Lecture 2: Architectural support

601.418/618 Operating Systems

David Hovemeyer

January 26, 2026

Assignment 0 update

A screencast video is available which walks through getting started on Assignment 0 using your ugrad (or grad) account.

Much of the info should also be relevant if you have set up a development environment on your own machine.

Agenda

- Hardware support required for robust virtualization of system resources
  - Privilege levels
  - Interrupts, traps, faults

Acknowledgments: These slides are based on Prof. Ryan Huang's Fall 2022 slides, which in turn are based on Prof. David Mazières's OS lecture notes. They also include some content developed by Prof. Phillipp Koehn for CSF.

Why start with hardware?

To understand how the OS kernel can robustly virtualize access to system resources, we need to know the underlying hardware mechanisms that make this virtualization possible.

Hardware features supporting OS functionality

- CPU privilege levels
- MMU (virtual memory)
- Exceptions
    - Hardware interrupts
    - Traps (especially system calls)
    - Faults
- The interval timer
- I/O devices (disk controllers, network controllers, etc.)
- Hardware support for synchronization

The OS kernel will need to use all of these!

- So you will need to understand what they do!

Types of architectural support

We'll consider two general types of architectural support needed to support OS functionality:

1. Manipulating privileged state
2. Handling and generating events

Manipulating privileged state

Privileged state

The OS kernel needs to create a "sandbox" for user processes. Within the sandbox, the running program can only access resources it has been granted access to.

User processes must *not* be allowed to access resources outside the sandbox:

- kernel memory
- memory belonging to other processes
- hardware devices
- etc.

In contrast, we want to allow the kernel to access all system resources without restriction.

Restricted operations

Preventing user processes from accessing forbidden resources can be achieved by preventing them from *executing CPU instructions* which access forbidden resources.

- Process can't execute any instruction accessing privileged state $\rightarrow$ process can never access privileged state

For example: if a process running on an x86 CPU could load a new value into the CR3 register, it could access memory outside its address space. (CR3 points to the "page directory", i.e., the root of the page table hierarchy.)

So, any `mov` instruction with the CR3 register as the destination must not be allowed for code executing in a user process.

Preventing execution of privileged instructions

One way to ensure that user processes can't execute privileged instructions would be to have the OS kernel *interpret* them.

- When the interpreter determines that a user process is trying to execute a privileged instruction, it doesn't execute the instruction

This would be quite slow!

Interpreters for machine instructions are possible (Bochs is exactly this!), but they generally impose a speed penalty in the neighborhood of 10x–100x compared to native execution on the CPU.

Execution modes

Better solution: CPU privilege levels.

Kernel mode: code executing in kernel mode can execute any instruction, and thus access any system resource without restriction.

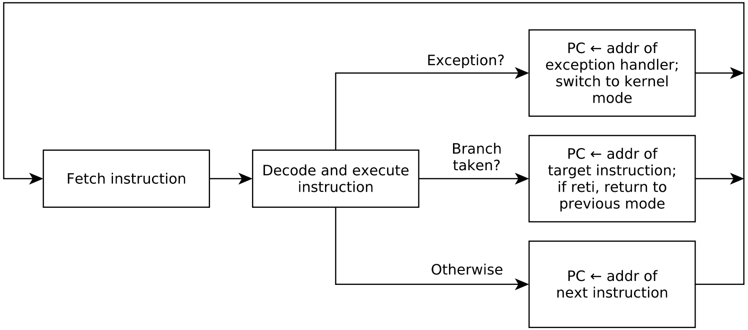
User mode: code executing in user mode can execute any *unprivileged* instruction without restriction, but any attempt to execute a *privileged* instruction causes an exception.

- We'll discuss exceptions soon!

All modern CPUs support at least these two privilege levels. (Some CPUs have fancier privilege models with more modes, but that's not strictly needed.)

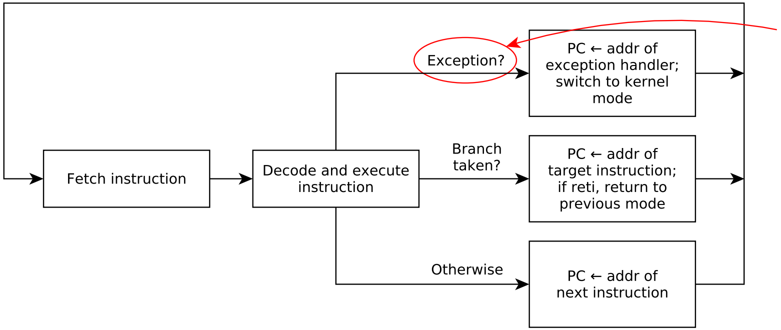
Executing code

Basic operation of a CPU:


