CS 333
Introduction to Operating Systems

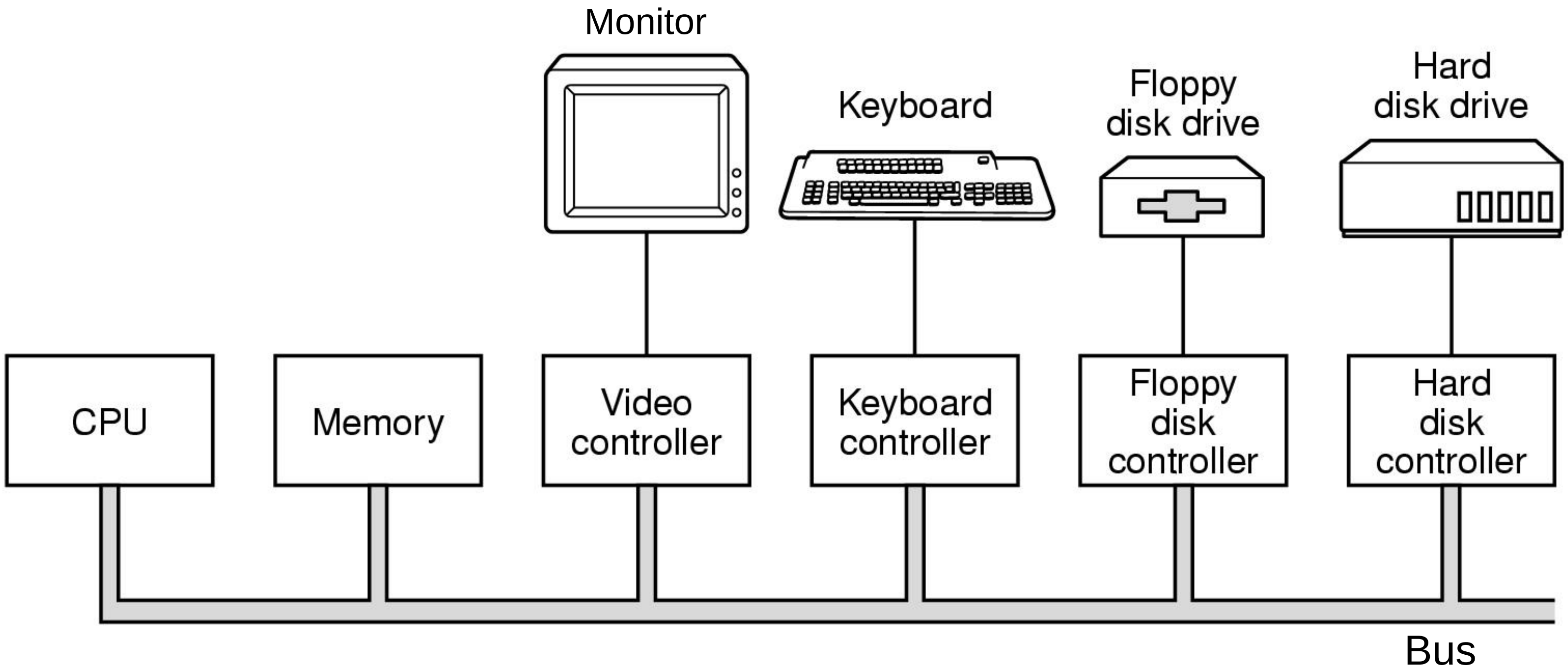
Class 15 - Input/Output

Jonathan Walpole
Computer Science
Portland State University

I/O devices - terminology

- Device (mechanical hardware)
- Device controller (electrical hardware)
- Device driver (software)

Example devices and their controllers



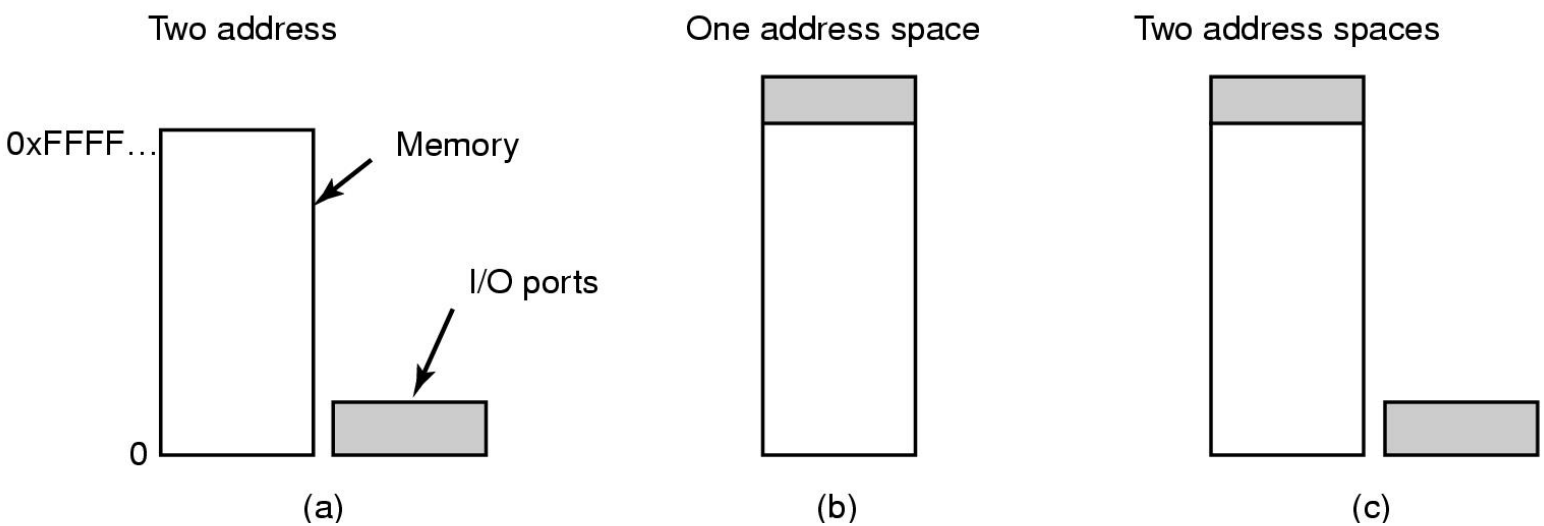
- **Components of a simple personal computer**

Device controllers

- **The Device vs. its Controller**
- **Some duties of a device controller:**
  - Interface between CPU and the Device
  - Start/Stop device activity
  - Convert serial bit stream to a block of bytes
  - Deal with errors
    - **Detection / Correction**
  - Move data to/from main memory
- **Some controllers may handle several (similar) devices**

How to communicate with a device?

- Hardware supports I/O ports or memory mapped I/O for accessing device controller registers and buffers



I/O ports

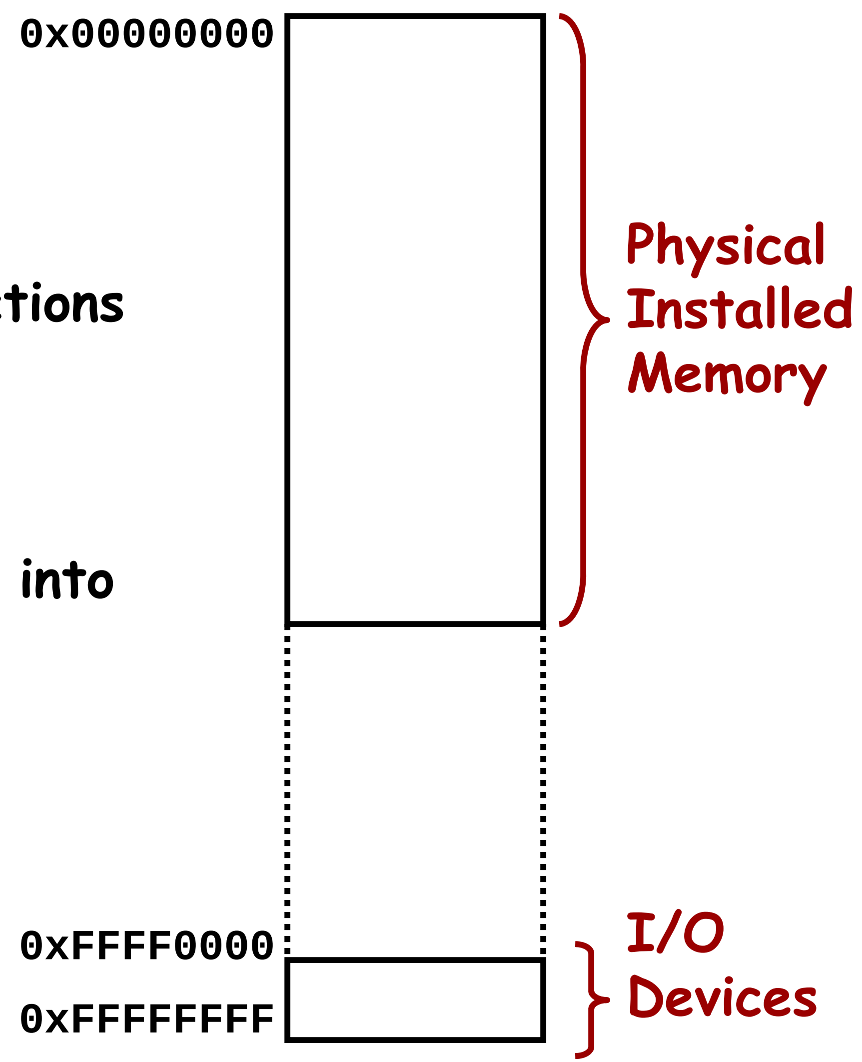
- Each port has a separate number.
- CPU has special I/O instructions
  - `in    r4,3`
  - `out   3,r4`

