CS 134
Operating Systems

Feb 18, 2019

Interrupts, Exceptions, and System Calls

Common theme

- The hardware wants attention now!

Why does HW want attention now?

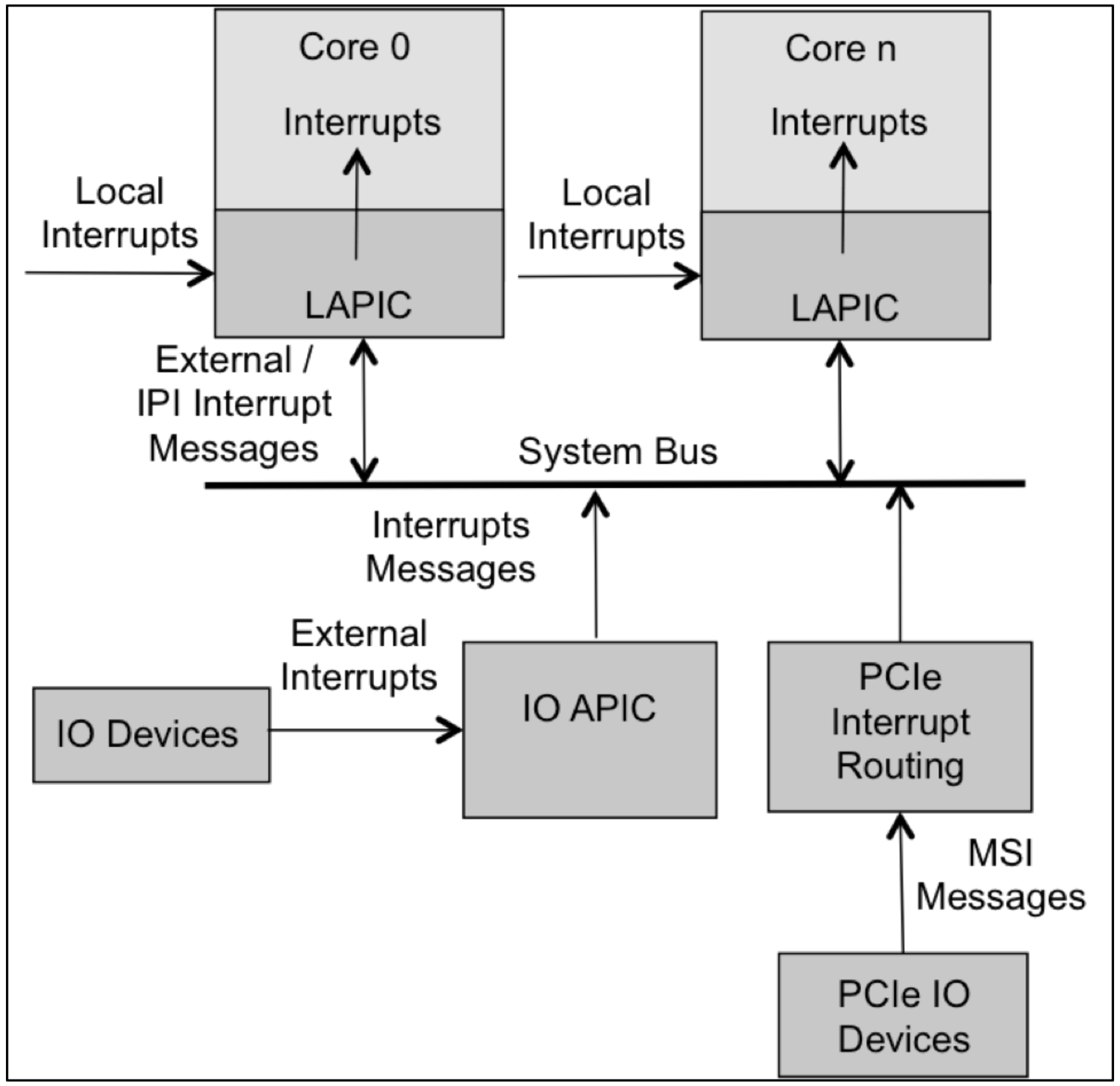
- MMU cannot translate address
- User program divides by zero
- User program wants to execute privileged instruction (INT)
- Network hardware wants to deliver a packet
- Timer hardware wants to deliver a “tick”
- Kernel CPU-to-CPU communication (e.g., to flush TLB)