Fetch instruction
Decode and execute instruction
Exception?
Branch taken?
Otherwise
PC ← addr of exception handler; switch to kernel mode
PC ← addr of target instruction; if reti, return to previous mode
PC ← addr of next instruction

Executing code

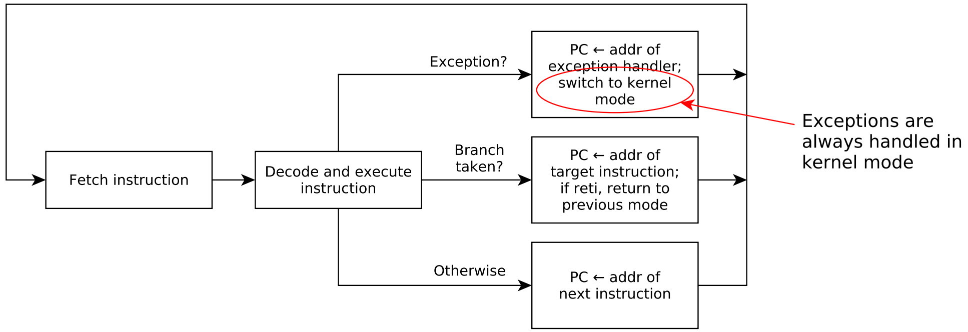
Basic operation of a CPU:


Fetch instruction
Decode and execute instruction
Exception?
PC ← addr of exception handler; switch to kernel mode
Branch taken?
PC ← addr of target instruction; if reti, return to previous mode
Otherwise
PC ← addr of next instruction

Among other possibilities: if instruction is privileged, and current mode is user mode, raise exception

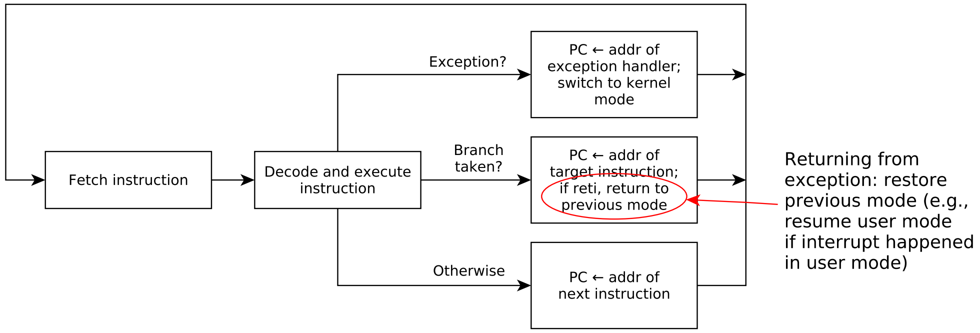
Executing code

Basic operation of a CPU:


Fetch instruction
Decode and execute instruction
Exception?
PC ← addr of exception handler; switch to kernel mode
Branch taken?
PC ← addr of target instruction; if reti, return to previous mode
Otherwise
PC ← addr of next instruction
Exceptions are always handled in kernel mode

Executing code

Basic operation of a CPU:


Fetch instruction
Decode and execute instruction
Exception?
PC ← addr of exception handler; switch to kernel mode
Branch taken?
PC ← addr of target instruction; if reti, return to previous mode
Otherwise
PC ← addr of next instruction
Returning from exception: restore previous mode (e.g., resume user mode if interrupt happened in user mode)

Which instructions must be privileged?

Any instruction that would allow a user process to

- access a resource belonging to the kernel or another process, or
- directly access hardware devices

must be a privileged instruction.

Example: `mov` to a control register (e.g., CR3).

From https://www.felixcloutier.com/x86/mov-1:

> *Moves the contents of a control register (CR0, CR2, CR3, CR4, or CR8) to a general-purpose register or the contents of a general-purpose register to a control register . . .* ***This instruction can be executed only when the current privilege level is 0.***

(On x86, privilege level 0 is kernel mode.)

Privilege models

More than 2 privilege levels are possible, and potentially useful.

For example, 32-bit x86 systems supported 4 privilege levels. “Gates” (descriptors for interrupt vectors) could specify a minimum privilege level required to use them.

However, two levels (kernel and user) are, in general, always sufficient. Fine-grained permissions can always be implemented by the kernel in software.

Memory protection

Memory is one of the most important kinds of system resources. We don't want a user process accessing memory belonging to another process or the kernel.