The I/O Port Number

- Port numbers form an "address space"... separate from main memory
- Contrast with
  - `load    r4,3`
  - `store   3,r4`

Memory-mapped I/O

- One address space for
  - main memory
  - I/O devices
- CPU has no special instructions
  - `load    r4,addr`
  - `store   addr,r4`
- I/O devices are "mapped" into
  - very high addresses



Wide range of I/O device speeds

❑

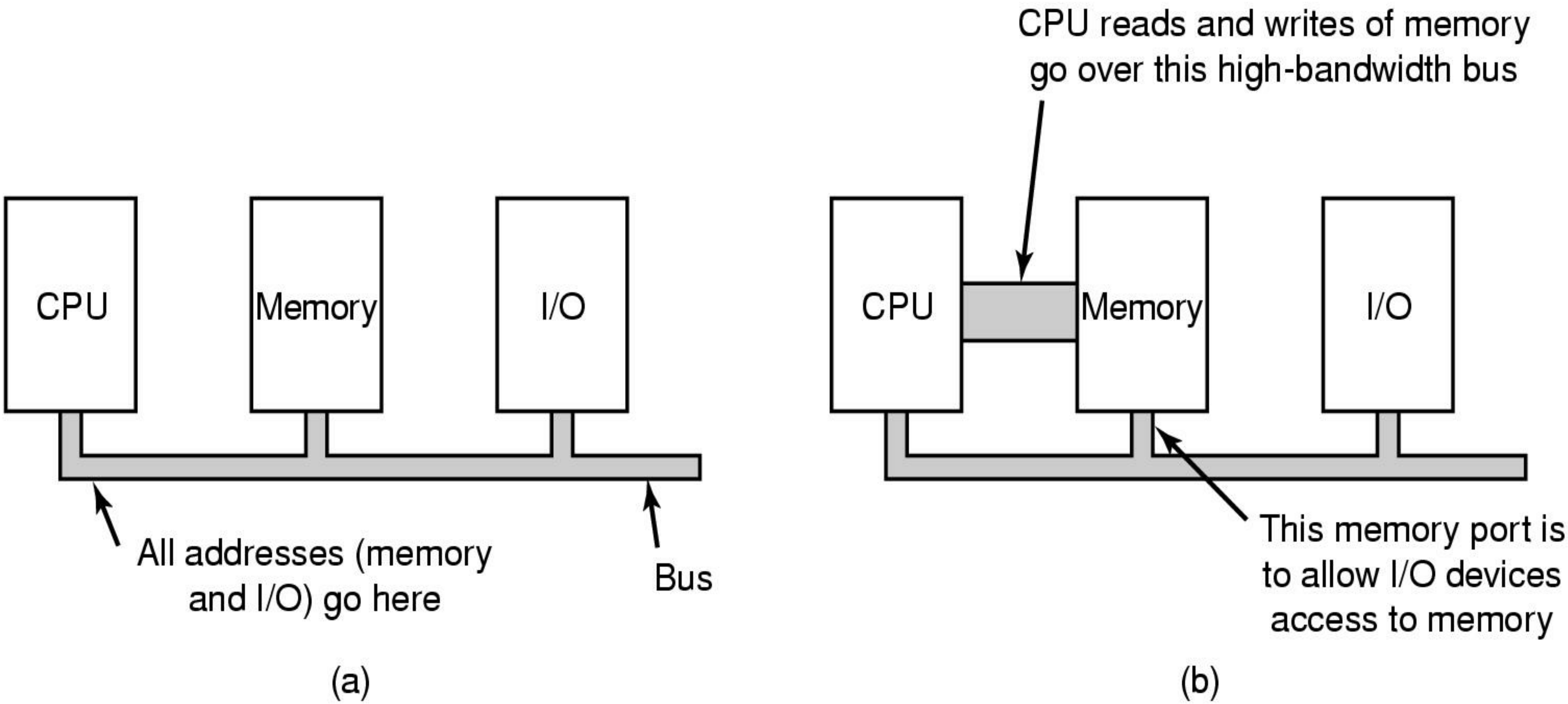
Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Telephone channel	8 KB/sec
Dual ISDN lines	16 KB/sec
Laser printer	100 KB/sec
Scanner	400 KB/sec
Classic Ethernet	1.25 MB/sec
USB (Universal Serial Bus)	1.5 MB/sec
Digital camcorder	4 MB/sec
IDE disk	5 MB/sec
40x CD-ROM	6 MB/sec
Fast Ethernet	12.5 MB/sec
ISA bus	16.7 MB/sec
EIDE (ATA-2) disk	16.7 MB/sec
FireWire (IEEE 1394)	50 MB/sec
XGA Monitor	60 MB/sec
SONET OC-12 network	78 MB/sec
SCSI Ultra 2 disk	80 MB/sec
Gigabit Ethernet	125 MB/sec
Ultrium tape	320 MB/sec
PCI bus	528 MB/sec
Sun Gigaplane XB backplane	20 GB/sec

Performance challenges: I/O hardware

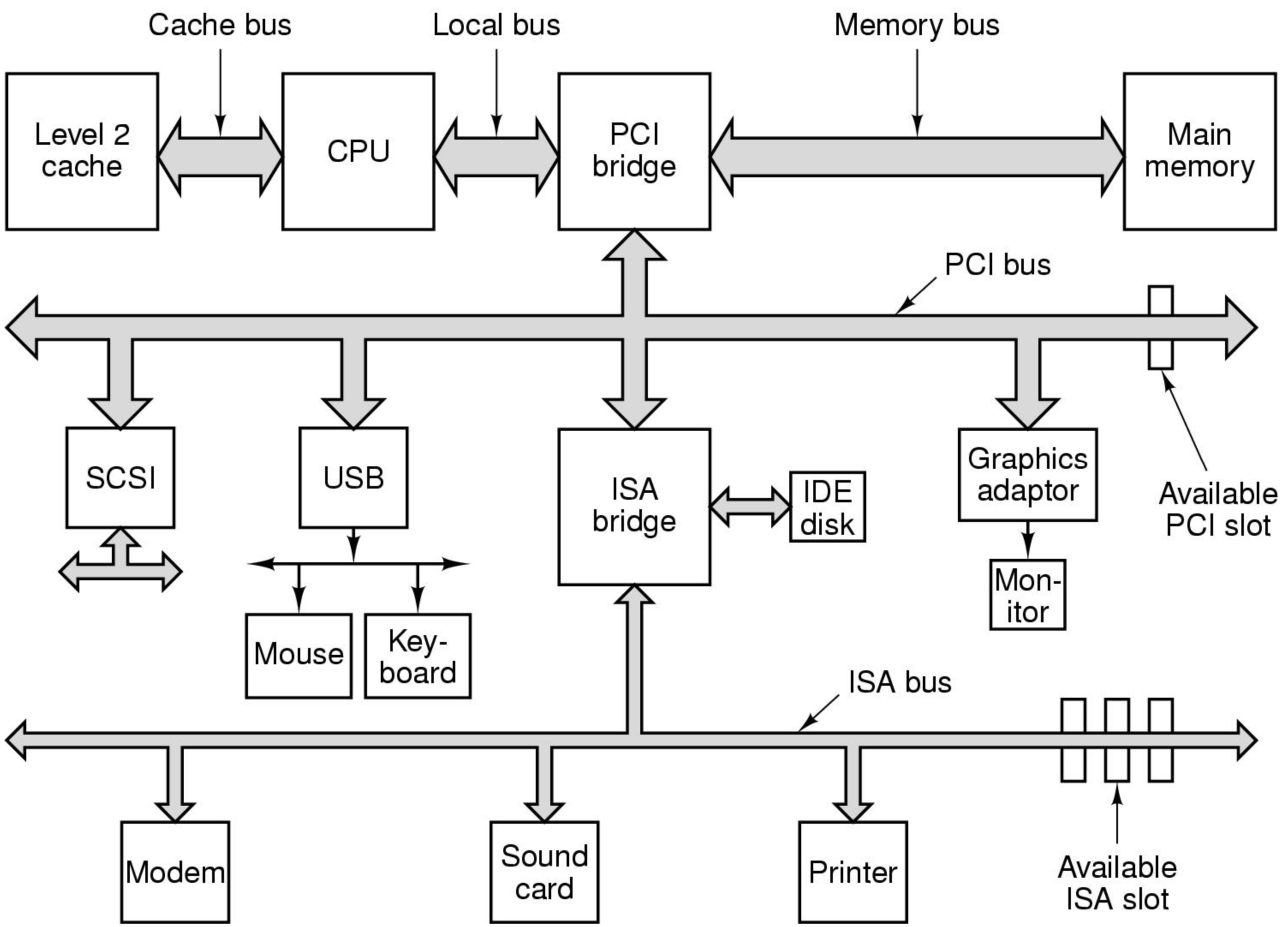
- **How to prevent slow devices from slowing down memory due to bus contention**
  - What is bus contention?

