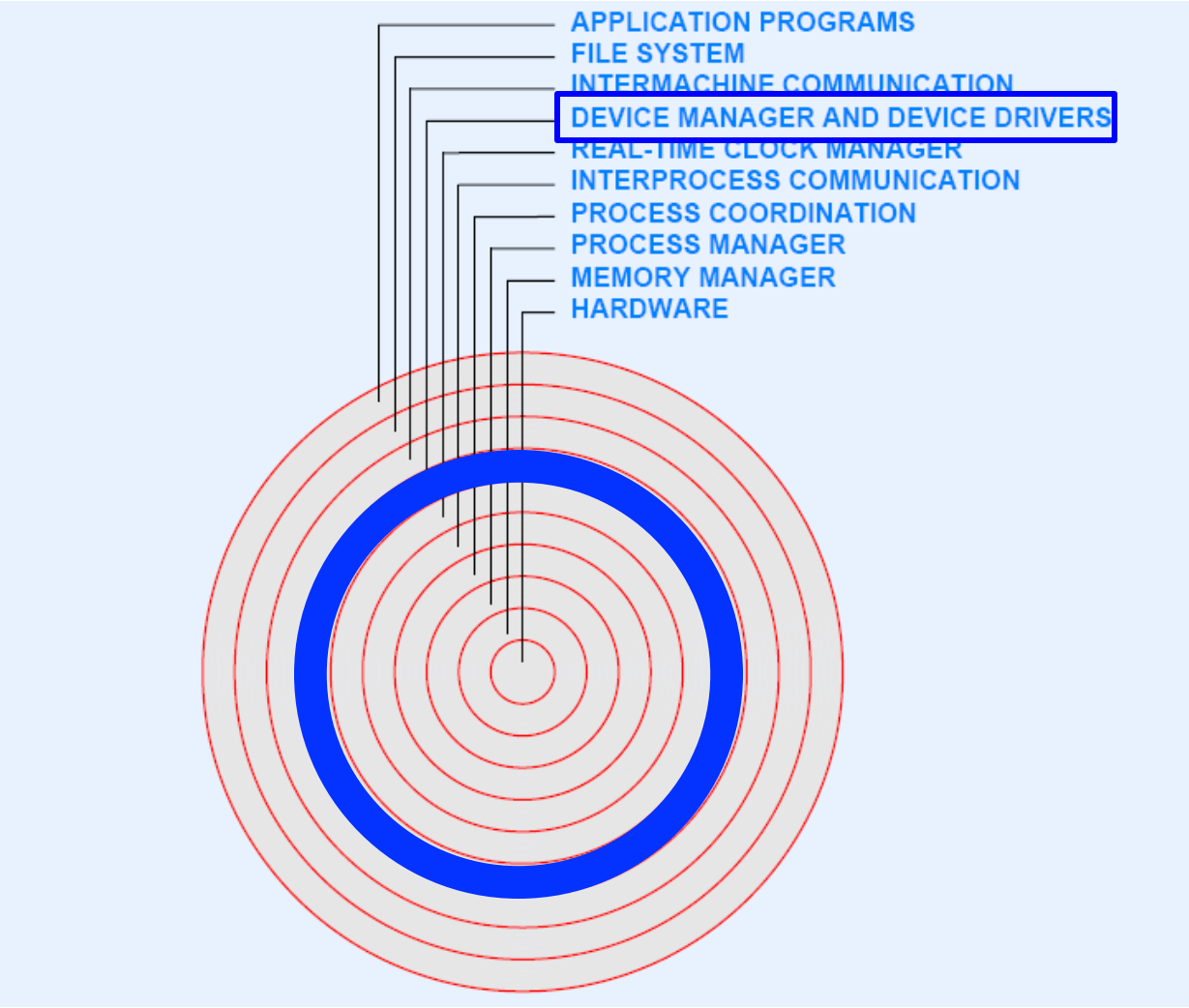
CSCI 8530
Advanced Operating Systems

Part 9

Device Management

Location of Device Management in the Hierarchy

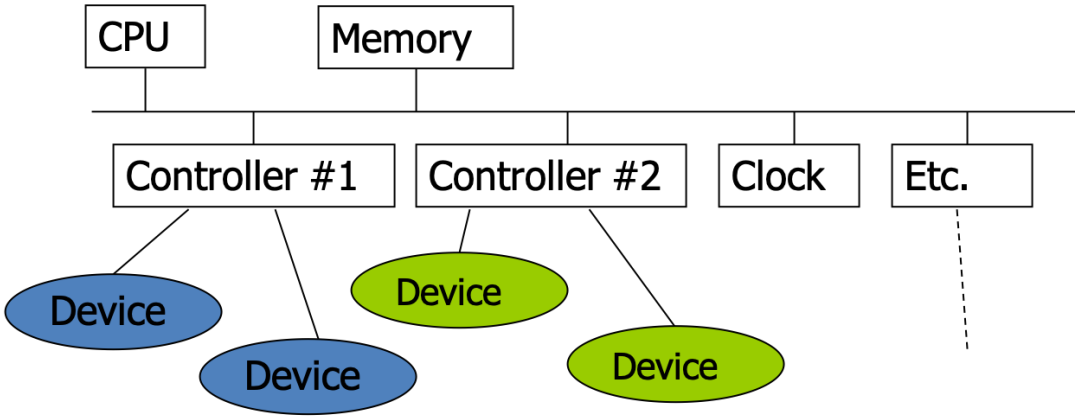


Ancient History

- Each device had unique hardware interface
- Code to communicate with device was part of application
- Application polled the device; interrupts were not used
- Disadvantages
  - Generality and portability were limited
  - Painful to program