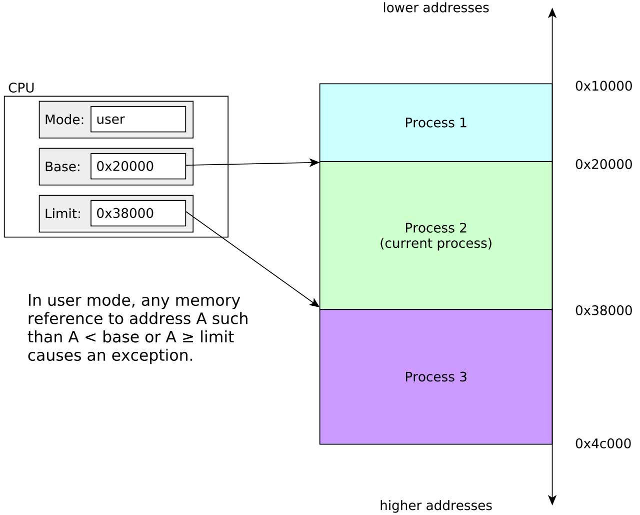
How to restrict memory accesses?

Segmentation

A simple form of memory protection is *segmentation*. When code runs in user mode, *base* and *limit* registers define the range of addresses the user code is permitted to access.

Instructions to set the base and limit registers are privileged (kernel-only) instructions.

Segmentation example



In user mode, any memory reference to address A such than A < base or A ≥ limit causes an exception.

Segmentation pros and cons

Pros:

- Simple to implement (hardware, software)

Cons:

- Process memory must be contiguous
  - Hard to have expandable memory areas (heap, stack)
- Poor utilization of system memory
  - Memory references are not uniformly distributed over the address space (some locations are accessed more frequently and some less frequently)

Paging

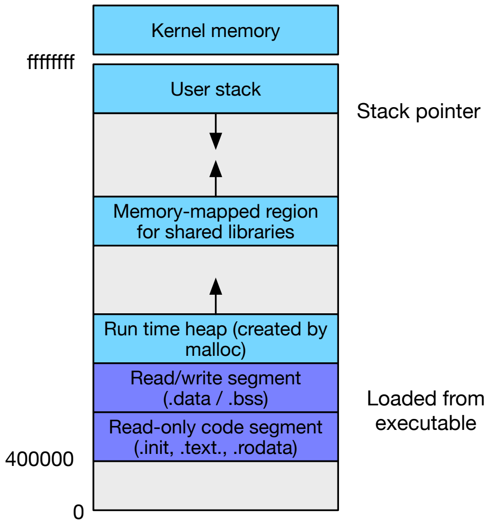
Paged virtual memory:

- Process address space is a sequence of *virtual pages*
- Each virtual page can be mapped either to
  - An arbitrary physical page, or
  - Nothing

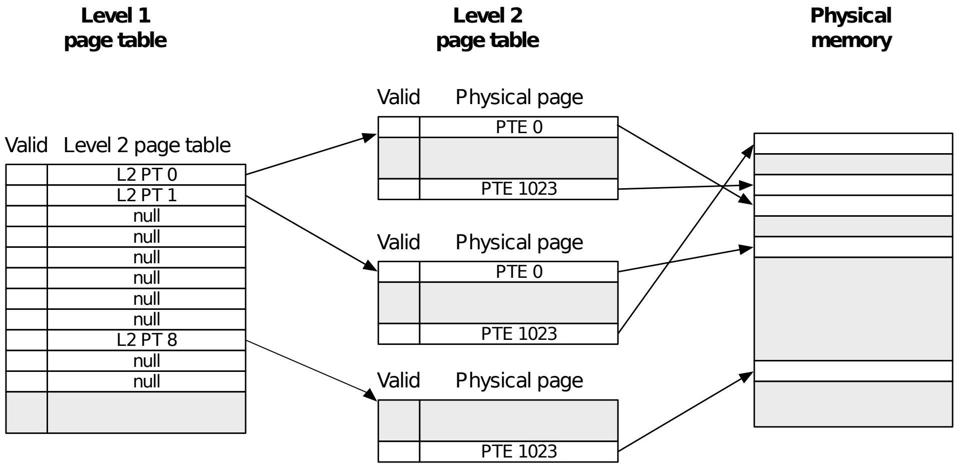
If a process executes a memory reference to a virtual page not currently mapped to a physical page, a *page fault* exception occurs.

