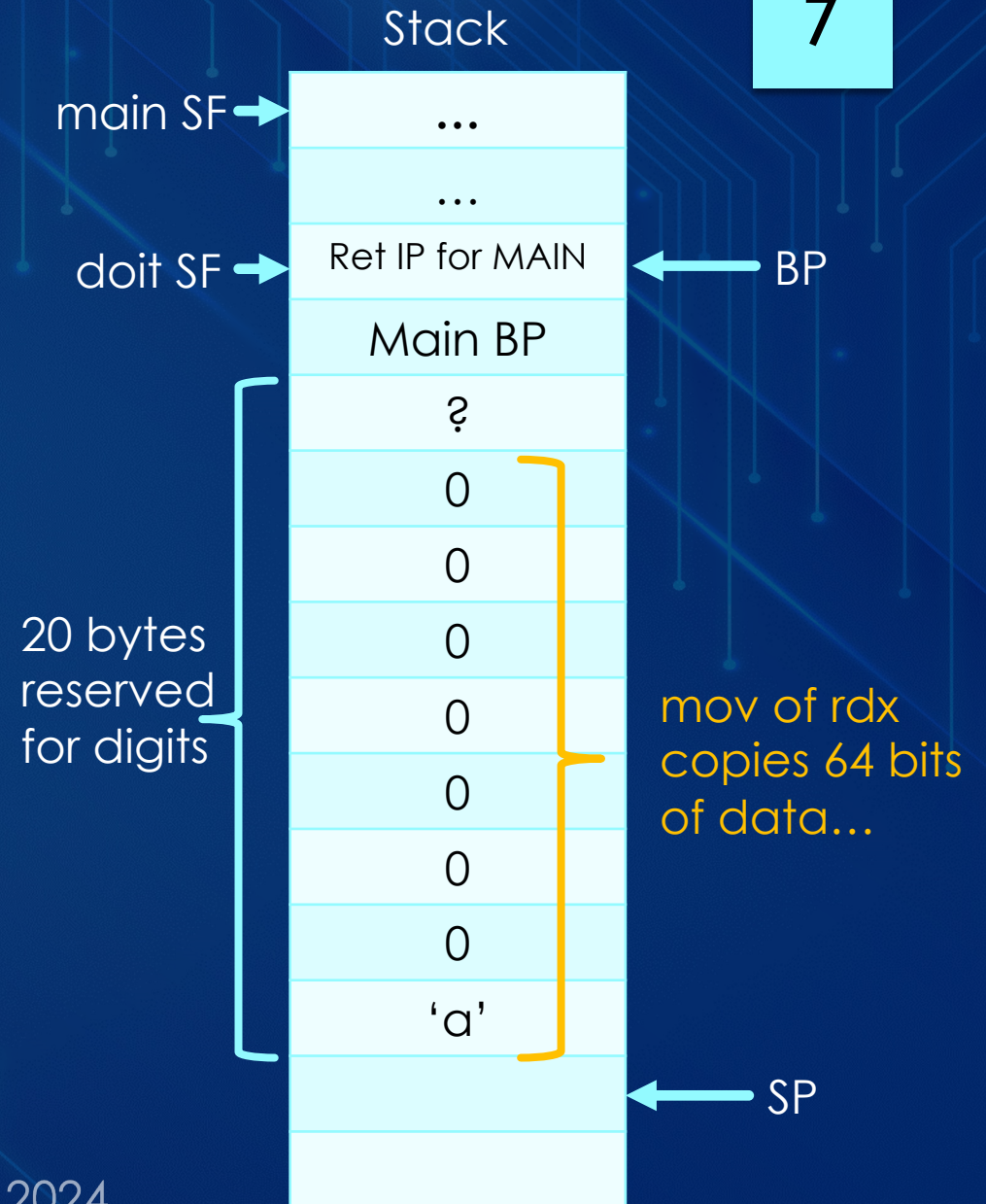
Print Integer Assembly

7

- Why is this a bug? Let's look at this code:

```
mov rdx, 'a'
mov [rsp+1], rdx ; Save digit/char on stack
```

- What does the stack look like after this line?
- How to fix?
 - Move only 8 bits (1 character)
 - `mov [rsp+1], dl`
 - `dl` is the `D` register's last 8 bits.