Three basic classes

- Exceptions (e.g., page fault, divide by zero)
  - Faults: Saved `%eip` is that of faulting instruction
    - Can often be fixed and restarted
  - Aborts: Saved `%eip` unclear
    - Must kill the associated process:
    - Example: Double-fault (fault while handling a fault)
- System calls (INT, intended exception)
  - Saved `%eip` is after the INT instruction
- Interrupts (device wants attention)
  - Saved `%eip` is next instruction to execute

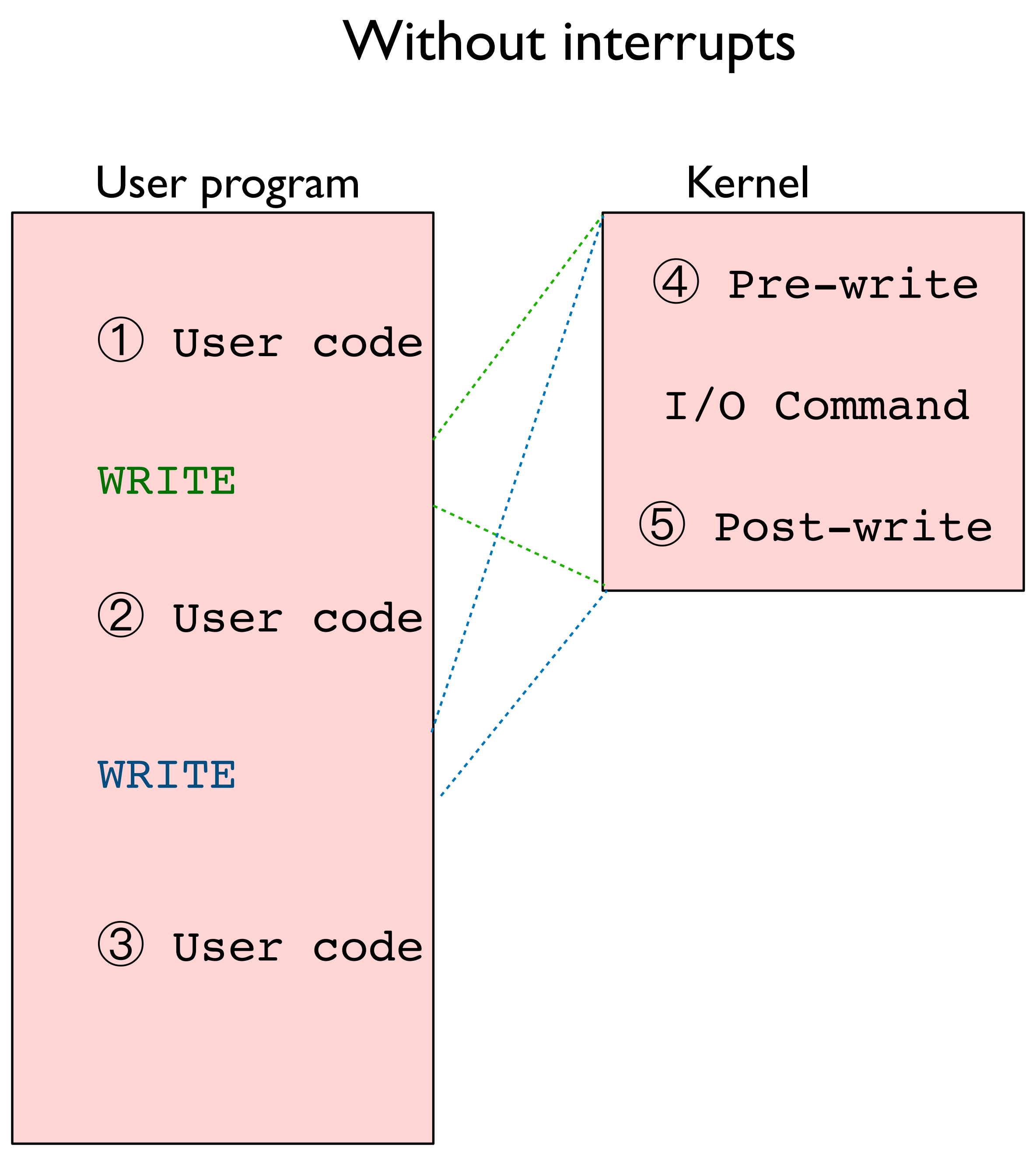
Where do device interrupts come from?

