

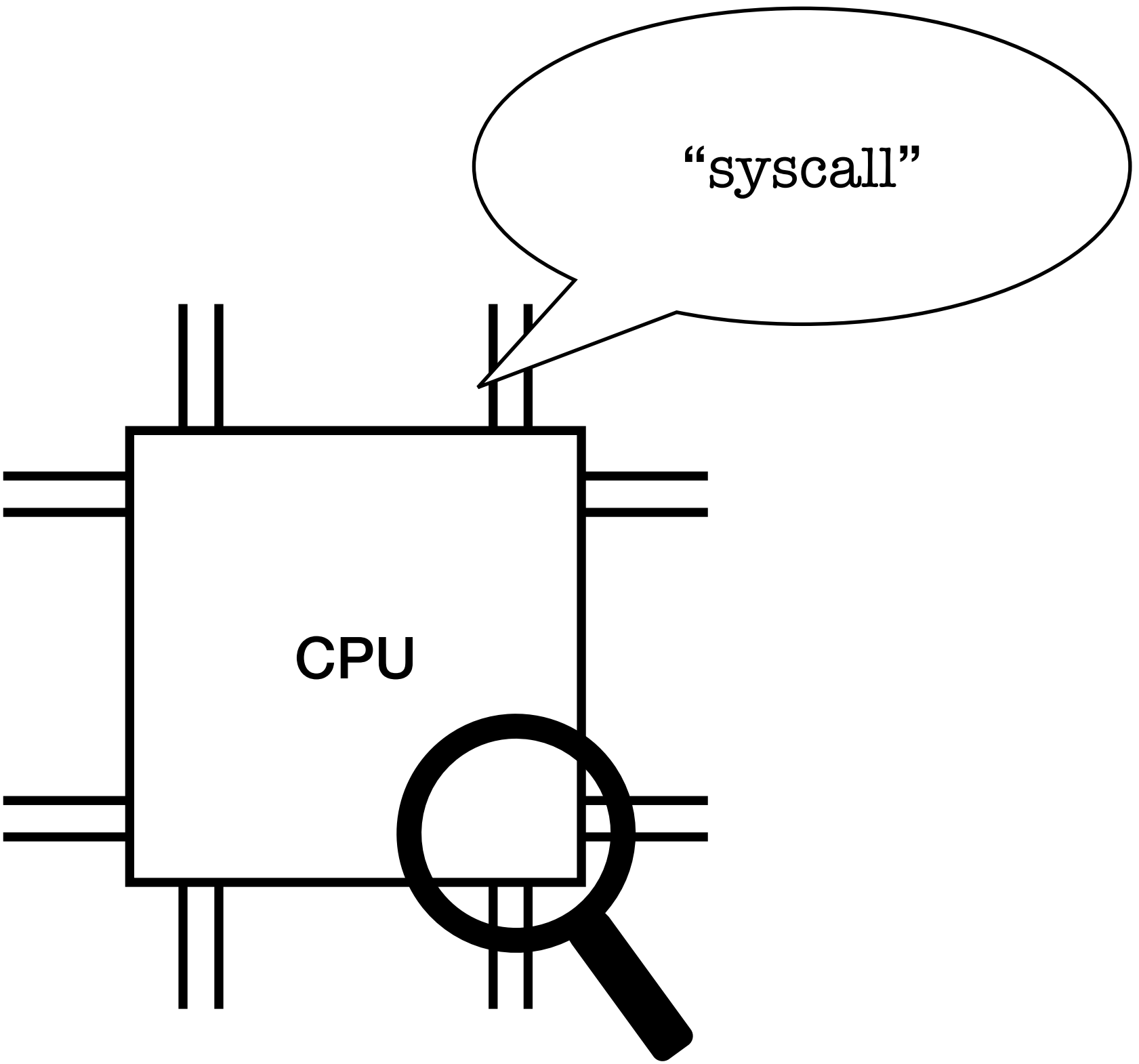
Interrupts and Scheduling

Code reviews to become one week long exercises;
Deadlines updated* (to be posted on the website after class);
Drivers assignment to be dropped

Outline

- Revisiting interrupt handling
- Implementing interrupt handlers
- Applying interrupts to scheduling
- Code that should not be interrupted

Review: “syscall” instruction in x86



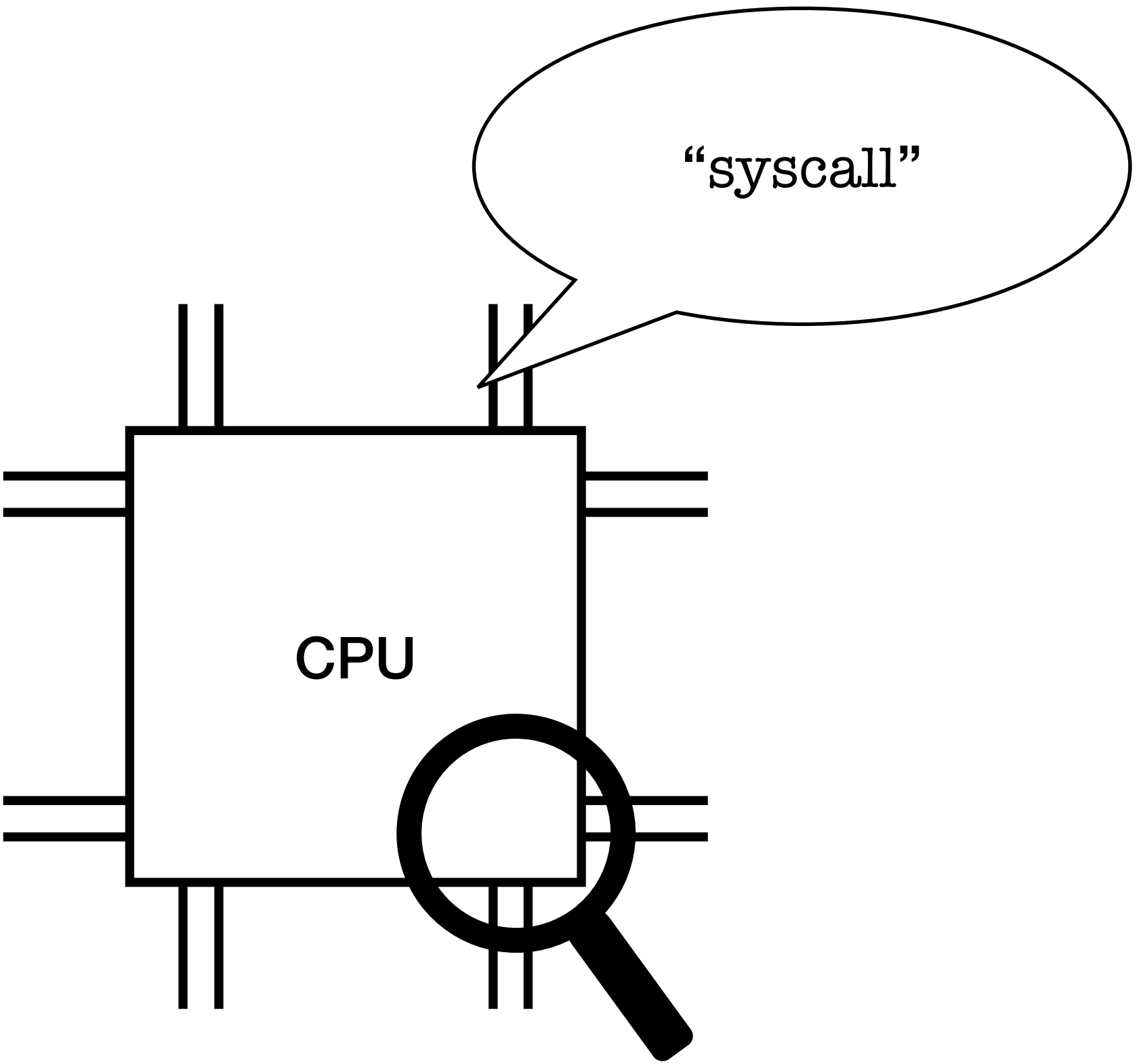
Interrupt	Handler
0	System traps and exceptions
1	...
...	...
32-127	Device interrupt handlers
128	syscall interface
...	...
255	...