Modern Approach

- Device manager is part of OS
- OS presents applications with uniform interface to all devices (as much as possible)
- All I /O is *interrupt-driven*
- Device Manager in an Operating System
  - Manages peripheral resources
  - Hides low-level hardware details
  - Provides API to applications
  - Synchronizes processes and I /O



A Conceptual Note

One of the most intellectually difficult aspects of operating systems arises from the interface between processes (an OS 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 Hardware Interrupts

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

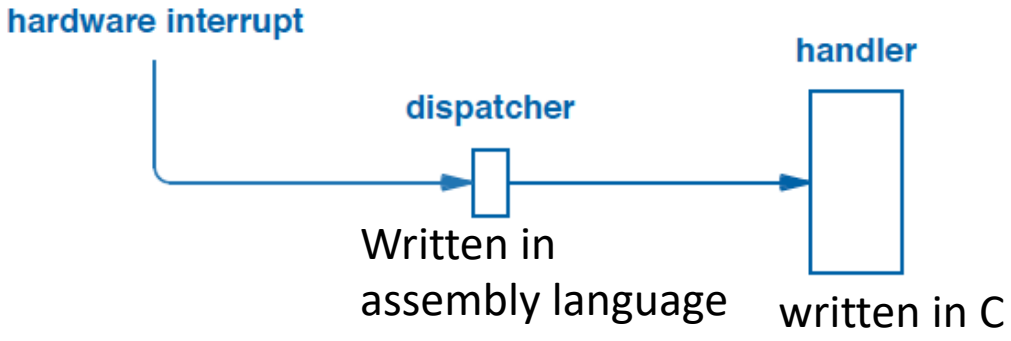
The current process executes the interrupt code.

Historic Interrupt Software

- Separate interrupt function for each device
  - Low-level
  - Handles housekeeping details
    - Saves / restores registers
    - Sets interrupt mask
  - Finds interrupting device on the bus
  - Interacts with the device to transfer data
  - Resets the device for the next interrupt
  - Returns from interrupt

Modern Interrupt Software (Two Pieces)

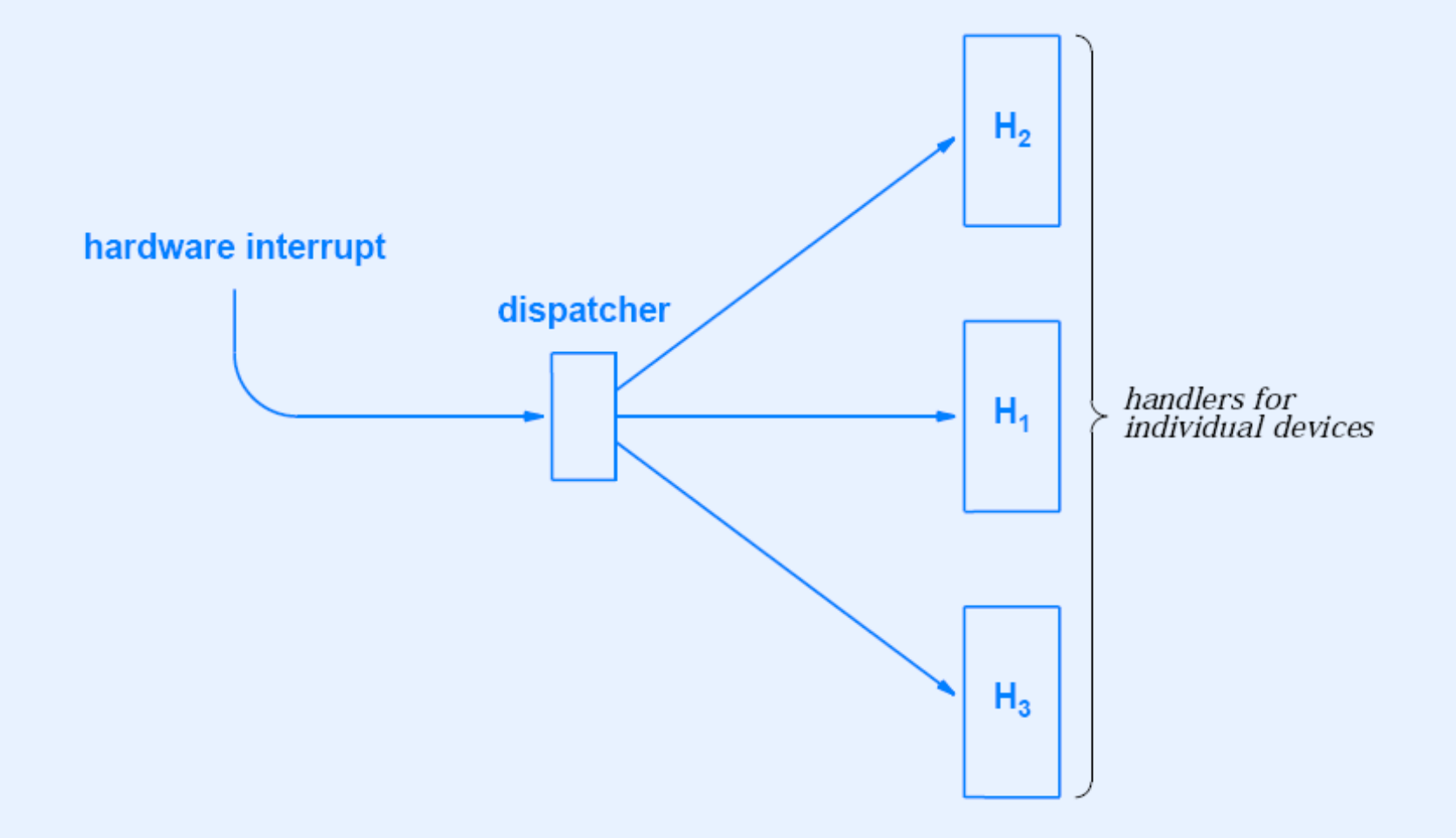
- Lower-level *Interrupt dispatcher*
  - Single function common to all interrupts
  - Performs housekeeping details
  - Finds interrupting device on the bus
  - Calls a device-specific function
- High-lever *Interrupt handler*
  - Separate code for each device
  - Invoked by the dispatcher
  - Performs all interaction with device



