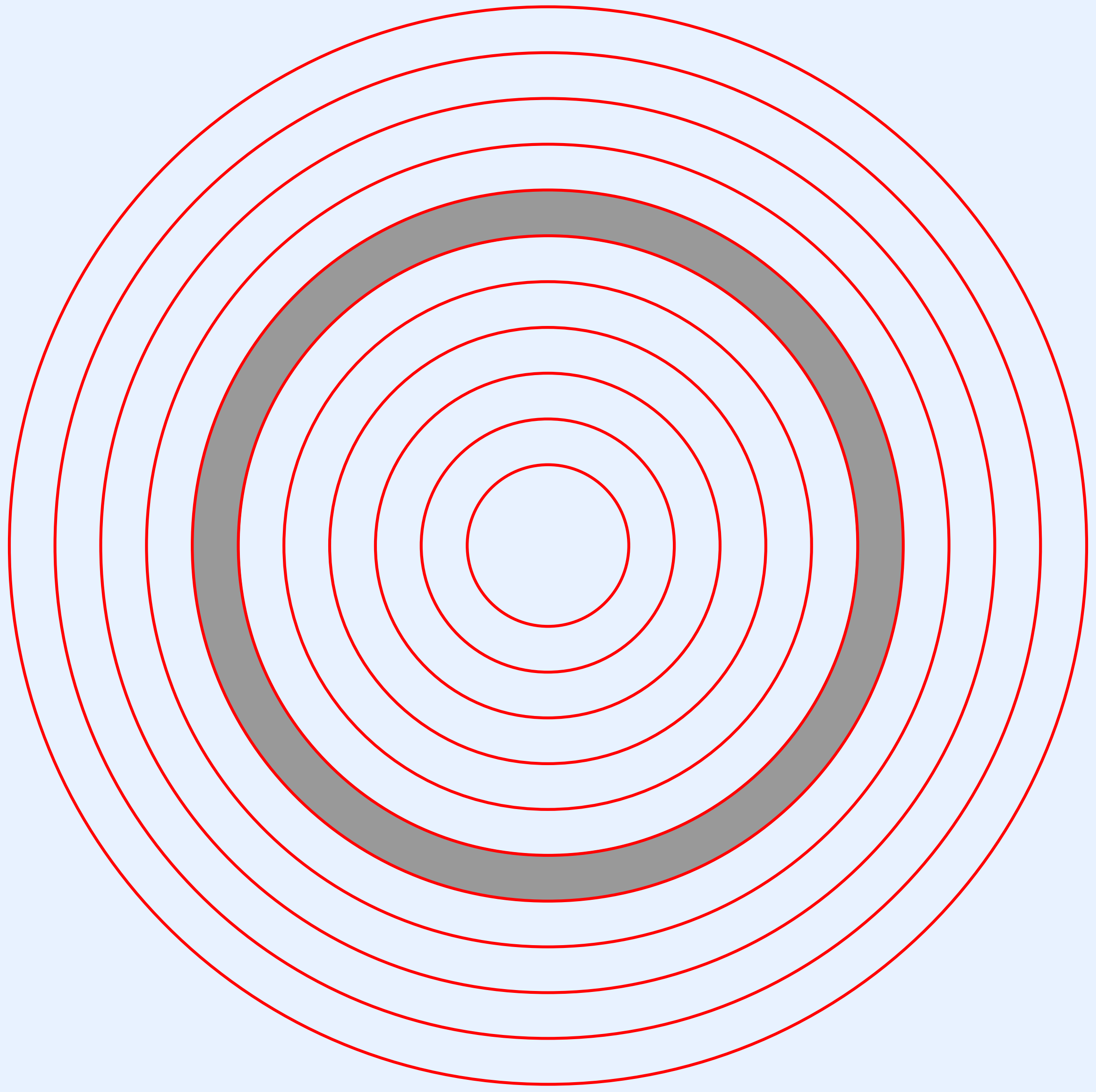
Module XII

Interrupt Processing

Location Of Interrupt Processing In The Hierarchy



Ancient History

- Each device had a unique hardware interface
- Code to communicate with device was built into applications
- An application polled the device; interrupts were not used
- Disadvantages
  - It was painful to create a program
  - A program could not use arbitrary devices (e.g., code for specific models of a printer and a disk were hard-wired into an application)
  - Upgrading a device meant rewriting applications!

