

---

# Lecture 5: Interrupts

CSC 469H1F / CSC 2208H1F

Fall 2007

Angela Demke Brown



---

# Topics

---


- What is an interrupt?
- How do operating systems handle interrupts?
  - FreeBSD example
  - Linux in tutorial on Thursday

# Interrupts

**Defn:** an event *external* to the currently executing process that causes a change in the normal flow of instruction execution; usually generated by hardware devices external to the CPU

- From "*Design and Implementation of the FreeBSD Operating System*", *Glossary*
- Key point is that interrupts are *asynchronous* w.r.t. current process
  - Typically indicate that some device needs service

# Why Interrupts?

- People like connecting devices
    - A computer is much more than the CPU
      - Keyboard, mouse, screen, disk drives
      - Scanner, printer, sound card, camera, etc.
  - These devices occasionally need CPU service
    - But we can't predict *when*
  - External events typically occur on a macroscopic timescale
    - we want to keep the CPU busy between events
-  Need a way for CPU to find out devices need attention

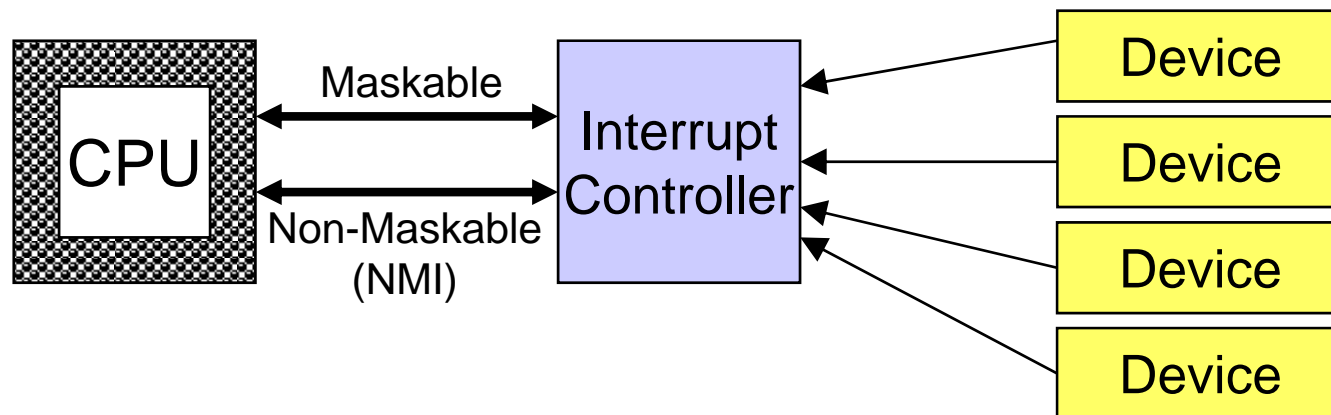
# Possible Solution: Polling

- CPU periodically checks each device to see if it needs service
  - ✗ takes CPU time even when no requests pending
  - ✗ overhead may be reduced at expense of response time
  - ✓ can be efficient if events arrive rapidly

*"Polling is like picking up your phone every few seconds to see if you have a call. ..."*