Interrupt Dispatcher

- Low-level function
- Invoked by hardware when interrupt occurs
  - Is invoked with correct mode (i.e., disables interrupts)
  - Hardware has saved the instruction pointer
- Dispatcher
  - Saves other machine state as necessary
  - Identifies interrupting device
  - Establishes high-level runtime environment (needed for a C function)
  - Calls a device-specific *interrupt handler*

Conceptual View of Interrupt Dispatching



- Note: the dispatcher is typically written in assembly language.

Recap: Return from Interrupt

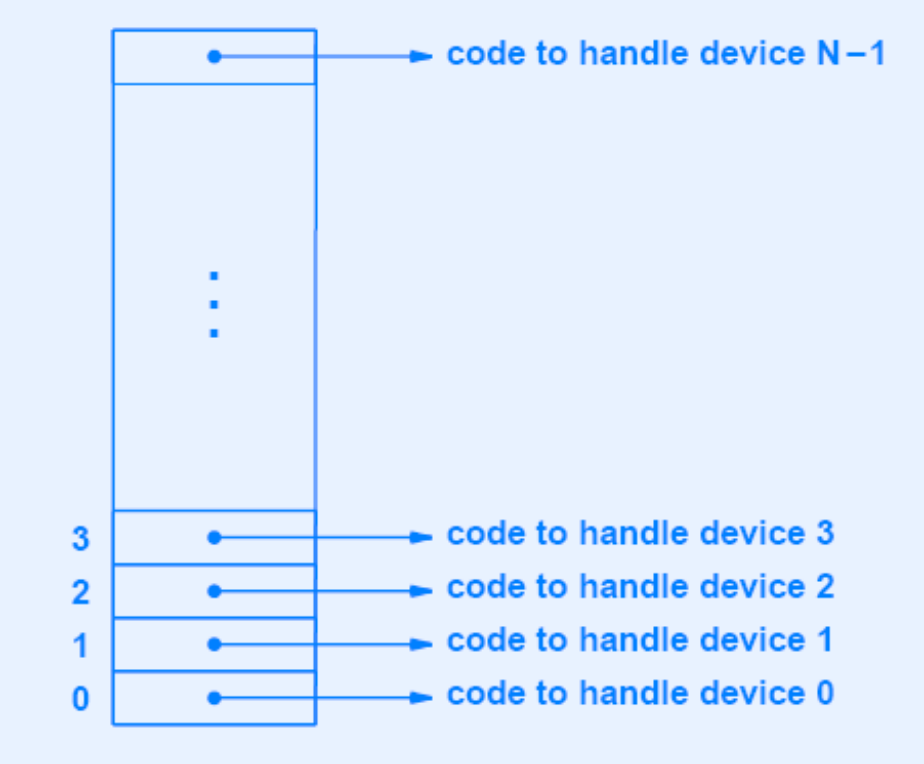
- Handler
  - Communicates with device
  - May restart next operation
  - Eventually returns to interrupt dispatcher
- Interrupt dispatcher
  - Executes special hardware instruction known as *return from interrupt*
- Return from interrupt instruction atomically
  - Resets instruction pointer to saved value
  - Enables interrupts