1. Save the information needed to return from the interrupt onto the stack
2. Execute kernel interrupt handler (in this case, the syscall interface)
 - In practice, this will be a table of functions to potentially execute depending on the value of %rax
3. Return from the interrupt handler

Review: Interrupt Handling in x86

- The `syscall` instruction is the primary mechanism to transition from user-mode to kernel-mode using the Linux x64 ABI
 - To do so, this instruction performs a “trap”  a mechanism that elicits an operating system response
 - The hardware sets the instruction pointer to a location defined by the *Interrupt Descriptor Table*, which has entries for all types of interrupts
 - The interrupt handler for the system call looks at the `%rax` register for which routine to perform (e.g., `do_open`)

Review: “syscall” instruction in x86



Interrupt	Handler
0	System traps and exceptions
1	...
...	...
32-127	Device interrupt handlers
128	syscall interface
...	...
255	...

User program explicitly invokes the operating system for the syscall interface handler

Other ways to invoke the handlers: various illegal executions, breakpoints, exceptions, etc..

Hardware also invokes the interrupt handler table!

Review: Implementing “context_switch”

```
thread_t *curthr; // a pointer to the current running thread
void saveContext(context_t *ctx); // saves the provided context to its TCB
void setContext(context_t *ctx); // set the provided context as active

void switchThread(thread_t *next_thread) {
    volatile int first = 1;
    saveContext(&curthr->ctx);
    if (first) {
        first = 0;
        curthr = next_thread;
        setContext(&curthr->ctx);
    }
}
```



Save the context from the currently executing thread to switch to the interrupt handling context

What is the “interrupt handling context”? Which registers/stack is used, etc...?

Resume the original context

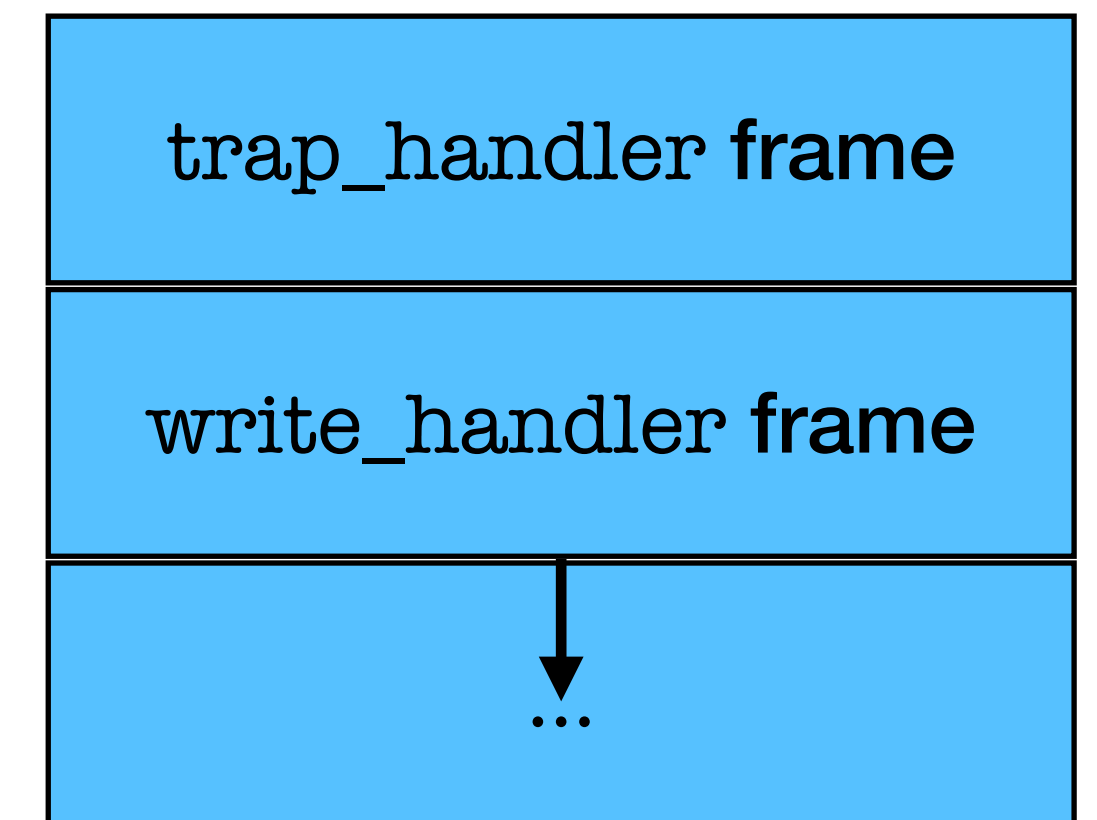
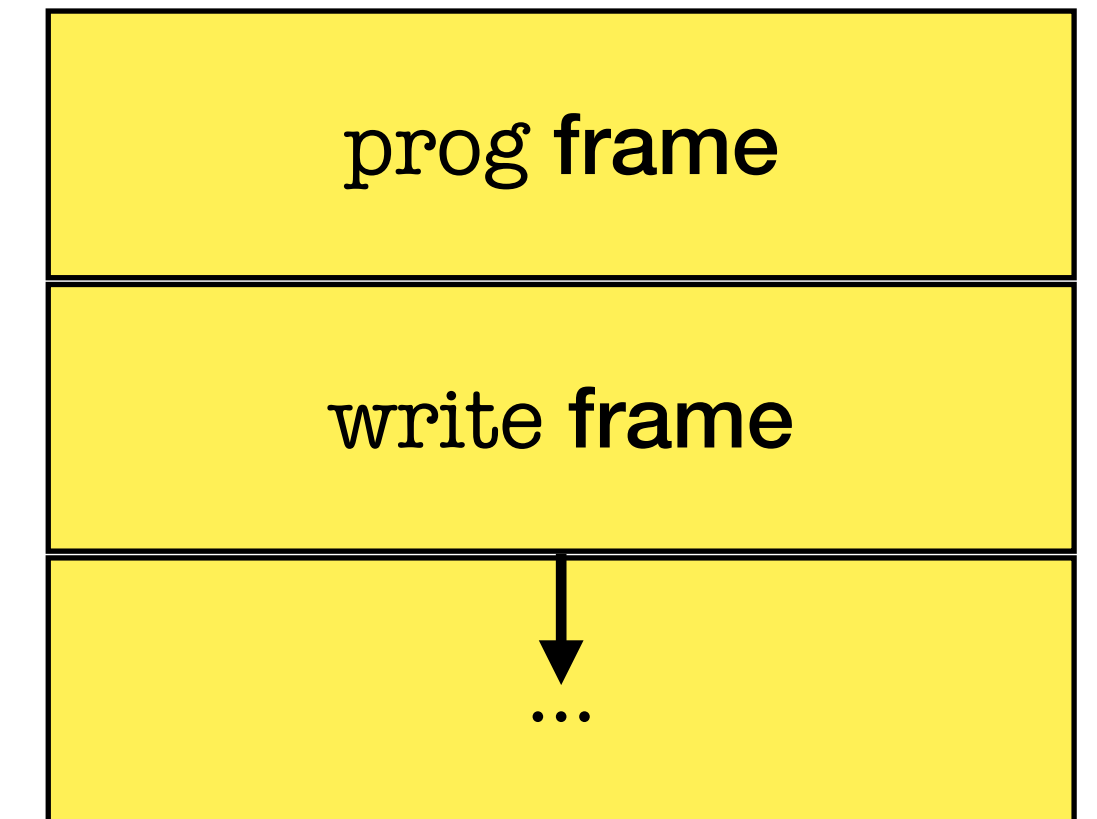
User Stacks and Kernel Stacks

```
prog() {  
    // ...  
    write(fd, buffer, size);  
    // ...  
}
```

```
write() {  
    // ...  
    trap(write_code);  
    // ...  
}
```

```
trap_handler(code) {  
    // ...  
    else if (code == write_code) {  
        write_handler();  
    }  
    // ...  
}
```