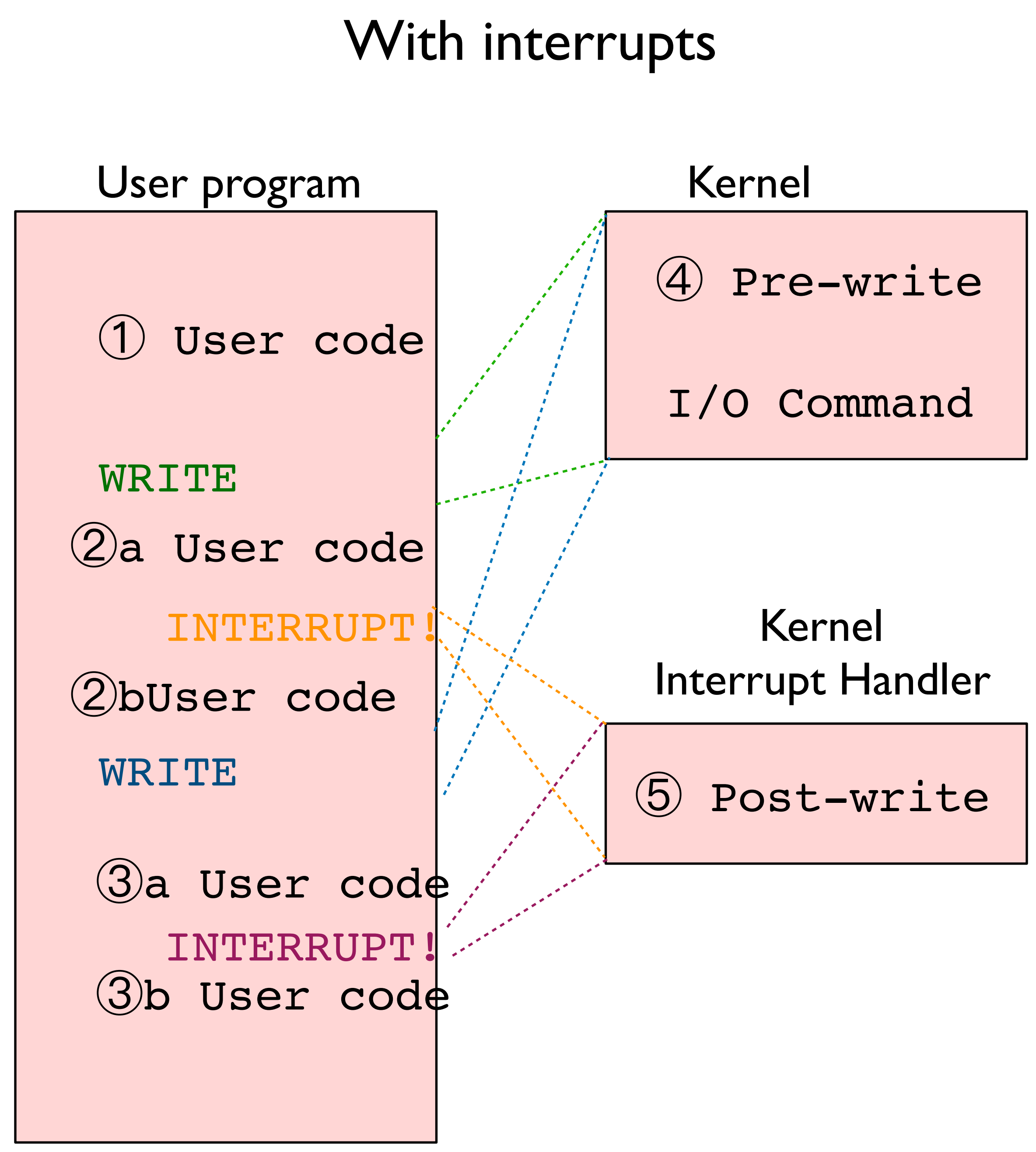
- Interrupt tells the kernel the device hardware wants attention
- The driver (in the kernel) knows how to tell the device to do things
- Often, the interrupt handler calls the relevant driver
  - Or, could be done differently (schedule a thread; poll)



APIC: Advanced Programmable Interrupt Controller
LAPIC: Local APIC: 1/processor
IO APIC: Input Output APIC: 1
MSI: Message Signaled Interrupts: don't need IO APIC