An Example: the Clock Dispatcher

```
/* clkdisp.s - clkdisp (x86) */

/*-------------------------------------------------------------------------
 * clkdisp  -  Interrupt dispatcher for clock interrupts (x86 version)
 *-------------------------------------------------------------------------
 */
#include <icu.s>
                .text
                .globl  clkdisp         # Clock interrupt dispatcher
clkdisp:
                pushal                  # Save registers
                cli                     # Disable further interrupts
                movb    $EOI,%al        # Reset interrupt
                outb    %al,$OCW1_2

                call    clkhandler      # Call high level handler

                sti                     # Restore interrupt status
                popal                   # Restore registers
                iret                    # Return from interrupt
```

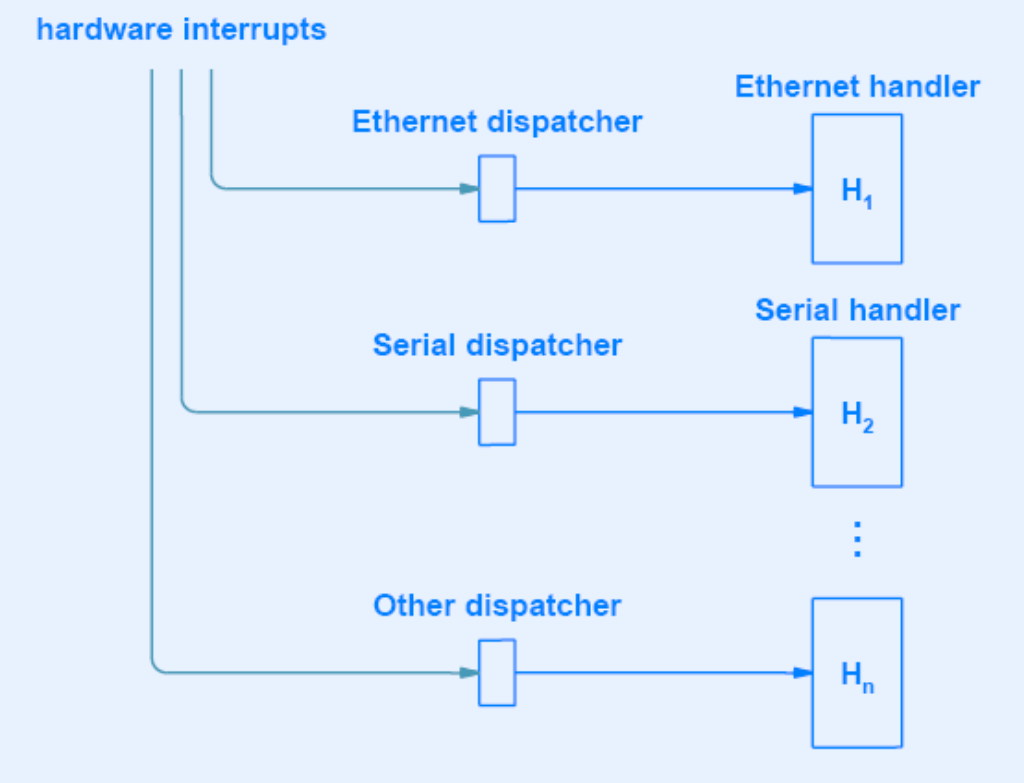
Recap: Interrupt Mechanism - A Vector

- Each possible interrupt is assigned a unique Interrupt Request Number (IRQ)
- Hardware uses IRQ as an index into an *interrupt vector* array
- When a device with IRQ i interrupts, control branches to:
  - Interrupt vectors as an array of pointers
    - interrupt_vector[i]