- The OS kernel's *page fault handler* can resolve the fault by mapping a physical page and re-executing the faulting instruction

Process virtual address space


Kernel memory
ffffffff
User stack
Stack pointer
Memory-mapped region for shared libraries
Run time heap (created by malloc)
Read/write segment (.data / .bss)
Read-only code segment (.init, .text., .rodata)
Loaded from executable
400000
0

Page tables


Level 1 page table
Level 2 page table
Physical memory
Valid
Level 2 page table
L2 PT 0
L2 PT 1
null
null
null
null
null
null
L2 PT 8
null
null
Valid
Physical page
PTE 0
PTE 1023
Valid
Physical page
PTE 0
PTE 1023
Valid
Physical page
PTE 1023

Paging pros and cons

Pros:

- Address space can be sparse
- Pages of physical memory can be allocated to processes on demand
  - Can prioritize use of physical memory for the pages that are “hot”
- OS kernel’s page fault handler can implement arbitrary semantics when a page is needed
  - Enabling things like memory-mapped file I/O

Cons:

- Hardware complexity
  - E.g., TLB needed to get good performance
- Software complexity

Opinion: paged virtual memory is really cool.

Handling/generating events

Events

To a large degree, the OS kernel is focused on handling *events*.

Architectural support is needed for:

- responding to events (both synchronous events generated by processes and asynchronous events generated by hardware devices)
- generating events (e.g., allowing processes to request a system call)

OS kernel = event handlers

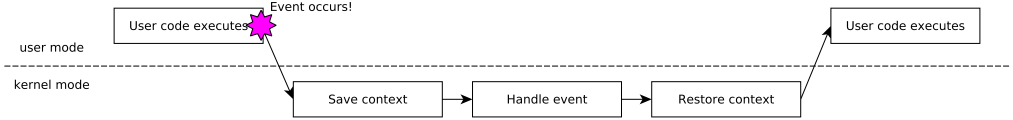
Code execution in the kernel is more or less always as the result of the occurrence of an event (hw interrupt, etc.) The kernel doesn't really have a `main`.

- The kernel may create long-running threads for housekeeping tasks, but ultimately its purpose is to provide services to processes

When an event occurs

- The CPU vectors to the handler routine (including switching to kernel mode)
- Handler routine saves context (values in CPU registers)
- Handler executes code to respond to the event
  - could involve switching to a different task!
- When handling is complete, restore context and return to interrupted code (often, a user mode process)

Handling an event



Asynchronous vs. synchronous events

Events can be *asynchronous* or *synchronous*.

Asynchronous events:

- Can occur at any time
- Examples: hardware interrupt

Synchronous events:

- Happen as a result of code execution
- Examples: traps (e.g., system calls), faults

Hardware events

- At (often) unpredictable times, hardware devices require attention
    - E.g., I/O hardware (disk controller, etc.)
        - Example event: data requested from disk is is ready
- When a hardware event occurs, the OS kernel needs to execute a *handler routine*
    - Do whatever is appropriate to deal with the event

How can we arrange for handler routines to be executed as hardware events occur?

Polling

Polling: OS kernel periodically checks to see whether a device requires attention.

- If it does, it executes the handler routine
- Example: check keyboard periodically to see if key was pressed or released

This works, but requires the OS kernel to spend CPU time checking for a situation that might not have actually happened.

- Also, depending on how frequently polling is done, could introduce a delay in handling an event (latency)

Note: Polling can be advantageous in some situations; for example, if a device generates interrupts very frequently, it may be better to poll and handle the hardware events in batches.

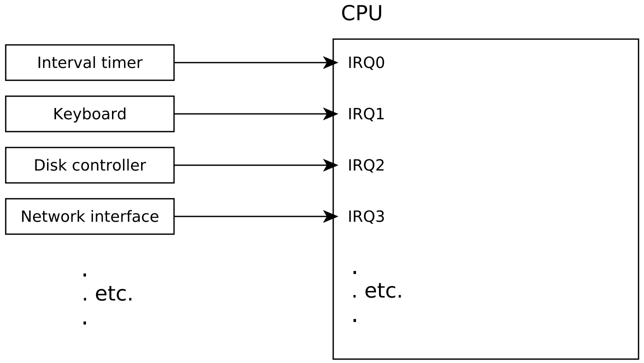
Hardware interrupts

- In a system supporting *hardware interrupts*, when a hardware even occurs, the hardware device immediately notifies the CPU
- In response, the CPU immediately transfers control from the currently-executing codee, and executes the handler routine associated with the interrupt
- Interrupts mean the OS kernel only uses CPU time if there is a situation needing attention
  - And, the kernel knows right away that the event occurred

Issue: how to devices get the kernel's attention?