Miscellaneous

8

- Reminder 1: MSD Student Panel at Noon today.
- Reminder 2: Please reserve Monday after class for labs...
- Week 4 Reading Assignment
 - Actually – vocabulary assignment.
 - Due Mon, Feb 5. Worth 25 Points.
- Assignment 3 Posted
 - Writing your own Unix Shell.
 - Due: Feb 16 – Plan your time accordingly.
 - Please read it over and go ahead and start thinking about the implementation.
 - Note, on Monday (Feb 5) we will have a lab that will address some of the basics of this assignment.
 - We've covered many of the concepts that will be used in this assignment
 - `fork()`, `exec()`, `wait()`

Limited Direct Execution (LDE)

- ▶ Issues

- ▶ Want to run at “full speed” (ie: Direct Execution)
- ▶ Must restrict access to sensitive state
- ▶ Must prevent denial-of-service; must share

- ▶ Solutions

- ▶ Kernel / User Mode
- ▶ Virtual Memory
 - ▶ Process has an array of “fake” addresses.
 - ▶ CPU converts fake addresses to real addresses using:
 - ▶ **MMU** (Memory Management Unit)
- ▶ Traps (syscalls) / Interrupts

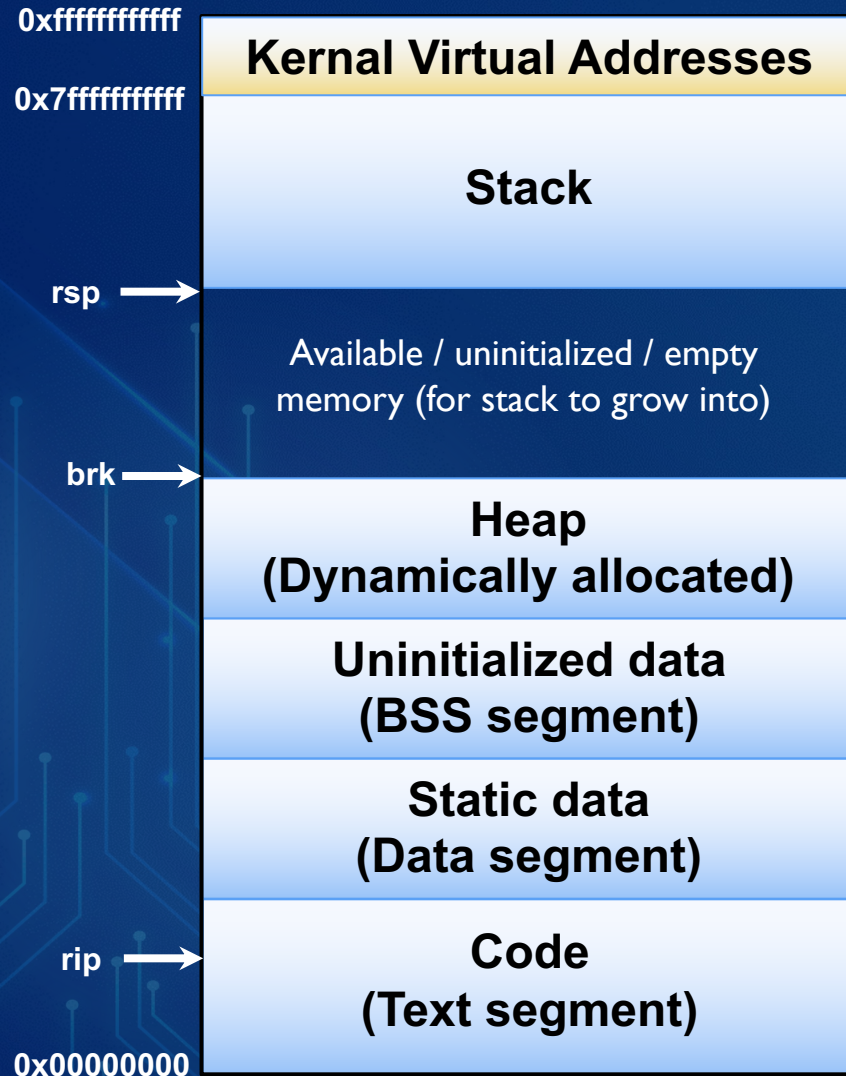
Processor Control Flow

10

- Kinds of CPU control interruptions
 - **Interrupt** - asynchronous, **resume** afterward
 - Network packet arrived, mouse has moved, key pressed, timer fired
 - **Trap** - synchronous, intentional, **resume** afterward
 - **syscall**, breakpoint, overflow
 - **Fault** - synchronous, unintentional, **retry** or **abort**
 - Div/0, illegal instruction, segment not present, general protection, FP-exception
 - **Abort** – unintentional, **abort**
 - Machine check exception, double fault
- **Resume**: restart on instruction after exception
- **Retry**: retry on instruction that cause exception
- **Abort**: terminate

Process Memory Layout

11



- Code / Data / Heap / Stack
- Virtual Addresses
- We are ready for more information about the process memory!
 - Each process also contains virtual addresses for data/code belonging to the kernel (including the KStack (Kernel Stack))