- **How to access I/O addresses without interfering with memory performance**

Single vs. dual bus architecture



Hardware view of Pentium



Structure of a large Pentium system

Performance challenges: I/O software

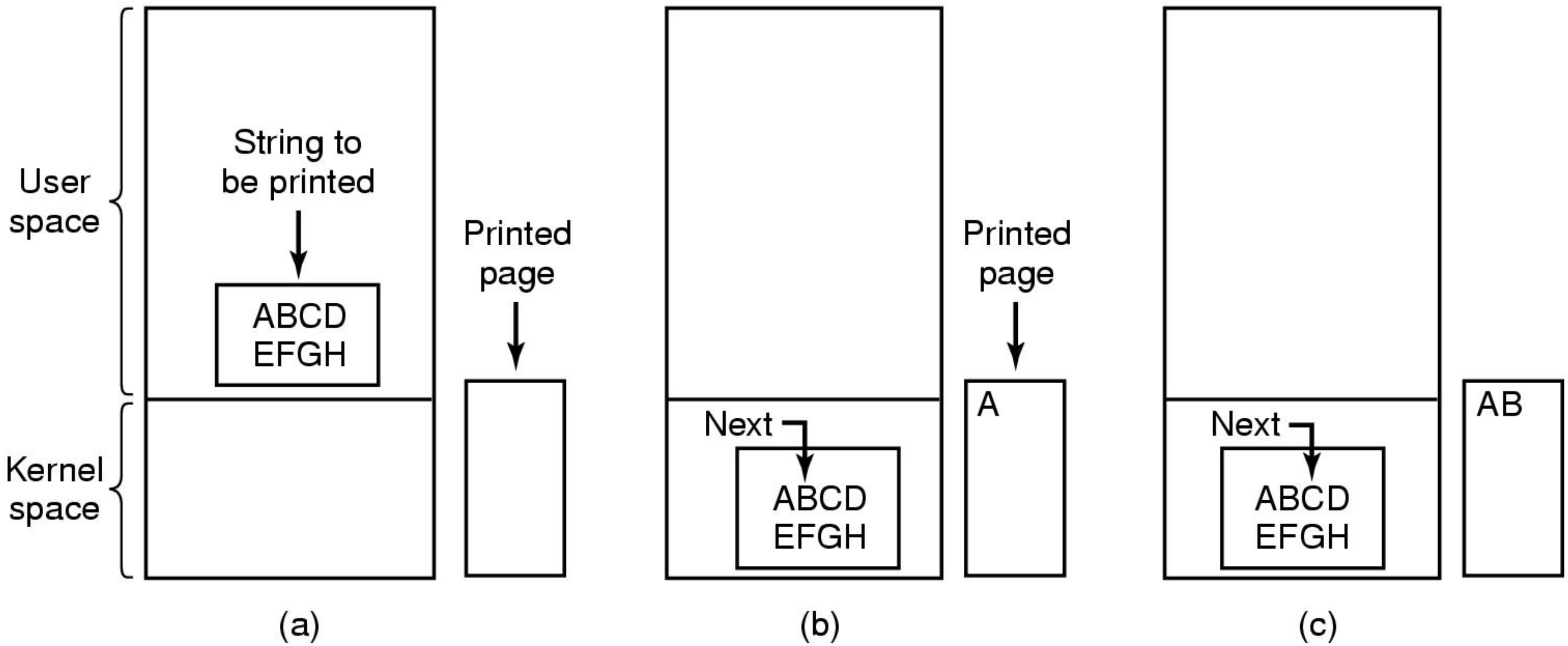
- **How to prevent CPU throughput from being limited by I/O device speed (for *slow devices*)**
  - Why would slow devices affect the CPU?

- **How to prevent I/O throughput from being limited by CPU speed (for *fast devices*)**
  - Why would device throughput be limited by the CPU?

- **How to achieve good utilization of CPU and I/O devices**

- **How to meet the real-time requirements of devices**

Programmed I/O



Steps in printing a string

Programmed I/O

- Example:
  - Writing a string to a serial output
    Printing a string on the printer

```
CopyFromUser(virtAddr, kernelBuffer, byteCount)
for i = 0 to byteCount-1
    while *serialStatusReg != READY
    endWhile
    *serialDataReg = kernelBuffer[i]
endFor
return
```

- Called “Busy Waiting” or “Polling”
- Problem: CPU is continually busy working on I/O!

Interrupt-Driven I/O

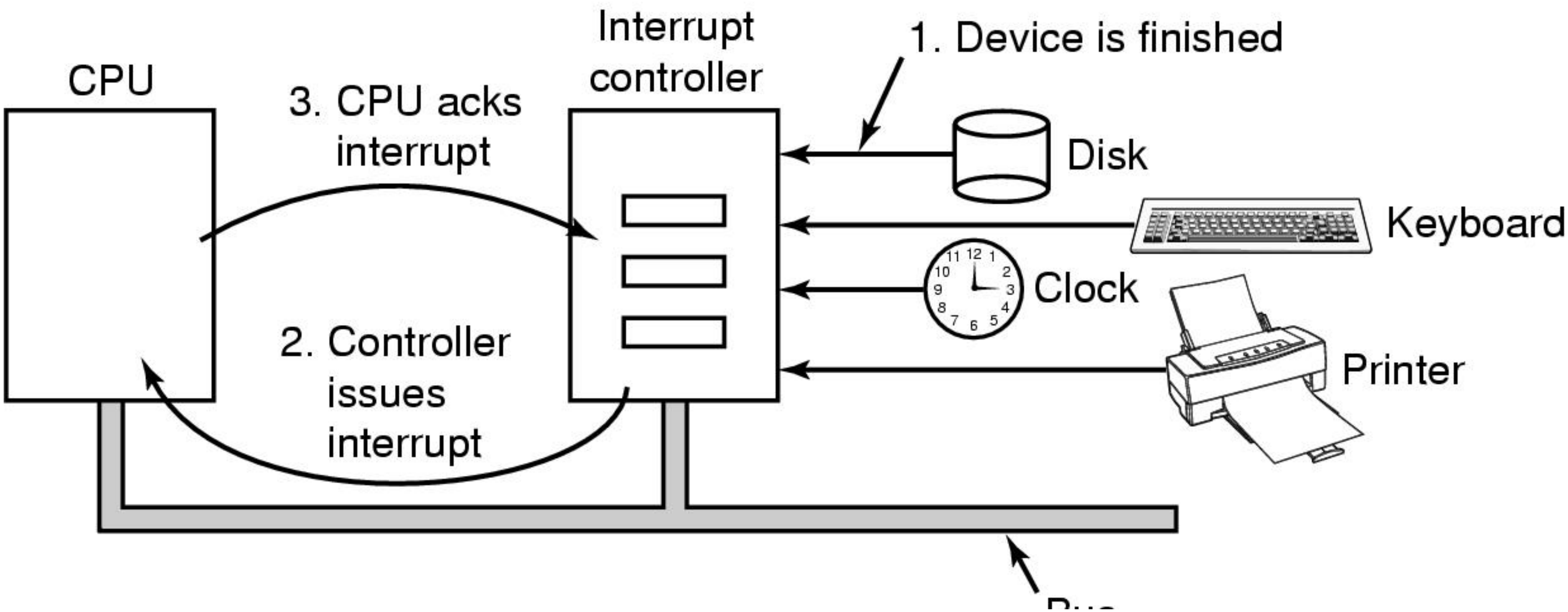
- Getting the I/O started:

```
CopyFromUser(virtAddr, kernelBuffer, byteCount)
EnableInterrupts()
while *serialStatusReg != READY
endWhile
*serialDataReg = kernelBuffer[0]
Sleep ()
```