# Alternative: Interrupts

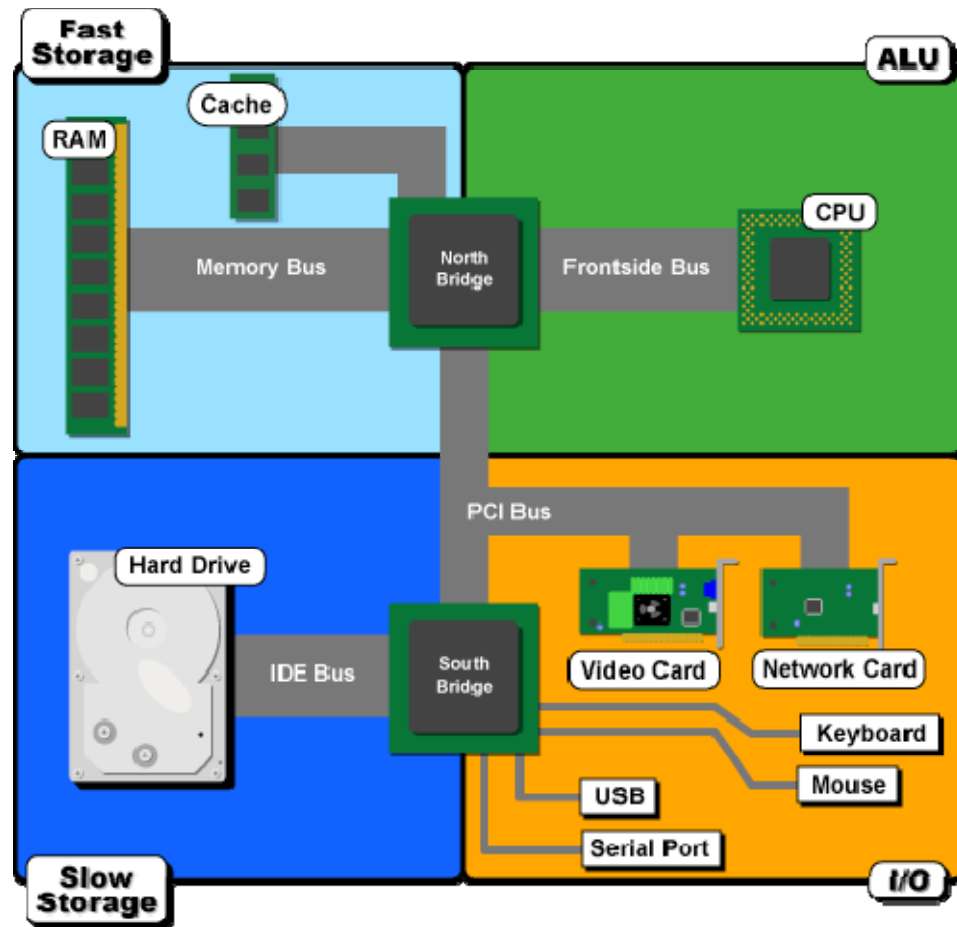
- Give each device a wire (interrupt line) that it can use to signal the processor
  - When interrupt signaled, processor executes a routine called an *interrupt handler* to deal with the interrupt
  - No overhead when no requests pending



# Intel 430HX Motherboard

- Programmable interrupt controller (PIC) part of the "Southbridge" chip
  - Commonly 8259A chip
  - 8 inputs, 1 output
  - Can be chained together
- Newer systems use "Advanced PIC" (APIC) for SMP support
  - Principle is the same

(image from The Ars Technica Motherboard Guide, Dec. 2005, Jon Hannibal Stokes)



# Polling vs. Interrupts

*"Polling is like picking up your phone every few seconds to see if you have a call. Interrupts are like waiting for the phone to ring."*

- Interrupts win if processor has other work to do *and* event response time is not critical
- Polling can be better if processor has to respond to an event ASAP
  - May be used in device controller that contains dedicated secondary processor