 - Contains the Interrupt Trap Vector and OS Handler Code

Virtual Addressing (VA): Page Table

cr3

12

- What is located at a processes (virtual) address of 0x0?
 - Program's code!
 - Present, not writable.
- What is located at VA 0x1000?
 - (Probably) More code... Not present?
 - Why not? What happens?
 - Not needed yet. Page it in (from disk)
- What is located at VA 0x7fffffffffff?
 - Program's stack memory
- So where, in real memory, is our program actually located?

Virtual Address	Present	Writable	User	Physical Address
0x0	1	0	-	
0x1000	0	0	1	
0x2000	1	1	1	
0x40000	0	-	-	
...	
0x7fffffffffff	1	1	1	
0x8008000	1	0	0	

Page Table

Virtual Addressing (VA): Isolating Memory

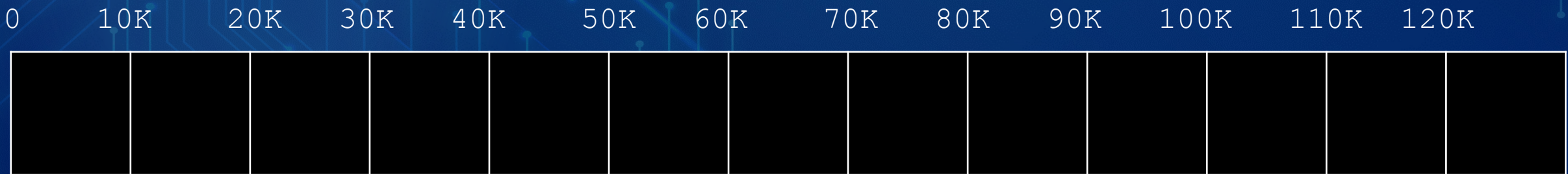
cr3

I3

- Memory is divided into **pages**
 - A fixed size of memory
 - Note, page “sizes” here are just for illustration.
 - The memory for this process is divided up into (many) pages
 - The page table maps the *virtual address* (the way a process looks at its memory), into the *physical address*.
- So our real memory looks like:


Virtual Address	Present	Writable	User	Physical Address
0x0	1	0	-	0x090000
0x1000	0	0	1	-
0x2000	1	1	1	0x120000
0x40000	0	-	-	-
...
0x7fffffffffff	1	1	1	0x20000
0x8008000	1	0	0	0x40000

Page Table



Virtual Addressing: Isolating Memory

- Note, process thinks its address space is contiguous – starts at 0, goes to 0x7fffffffffffffff.
- But its memory is really found anywhere there was room in physical memory for it.



cr3

Virtual Address	Present	Writable	User	Physical Address
0x0	1	0	-	0x090000
0x1000	0	0	1	-
0x2000	1	1	1	0x070000
0x40000	0	-	-	-
...
0x7fffffffffff	1	1	1	0x20000
0x8008000	1	0	0	0x40000

Page Table



Virtual Addressing: Isolating Memory

cr3

15

- What is (might be) at VA 0x40000?
 - Heap Data
- Why is it not *Present*?
 - Paged out – not being used by the program currently.
- What will cause it to be mapped?
 - `x = new int(3)` // When that
 - `*x = 99` // memory is accessed