User Stack



Interrupt Stack

User Stacks and Kernel Stacks

- The minimum functional context switching sequence must provide:
 - (1) Limited entry into the kernel: user programs cannot have the ability to jump to arbitrary locations within the kernel, otherwise isolation between processes is broken!
 - (2) Atomic changes to processor state: when context switching into the kernel (or vice versa), the user process' program counter, stack, and registers must remain consistent before the transfer is made
 - (3) Transparent, restartable execution: interrupts may occur between any two instructions, so the operating system needs to be able to restore this state exactly as it was before the transfer mode occurs

Chat with your neighbor(s)!

In the CS134 assignment, contexts maintain a pointer to a stack. Which stack does this refer to? What is a limitation of this design?

It exclusively implements the kernel stack... the user “code” executes in the kernel space!

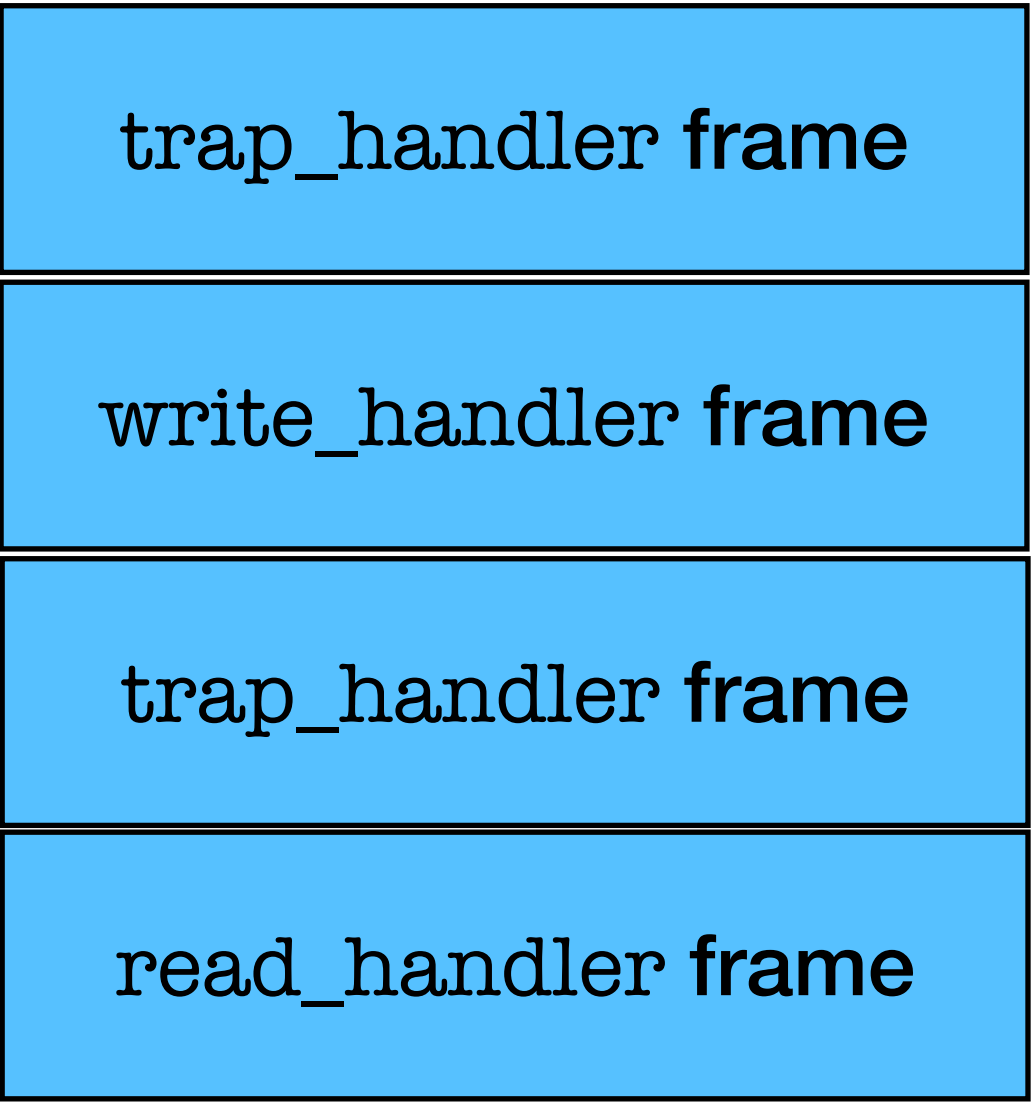
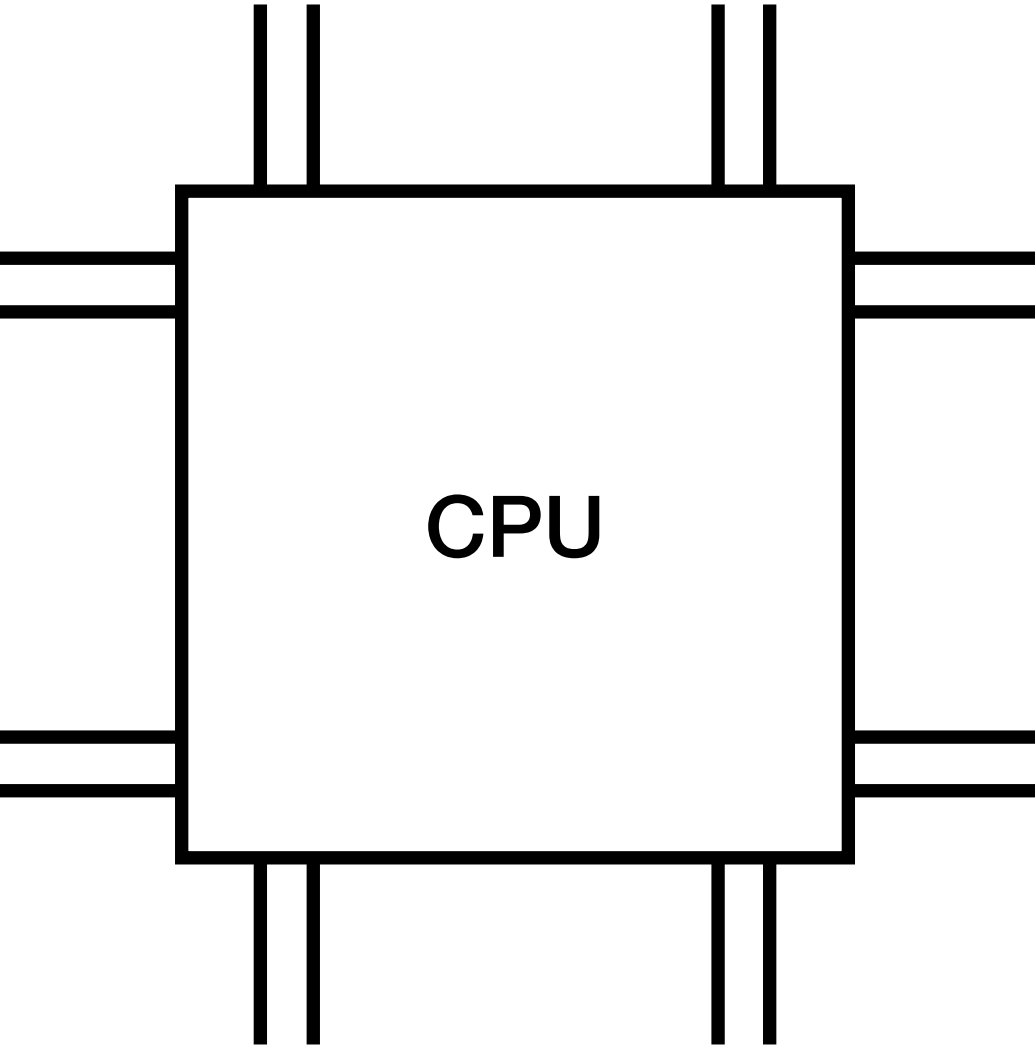
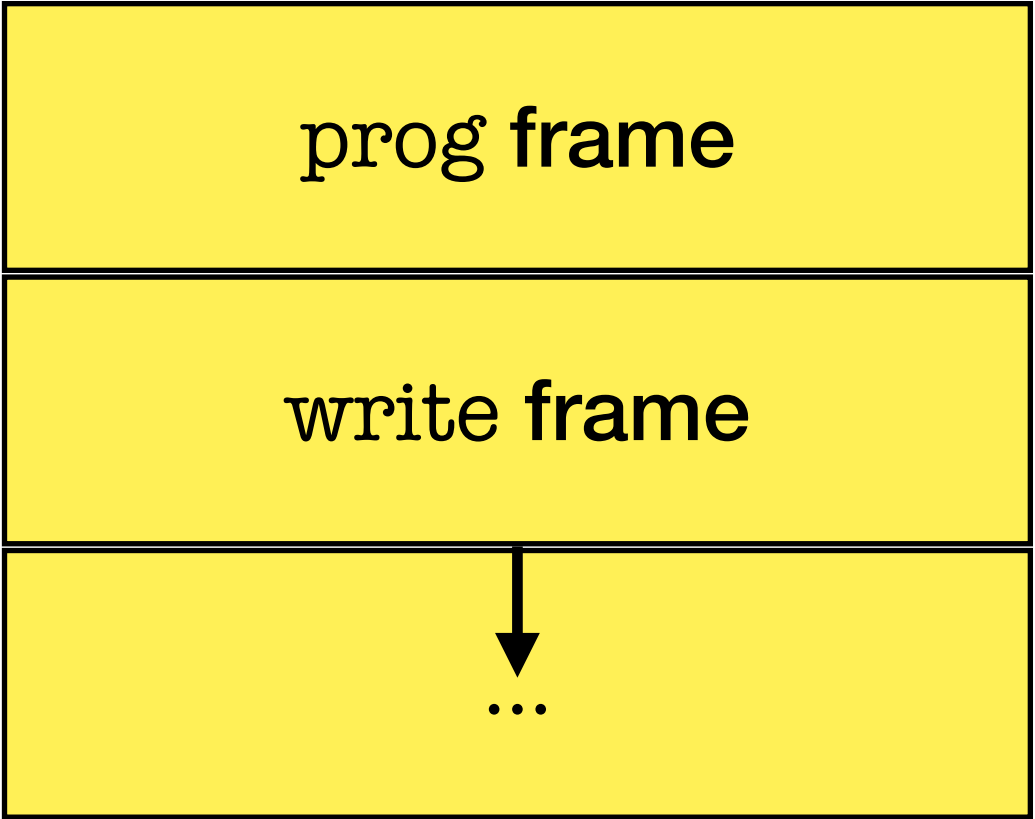
We can work around this by maintaining two different stack pointers — one for the user stack and one for the kernel/interrupt stack!