The Modern Approach

- A device manager is part of an operating system
- The operating system presents applications with a uniform interface to all devices (as much as possible)
- All I / O is *interrupt-driven*

The Advantage Of Interrupts

An interrupt mechanism permits the processor and I/O devices to operate in parallel. Although the details differ, the hardware includes a mechanism that automatically ‘interrupts” normal processing and informs the operating system when a device completes an operation or needs attention.

A Device Manager In An Operating System

- Manages peripheral resources
- Hides low-level hardware details
- Provides an API that applications use
- Synchronizes processes and I / O

A Conceptual Note

One of the most intellectually difficult aspects of operating systems arises from the interaction between processes (an operating system abstraction) and devices (a hardware reality). Specifically, the connection between interrupts and scheduling can be tricky because an interrupt that occurs in one process can enable another.

Review Of I/O Using Interrupts

- The processor
  - Starts a device
  - Enables interrupts and continues with other computation
- The device
  - Performs the requested operation
  - Raises an interrupt on the bus
- Processor hardware
  - Checks for interrupts after each instruction is executed, and invokes an interrupt function if an interrupt is pending
  - Has a special instruction used to return from interrupt mode and resume normal processing

Processes And Interrupts

- Key ideas
  - Recall that at any time, a process is running
  - We think of an interrupt as a function call that occurs “between” two instructions
  - Processes are an operating system abstraction, not part of the hardware
  - An operating system cannot afford to switch context whenever an interrupt occurs
- Consequence:

The currently executing process executes interrupt code

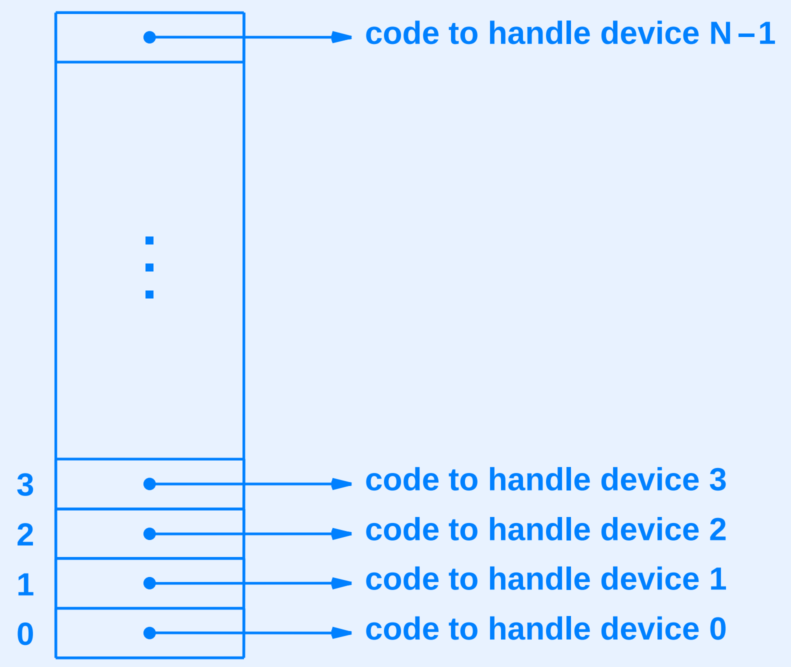
Historic Interrupt Software

- A separate interrupt function was created for each device
  - Very low-level code
  - Interrupt code must handle many details
    - Saves / restores registers
    - Sets the interrupt mask
    - Finds the interrupting device on the bus
    - Interacts with the device to transfer data
    - Resets the device for the next interrupt
    - Returns from the interrupt to normal processing