I/O with and without interrupts





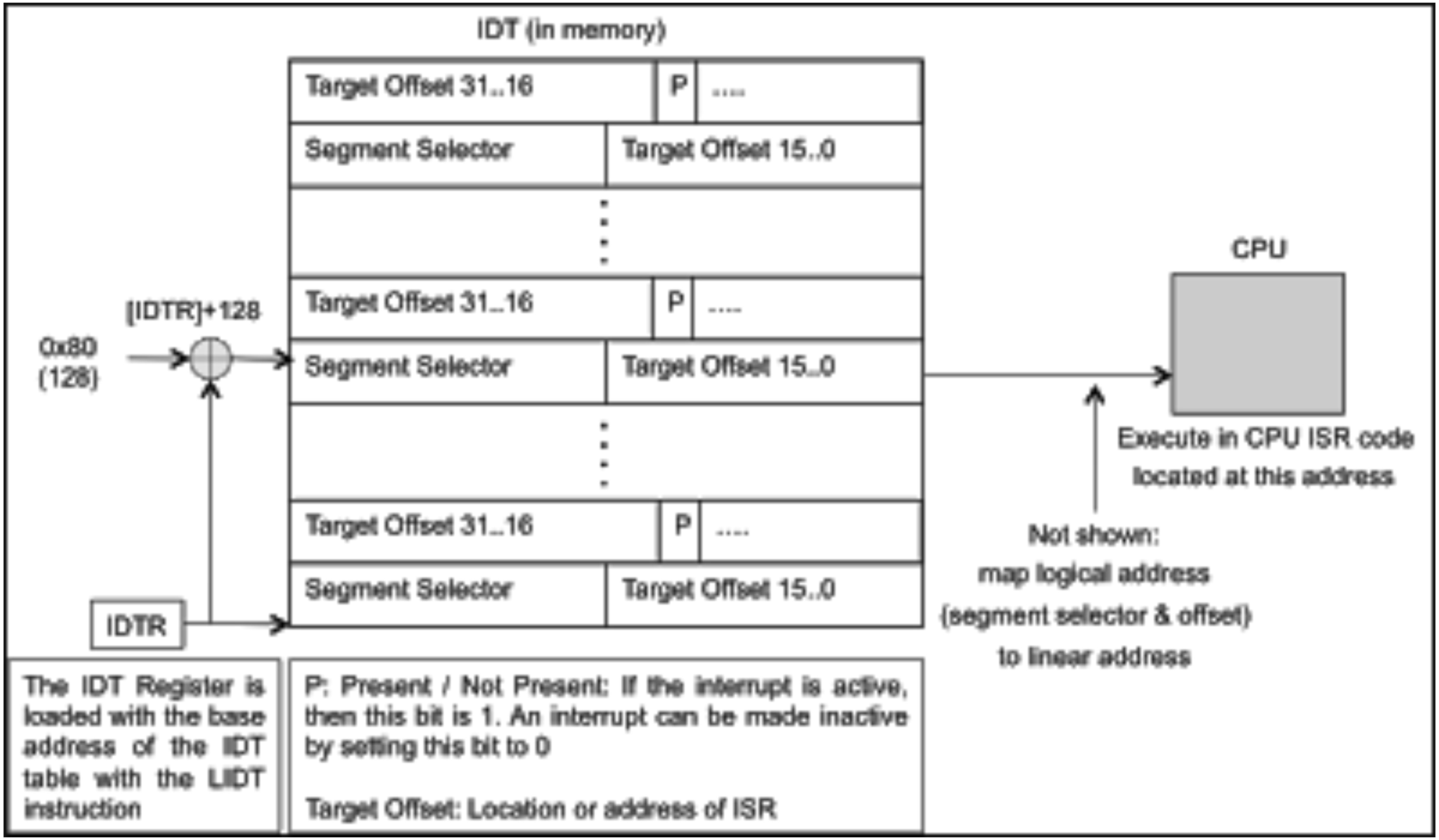
Interrupt cycle

- At beginning of FDE (Fetch-Decode-Execute) cycle, CPU checks for interrupt
- If no interrupt, fetch next instruction
- If interrupt pending:
  - Suspend execution of current program
  - Save context
  - Set PC to start address of Interrupt service routine (ISR) (via IDT)
  - Process interrupt (execute ISR)
  - Restore context (IRET), returning to interrupted code

How does `trap()` know which device interrupted?

- Where did:
  `tf->trapno == T_IRQ0 + IRQ_TIMER`
  come from
- Kernel tells IOAPIC/LAPIC what vector number to use (within IDT)
  - Page faults, traps also have vector numbers
- IDT associates an instruction with each vector number
- Each vector jumps to `alltraps` (pushing vector # first)
- CPU sends many kinds of traps through IDT
  - Low 32 IDT entries have special fixed meaning

How does `trap()` know which device interrupted?