Implementing Interrupt Handlers

- Whenever we are executing some code, we need to execute a code associated with a particular context (referred to as the “interrupt handler context”)
- There are two ways in which the interrupt handler stack may be implemented
 - (1) The interrupt handler stack could be implemented as part of the OS kernel with one shared stack for all interrupts to be caught and handled on
 - (2) The interrupt handler stack could be implemented as part of the interrupted thread’s context

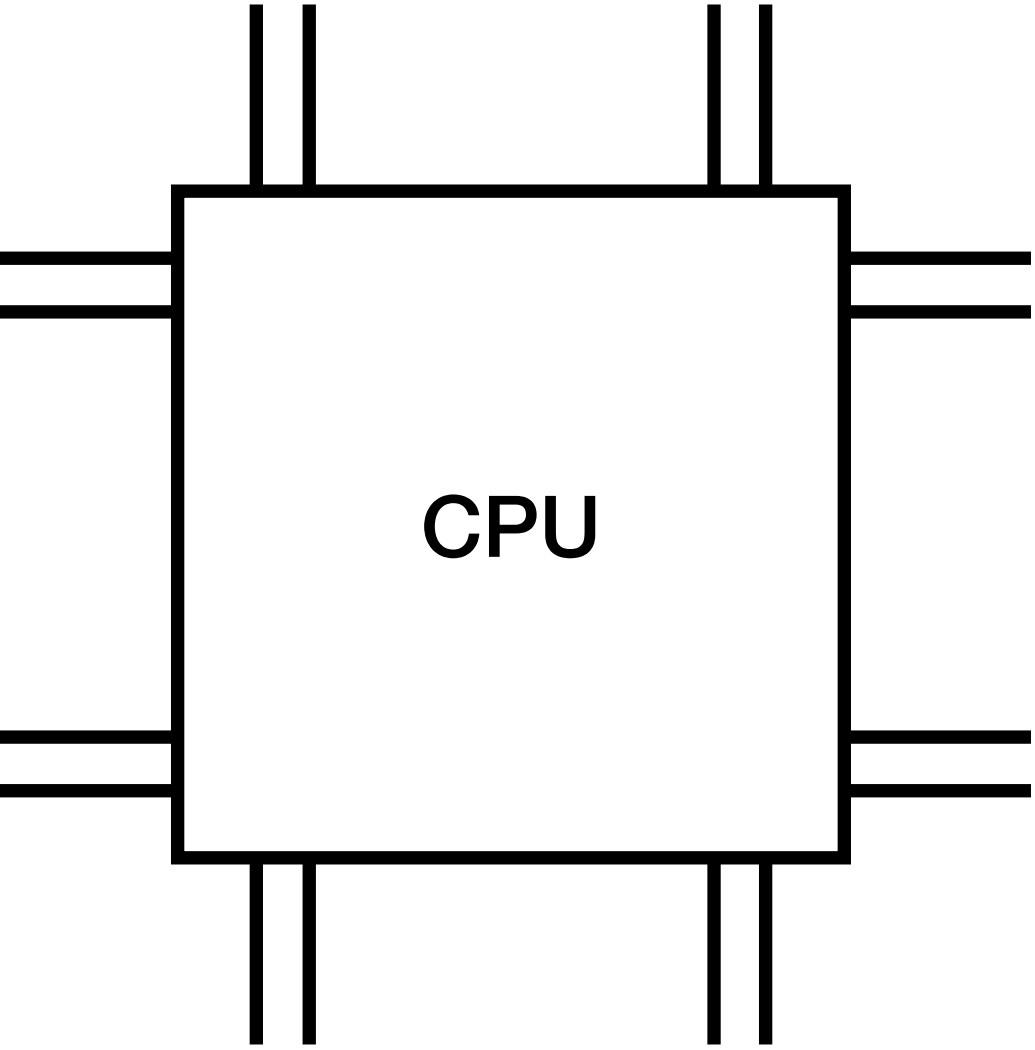
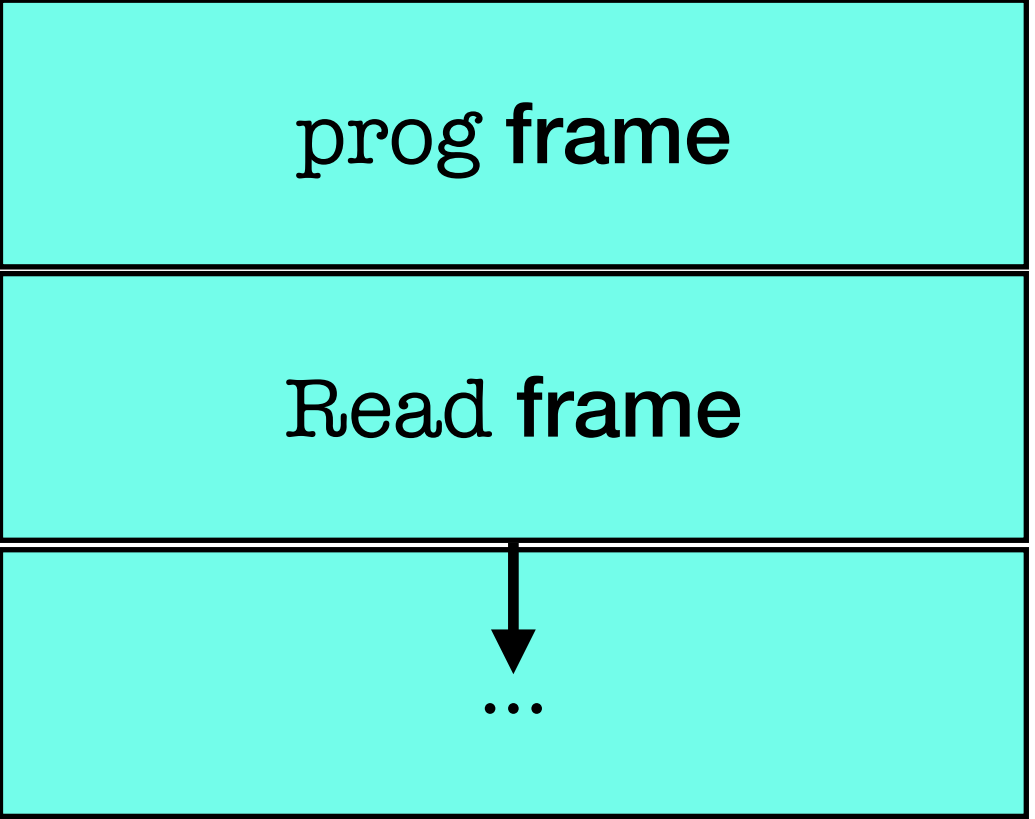
Implementing a Central Interrupt Handler

User Stack



Interrupt Stack

User Stack



Chat with your neighbor(s)!

Insight: our interrupts should be given some notion of priority... we'll come back to this!

