Device management 2

CS503: Operating systems, Spring 2019

Pedro Fonseca
Department of Computer Science
Purdue University

Admin

- Course feedback survey for feedback on Piazza
  - Only 3 questions :)

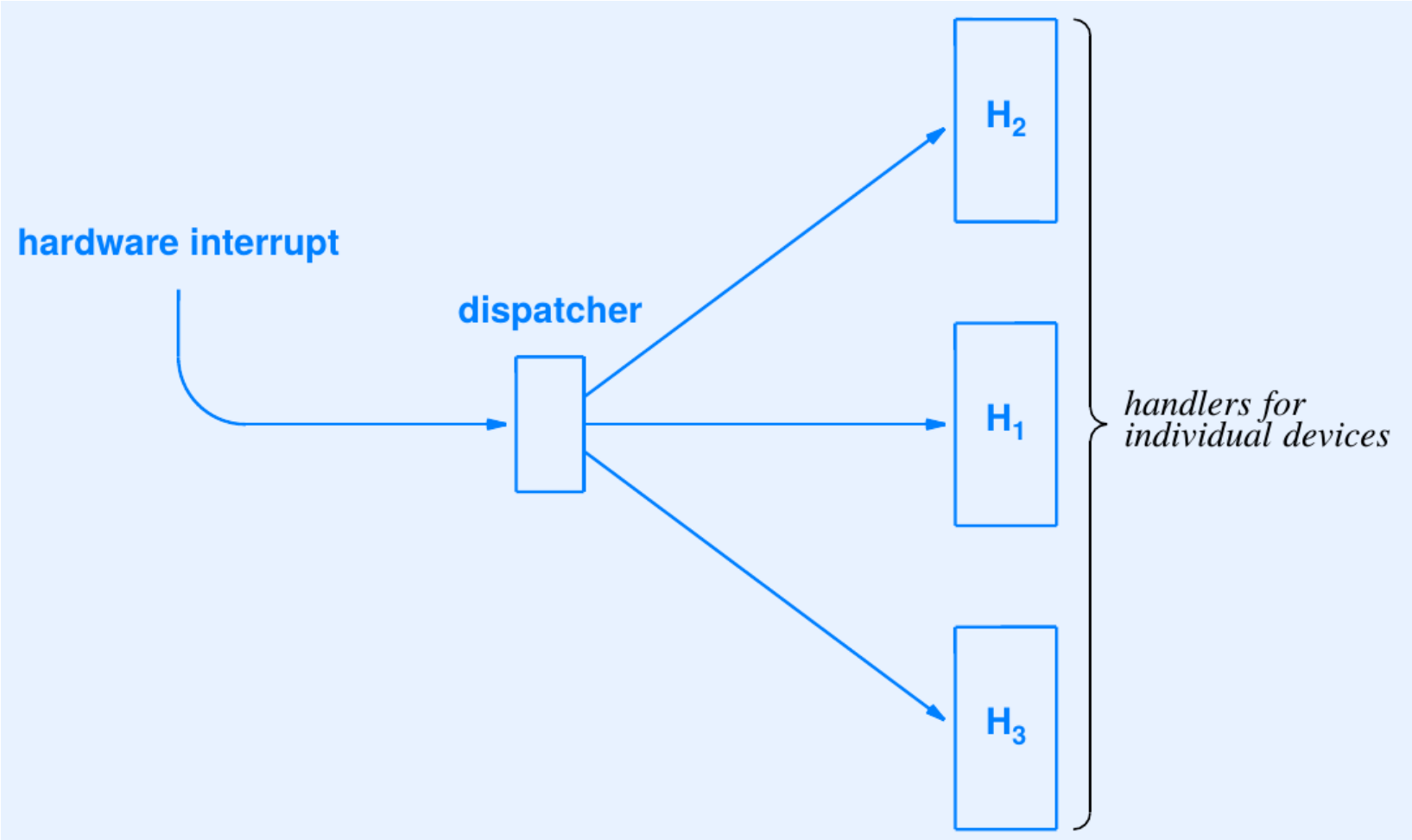
Previous lecture

- Device manager is part of OS:
  - Manages peripheral resources
  - Hides low-level hardware details
  - Provides API to applications
  - Synchronizes processes and IO
- OS presents applications with uniform interface to all devices (as much as possible)
- IO is typically interrupt driven