Vectored Interrupts

- Each device is assigned a unique integer known as an *Interrupt Request Number* (*IRQ*)
- The operating system stores an array of pointers in memory called an *interrupt vector*
- The device supplies its IRQ when interrupting the processor
- Interrupt hardware (or software) uses the IRQ as index into the interrupt vector array and jumps to the specified location

Illustration Of Interrupt Vectors



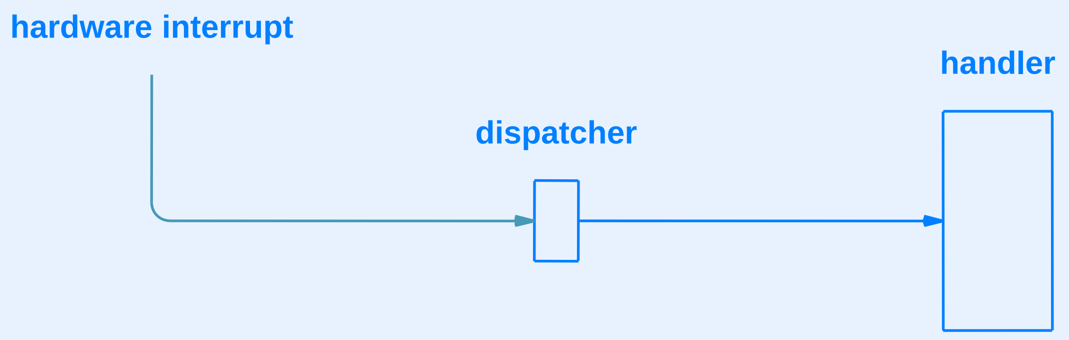
- When device *i* interrupts, the process executes code starting at location

*interrupt_vector[i]

Modern Interrupt Software (Two Pieces)

- An *interrupt dispatcher*
  - Is a single function common to all interrupts
  - Handles low-level details, such as finding the interrupting device on the bus
  - Sets up the environment needed for a function call and calls a device-specific function
  - Some functionality may be incorporated into an *interrupt controller chip*
- An *interrupt handler*
  - One handler for each device
  - Is invoked by the dispatcher
  - Performs all interaction with a specific device

The Conceptual Structure Of Interrupt Software



- Note: we will see that in practice, many hardware details complicate interrupt software