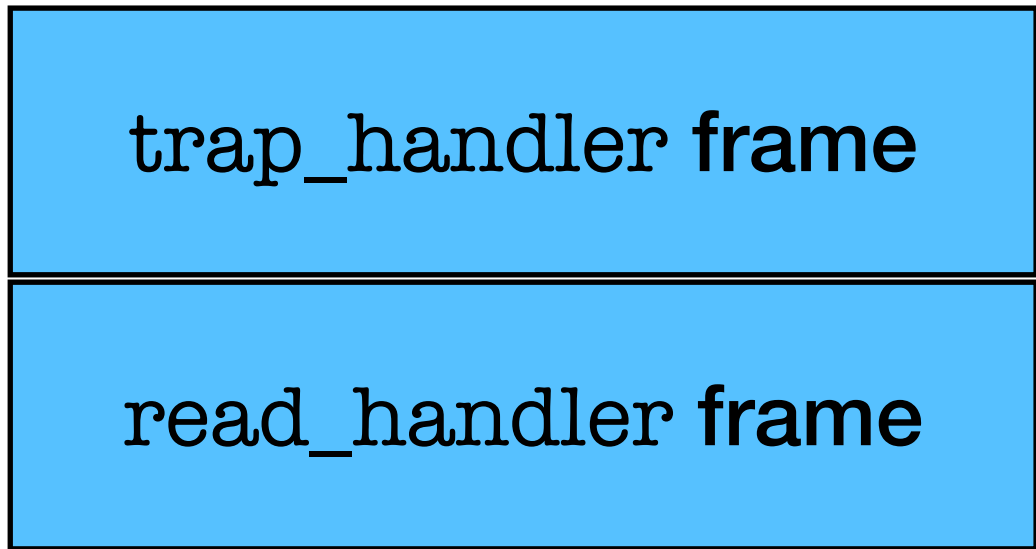
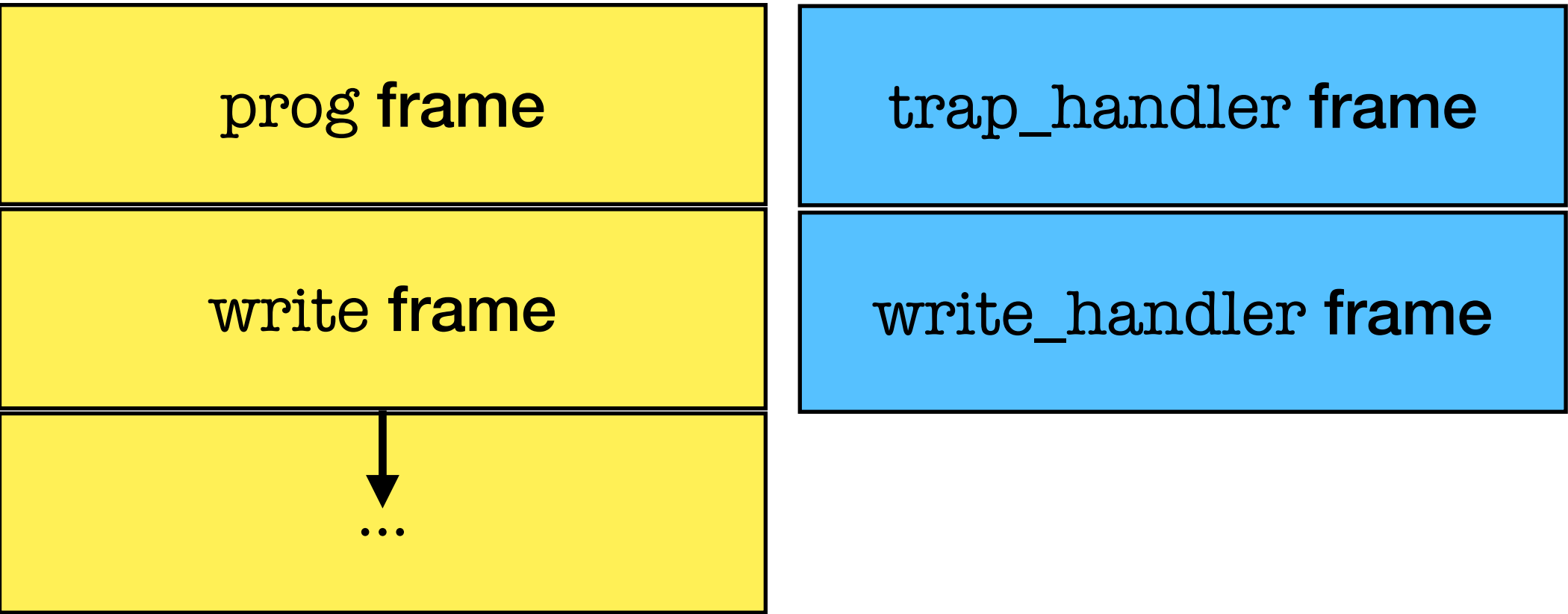
Think about the implementation with a centralized interrupt handler context/stack. What must be true about the data structure in this implementation for multicore architectures? What are the implications of this approach?

Must be a synchronized data structure! Two processors executing the interrupt handler should only add to the stack one at a time