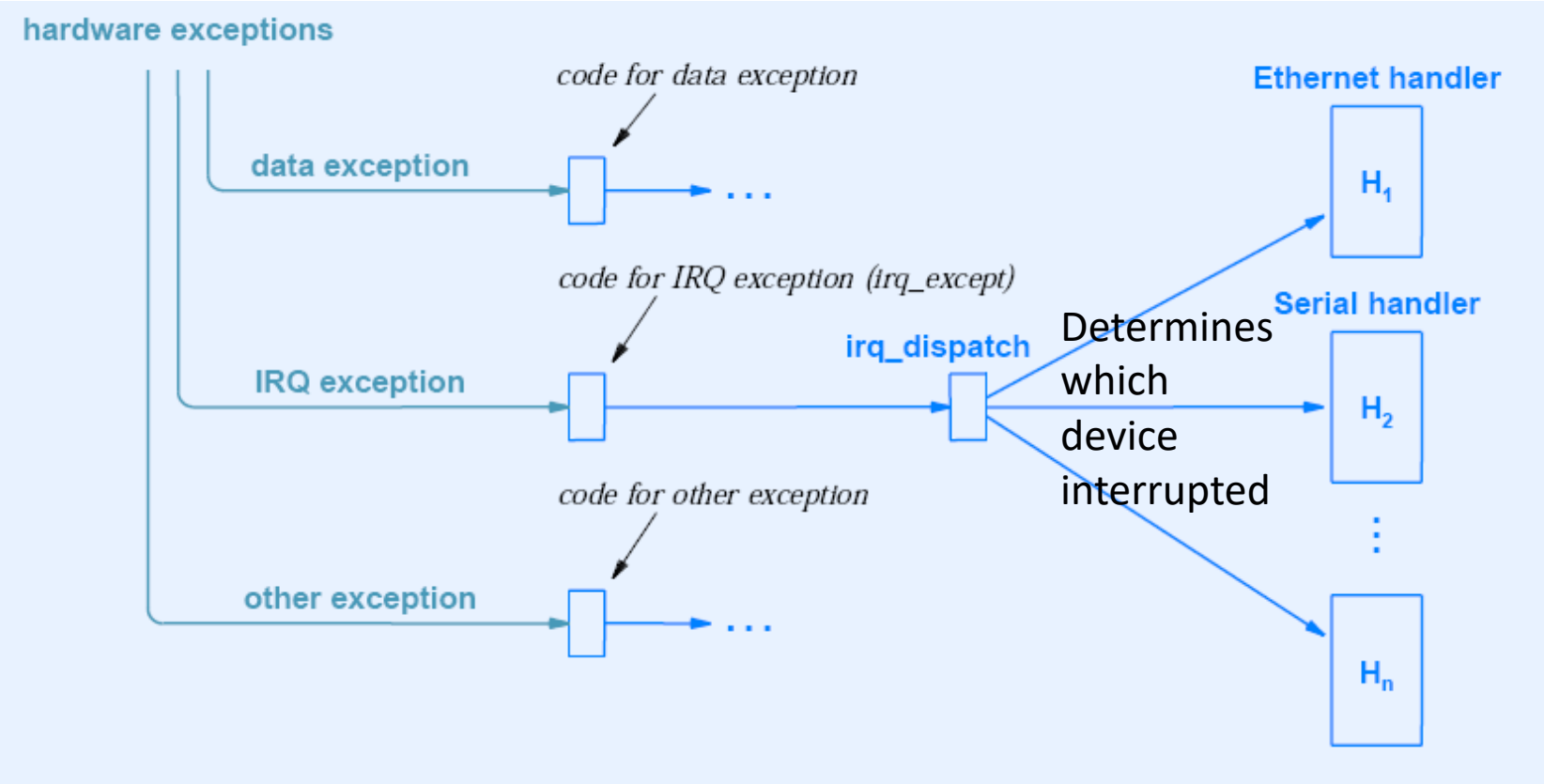
Interrupt Code on a Galileo (x86)

- Operating system loads the interrupt controller with the address of a dispatcher for each device
- Hardware: Controller invokes the correct dispatcher (save time)
- Each driver has both a dispatch function and a handler function



Ethernet device interrupts

Interrupt Code on a BeagleBone Black (ARM)



- Two-level scheme where controller uses *IRQ exception* for all device interrupts
- The hardware maps the interrupt from any device into an IRQ exception, jumps to the interrupt vector
- Exception code invokes the IRQ dispatcher

A Basic Rule for Interrupt Processing

- Facts
  - The processor disables interrupts before invoking the interrupt dispatcher
  - Interrupts remain disabled when the dispatcher calls a device-specific interrupt handler
  - But… How long?
- 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

A Question About Scheduling during an Interrupt

- Recall
  - The scheduling invariant specifies that at any time, a highest priority eligible process must be executing.
  - When an I/O operation completes, a high-priority process may become eligible to execute.
- Suppose process *X* is executing when an interrupt occurs
  - Process *X* remains executing (that is, current) 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 hander merely returns from an interrupt, process *X* will continue to execute (remain current).

  - Should the interrupt handler call *resched*?
  - If not, how is the scheduling invariant established?

Possible Answer

- An OS may
  - Have an interrupt handler reestablish the scheduling invariant.
  - 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-slice expires).
- Placing a check in the dispatcher incurs more overhead.

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* (an infinite loop that does not make function calls) 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 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*.

DEVICE DRIVER ORGANIZATION

Device Driver

- Set of functions that perform I /O 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 conceptual parts
  - an Upper half
  - a Lower half

Two Conceptual Parts of a Device Driver

- Upper half
  - Functions executed by an application
  - Used to request I /O
  - May copy data between user and kernel address spaces
- 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