Interrupt Dispatcher

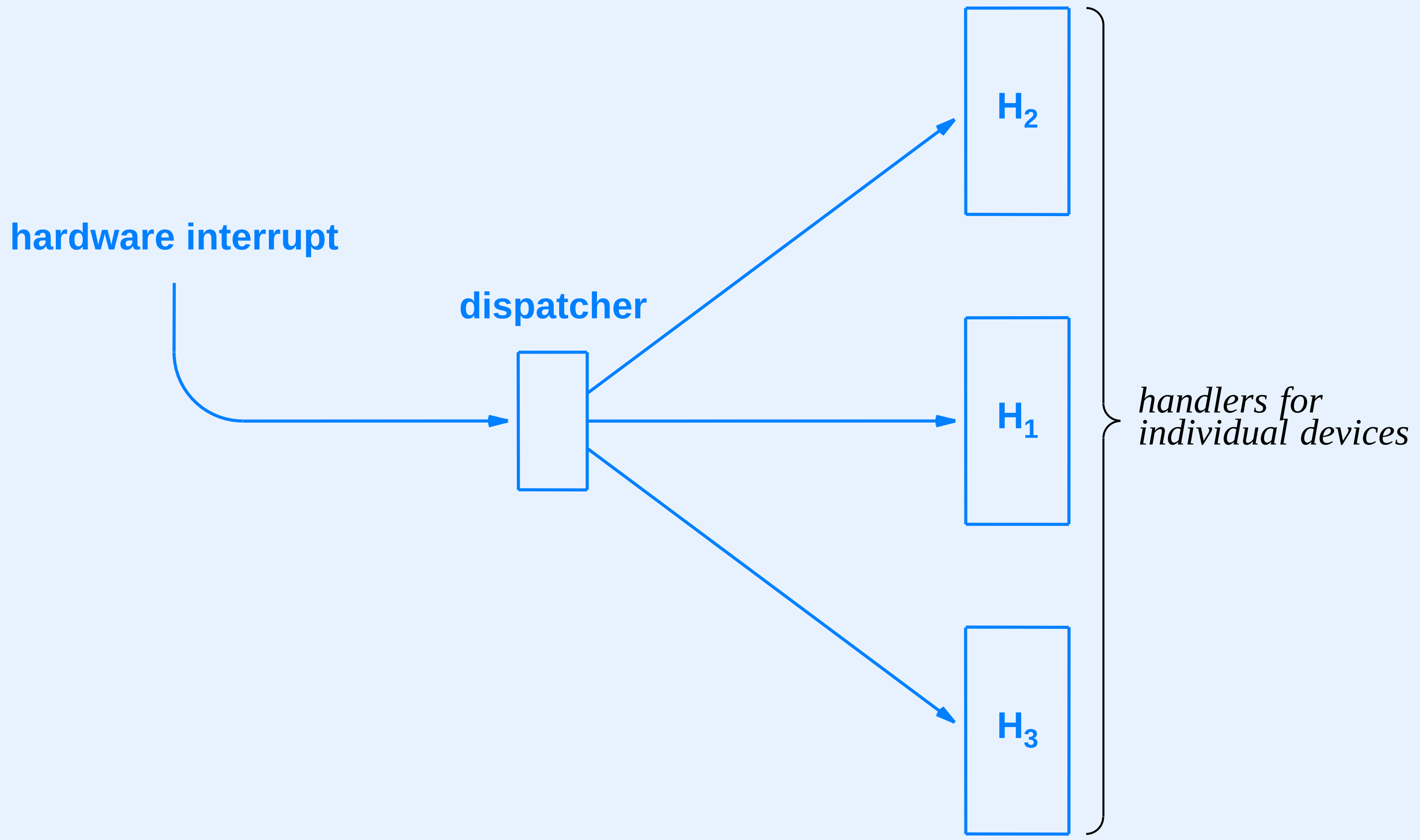
- A low-level piece of code written in assembly language
- Is invoked by the hardware when interrupt occurs
  - Runs in interrupt mode (i.e., with further interrupts disabled)
  - The hardware has saved the instruction pointer for a return
- The dispatcher
  - Saves other machine state as necessary
  - Identifies the interrupting device
  - Establishes the high-level runtime environment needed by a C function
  - Calls a device-specific *interrupt handler*, which is written in C

Return From Interrupt

- The interrupt handler
  - Communicates with the device
  - May restart the next operation on the device
  - Eventually returns to the interrupt dispatcher
- The interrupt dispatcher
  - Executes a special hardware instruction known as *return from interrupt*
- The *return from interrupt* instruction atomically
  - Resets the instruction pointer to the saved value
  - Enables interrupts

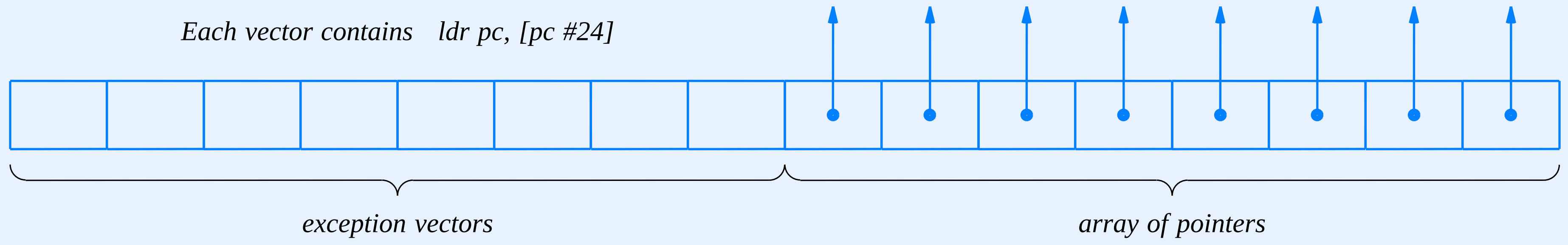
ARM Exception Vectors

- Each contain code that is executed, not a pointer to code
- Trick: use a parallel array of *indirect jump* instructions