The IDT Register is loaded with the base address of the IDT table with the LIDT instruction

P: Present / Not Present: If the interrupt is active, then this bit is 1. An interrupt can be made inactive by setting this bit to 0

Target Offset: Location or address of ISR

Diagram by Masum Z. Hasan

How xv6 uses interrupt vector machinery

- `lapic.c:lapicinnit()`—tells LAPIC HW to use vector 32 for timer

```
lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
```

- `trap.c:tvinit()`—initializes IDT so entry i points to code at vector i

```
for(i = 0; i < 256; i++)
  SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
```

  - But,

```
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER)
```

    `T_SYSCALL`'s 1 says to enable interrupts during system calls
    - Why allow interrupts during system calls?
    - Why not allow interrupts during interrupt handling?
  - `DPL_USER` allows the interrupt from user mode

How does the HW know what stack to use for an interrupt?

- User stack not OK (`%esp` might point anywhere)
- When it switches from user to kernel mode:
  - Hardware-defined TSS (Task State Segment) lets kernel configure CPU:
    - one per CPU
      - So, each CPU can take traps on different stacks
  - `proc.c:scheduler()`
    - one scheduler running for each CPU
  - `vm.c:switchuvm()`
    - Tells CPU what kernel stack to use (`proc->kstack`)
    - Tells CPU what page table to use (`proc->pgdir`)

An OS may allow nested interrupt handling

- Interrupts have a priority level
  - Higher priority interrupts are handled first
  - What if low-priority ISR is running and a higher-priority interrupt is pending
    - xv6: wait for ISR to finish
    - Or, could execute higher-priority ISR immediately
      - What happens to kernel stack?
      - How far could this go?

DMA (Direct Memory Access)

- Rather than having ISR read data from peripheral,

Without DMA

```
Device driver for read into buffer at addr XXXX:
  Tell I/O device to do a read
  Wait for interrupt to be generated
  Ask I/O device for data and copy to XXXX
  Read is complete
```

With DMA

```
Device driver for read into buffer at addr XXXX:
  Tell I/O device to do a read into PA(XXXX)
  Wait for interrupt to be generated
  Read is complete
```

HW 5: xv6 CPU alarm

- Interrupts plus system calls
- Challenges:
  - Get it to work at all
  - Maintain isolation (not easy to test!)

alarmtest.c

```
int
main(int argc, char *argv[])
{
  printf(1, "alarmtest starting\n");
  alarm(10, periodic);
  for(int i = 0; i < 25*500000; i++){
    if((i % 250000) == 0)
      write(2, ".", 1);
  }
  exit();
}

void
periodic()
{
  printf(1, "alarm!\n");
}
```

asks kernel to call `periodic()` every 10 "ticks" of CPU Time this process consumes

Alarm

- Need 3 parts:
  - New system call
  - Count ticks as the user program runs (timer interrupt)
  - Call back to user's registered callback ("upcall")

Glue for new system call

- Like HW 3: new system call

```
#define SYS_alarm  22
```

syscall.h

```
SYSCALL(alarm)
```

usys.S

```
extern int sys_alarm(void);
…
[SYS_alarm]   sys_alarm,
```

syscall.c

```
int
sys_alarm(void)
{
  int ticks;
  void (*handler)();

  if(argint(0, &ticks) < 0)
    return -1;
  if (argptr(1, (char **) &handler, 1) < 0)
    return -1;
  myproc()->alarmticks = ticks;
  myproc()->ticksuntilhandler = ticks;
  myproc()->alarmhandler = handler;
  return 0;
}
```

sysproc.c

```
struct proc {
  …
  int ticksuntilhandler;       // Num ticks left until calling alarm handler
  int alarmticks;
  void (*alarmhandler)();      // Call this function every alarmticks ticks
}
```

proc.h

Must take action when timer HW interrupts

```
case T_IRQ0 + IRQ_TIMER:
  …
  if (myproc() != 0 && (tf->cs & 3) == 3) {
    // Only if timer interrupt came from user space
    if (myproc()->ticksuntilhandler > 0) {
      if (--myproc()->ticksuntilhandler == 0) {
        myproc()->ticksuntilhandler = myproc()->alarmticks;
        // When alarm handler returns, we want it to return to the
        // code that was executing when this interrupt occurred.
        // Save space on the stack for return address;
        tf->esp -= 4;

        *((uint *) tf->esp) = tf->eip;
        // cause instruction pointer to be alarmhandler
        tf->eip = (uint) myproc()->alarmhandler;
      }
    }
  }
  lapiceoi();
  break;
```

