Device management

CS503: Operating systems, Spring 2019

Pedro Fonseca
Department of Computer Science
Purdue University

Edited by Byoungyoung Lee and Pedro Fonseca

Previous lecture

- Paging
  - Processor modes
  - Virtual memory mechanisms:
    - Paging
    - Segmentation
  - Page tables, multi-level paging, PDEs/PTEs, etc.
  - TLB
  - Memory protection, memory layout with virtual memory

Previous lecture

- On demand paging:
  - Lazy loading of executables
  - Simulating/virtualizing more memory than the available physical RAM
- Intercepting page accesses using the valid bit
- Trapping on the page faults

Ancient history

- Device manager is part of OS (typically called device drivers)
- OS presents applications with uniform interface to all devices (as much as possible)
- IO is typically interrupt driven

Device manager in an operating system

- Manages peripheral resources
- Hides low-level hardware details
- Provides API to applications
- Synchronizes processes and IO

Review of hardware interrupts

- Processor:
    - Starts a device
    - Enables interrupts
- Device
    - Performs the requested operation
    - Raises an interrupt on bus
- Processor hardware
    - Checks for interrupts after each instruction is executed
    - Invokes an interrupt function, if an interrupt is pending
    - Provides a mechanism for return

Processes and interrupts

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

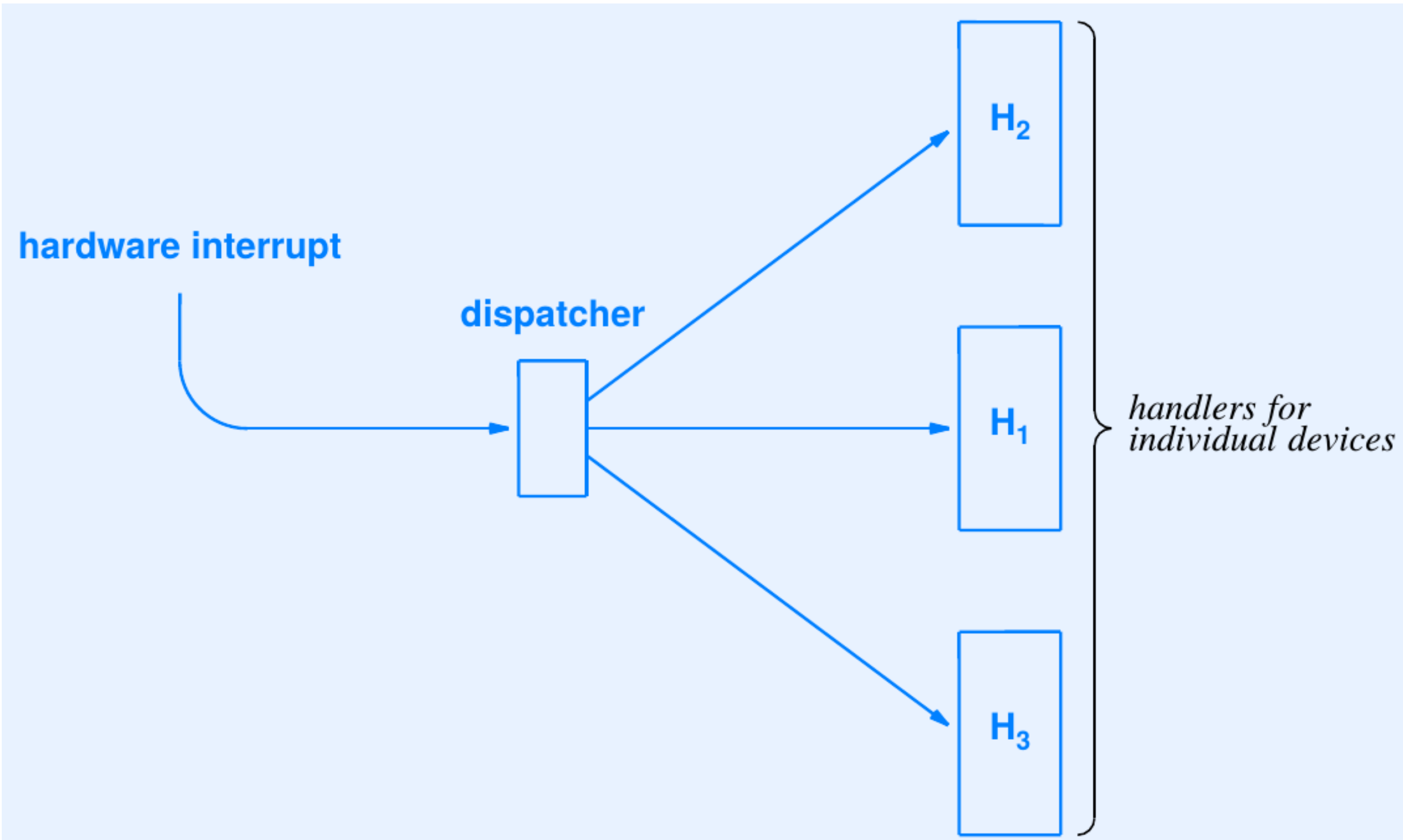
Interrupt software (two pieces)