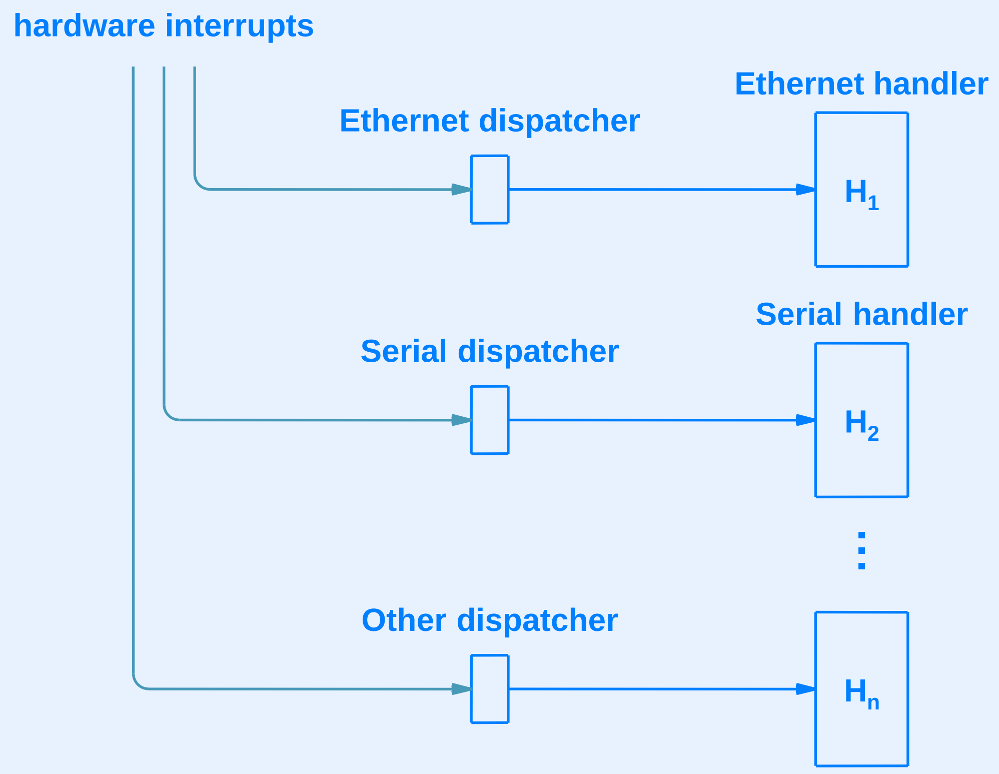


ARM Exception Vectors In Memory

- Restriction: an indirect jump uses a relative offset, which must be small
- To keep offsets small, Xinu places the two arrays in contiguous memory