Virtual Address	Present	Writable	User	Physical Address
0x0	1	0	-	0x090000
0x1000	0	0	1	-
0x2000	1	1	1	0x070000
0x40000	0	-	-	-
...
0x7fffffffffff	1	1	1	0x20000
0x8008000	1	0	0	0x40000

Page Table

0 10K 20K 30K 40K 50K 60K 70K 80K 90K 100K 110K 120K

		P1 Stack Mem		P1 Kernel Mem			P1 Static Vars		P1 Code			
--	--	--------------------	--	---------------------	--	--	----------------------	--	------------	--	--	--

Virtual Addressing: Isolating Memory

- Why was it placed in physical memory at 0x10000?
 - There was room for it there.

cr3

16

Virtual Address	Present	Writable	User	Physical Address
0x0	1	0	-	0x090000
0x1000	0	0	1	-
0x2000	1	1	1	0x070000
0x40000	1	1	1	0x10000
...
0x7fffffffffff	1	1	1	0x20000
0x8008000	1	0	0	0x40000

Page Table

0 10K 20K 30K 40K 50K 60K 70K 80K 90K 100K 110K 120K

	P1 Heap Mem	P1 Stack Mem		P1 Kernel Mem			P1 Static Vars		P1 Code			
--	-------------------	--------------------	--	---------------------	--	--	----------------------	--	------------	--	--	--

Physical (real) memory (shown in ~PAGE~ sizes)

CS 6013 – Spring 2024

Virtual Addressing: Process 2

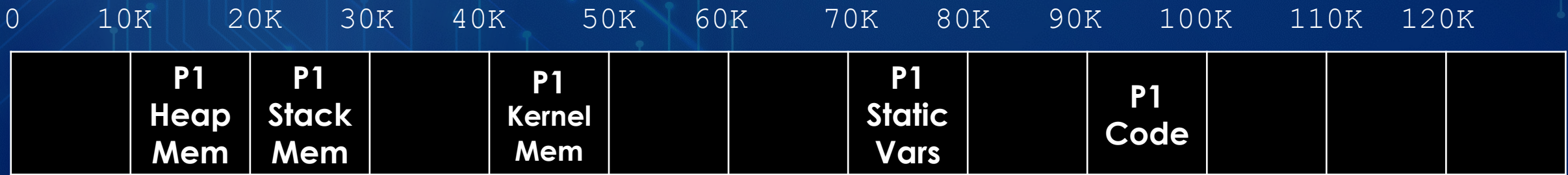
- Now let's look at Process 2's memory.
 - Notice its Virtual addresses are the same as Process 1.

cr3

17

Virtual Address	Present	Writable	User	Physical Address
0x0	1	0	-	0x030000
0x1000	0	-	-	-
0x2000	0	-	-	-
0x40000	1	1	1	0x080000
...
0x7fffffffffff	0	-	-	-
0x8008000	0	-	-	-

Page Table



Virtual Addressing: Process 2

cr3

18

- What if P2 could grow its heap (without the OS monitoring it)?
 - Say it grows from 10K bytes, to 20K bytes?
 - What could it do?
 - Place any (object) code it wants into P1's Code segment...
 - Then what happens when P1 begins to run again? Especially if P1 was running as the *root* user...

Virtual Address	Present	Writable	User	Physical Address
0x0	1	0	-	0x30000
0x1000	0	-	-	-
0x2000	0	-	-	-
0x40000	1	1	1	0x80000
...
0x7fffffffffff	0	-	-	-
0x8008000	0	-	-	-

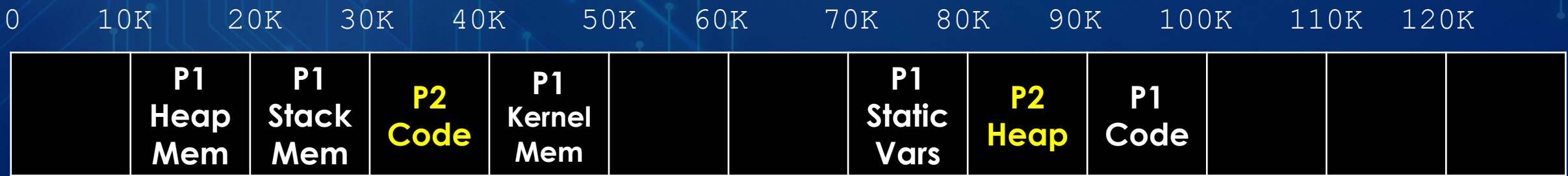
Page Table

0 10K 20K 30K 40K 50K 60K 70K 80K 90K 100K 110K 120K

	P1 Heap Mem	P1 Stack Mem	P2 Code	P1 Kernel Mem			P1 Static Vars	P2 Heap	P1 Code			
--	-------------------	--------------------	------------	---------------------	--	--	----------------------	------------	------------	--	--	--