- The Interrupt Handler:

```
if i == byteCount
  Wake up the user process
else
  *serialDataReg = kernelBuffer[i]
  i = i + 1
endIf
Return from interrupt
```

Hardware support for interrupts



How interrupts happen. Connections between devices and interrupt controller actually use interrupt lines on the bus rather than dedicated wires

Problem with Interrupt driven I/O

- **Problem:**
  - CPU is still involved in every data transfer
  - Interrupt handling overhead is high
  - Overhead cost is not amortized over much data
  - Overhead is too high for fast devices
    - **Gbps networks**
    - **Disk drives**

Direct Memory Access (DMA)

- **Data transferred from device straight to/from memory**
- **CPU not involved**
- ***The DMA controller:***
  - Does the work of moving the data
  - CPU sets up the DMA controller ("programs it")
  - CPU continues
  - The DMA controller moves the bytes

Sending data to a device using DMA

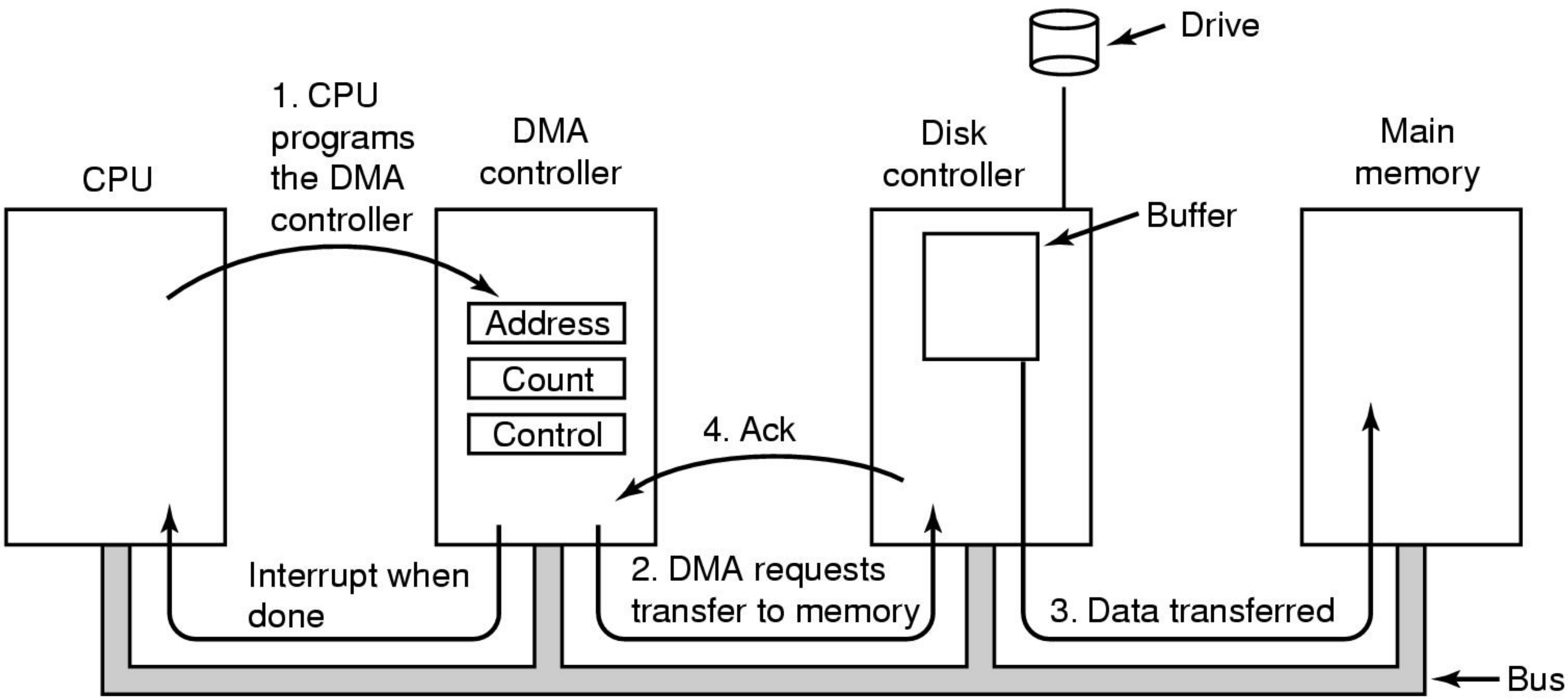
- **Getting the I/O started:**

  ```
  CopyFromUser(virtAddr, kernelBuffer, byteCount)
  Set up DMA controller
  Sleep ()
  ```

- **The Interrupt Handler:**

  ```
  Acknowledge interrupt
  Wake up the user process
  Return from interrupt
  ```

Direct Memory Access (DMA)



Direct Memory Access (DMA)

- **Cycle Stealing**
  - DMA Controller acquires control of bus
  - Transfers a single byte (or word)
  - Releases the bus
  - The CPU is slowed down due to bus contention
- **Burst Mode**
  - DMA Controller acquires control of bus
  - Transfers all the data
  - Releases the bus
  - The CPU operation is temporarily suspended

Direct Memory Access (DMA)

- **Cycle Stealing**
  - DMA controller acquires control of bus
  - Transfers a single byte (or word)
  - Releases the bus
  - The CPU is slowed down due to bus contention
  - *Responsive but not very efficient*

- **Burst Mode**
  - DMA Controller acquires control of bus
  - Transfers all the data
  - Releases the bus
  - The CPU operation is suspended
  - *Efficient but interrupts may not be serviced in a timely way*

Principles of I/O software

- **Device Independence**
  - Programs can access any I/O device
    - **Hard Drive, CD-ROM, Floppy,...**
    - **... without specifying the device in advance**
- **Uniform Naming**
  - Devices / Files are named with simple strings
  - Names should not depend on the device
- **Error Handling**
  - ...should be as close to the hardware as possible
  - ... because its often device-specific

Principles of I/O software

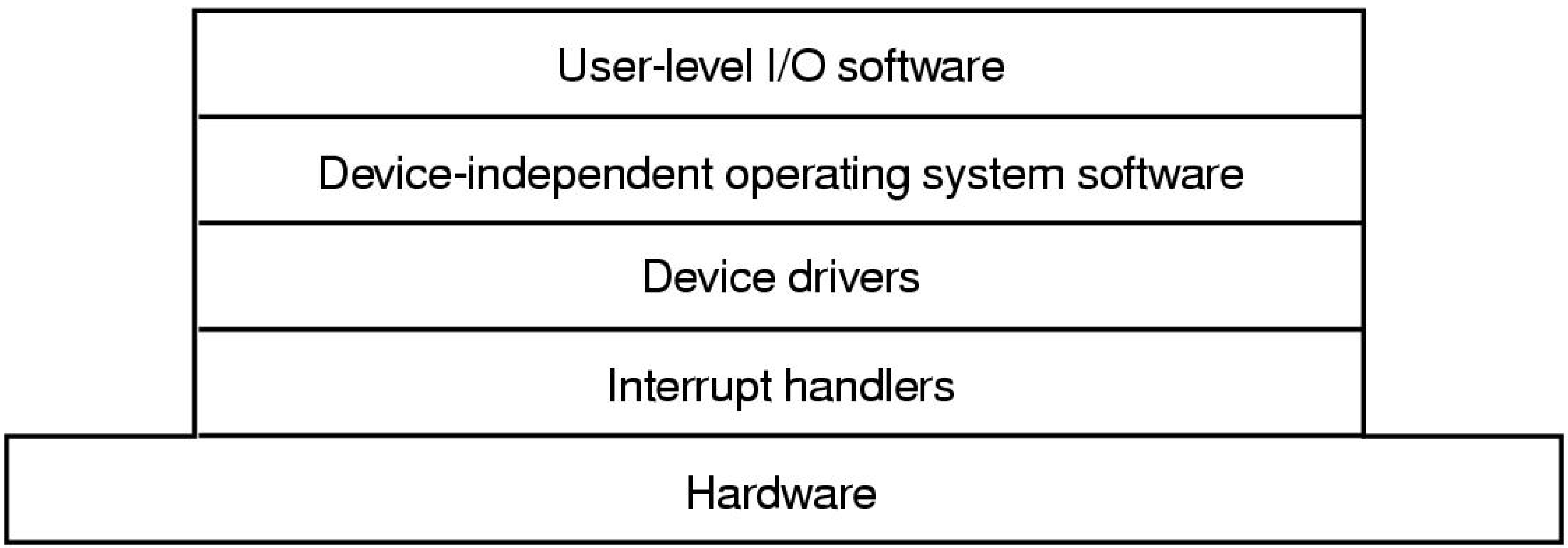
- **Synchronous vs. Asynchronous Transfers**
  - Process is blocked vs. interrupt-driven or polling approaches
- **Buffering**
  - Data comes off a device
  - May not know the final destination of the data
    - **e.g., a network packet... Where to put it???**
- **Sharable vs. Dedicated Devices**
  - Disk should be sharable
  - Keyboard, Screen dedicated to one process