trap.c

Why can't we just call `alarmhandler` directly?

```
case T_IRQ0 + IRQ_TIMER:
  …
  if (myproc() != 0 && (tf->cs & 3) == 3) {
    // Only if timer interrupt came from user space
    if (myproc()->ticksuntilhandler > 0) {
      if (--myproc()->ticksuntilhandler == 0) {
        myproc()->ticksuntilhandler = myproc()->alarmticks;
        myproc()->alarmhandler();
     }
    }
  }
  lapiceoi();
  break;
```

trap.c

Scary how close it came to working

- Why can we call from kernel code jump directly into user instructions?
- Why can user instructions modify the kernel stack?
- Why do system calls (INT) work from the kernel?

- We don’t want any of these behaviors in xv6!
  - x86 HW doesn’t directly provide isolation
  - Many separate x86 features (page tables, INT, user/kernel mode)
  - Possible to use these features to ensure isolation
    - Not the default!

What happens if we don’t reserve stack space?

```
  case T_IRQ0 + IRQ_TIMER:
    …
    if (myproc() != 0 && (tf->cs & 3) == 3) {
      // Only if timer interrupt came from user space
      if (myproc()->ticksuntilhandler > 0) {
        if (--myproc()->ticksuntilhandler == 0) {
          myproc()->ticksuntilhandler = myproc()->alarmticks;
          ~~// When alarm handler returns, we want it to return to the~~
          ~~// code that was executing when this interrupt occurred.~~
          ~~// Save space on the stack for return address;~~
          ~~tf->esp -= 4;~~

          ~~*((uint *) tf->esp) = tf->eip;~~
          // cause instruction pointer to be alarmhandler
          tf->eip = (uint) myproc()->alarmhandler;
        }
      }
    }
    lapiceoi();
    break;
```

trap.c

Where will alarmhandler return to after RET instruction?

What it trap didn’t check for CPL 3?

```
  case T_IRQ0 + IRQ_TIMER:
    …
    if (myproc() != 0 ~~&& (tf->cs & 3) == 3~~) {
~~      // Only if timer interrupt came from user space~~
      if (myproc()->ticksuntilhandler > 0) {
        if (--myproc()->ticksuntilhandler == 0) {
          myproc()->ticksuntilhandler = myproc()->alarmticks;
          // When alarm handler returns, we want it to return to the
          // code that was executing when this interrupt occurred.
          // Save space on the stack for return address;
          tf->esp -= 4;

          *((uint *) tf->esp) = tf->eip;
          // cause instruction pointer to be alarmhandler
          tf->eip = (uint) myproc()->alarmhandler;
        }
      }
    }
    lapiceoi();
    break;
```

trap.c

```
unexpected trap 14 from cpu 1 eip 8010517d (cr2=0x8010062d)
lapicid 1: panic: trap
```

```
        *((uint *) tf->esp) = tf->eip;
8010517a:       8b 57 38                mov    0x38(%edi),%edx
8010517d:       89 50 fc                mov    %edx,-0x4(%eax)
```

Sanity checking

- What if user-supplied alarm callback points to kernel code?

What if another timer interrupt happens while in `periodic()`

- Works, but is confusing
- Maybe kernel shouldn’t restart timer until handler function finishes?

Is it a problem if `periodic()` modifies registers?

- Yes!
- Interrupt can happen between any two instructions in `main()`
- How could we restore registers before returning from `periodic()`?

Interrupt handlers introduce concurrency

- Interrupt can happen between any two instructions
- Other code runs between those two instructions
- User code:
  - not so bad, but must be OK with periodic() running between any two instructions.
- Kernel code:
  - Could be a big issue. To make code in kernel atomic, surround with:
    - CLI: clear interrupt flag
    - STI: set interrupt flag

Interrupts vs. polling

- Interrupts take on the order of 1 microsecond
  - Cache miss, Save/restore state
- Some devices can generate interrupts faster than 1/microsecond:
  - Gigabit ethernet, for example
- What do do if interrupts come in faster?
  - Poll: processor spins waiting for device
  - No saving of registers
- Interrupt for low-rate devices (e.g, keyboard)
  - No wasting CPU time polling
- Poll for high-rate devices
  - No wasting CPU time interrupting
- Or, switch dynamically based on interrupt rate