Physical Memory

- What else is stored in physical memory?
 - Other processes.
 - System (OS) information
 - Network Device Data
 - Video Data
 - Interrupt Descriptor Table



Physical Memory

- What else is stored in physical memory?
 - Other processes.
 - System (OS) information
 - Network Device Data
 - Video Data
 - Interrupt Descriptor Table



Interrupt Descriptor Table (IDT)

21

- IDT tells CPU what to do...
 - On each type of trap
 - On each type of interrupt
- IDTR points to it
 - Protected `lidt` instruction loads it
- `cli/sti` instructions enable/disable interrupts
 - Avoids interrupts during interrupt handling
- Note, only one entry for `syscall`.
 - This is because we load registers (eg: `ax`) with the info for the `syscall`, and the `syscall` code will then determine what “real” code to execute based on that.
 - As an aside, note that `Int 0x80` is the `syscall` trap identifier, hence using assembly:
 - `int 0x80 ; to make a syscall.`
 - Because only one point of entry, the OS can more easily make sure it is secure.

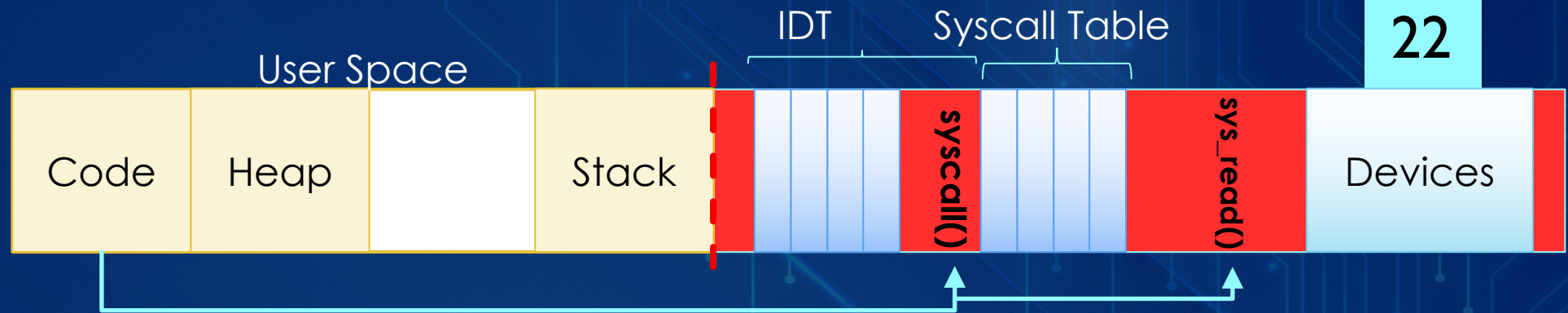
idtr →

Int	Handler Addr	Purpose
0x00	0x12001010	Division by zero
0x01	0x12006028	Single-step (trap flag)
...		
0x06	0x12008080	Invalid Opcode
...		
0x0D	0x12001080	General protection fault
0x0E	0x12011020	Page fault
...		
0x20	0x12004088	Timer
0x21	0x12012020	Keyboard
...		
0x2A	0x12004440	Network card
0x2B	0x12004448	USB, sound card
0x2C	0x120121A8	PS/2 mouse
...		
0x80	0x12001100	Linux Syscall

Note: IDT and Trap Vector Table are for our purposes just different names for the same thing.