Software engineering-related challenges

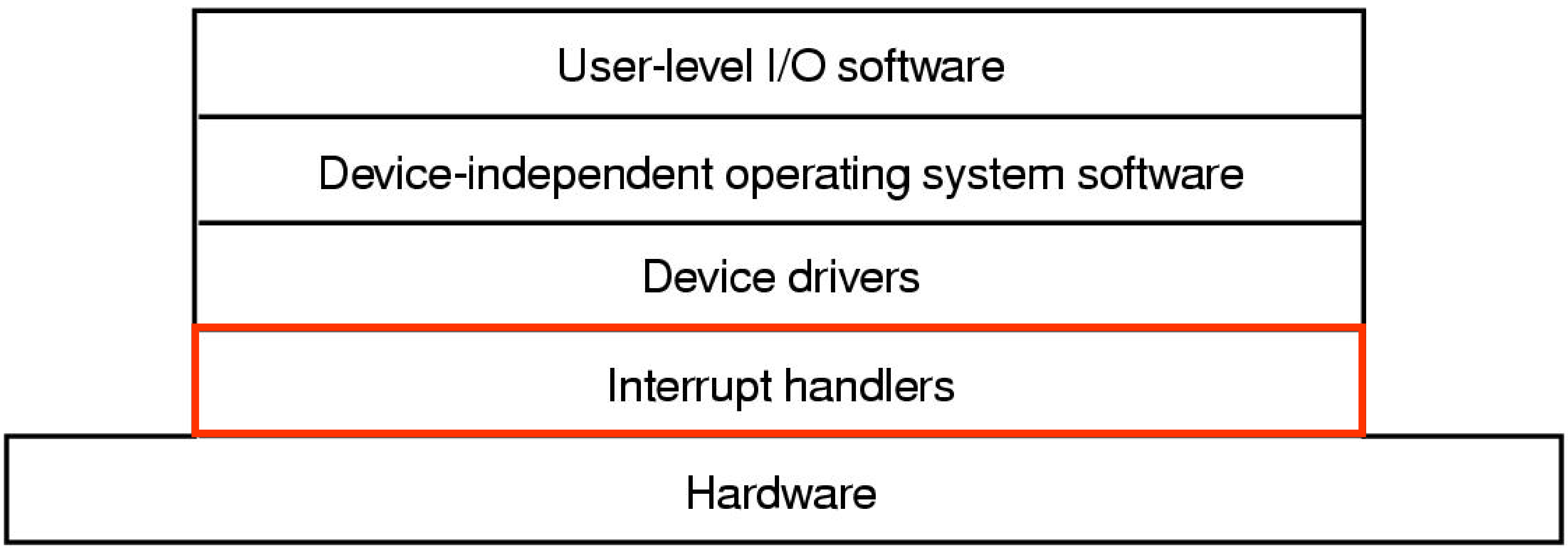
- **How to remove the complexities of I/O handling from application programs**
  - Solution
    - **standard I/O APIs (libraries and system calls)**

- **How to support a wide range of device types on a wide range of operating systems**
  - Solution
    - **standard interfaces for device drivers (DDI)**
    - **standard/published interfaces for access to kernel facilities (DKI)**

I/O software layers



I/O software layers



Interrupt handling

- **I/O Device Driver starts the operation**
  - Then blocks until an interrupt occurs
  - Then it wakes up, finishes, & returns
- **The Interrupt Handler**
  - Does whatever is immediately necessary
  - Then unblocks the driver
- **Example: The BLITZ "DiskDriver"**
  - Start I/O and block (waits on semaphore)
  - Interrupt routine signals the semaphore & returns

Interrupt handlers – top/bottom halves

- **Interrupt handlers are divided into *scheduled* and *non scheduled* tasks**

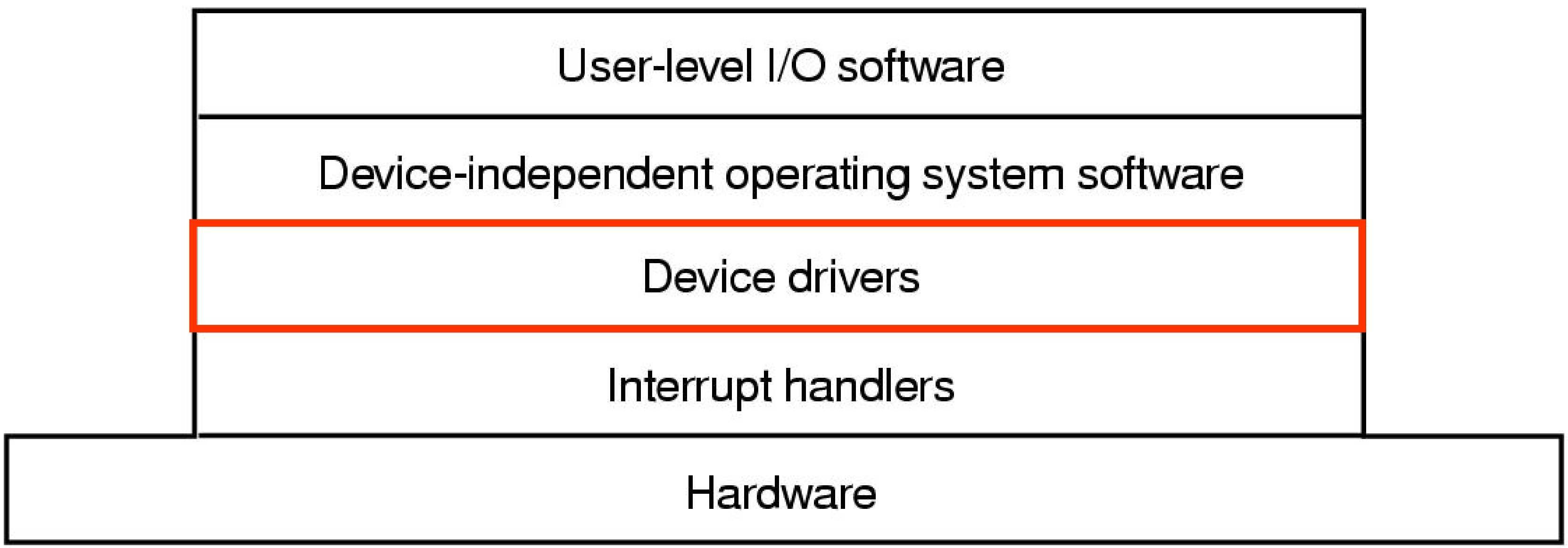
- **Non-scheduled tasks execute immediately on interrupt and run in the context of the interrupted thread**
  - Ie. There is no VM context switch
  - They should do a minimum amount of work so as not to disrupt progress of interrupted thread
  - They should minimize time during which interrupts are disabled

- **Scheduled tasks are queued for processing by a designated thread**
  - This thread will be scheduled to run later
  - May be scheduled preemptively or nonpreemptively