One option: one CPU input per device

- One option: CPU has an input pin per device, hw device asserts when it needs attention
    - Signal that a device requires attention: “interrupt request”, a.k.a. “IRQ”



Drawbacks to one input per device

- Could require lots of pins (what if we run out?)
- If multiple devices assert, how does the CPU decide which to service?
- Sometimes the kernel may need to defer handling a particular interrupt (because it is doing something critical)

Better option: interrupt controller

Interrupt controller:

- Have a dedicated device (or devices) to monitor IRQs
- Interrupt controller notifies CPU when device(s) require attention
- Interrupt controllers can
  - Support “masking” (temporarily deferring handling of some IRQs)
  - Support priorities (if multiple interrupts are active, priorities to determine which gets sent to CPU next)

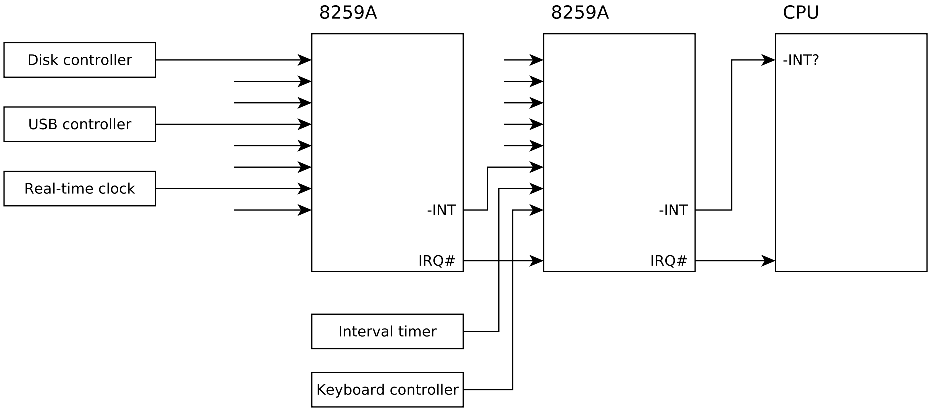
8259A PIC

32-bit x86 PCs use two Intel 8259A Peripheral Interrupt Controller (PIC) chips.

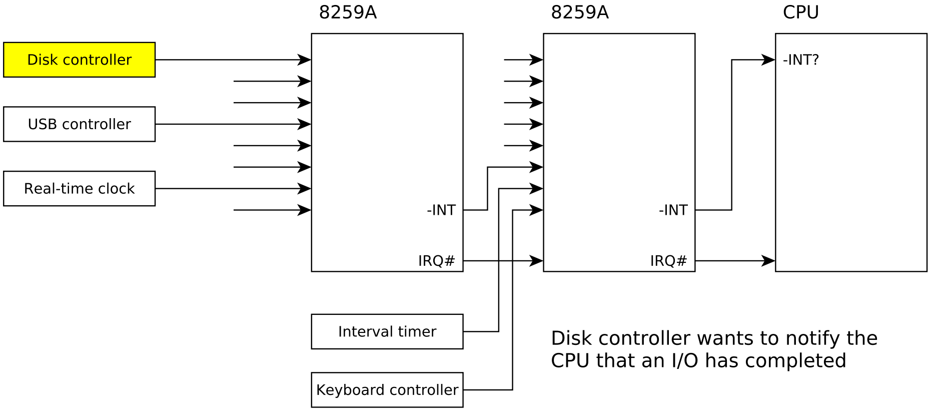
- This hardware design originates in the IBM PC/AT in 1984!
  - Which in turn is based on the design of the original IBM PC in 1981, which used a single 8259A
- In practice, most 32-bit x86 PCs use equivalent logic integrated into the motherboard chipset rather than actual 8259A chips

Each 8259A has 8 interrupt request (IRQ) inputs. However, the IRQs on one device (handling IRQs 8–15) are "cascaded" to the input for IRQ2 on the second device.

8259A PIC

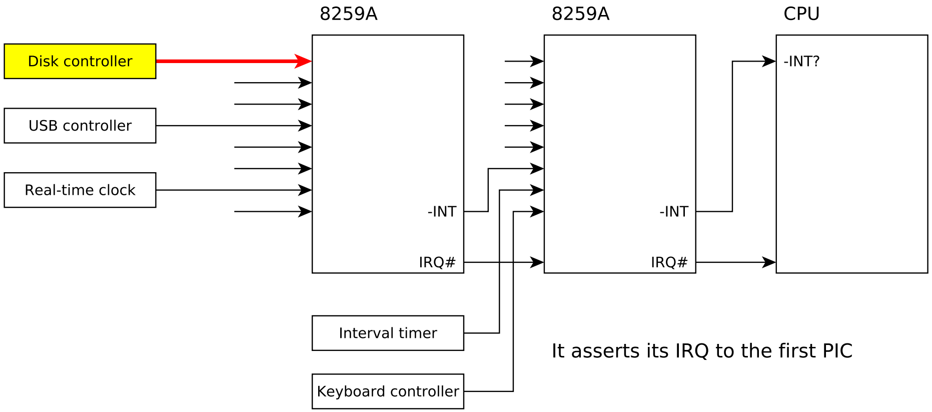

8259A
8259A
CPU
Disk controller
USB controller
Real-time clock
-INT
IRQ#
-INT
IRQ#
-INT?
Interval timer
Keyboard controller