- Interrupt dispatcher:
  - Single function common to all interrupts
  - Finds interrupting device on the bus
  - Calls a device-specific function
- Interrupt handler:
  - Separate code for each device
  - Invoked by the dispatcher
  - Performs all interaction with device

Interrupt dispatcher

- Low-level function
- Invoked by hardware when interrupt occurs
  - CPU has saved the instruction pointer (and a flag register)
- Dispatcher
  - Saves other machine state as necessary
  - Identifies interrupting device
  - Calls a device-specific function

Conceptual view of interrupt dispatching



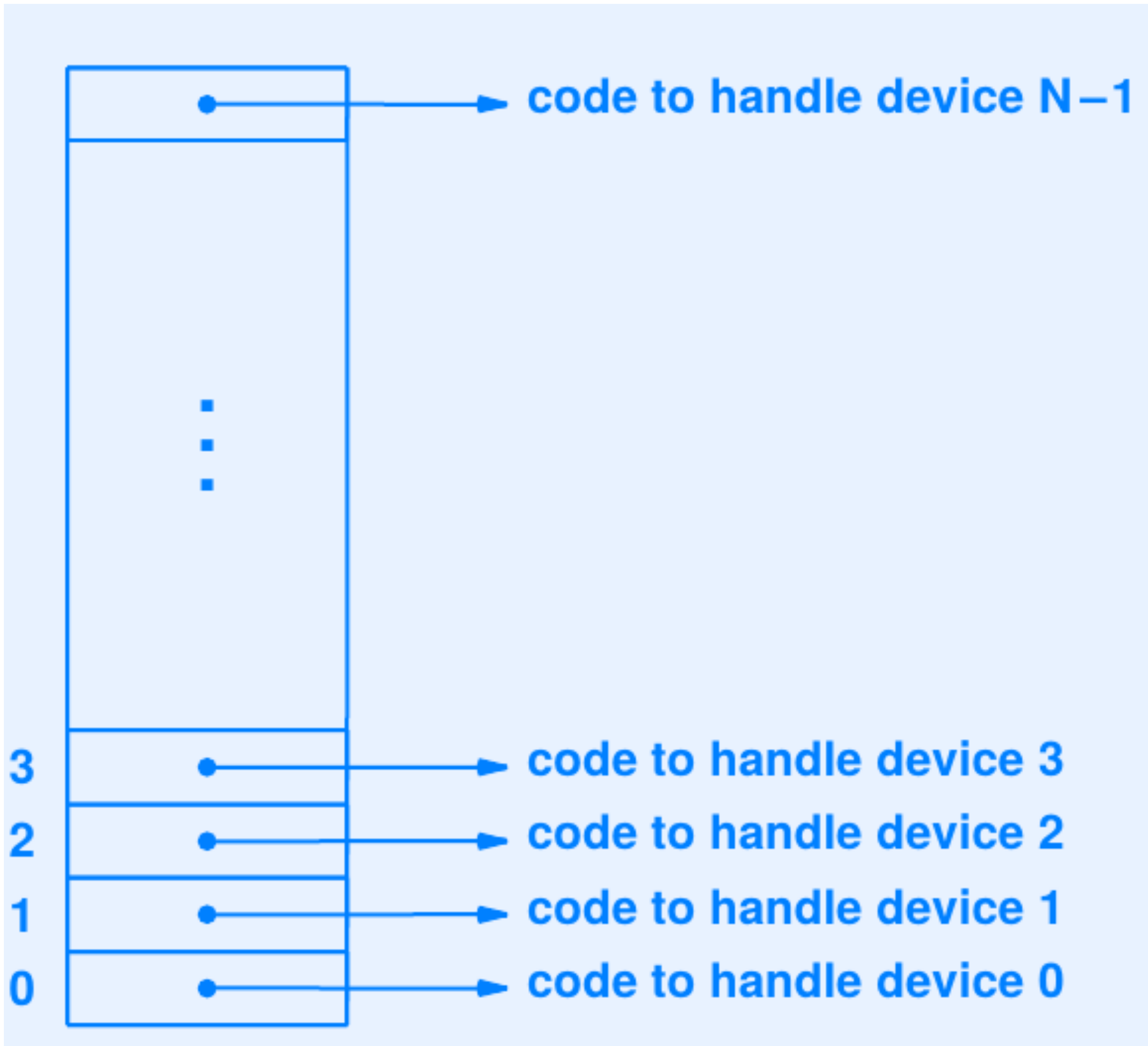
Return from an interrupt

- Interrupt dispatcher
  - Executes special hardware instruction known as return from interrupt:
    - e.g., **iret** in x86
- Interrupt handler
  - Communicates with device (sending and receiving data)
  - Eventually returns to interrupt dispatcher
- Return from interrupt instruction atomically
  - resets instruction pointer to save value
  - Enables interrupts