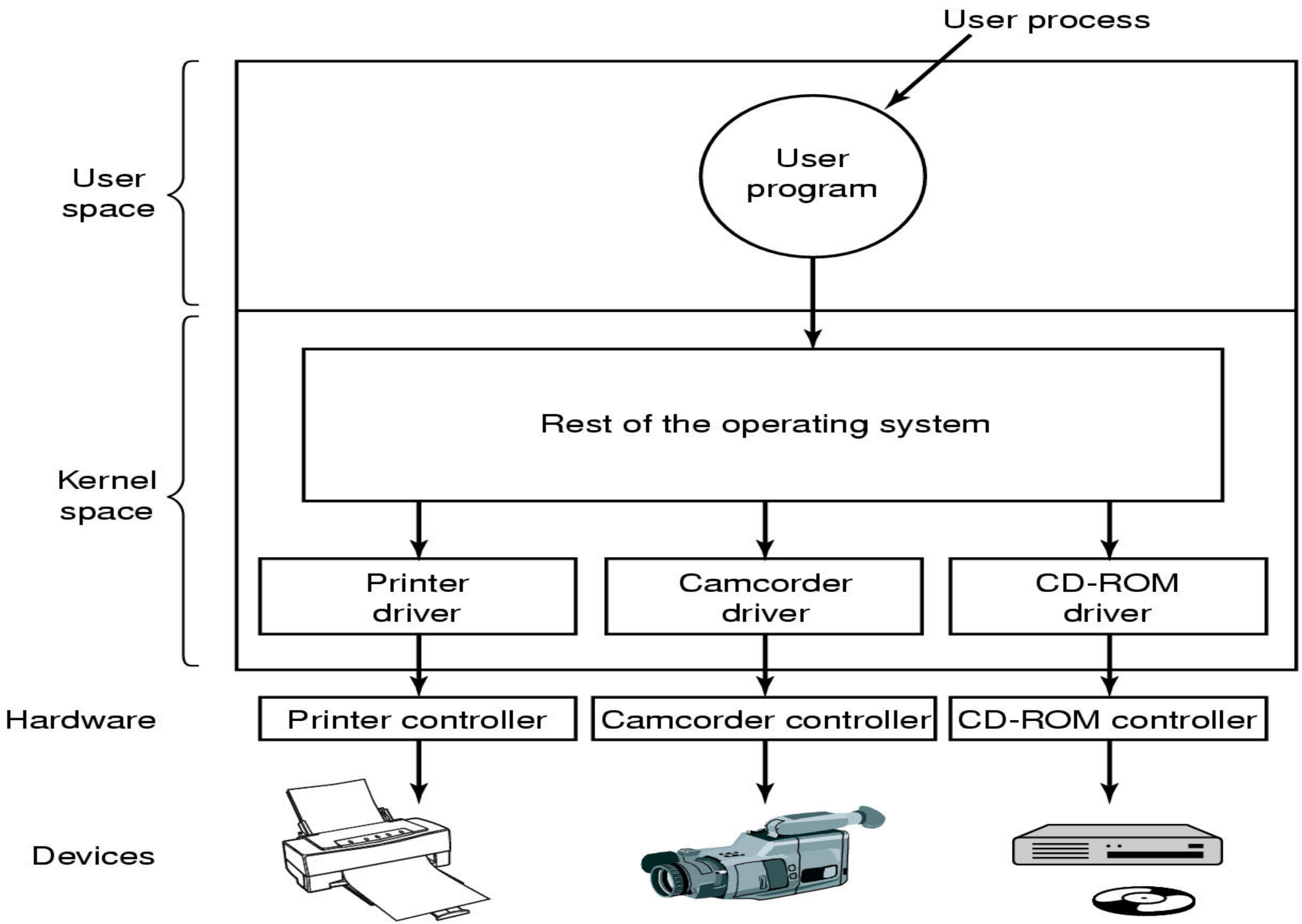
Basic activities of an interrupt handler

- Set up stack for interrupt service procedure
- Ack interrupt controller, reenable interrupts
- Copy registers from where saved
- Run service procedure

I/O software layers



Device drivers in kernel space

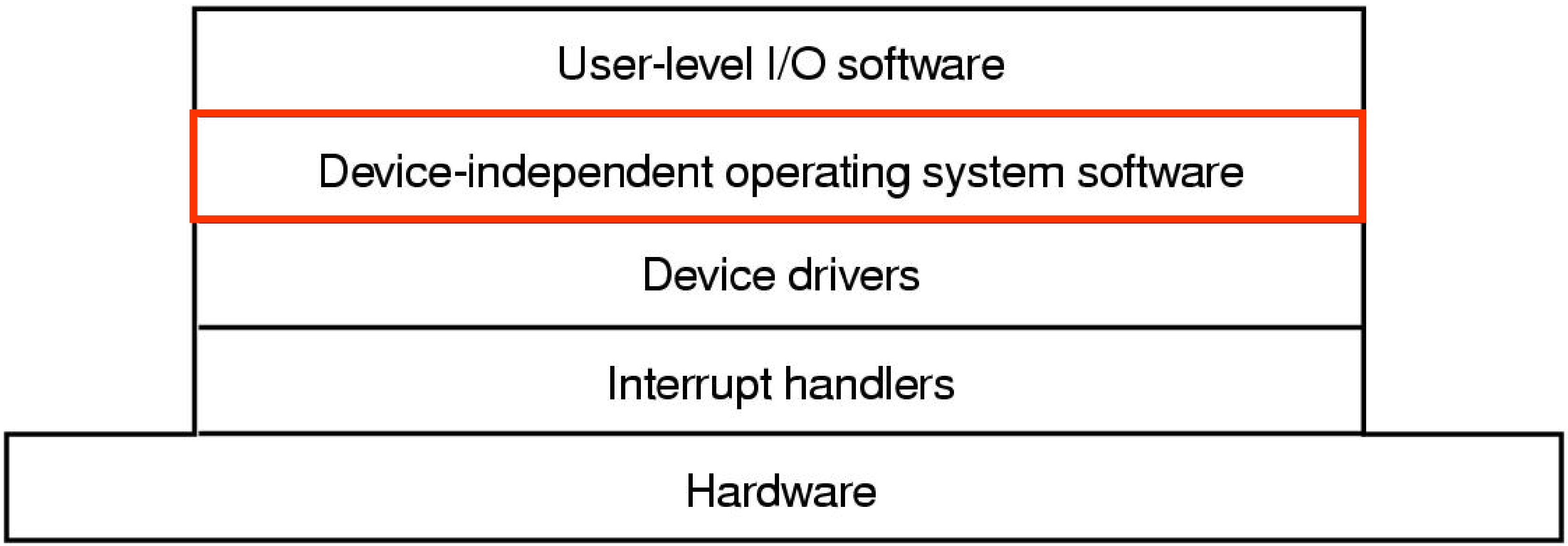


Device drivers

- **Device drivers are device-specific software that connects devices with the operating system**
  - Typically a nasty assembly-level job
    - **Must deal with hardware-specific details (and changes)**
    - **Must deal with O.S. specific details (and changes)**
  - Goal: hide as many device-specific details as possible from higher level software

- **Device drivers are typically given kernel privileges for efficiency**
  - Bugs can bring down the O.S.!
  - Open challenge: how to provide efficiency <u>and</u> safety???