Syscalls

22

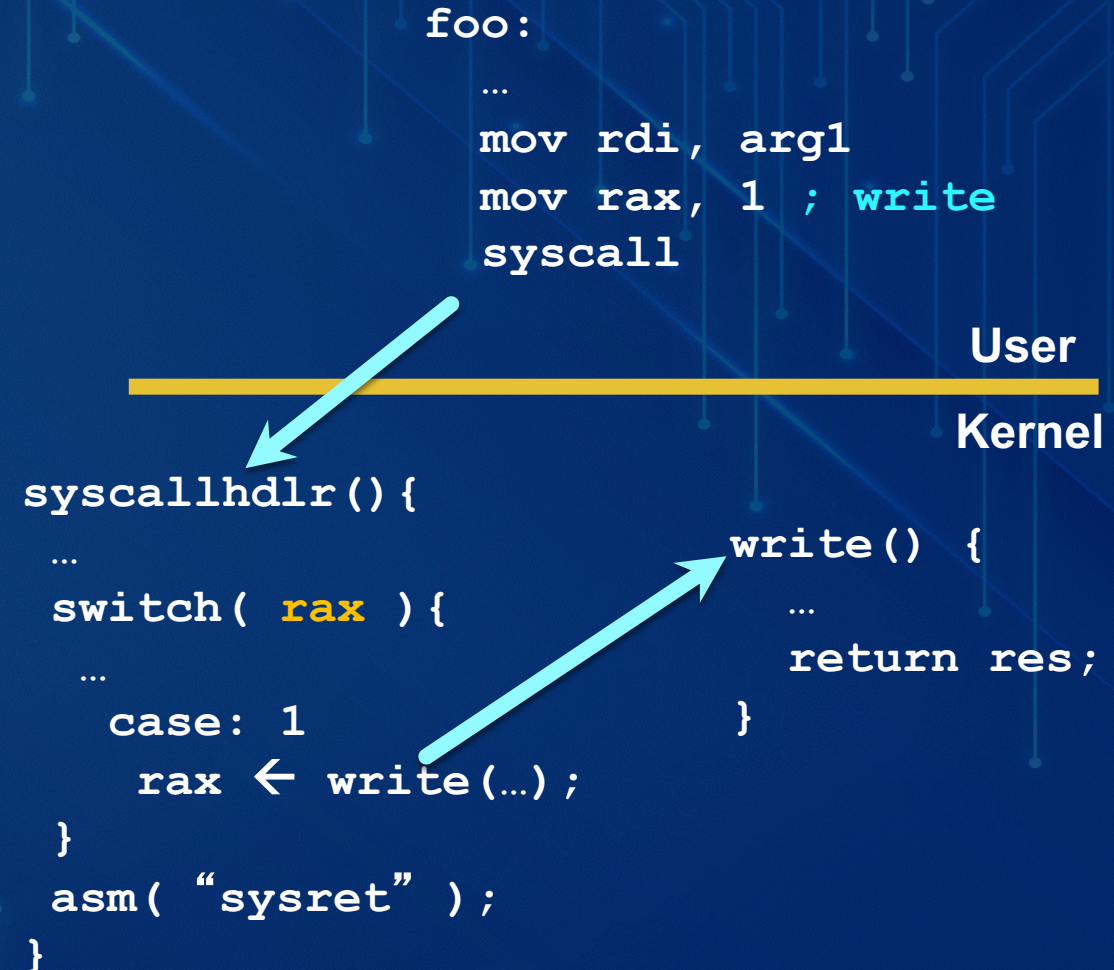


- Processes induce syscalls with a **trap**
- Multiple ways on x86 Linux
 - `int 0x80` – ie, entry 128 in the interrupt descriptor table (why location 128?)
 - Newer instructions (especially 64-bit systems): `syscall`, `sysenter`
- CPU consults its interrupt descriptor table (IDT)
 - IDT tells CPU what code to run on **trap**, **interrupt**, **fault**
 - CPU raises privilege level to ring 0 on trap
 - (Note, Kernel sets up IDT at boot time), process cannot change it
- Opcode passed on stack or in register tells kernel what syscall the process wants to perform

Anatomy of a System Call

23

- Let's review everything that happens in making a system call.
- 1st?
 - Program puts syscall params in registers
- 2nd?
 - Call syscall... what does this actually do?
 - executes a trap on the CPU.
 - This does what?
 - » Switches mode to ring 0
 - » Switches to process' kstack
 - » Pushes process rip (+1) to kstack (Why?)
 - » Vectors (moves) to trap handler from IDT
 - » OS syscall dispatch code gains control
 - Trap handler uses param to jump to desired handler (e.g., fork, exec, "foo"...)
 - When complete, reverse operation
 - » Place return code in register (Which one?)
 - » Return from trap (Notice "sysret"... What does this do?)