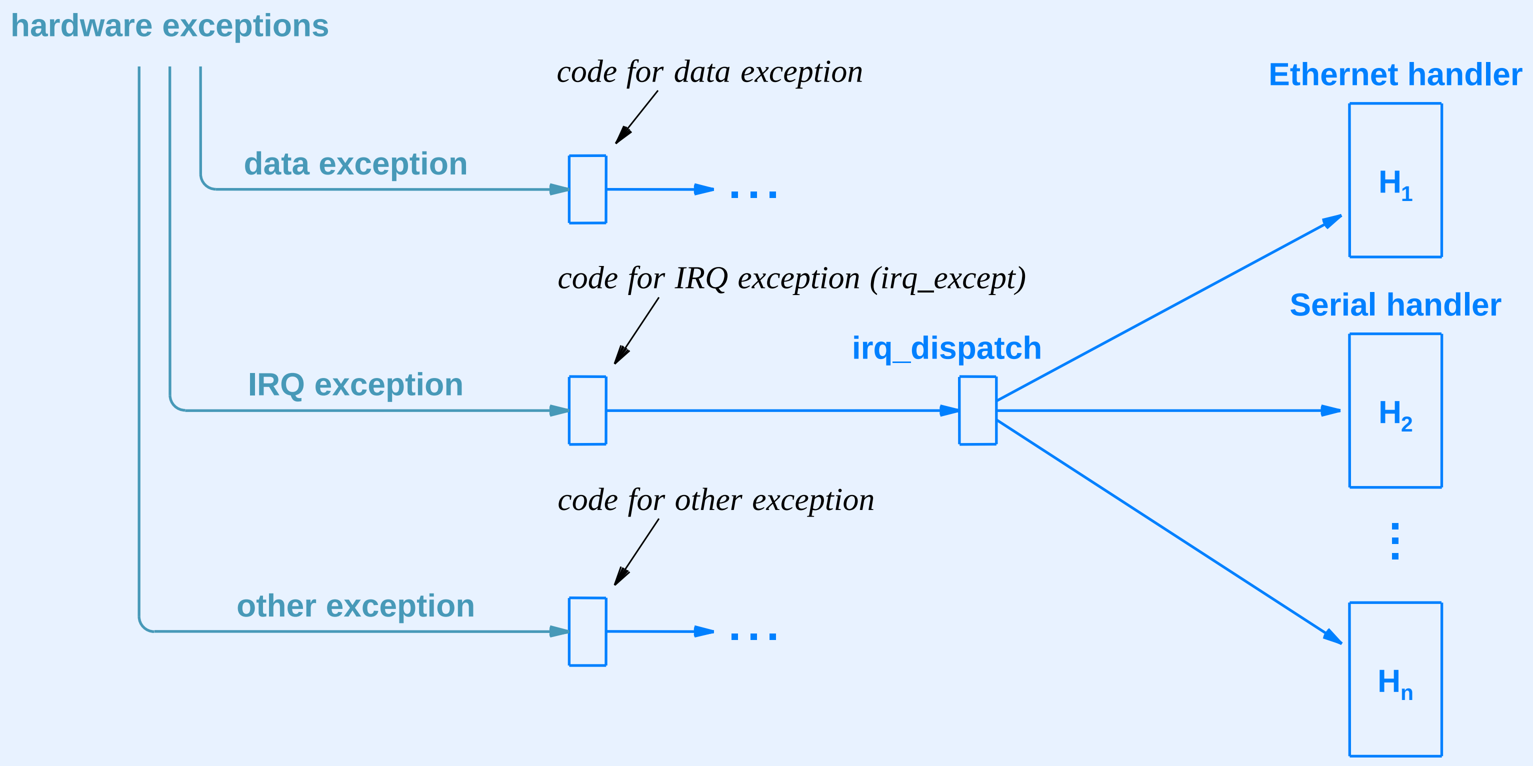


Interrupts On A Galileo (x86)



- The operating system preloads the *interrupt controller* unit with the address of a dispatcher for each type of device
- The controller hardware invokes the correct dispatcher

Interrupts On A BeagleBone Black (ARM)



- Uses a two-level scheme where the hardware raises an *IRQ exception* when a device interrupts
- The IRQ exception code invokes the IRQ dispatcher, which calls the correct handler

Integration Of Interrupts And Exceptions

Exceptions, such as divide-by-zero and page faults, follow a vectored approach. The Galileo illustrates interrupts and exceptions integrated into a single exception vector. *The BeagleBone Black illustrates a two-level scheme in which device interrupts correspond to one particular exception and the operating system must use a second level of indirection to reach the handler for a specific device.*