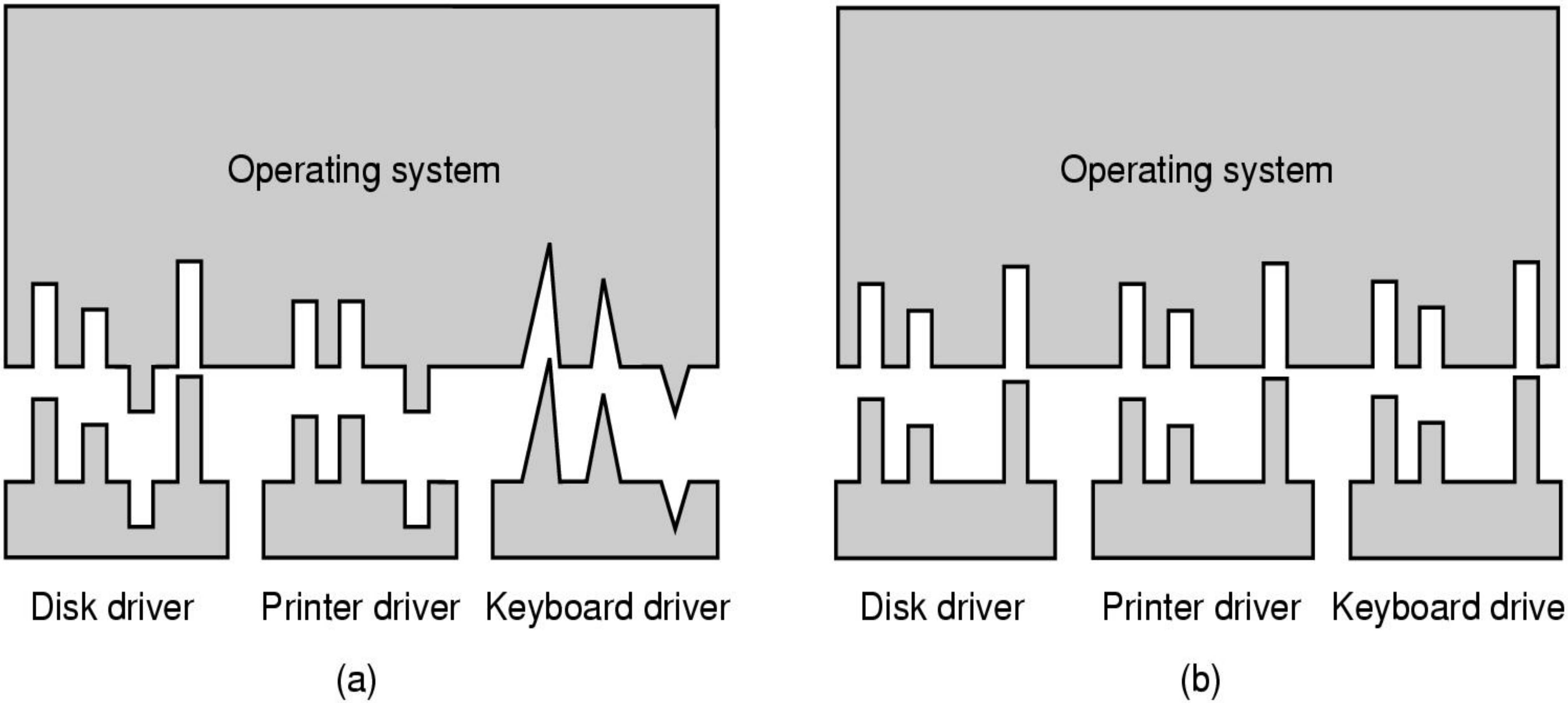
I/O software layers



Device-independent I/O software

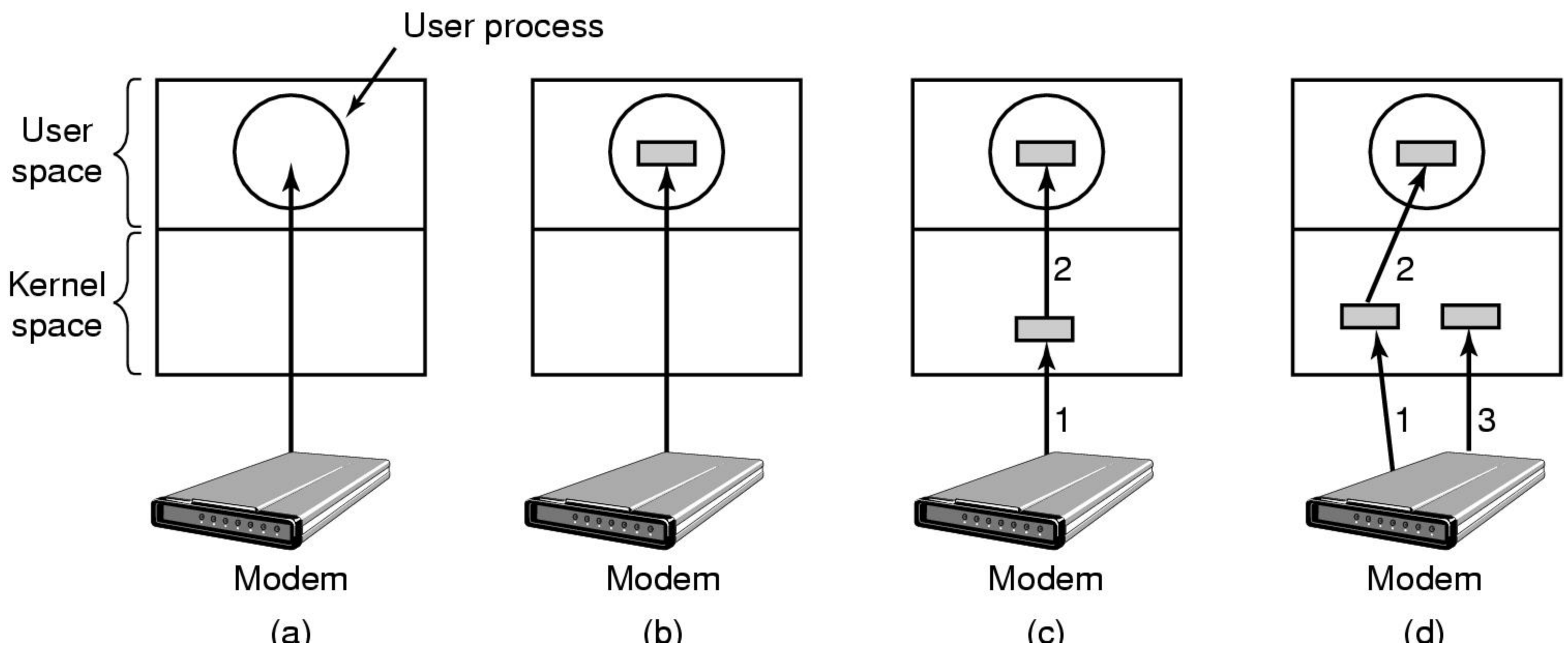
- **Functions and responsibilities**
  - Uniform interfacing for device drivers
  - Buffering
  - Error reporting
  - Allocating and releasing dedicated devices
  - Providing a device-independent block size

Device-independent I/O software



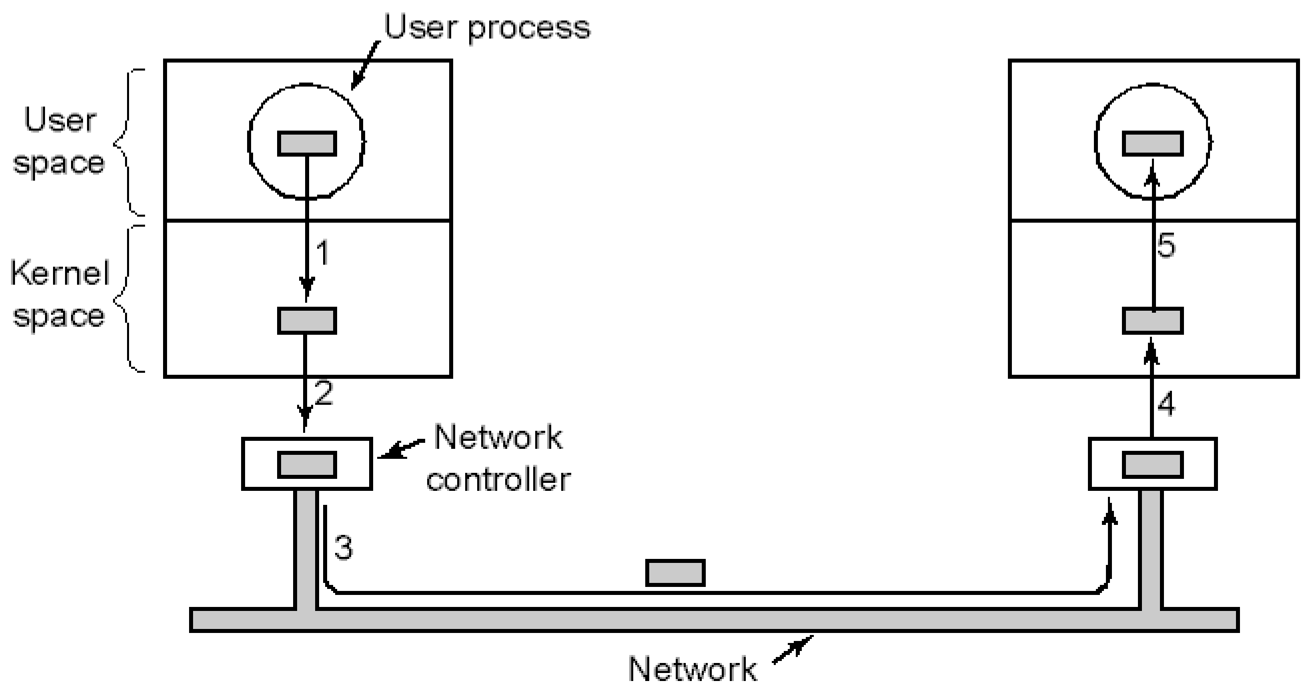
Device Driver Interface (DDI) and Device Kernel Interface (DKI) without/with standardization