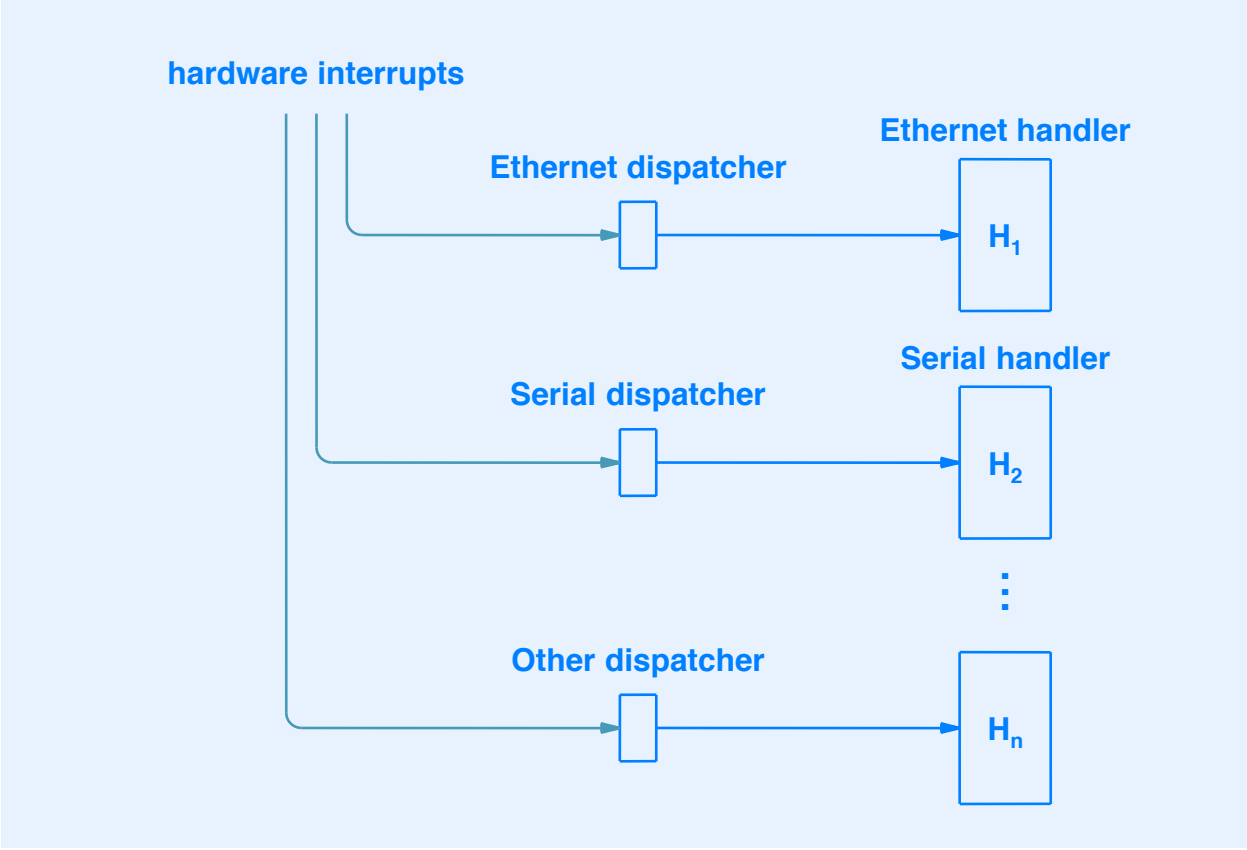
Recall: Interrupt dispatcher

- 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

Recall:
Conceptual view of interrupt dispatching



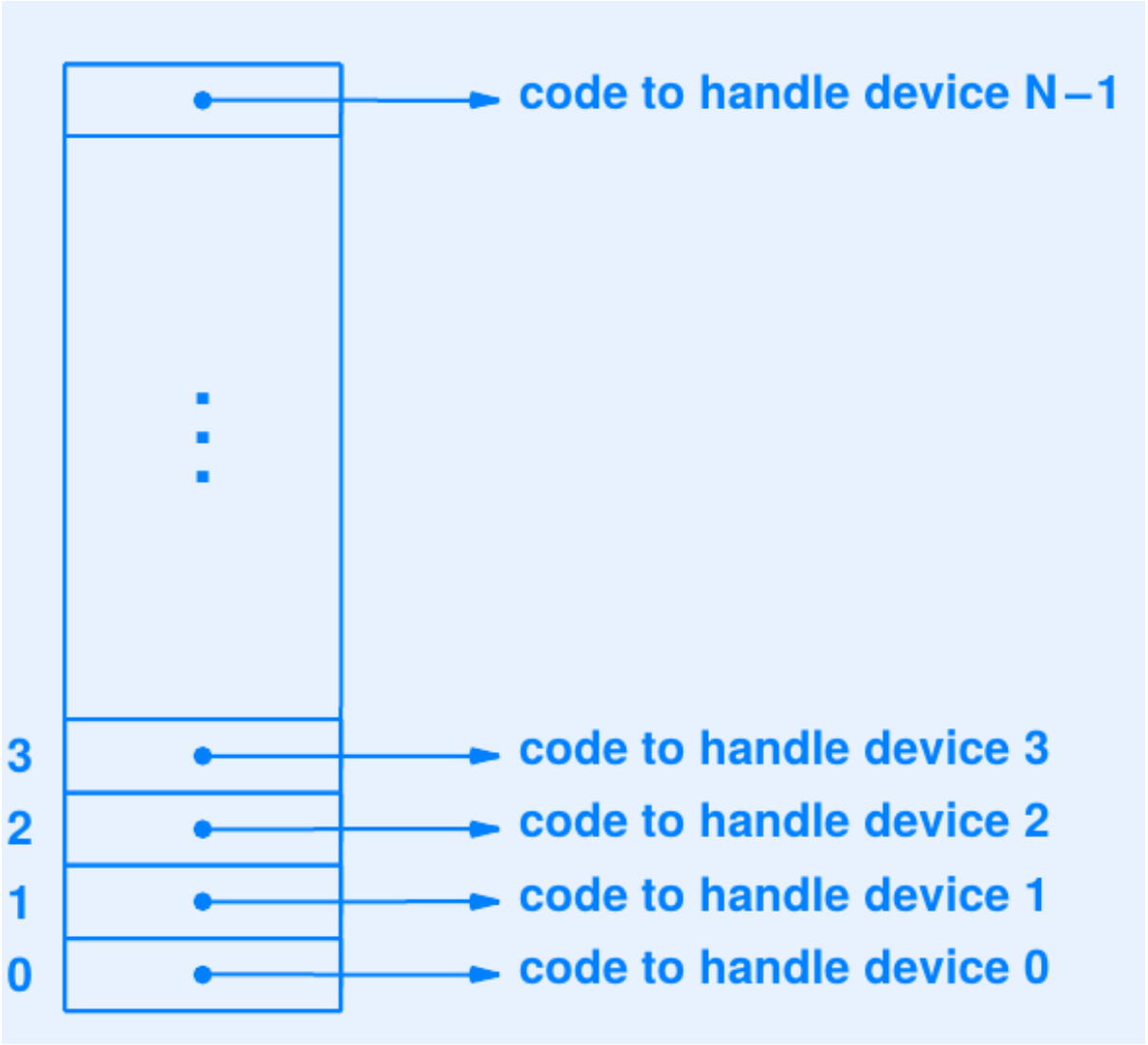
Interrupt dispatcher in Xinu for x86



Recall: 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

Recall: Conceptual organization of the interrupt vector



Intel manual: meaning of each vector number

Table 9-1. Interrupt and Exception ID Assignments

Identifier	Description
0	Divide error
1	Debug exceptions
2	Nonmaskable interrupt
3	Breakpoint (one-byte INT 3 instruction)
4	Overflow (INTO instruction)
5	Bounds check (BOUND instruction)
6	Invalid opcode
7	Coprocessor not available
8	Double fault
9	(reserved)
10	Invalid TSS
11	Segment not present
12	Stack exception
13	General protection
14	Page fault
15	(reserved)
16	Coprecessor error
17-31	(reserved)
32-255	Available for external interrupts via INTR pin