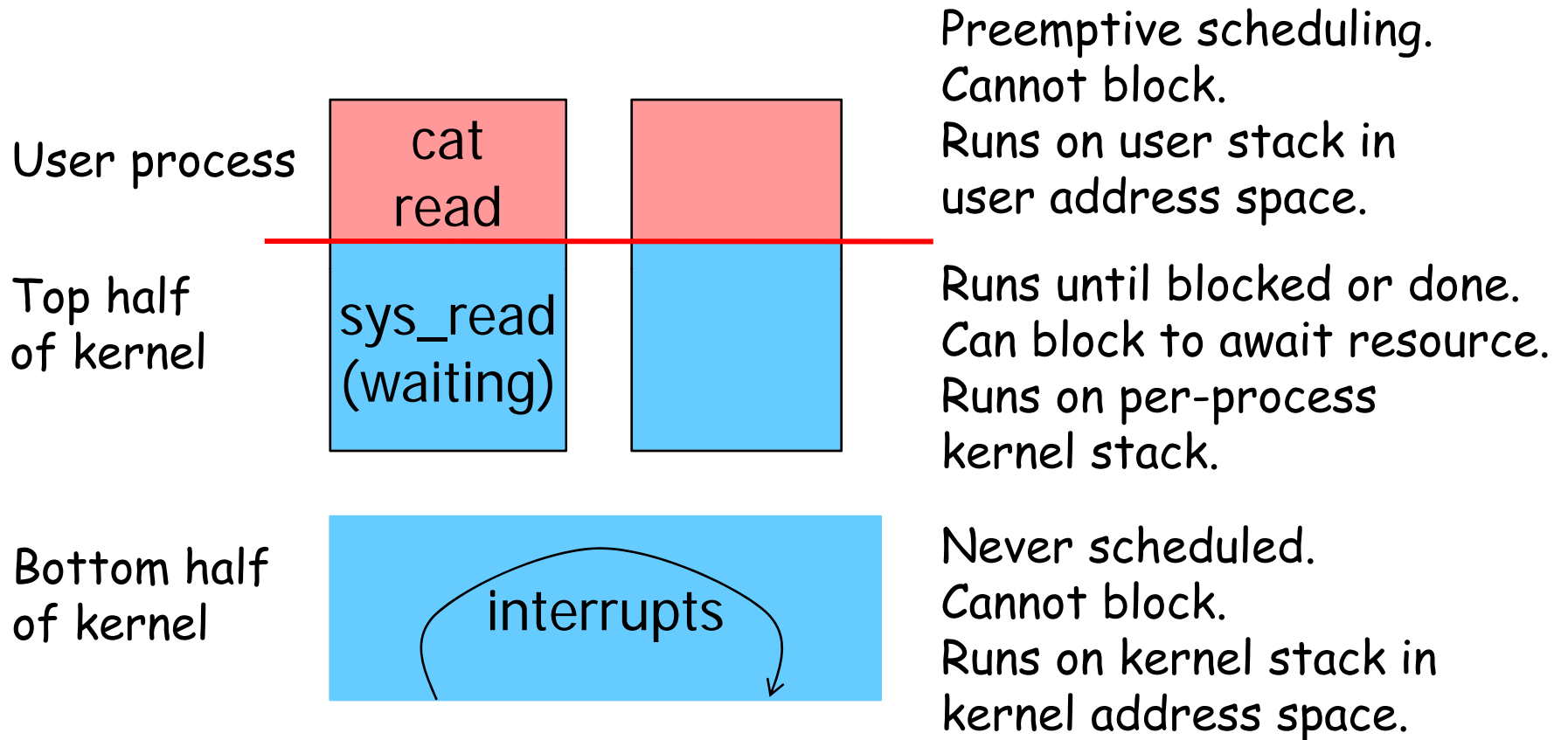
# Hardware Interrupt Handling

- Details are architecture dependent!
- Interrupt controller signals CPU that interrupt has occurred, passes interrupt number
  - Interrupts are assigned priorities to handle simultaneous interrupts
  - Lower priority interrupts may be disabled during service
- CPU senses (checks) *interrupt request line* after every instruction; if raised, then:
  - uses interrupt number to determine which handler to start
  - *interrupt vector* associates handlers with interrupts
- Basic program state saved (as for system call)
- CPU jumps to interrupt handler
- When interrupt done, program state reloaded and program resumes

# Software Interrupt Handling

- Typically two parts to interrupt handling
  - The part that has to be done immediately
    - So that device can continue working
  - The part that should be deferred for later
    - So that we can respond to the device faster
    - So that we have a more convenient execution context
      - What does *that* mean?

# BSD Unix Kernel Structure



# Interrupt Context

- Execution of first part of interrupt handler “borrows” the context of whatever was interrupted
  - Interrupted process state is saved in process structure
  - Handler uses interrupted thread’s kernel stack
    - Have to be *very* careful about stack-allocated data
  - Handler is not allowed to block
    - Has no process structure of its own to save state or allow rescheduling
    - Can’t call functions that might block (like `kmalloc`)
- Handler needs to be kept fast and simple
  - Typically sets up work for second part, flags that second part needs to execute, and re-enables interrupt

# Software Interrupts

- The deferred parts of interrupt handling are sometimes referred to as “software interrupts”
  - In Linux, they are referred to as “bottom halves”
  - The terminology here is inconsistent and confusing
- What things can be deferred?
  - Networking
    - time-critical work → copy packet off hardware, respond to hardware
    - Deferred work → process packet, pass to correct application
  - Timers
    - Time-critical → increment current time-of-day
    - Deferred → recalculate process priorities

# FreeBSD 5.2 & up

- All hardware devices and other interrupt events have an associated kernel thread with suitable priority
- First part of interrupt handling just schedules proper thread to run
  - Interrupted thread is marked as needing reschedule
  - High-priority handler thread is then scheduled on "return from interrupt"
- Handlers have full context, separate stack
  - So they can block now, but they usually don't
- Handling is often still divided into two parts, second part is performed by a lower-priority software interrupt thread
- Some interrupts that have to be very fast still run entirely in interrupt context (e.g. clock interrupt handler)

# Signals

- Software equivalent of hardware interrupts
- Allows process to respond to asynchronous external events
  - Process may specify its own *signal handlers* or may use OS *default action*
  - Defaults include
    - Ignoring the signal
    - Terminating all threads in the process (with or without a *core dump*)
    - Stopping all threads in the process
    - Resuming all threads in the process
- Provide a simple form of inter-process communication (IPC)

---

# Basics

---

- Process structure has flags for possible signals and actions to take
- When signal is posted to process, signal pending flag is marked
- When process is next scheduled to run, pending signals are checked and appropriate action is taken
  - Signal delivery is *not* instantaneous