8259A PIC



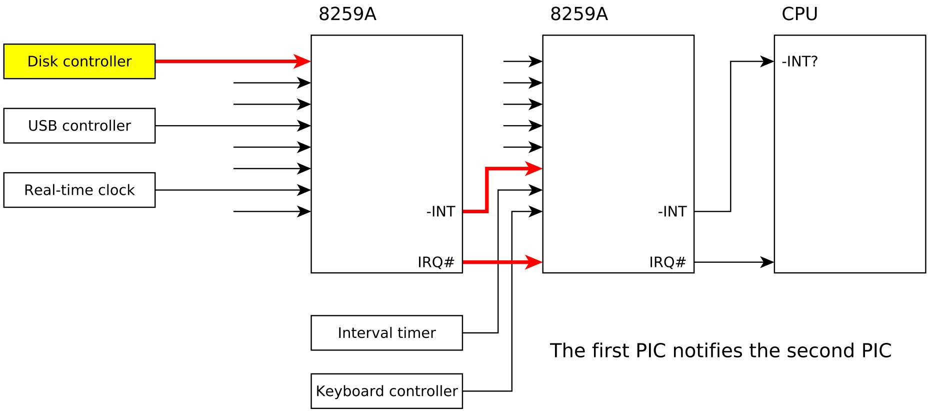
Disk controller wants to notify the CPU that an I/O has completed

8259A PIC



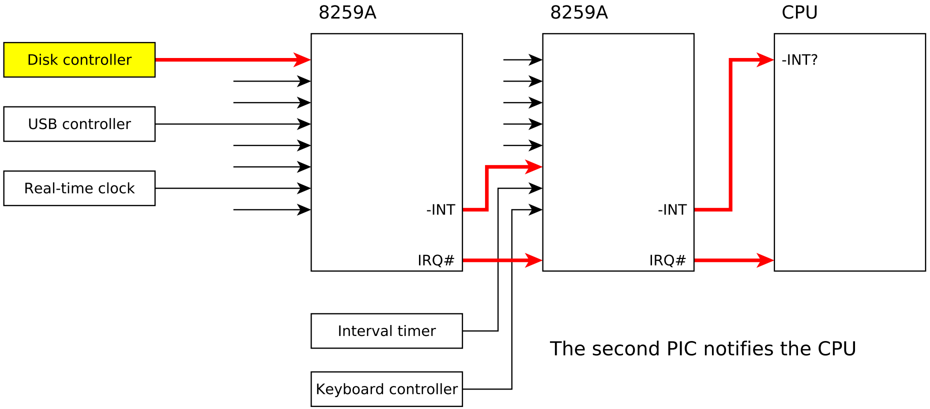
It asserts its IRQ to the first PIC

8259A PIC



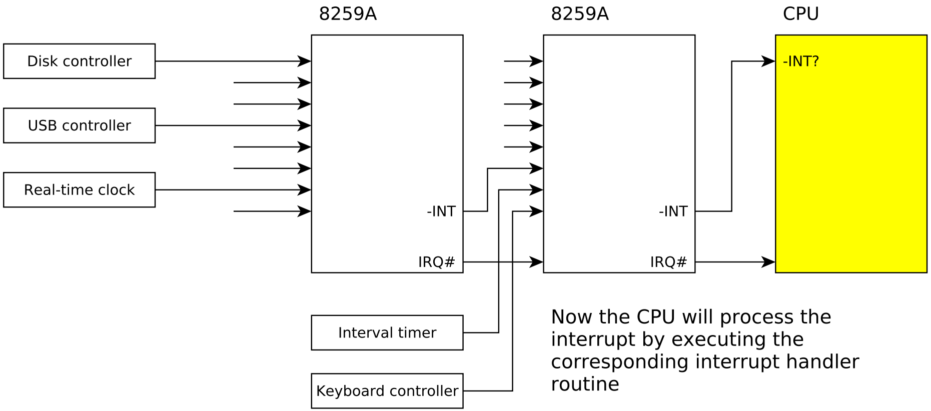
The first PIC notifies the second PIC

8259A PIC



The second PIC notifies the CPU

8259A PIC



Now the CPU will process the interrupt by executing the corresponding interrupt handler routine

IRQs in Pintos

In Pintos, IRQs 0–15 are mapped to the CPU interrupt vectors 32–47 in `src/threads/interrupt.c`.