Interrupt dispatching and handling

- Dispatcher involves saving and restoring system state -> generally written with assembly
- Handler -> generally bigger (bulk of the device handling logic) and written in higher-level languages (more convenient for developers)

Rescheduling during an interrupt (v2)

- Process U running with interrupts enabled
- Interrupt occurs, interrupt dispatching uses U's stack
- Suppose the interrupt handler calls resched() and context switches to T
- T may be running with interrupts enabled and another interrupt may occur
- What prevents a stack overflow if interrupts keep happening?

Rescheduling during an interrupt (v2)

- Process U running with interrupts enabled
- Interrupt occurs, interrupt dispatching uses U's stack
- Suppose the interrupt handler calls resched() and context switches to T
- T may be running with interrupts enabled and another interrupt may occur
- Even if we switch to U again:
  - U will be running with interrupts disabled until it reenables them
  - Implementation guarantee: only one interrupt can be running at any point in each process
  - Use of disable/restore
  - Rescheduling during interrupt processing is safe provided that: (1) interrupt code leaves global data in a consistent state and (2) no function enables interrupts unless it disabled them

Xinu code

- system/intr.S: _extint
- system/evec.c: int_dispatch

Interrupt assignment

- Manual device configuration
- Automatic assignment during bootstrap
- Dynamic assignment for pluggable devices

Recall: 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?

Recall: Retrofitting process coordination mechanism

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

Recall: 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

Recall: device management topics

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

Application interface to devices

- Desires:
  - Portability across machines
  - Generality sufficient for all devices
- Solution:
  - Isolate application processes from device drivers
  - Use a common paradigm across all devices
  - Integrate with other operating system facilities

Approach to device-indepent IO

- Define a set of abstract functions
- Build a general purpose mechanism:
  - Use a generic operations (e.g., read or write)
  - Include parameters to specify device instance
  - Arrange an efficient way to map generic operation onto 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 implement the generic operations wrt the device

Device independent I/O in Xinu

- `init`: initialize device (invoked once, at system startup)
- `open`: make the device ready
- `close`: terminate use of the device
- `read`: input arbitrary data from the device
- `write`: output arbitrary data to the device
- `getc`: input a single character from the device
- `putc`: output a single character to the device
- `seek`: position the device
- `control`: control the device and / or the driver
- Note: some abstract functions may not apply to a given 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 read for ETHER)

Mapping generic IO function

- Mapping must be efficient
- Is performed with a device switch table
  - Kernel data structure initialized when the system loads
  - 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