Interrupt mechanism: a vector

- Each possible interrupt is assigned a unique IRQ
- Hardware uses IRQ as an index into an interrupt vector array
  - See `set_evec()` how Xinu initializes the vector

Conceptual organization of the interrupt vector



Basic rules for interrupt processing

- **1. Data consistency:**
  - Need synchronization mechanism to ensure data consistency
  - Sharing data between:
    - Interrupt handler and device drivers
    - Interrupt handler and kernel threads
  - Possible synchronization mechanisms in the interrupt handler
    - Disable interrupt
    - Considering SMP makes the synchronization problem complicated
    - Q: How about using:
      - Spin lock
      - Semaphores / mutex?

Basic rules for interrupt processing

- 2. Speed: interrupt processing has to be **quick**
  - Linux: top and bottom halves, where top-half quickly handles the interrupt and the bottom-half processes heavy long tasks

Interrupts and processes

- When an interrupt occurs, I/O has completed
- Either:
  - Data has arrived
  - Space has become available in an output buffer
- A process may have been blocked waiting
  - To read data
  - To write data
- The blocked process may have a higher priority than the currently executing process
- The scheduling invariant needs to be upheld

A question about scheduling

- Suppose process X is executing when an interrupt occurs
- 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 Y is waiting for the data
- If the handler merely returns from an interrupt, process X will continue to execute
- Should the interrupt handler call resched()
- If not, how is the scheduling variant established?

Possible solutions

- An OS may:
  - Arrange for the dispatcher to reestablish the scheduling invariant just before returning from the interrupt
  - Postpone rescheduling until a later time (e.g., when the current process's time-splice expires)
- Any of the above works

Interrupts and the null process

- In the concurrent processing world:
  - A process is always running
  - Interrupts can occur asynchronously
  - The currently executing process executes interrupt code
- An important consequence
  - The null process may be running when an interrupt occurs
  - If interrupted, the null process will execute the interrupt handler
- Keep in mind: the null process must always remain eligible to run

A restriction 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

Scheduling and interrupts

- Recall that interrupts are disabled when dispatcher calls 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
  - Q: Would acquire locks also work in this situation?

Rescheduling during interrupt processing

- Suppose:
  - Interrupt handler calls signal
  - Signal calls resched()
  - Resched() switches to a new process
  - The new process executes the interrupts enabled
- Q: will interrupts pile up indefinitely?

An example

- Let T be the current process
- When interrupt occurs, T executes an interrupt handler
- The interrupt handler calls signal
- Signal calls resched
- A context switch occurs and process S runs
- S may run with interrupts enabled

The answer

- 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

Device driver in Xinu

- Set of functions that perform IO on a given device
- Contains device-specific code
- Includes functions used to read or write data and control the device as well as interrupt handler code
- Code divided into two parts

Two parts of a device driver

- Upper half:
  - Functions executed by applications
  - Used to request IO
  - May copy data between user and kernel address space
- Lower half:
  - Device specific interrupt handler
  - Invoked by interrupt when operation completes
  - Executed by whatever process is executing
  - May restart the device for next operation

Devision of duties in a driver

- Upper half:
  - Minimal interaction with device hardware
  - Enqueues request
  - Starts devices if idle
- Lower half
  - Minimal interaction with application
  - Talks to the device
    - Obtains incoming data

Coordination of processes performing IO

- Processes may need to block when attempting to perform IO
- Example:
  - Application waits for incoming data to arrive
  - Other examples?
- How should coordination be done?

Retrofitting process coordination mechanism

- Answer: Retrofitting process coordination mechanism
  - Message passing
  - Semaphores

Using semaphores for input synchronization

- A shared buffer is created with N slots
- A semaphore is created with initial count 0
- Upper-half
  - Calls `wait` on the semaphore
  - Extracts the next item from buffer and returns
  - Can restart the device if the device is idle
- Lower-half
  - Places the incoming item in the buffer
  - Calls `signal` on the semaphore
- Note: the semaphore counts the # of items in the buffer

More device management topics

- API for devices
- Device independent IO
- Xinu primitives
- Driver examples
- Internal communication