A Basic Rule For Interrupt Processing

- Facts
  - The hardware disables interrupts before invoking the interrupt dispatcher
  - Interrupts remain disabled when the dispatcher calls a device-specific interrupt handler
- Rule
  - To prevent interference, an interrupt handler must keep interrupts disabled until it finishes touching global data structures, ensures all data structures are in a consistent state, and returns from the interrupt
- Note: we will consider a more subtle version of the rule later

Interrupts And Processes

- When an interrupt occurs, I/O has completed
- Either
  - The device has received incoming data (an input interrupt occurs)
  - The device has finished sending outgoing data (an output interrupt occurs)
- A process may have been blocked waiting
  - To read the data that arrived
  - To write more outgoing data
- The blocked process may have a higher priority than the currently executing process
- The scheduling invariant *must* be upheld

The Scheduling Invariant

- Suppose process *X* is executing when an interrupt occurs
- We said that process *X* remains executing when the interrupt dispatcher is invoked and when the dispatcher calls a handler
- Suppose data has arrived and a higher-priority process, process *Y*, is waiting for the data
- If the hander merely returns from the interrupt, process *X* will continue to execute
- To maintain the scheduling invariant, the handler must call *resched*

Interrupts And The Null Process

- In the concurrent processing world
  - A process is always running
  - An interrupt can occur at any time
  - The currently executing process executes interrupt code
- An important consequence: the null process may be running when an interrupt occurs, which means the null process will execute the interrupt handler
- We know that the null process must always remain eligible to execute

A Restriction On Interrupt Handlers Imposed By The Null Process

Because an interrupt can occur while the null process is executing, an interrupt hander can only call functions that leave the executing process in the current or ready states. For example: an interrupt handler can call send **or** signal**, but cannot call** wait**.**

A Question About Scheduling And Interrupts

- Recall that
  - The hardware disables further interrupts before invoking a dispatcher
  - Interrupts remain disabled when the dispatcher calls a device-specific interrupt handler
- To remain safe
  - A device-specific interrupt handler must keep further interrupts disabled until it completes changes to global data structures
- What happens if an interrupt calls a function that calls *resched* and the new process has interrupts enabled?

An Example Of Rescheduling During Interrupt Processing

- As an example, suppose
  - An interrupt handler calls *signal*
  - *Signal* calls *resched*
  - *Resched* switches to a new process
  - The new process executes with interrupts enabled
- Will interrupts pile up indefinitely?

The Answer

- No, interrupts will not pile up indefinitely
- Reason:
  - Interrupt status is associated with each *process*, not with the hardware
  - After switching to a process that has interrupts enabled, that process can be interrupted
  - In the worst case, all processes can end up executing interrupt code with further interrupts disabled
  - If another context switch occurs, it will be to a process that has interrupts disabled, and the system must return from the interrupt to have interrupts enabled again

The Answer
(continued)

- As an example, let *T* be the current process
- When an interrupt occurs, *T* executes an interrupt handler with interrupts disabled
- If the handler that *T* is executing calls *signal*
  - *Signal* may call *resched*
  - A context switch may occur and process *S* may run
  - *S* may run with interrupts enabled
  - If a second interrupt occurs, *S* may execute an interrupt handler with interrupts disabled
- Only NPROC interrupts can occur before all processes are running with interrupts disabled

The Principle

Rescheduling during interrupt processing is safe provided that each interrupt handler leaves global data in a valid state before rescheduling and no function enables interrupts unless it previously disabled them (i.e., disable / restore **is used instead of** enable**).**

Summary

- Interrupts allow processors and devices to operate simultaneously
- Vectored interrupts make passing control to the correct handler efficient
- An interrupt handler must leave the process executing the interrupt in the current or ready states
- To preserve the scheduling invariant, an interrupt handler must reschedule whenever it makes a waiting process ready
- Rescheduling is allowed provided that the global data is in a valid state and no function enables interrupts unless it disabled them
- Most processors use the hardware interrupt mechanism for exceptions (e.g., divide by zero)