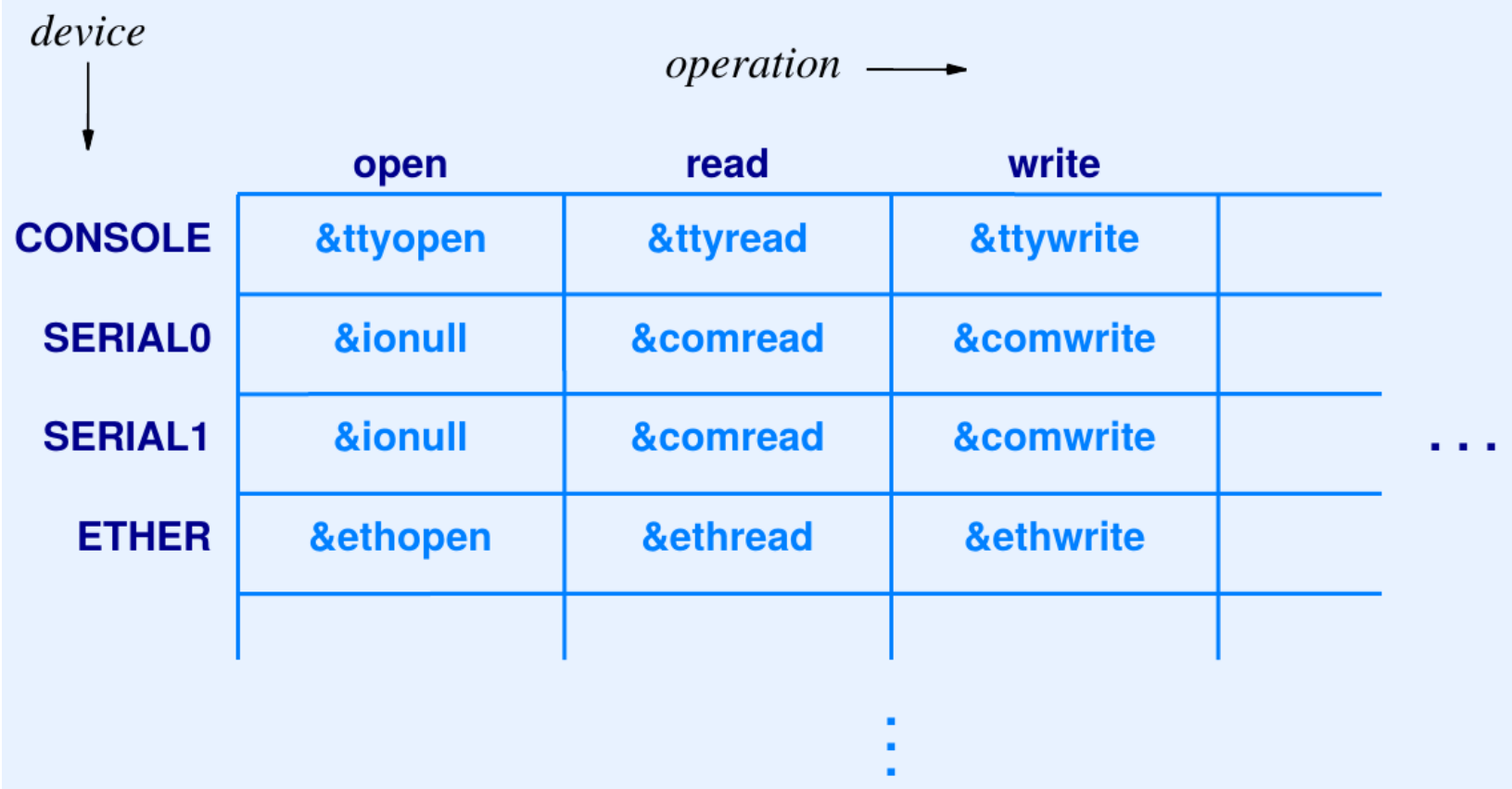
Semantics of device-independent IO

- Each device-independent operation is generic
- An operation may not make sense for a given device
  - Seek on keyboard, network, or display screen
- However, all entries in the device switch table must be valid
- One solution: create functions that can be used to fill in meaningless entires

Special entries in the device switch table

- ionull
  - Used for innocuous operations
  - Always return OK
- ioerr
  - Used for incorrect operation
  - Always returns SYSERR

Device switch table



Parameterized device drivers

- A driver must:
  - Know which physical copy of a device to use
  - Keep information for one copy separate from the information for other copies
- Solution:
  - Assign each instance a device a unique minor number (0, 1, 2…) known as a minor device number
  - Store the minor device number in the device switch table

Device names

- Some examples:
  - CONSOLE
  - SERIAL0
  - SERIAL1
  - ETHER
- The device switch table is an array, and each device name is an index
- Q: How are unique values assigned to names, such as CONSOLE?
- A: OS designer specifies names during configuration

Initializing the IO 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:
  - Static ID (major/minor) allocation
    - Device switch table is configured at compile time
  - At startup, Xinu calls **init** for each device

Example: tty

Driver organization

- tty device:
  - Used for IO on serial lines
  - Most common use: keyboard IO
- Hardware:
  - Transfers a single character at a time
  - Has on-board input and output FIFOs
- The driver maintains separate input and output buffers
- Semaphores are used to synchronize upper and lower halves

tty functions

Upper-Half	Lower-Half
ttyinit	ttyhandler (interrupt handler)
ttyopen	ttyhandle_in (input interrupt)
ttyclose	ttyhandle_out (output interrupt)
ttyread	
ttywrite	
ttyputc	
ttygetc	
ttycontrol	