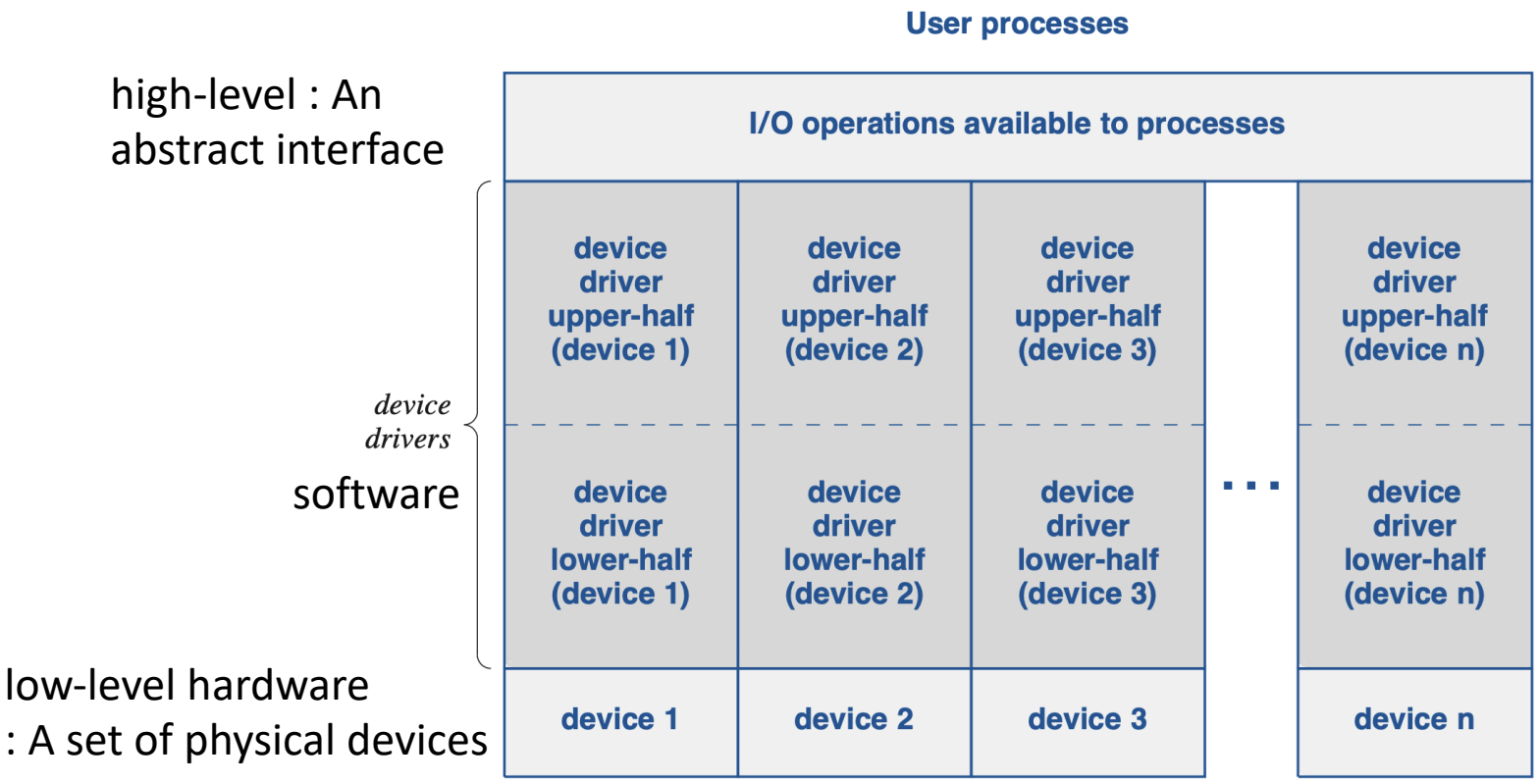
Division of Duties in a Driver

- Upper-half
  - Minimal interaction with device hardware
  - Enqueues request: applications specify
  - Starts device if idle
- Lower-half
  - Minimal interaction with application
  - Talks to the device
    - Obtains incoming data
    - Starts next operation
  - Awakens a process (if any waiting)

Why partitioning a driver?

When creating a device driver, a programmer must be careful to preserve the division between upper-half and lower-half functions because upper-half functions are called by application processes and lower-half functions are invoked by interrupts.

Conceptual Organization of Device Software



Coordination of Processes Performing I/O

- Process may need to block when attempting to perform I/O
  - A *synchronous* interface
  - An *asynchronous* I/O interface

> *When using a synchronous I/O interface, a process is blocked until the operation completes. When using an asynchronous I/O interface, a process continues to execute and is notified when the operation completes.*

- Examples
  - Application waits for incoming data to arrive
  - Application blocks until the device is ready to send outgoing data
- How should coordination be performed?

Answer

- No need to invent new mechanisms; standard process coordination mechanisms suffice:
  - Message passing
  - Semaphores
  - Suspend /resume
- However: a programmer must exercise caution because a lower-half function must *not* block the calling process.

Using Message Passing for Input Synchronization

- During input operation (*read*)
  - Upper-half
    - Starts the device.
    - Places the current process ID in a data structure associated with device.
    - Calls *receive* to block.
  - Lower-half
    - Is invoked during the interrupt when input completes.
    - Calls *send* to send a message to the blocked process.

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 items in the buffer.

Semaphores and *Output* Synchronization

- A flawed approach
  - A semaphore counts items in the output buffer.
  - The upper-half deposits an item and calls *signal.*
  - The lower-half calls *wait* to block until an item is present.
- Why is the above flawed?

  Lower-half functions cannot execute *wait*!