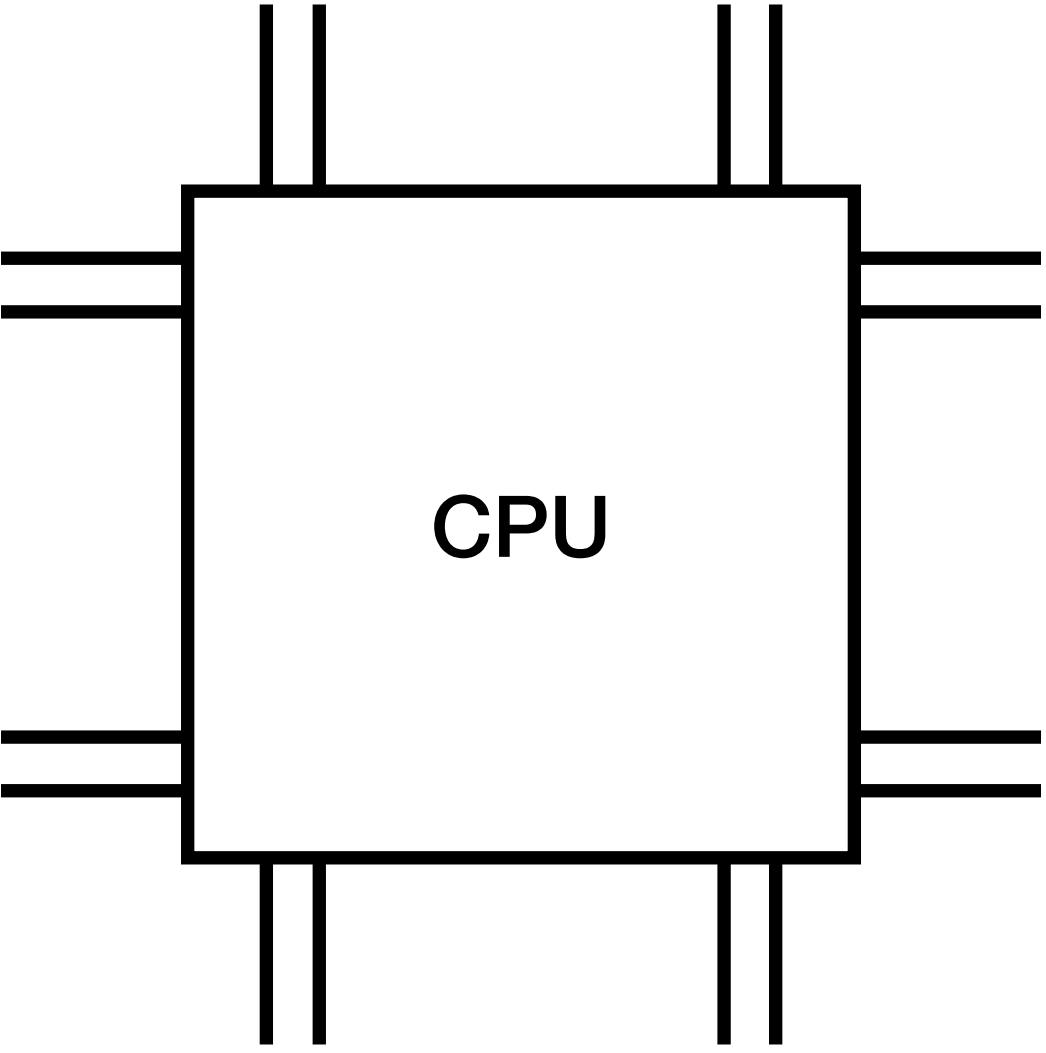
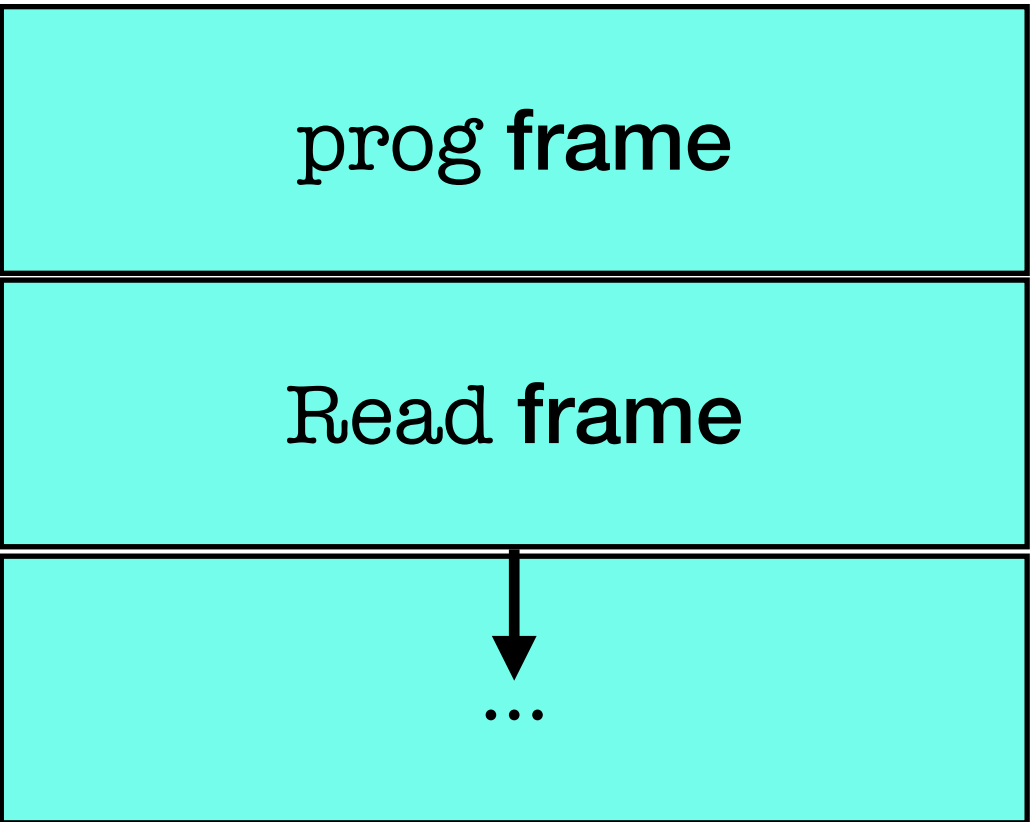
If different interrupts require different priority, then it is possible that a segmentation fault or floating point error is waiting for a different process to complete before the error message is thrown...

Implementing Per-Thread Interrupt Handlers

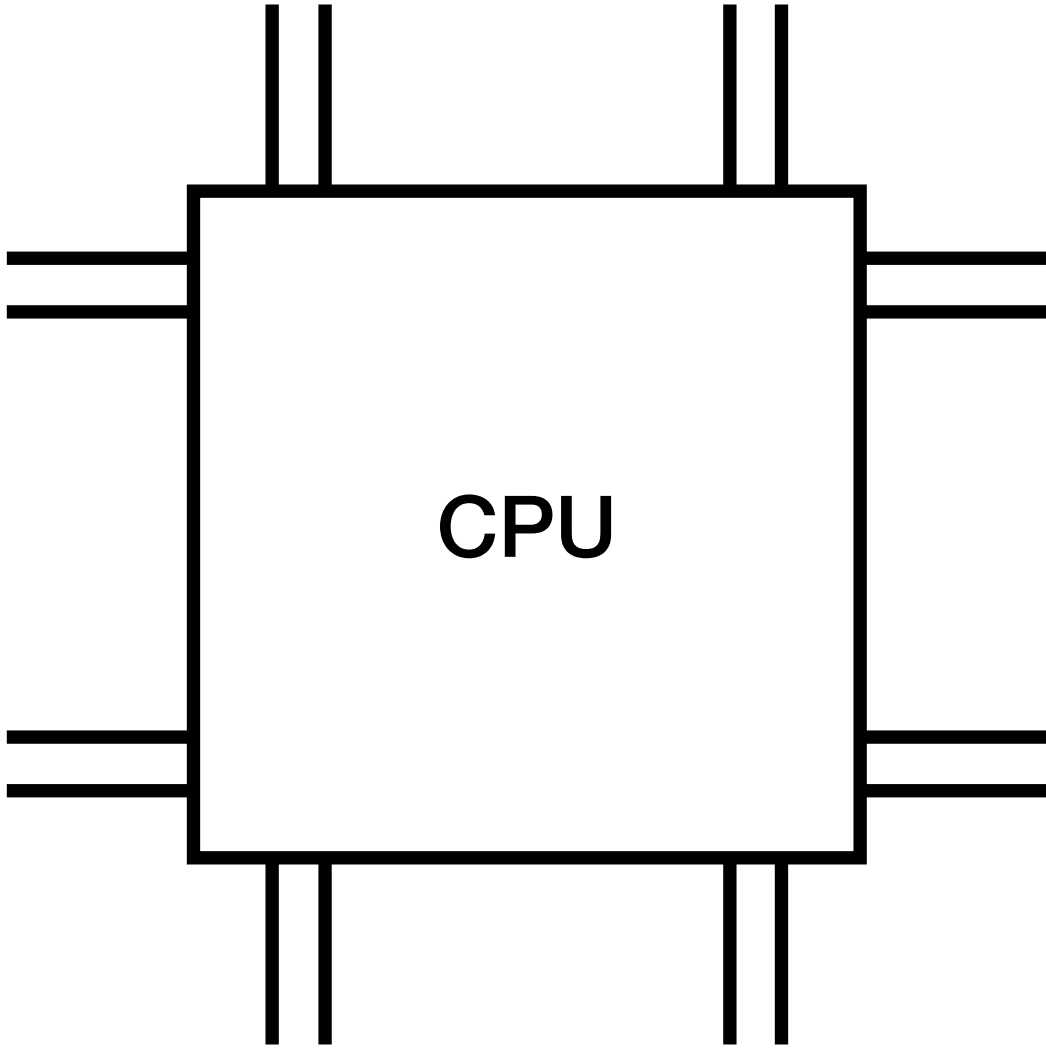
User Stack



User Stack



Requires lots of space!