The PIC(s) allow the kernel to "mask" any subset of IRQs to temporarily disable them. (If a hardware device asserts its IRQ while the IRQ is masked, handling the interrupt is deferred until the OS kernel unmasks the IRQ.)

- Pintos does not appear to use this capability

The CPU can also disable processing of all external interrupts.

- Pintos definitely *does* use this capability: `intr_get_level()`, `intr_set_level()` functions

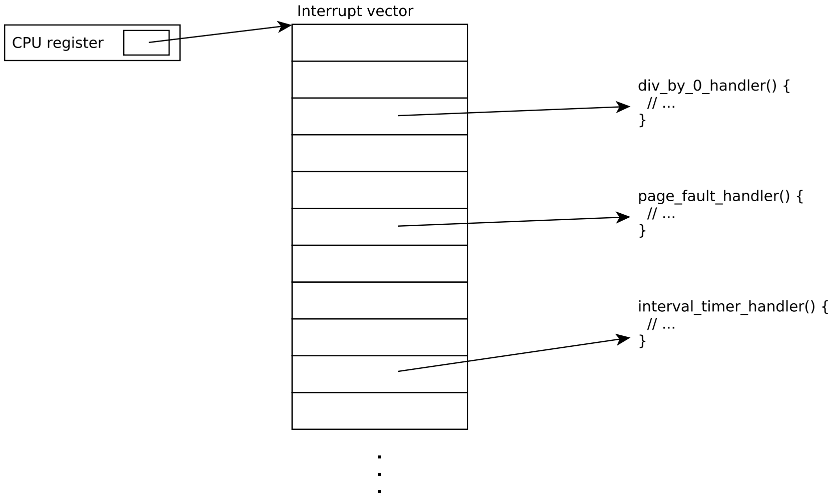
Interrupt vector

CPUs supporting hardware interrupts need a mechanism to allow the OS kernel to specify the handler routines to be executed for each IRQ.

This is typically implemented using an *interrupt vector*, which is (essentially) an array of pointers to the code addresses of handler routines.

Often there is a special CPU register storing the base address of the interrupt vector.

Interrupt vector



x86 interrupt vector

x86 interrupt vector

- Called the “Interrupt Descriptor Table (IDT)”
- Array of “Interrupt Gates”
- Special CPU register (IDTR) points to the start of this table
- In Pintos: `make_intr_gate` (`src/threads/interrupt.c`)

Example use of interrupts: the interval timer

- On standard x86 PC, the Intel 8254 (programmable interval timer, a.k.a. “PIT”)
  - As with the 8259A, usually integrated into the motherboard chipset rather than a discrete device
- The OS kernel can program the PIT to interrupt the CPU at regular intervals
- Ensures that OS kernel regains control of the CPU regularly
  - the OS kernel can use timer interrupts to make scheduling decisions
  - avoid any process from monopolizing use of the CPU
- You will make use of the interval timer in Assignment 1

Interval timer in Pintos

```
/* in devices/timer.c */

/* Sets up the timer to interrupt TIMER_FREQ times per second,
   and registers the corresponding interrupt. */
void
timer_init (void)
{
  pit_configure_channel (0, 2, TIMER_FREQ);
  intr_register_ext (0x20, timer_interrupt, "8254 Timer");
}

/* Timer interrupt handler. */
static void timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();
}
```

Interval timer in Pintos

```c
/* in threads/thread.c */

/* Called by the timer interrupt handler at each timer tick.
   Thus, this function runs in an external interrupt context. */
void
thread_tick (void)
{
  struct thread *t = thread_current ();

  /* Update statistics. */
  if (t == idle_thread)
    idle_ticks++;
  else
    kernel_ticks++;

  /* Enforce preemption. */
  if (++thread_ticks >= TIME_SLICE)
    intr_yield_on_return ();
}
```

Example use of interrupts: I/O devices

E.g., hard drive controller, network controller, etc.