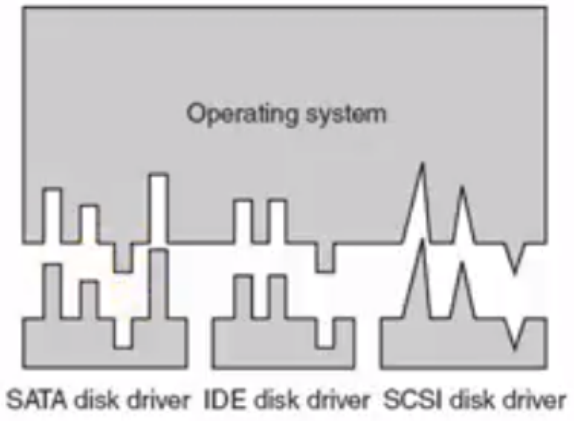
Using Semaphores for Output Synchronization

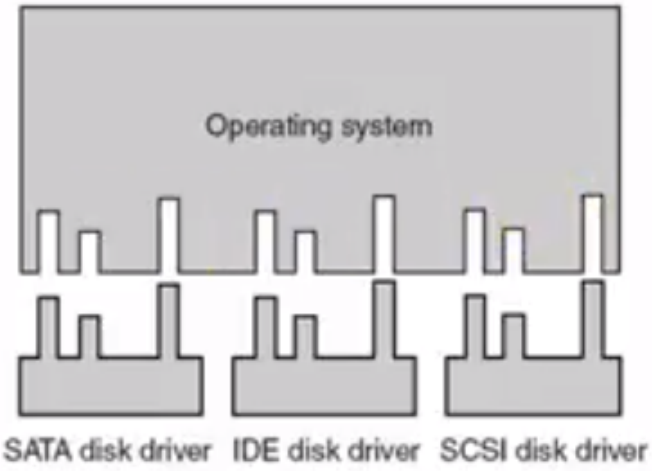
- Trick: semaphore count interpreted as “space available”
  - Initialized to buffer size.
- Upper-half
  - Calls *wait* on the semaphore.
  - Deposits a data item in the buffer.
  - Starts the device, if necessary.
- Lower-half
  - Is invoked when an output operation completes.
  - Starts next output operation, if the buffer is not empty.
  - Signals the semaphore to indicate that space is available.

DEVICE-INDEPENDENT I/O

Application Interface to Devices

- Desires
  - Portability across machines
  - Generality sufficient for all devices
  - Elegance (minimalism rather than a potpourri of functions)
- Solution
  - Isolate application processes from device drivers
  - Use a common paradigm across all devices
  - Integrate with other operating system facilities





(b) With a standard driver interface

Approach for Device-Independent I/O

- Define a set of abstract operations
- Build a general-purpose mechanism
  - Use generic operations (e.g., *read*).
  - Include parameters to specify device instance.
  - Arrange an efficient way to map generic operations onto the code for a specific device.
- Notes
  - The set of generic operations form an abstract data type.
  - The upper-half of each driver must define functions that apply each generic operation to the device.

Device-Independent I/O Primitives in Xinu

Operation	Purpose
close	Terminate use of a device
control	Perform operations other than data transfer
getc	Input a single byte of data
init	Initialize the device at system startup
open	Prepare the device for use
putc	Output a single byte of data
read	Input multiple bytes of data
seek	Move to specific data (usually a disk)
write	Output multiple bytes of data

- Note: some abstract functions may not apply to a particular device

Implementation of Device-Independent I/O in Xinu

- Application process
  - Makes calls to device-independent functions (e.g., *read*).
  - Supplies the device ID as parameter (e.g., ETHER).
- OS
  - Maps the device ID to an actual device.
  - Invokes appropriate device-specific function (e.g., *ethread* to read from an Ethernet).

Mapping Generic I/O Function to a Device-Specific Function

- Mapping must be efficient
- Is performed with a *device switch table*
  - Kernel data structure initialized when system loaded
  - One row per device
  - One column per operation
  - Each entry in the table points to a function
- The device ID is used as an index into the table