Interrupt Handling

- An OS kernel may choose to implement a central interrupt handling stack (generally as a synchronized stack within the OS kernel body) or to implement interrupt handling on a per-thread/context basis
 - Early Digital Equipment Corporation VAX machines implemented a centralized data structure across the operating system, and later versions of this kernel implemented centralized interrupt handling per processor
- Regardless of which implementation is selected, both of these interrupt handling implementations require full execution of the interrupt before any other interrupt can be handled

Chat with your neighbor(s)!

Suppose a processor is executing an interrupt handler associated with a system call made by a user program. Is it possible for another interrupt to occur while the kernel is in the middle of handling the system call interrupt?

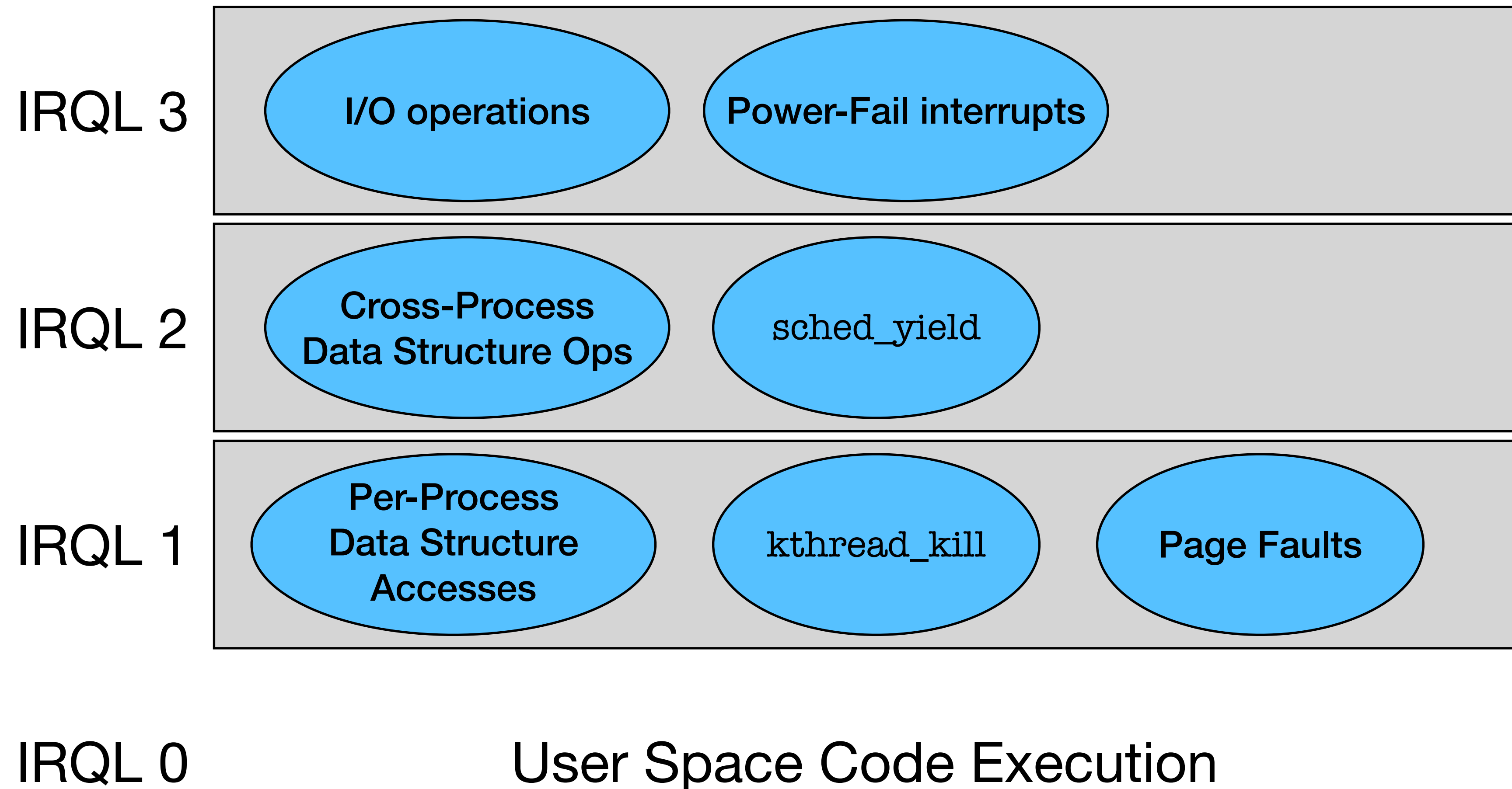
Hardware also invokes the
interrupt handler table!

Hardware Interrupts

- When a machine invokes an interrupt handler due to a hardware event (e.g., a key press, a network packet arrives, disk I/O event completes, etc...), the processor (hardware) typically disables all other interrupts until the handling of the first interrupt completes ➡ only when interrupt handling is reenabled are other events processed
- Some interrupt handlers require a lot of processing which means that interrupt handling may be disabled for long periods of time ➡ this means hardware may need to drop information to perform the handling
- Hardware interrupts are typically partitioned into top- and bottom-half interrupts to ensure that the handling is efficient... bottom half interrupts typically save the hardware state in a kernel buffer for the top-half interrupt to later manage

Handling Different Interrupt Levels

Insight: interrupts are merely instructions that may be interrupted (depending on who is being interrupted and who is doing the interrupting)!



Interrupts above this may not assume there is an active process/context

Chat with your neighbor(s)!

To implement scheduling, the processor has a regular hardware interrupt called a “timer,” which goes off on a regular cycle. Timer interrupts are categorized as IRQ 2. Given what we know about interrupts, how can we use a timer interrupts to implement a scheduling system?

Summary

- We can implement interrupt handlers in such a way that preserves user-space versus kernel-space semantics
- By implementing the user-space/kernel-space context switch, the kernel stack needs to be allocated as a separate data structure
- Interrupts themselves may need to be interrupted, so interrupt priority is used to reason about varying degrees of interrupts
- We can use hardware timer interrupts as the means to implement context switching between processes to implement ideas like scheduling