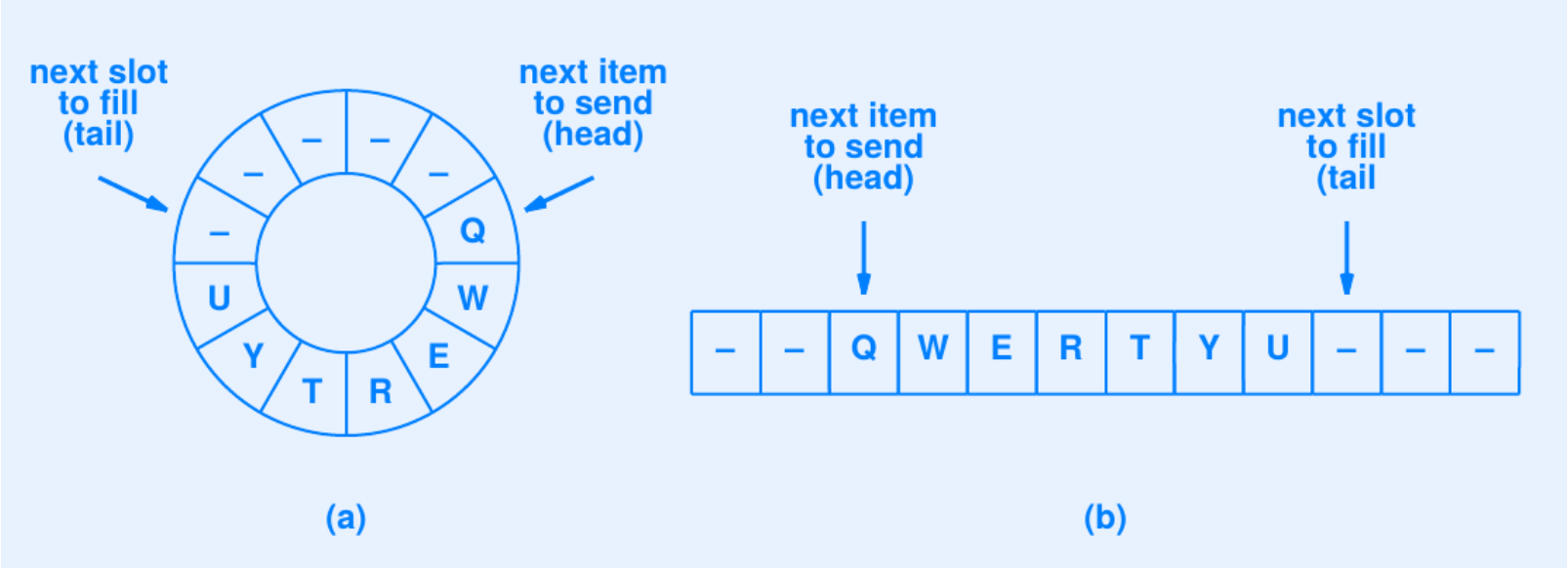
Actions during character output

- Output semaphore counts spaces in buffer
- Upper-half:
  - Waits on semaphore for buffer space
  - Deposits character in next buffer slot
  - Kicks the device, which causes it to interrupt
- Lower-half:
  - Extracts character from next filled slot and passes to device
  - Signals semaphores to indicate space in buffer

Tty driver complexity

- Hardware: onboard FIFO
- Software features:
  - Modes (raw, cooked, and cbreak)
  - CRLF mapping
  - input character echo
  - Visualization of echoed control characters (e.g., ^A)
  - Editing, including backspace over control characters

Circular buffer implementation with an array



- Mode determines how characters are processed
- Application can change the mode at any point in time

Perspective

- The hardware is primitive
  - There is no direct hardware connection between keyboard input and the output seen on a screen; a driver must choose how to display characters
- Most features, such as erasing backspace, are handled entirely by software
- Small details may complicate the implementation

Summary

- Interrupt vectors
- Scheduling during interrupts
- Device interface; device switch table
- Tty device