Typical scenario:

1. Process uses system call to request I/O (let's say, reading from a disk)
2. OS kernel instructs the disk controller to begin reading specified data
    - Usually this means finding a different task to execute while the I/O is pending!
3. When the I/O completes, the disk controller asserts its IRQ
4. OS kernel interrupt transfers the data read from disk to the user process's memory
5. Process can be resumed (read from disk is complete)

Hardware interrupts vs. exceptions

Hardware interrupts: are asynchronous, can happen at any time

- We've just discussed these

Exceptions (traps, faults): are *synchronous*, happen as a result of executing an instruction

Most important kind of trap: *system call*

- Used by user process to request service from the OS kernel

Examples of faults:

- Page fault (usually, process accessed a virtual page that isn't mapped to a physical page)
- Divide by zero (and other error conditions)

System calls on x86

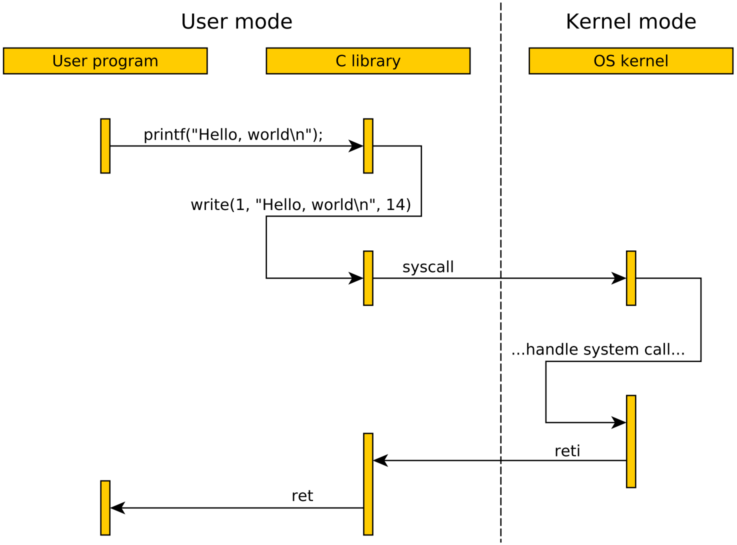
- Linux on 32-bit x86 used `int 0x80` instruction to initiate a system call
- Pintos uses `int 0x30`
- Linux on x86-64 uses the `syscall` instruction (you might recall this from CSF)
  - IIRC `int 0x80` still works, though

Hardware support for trap instructions

When a trap is executed, the CPU must

1. Switch to kernel mode
2. Save any context information that will be needed to seamlessly resume execution of the user program
   - E.g., the address of the instruction following the trap
3. Look up the address of the handler routine for the trap in the interrupt vector table
4. Execute the interrupt handler routine
   - When it is done, the interrupt handler routine will need to restore the CPU state (program counter, CPU register values, return to user mode)
     - In x86, the `reti` instruction ("return from interrupt" implements the return to the interrupted code, including the mode switch)

System call example



Naming resources

A significant issue in the design of the system call interface is *naming* resources involved in the request.

Unix/Linux use *file descriptors* to describe many resources, especially files and communication channels.

Important!

Checking system call arguments to verify whether the invoking process is allowed to access a specific resource is hugely significant from a security perspective.

Faults

A *fault* is similar to a trap, but

1. A trap is always executed deliberately by the process
2. A fault indicates an error or a runtime condition indicating that an instruction can't be executed for some reason

Example faults:

- divide by 0
- page fault

Difference between traps and faults

- Trap: resume execution at instruction *following* the one that caused the trap
- Fault: handled by
  - *Repeating* the execution of the instruction that caused the fault once the situation is resolved (e.g., after page fault, kernel maps a physical page and reexecutes the faulting instruction)
  - Notifying the process (e.g., sending SIGFPE signal for divide by 0)
    - execution of process could continue by executing a signal handler

Unrecoverable faults

If the OS kernel determines that a fault is unrecoverable (e.g., divide by 0, but no SIGFPE signal handler), it (typically) terminates the process.

Fault in kernel code

A fault in kernel code generally can't be recovered from.

- The kernel should strictly avoid executing any instruction that would cause a fault: dereferencing a null pointer (page fault), dividing by zero, etc.
- As a diagnostic, the kernel typically displays an error message and halts the system (e.g., Linux kernel panic, Windows BSOD)

Synchronization and interrupts

- Interrupts can happen any time
- What if an interrupt handler needs to access a data structure?
    - What if the interrupt occurs while other kernel code is modifying that data structure?
- Mechanisms for synchronizing code with interrupt handlers:
    - Mask the IRQ: disable handling of hardware interrupt from a particular device, reenable later
    - Disable interrupts: have the CPU disable all interrupts, reenable later (x86: `cli`/`sei` instructions)
    - Atomic machine instructions: atomically inspect and/or modify a memory location (x86: `lock` prefix, `xchg`, `cmpxchg` instructions)