LDE Problem #2

- Preventing denial of service / allowing hardware sharing
- OS requirements for **multiprogramming** (**multitasking**) – two considerations:
 1. Policy: Choose what to run next (in a consistent way)
 - Need a process **Scheduler**
 2. Mechanism: Low-level code that implements the decision
 - **Dispatcher** and **Context Switch**: How?
- Example of separation of policy and mechanism

Dispatch Mechanism

- OS runs dispatch loop

```
while( true ) {  
    run process A for some time-slice  
    stop process A and save its context  
    load context of another process B (which starts B running)  
}
```

} Context-switch

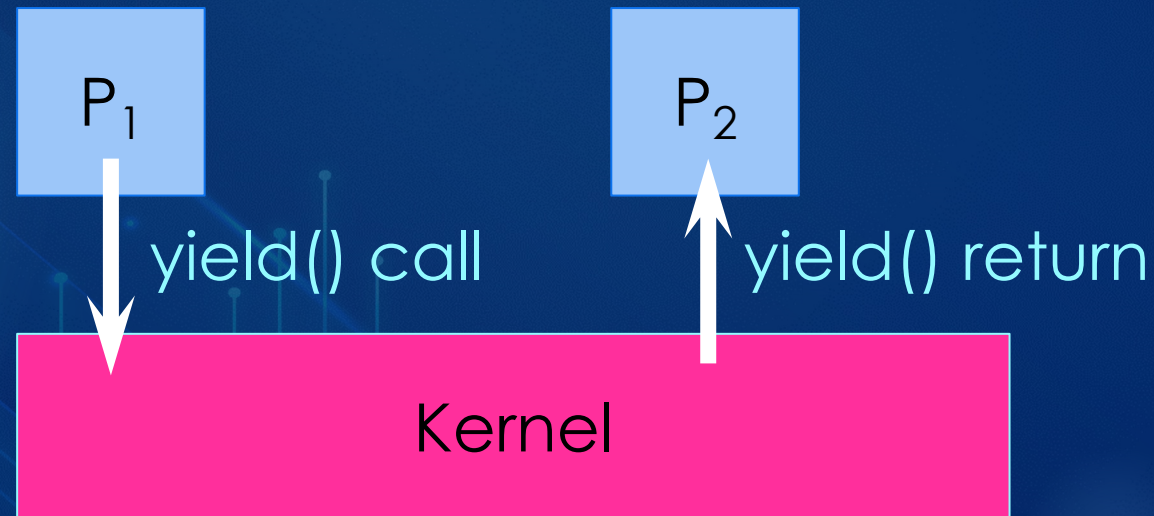
- Question 1: What execution *context* (information) must be saved (for a process) – and later restored?
 - Registers! Memory? (More on this later)
 - Not Memory! It's already “saved”
- Question 2: How does OS (scheduler) regain control? (See next slide)

So, how does OS get (regain) control?

26

Option I: **Cooperative Multitasking**

- **Trust process** to relinquish CPU to OS through traps
 - Examples: System call, page fault (access page not in main memory), or error (illegal instruction or divide by zero)
 - Provide special `yield()` system call



So, how does OS get (regain) control?

27

- Problem with cooperative approach?
- Disadvantages: Processes can misbehave
 - By avoiding all traps and performing no I/O, can take over entire machine
 - Accidentally go into an infinite loop.
 - Only solution: Reboot!
- Not performed in modern operating systems

So, how does OS get (regain) control?