Semantics of Device-Independent I/O

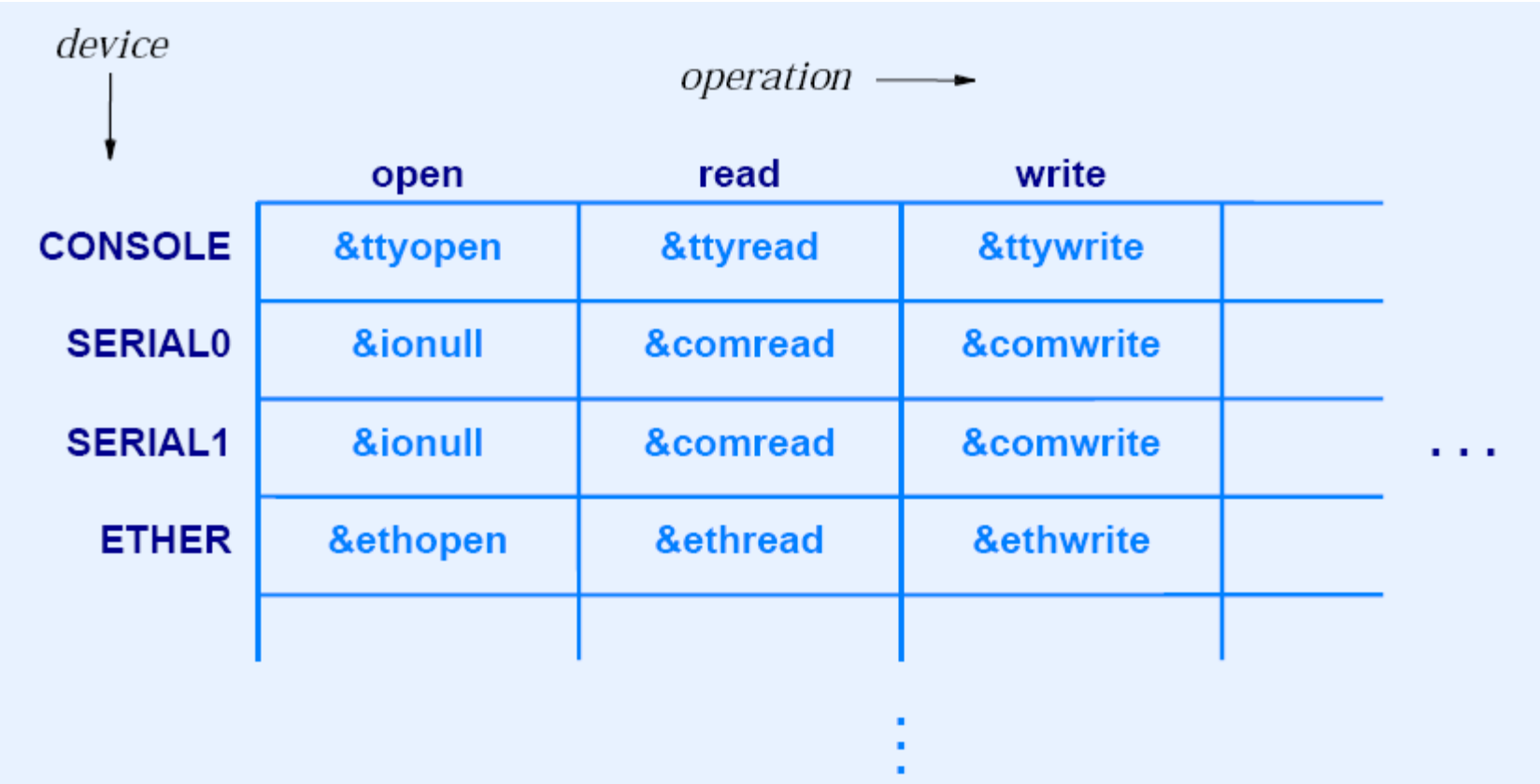
- Each device-independent operation is generic
- An operation may not make sense for a given device
  - *Seek* on a keyboard, network, or display screen
  - *Close* on a mouse
- However...

  all entries in device switch table must be valid
- Our solution: create functions that can be used to fill in meaningless entries

Special Entries Using in the Device Switch Table

- *ionull*
  - Used for innocuous operation
  - Returns *OK*
- *ioerr*
  - Used for incorrect operation
  - Returns *SYSERR*

Illustration of Device Switch Table



- Each row corresponds to a device
- Each column corresponds to an operation
- Each entry specifies the address of a function to invoke

Replicated Devices

- Computer may contain multiple instances of a physical device
- Example: two serial lines or two Ethernet NICs
- Goal
  - Have a single copy of the device driver functions
  - Allow a function to be used with any copy of the device

Parameterized Device Drivers

- A driver must
  - Know which physical instance of a device to use
  - Keep the information for one instance separate from the information for other instances
- Technique
  - Assign each instance of a replicated device a unique minor number (e.g., 0, 1, 2, ...) known as a *minor device number*
  - Store the minor device number in the device switch table (each column includes a minor device number; see config/conf.c for example)

Device Names in Xinu (1/2)

- Previous examples have shown device names used in code
  - CONSOLE
  - SERIAL0
  - SERIAL1
  - ETHER
- The device switch table is an array, and each device name is an index
- How are unique values assigned to names, such as *CONSOLE*?
  - Answer: OS designer specifies names during configuration

Device Names in Xinu (2/2)

- The configuration program assigns each device name a unique integer value known as a *device descriptor*.
- The configuration program produces a header file that contains *#define* statements for each name.
  - If CONSOLE has been assigned descriptor zero, the call:

```
read(CONSOLE, buf, 100);
read(0, buf, 100);
```

Xinu uses a static binding for device names. Each device name is bound to an integer descriptor at configuration time before the operating system is compiled.

Initializing the I/O Subsystem

- Required steps
  - Fill in the device switch table
  - Fill in interrupt vectors
  - Initialize data structures, such as shared buffers
  - Create semaphores used for coordination
- Xinu approach
  - Device switch table configured at compile time
  - At startup, Xinu calls *init* for each device

Summary (1/2)

- OS component to handle I /O known as *device manager*
- Device-independent routines
  - Provide uniform interface
  - Define generic operations that must be mapped to device-specific functions
- Interrupt code
  - Consists of single dispatcher and handler for each device
  - Is executed by whatever process was running when interrupt occurred
- To accommodate null process, interrupt handler must leave executing process in *current* or *ready* states

Summary
(2/2)

- Rescheduling during interrupt safe if
  - Global data structures valid
  - No process explicitly enables interrupts
- Device driver functions
  - Are divided into upper-half and lower-half
  - Can use process synchronization primitives