28

Option 2: **Preemptive Multitasking**

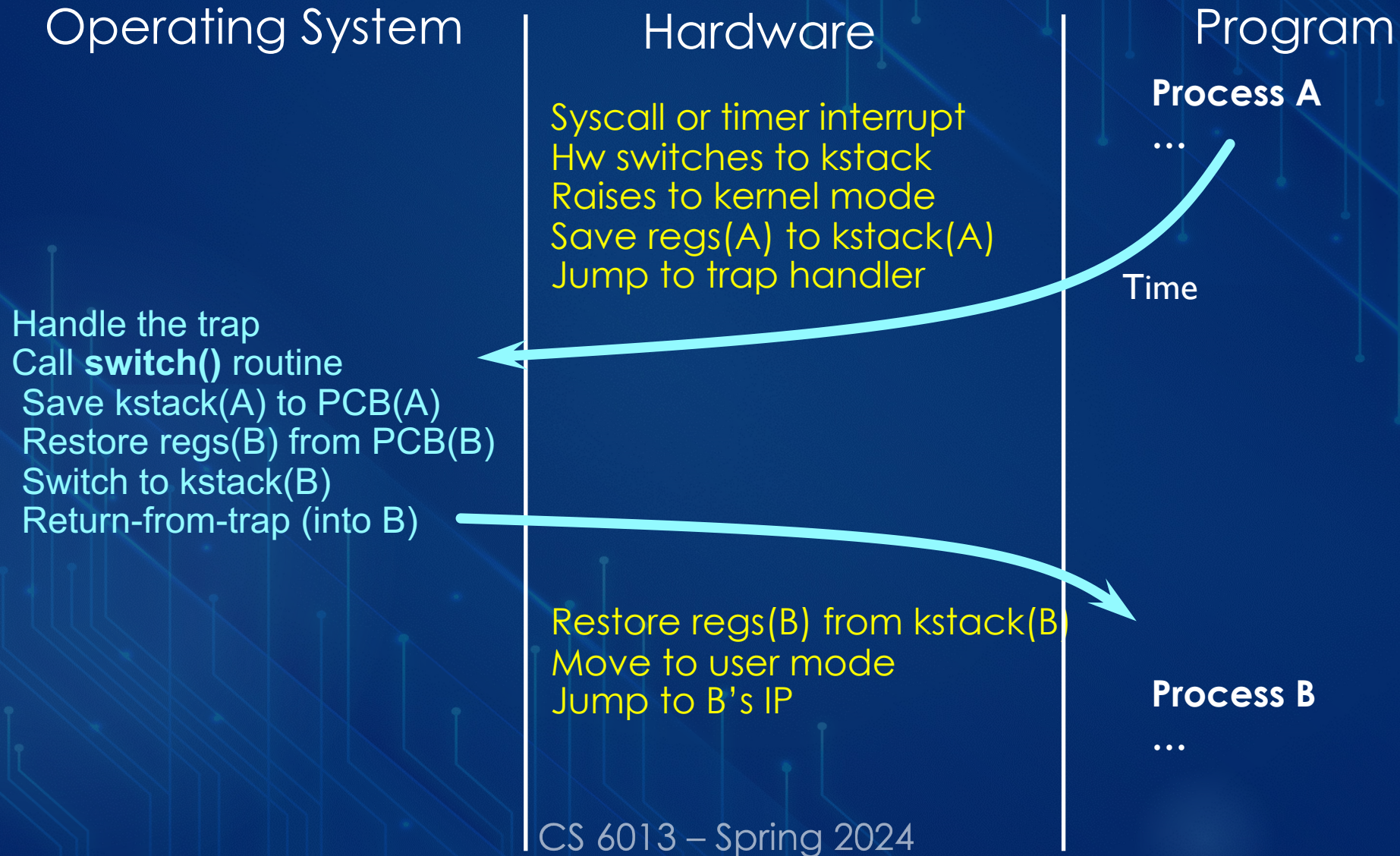
- Guarantee OS can obtain control periodically
- Enter OS by enabling periodic alarm clock
 - Hardware generates timer interrupt (CPU or separate chip)
 - Example: Every 10ms
- User must not be able to mask timer interrupt
- Dispatcher counts interrupts between context switches
 - Example: Waiting 20 timer ticks gives 200 ms time slice
 - Common time slices range from 10 ms to 200 ms
- How long is 200 ms?
 - .2 seconds

Q1: What context to save?

- On interrupt (sometimes called Interrupt Request aka IRQ)
 - HW switches to process's kstack
 - Records user process's IP and SP and registers (**trapframe**)
 - Then HW enters the OS dispatcher
- Dispatcher must track context of process when not running
 - Save context in PCB, restore context from another PCB
 - Called **context switch**

Context Switch Diagram

30



~ Fin ~