Device-independent I/O software buffering



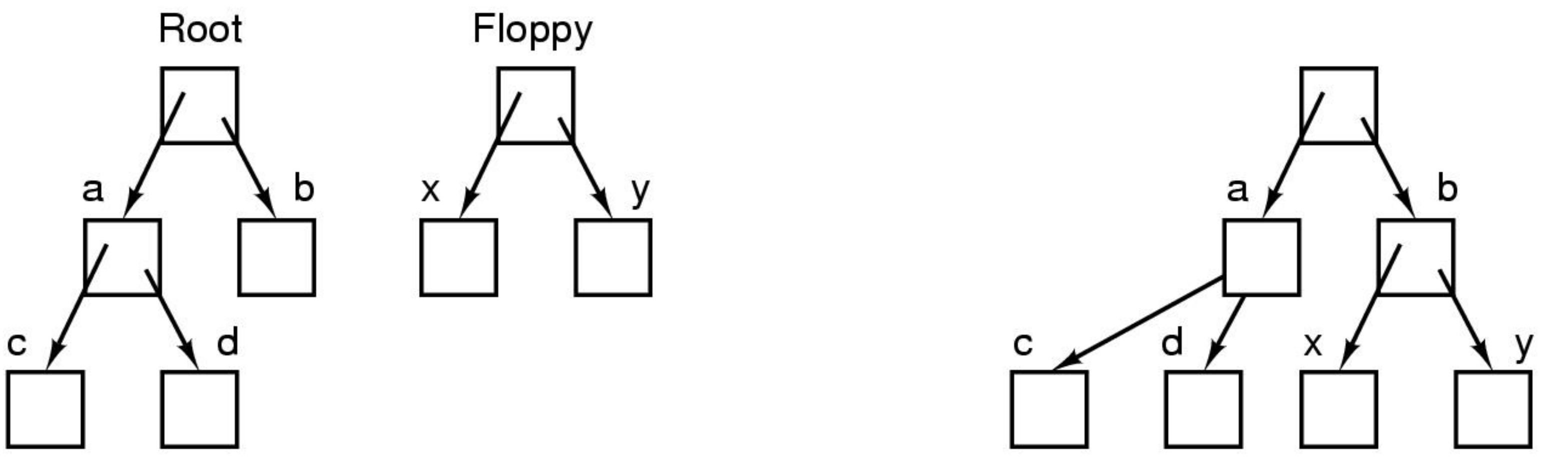
(a) Unbuffered input
(b) Buffering in user space
(c) Buffering in the kernel followed by copying to user space
(d) Double buffering in the kernel

Copying overhead in network I/O



Networking may involve many copies

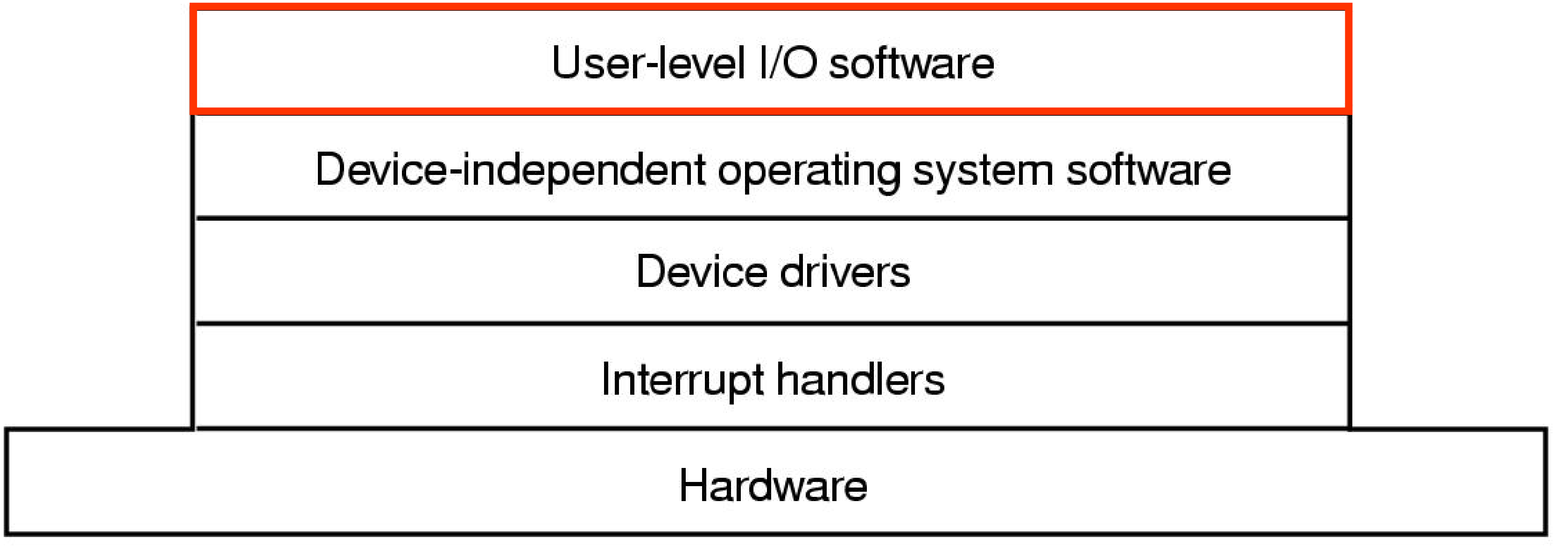
Devices as files



(a) (b)

- **Before mounting,**
  - files on floppy are inaccessible
- **After mounting floppy on b,**
  - files on floppy are part of file hierarchy

I/O software layers



User-space I/O software

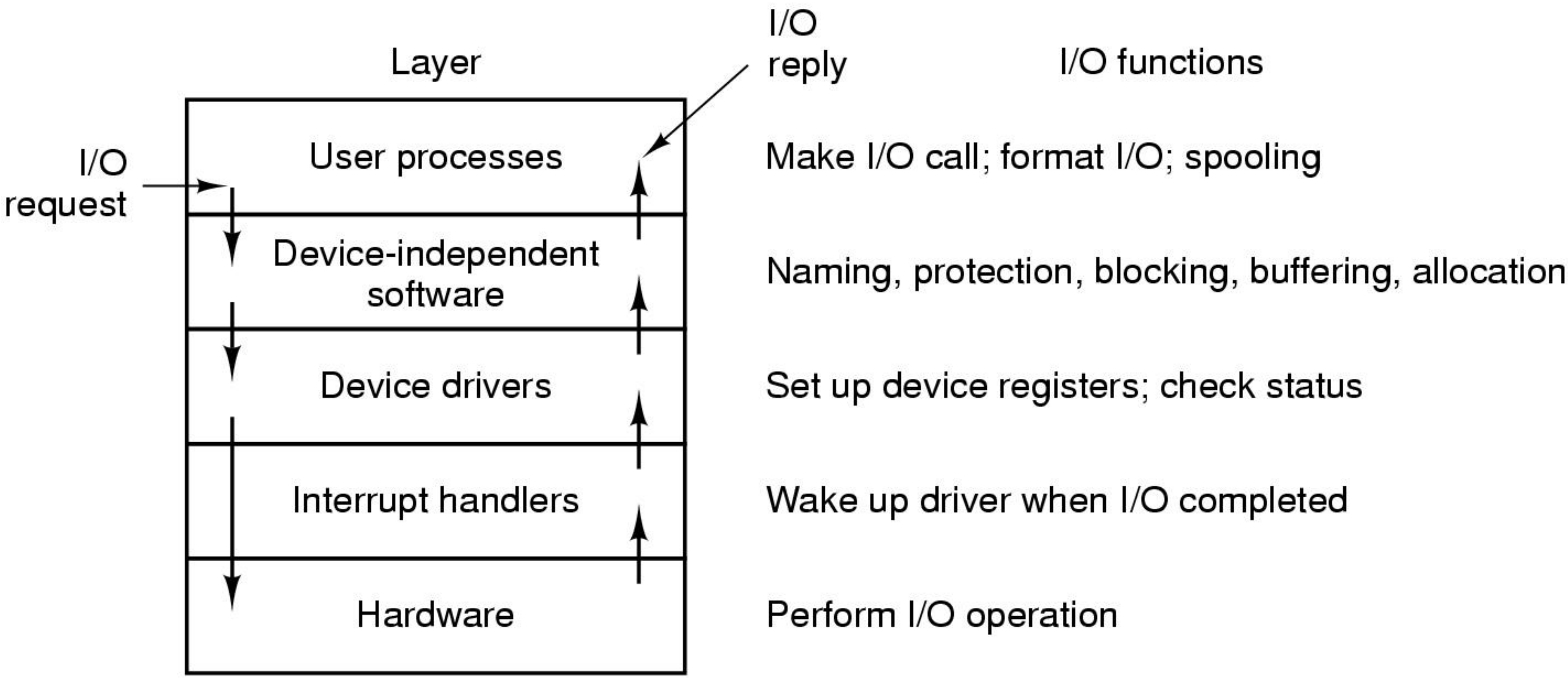
- **In user's (C) program**

```
count = write (fd, buffer, nbytes);
printf ("The value of %s is %d\n", str, i);
```

- **Linked with library routines.**

- **The library routines contain:**
  - Lots of code
  - Buffering
  - The syscall to trap into the kernel

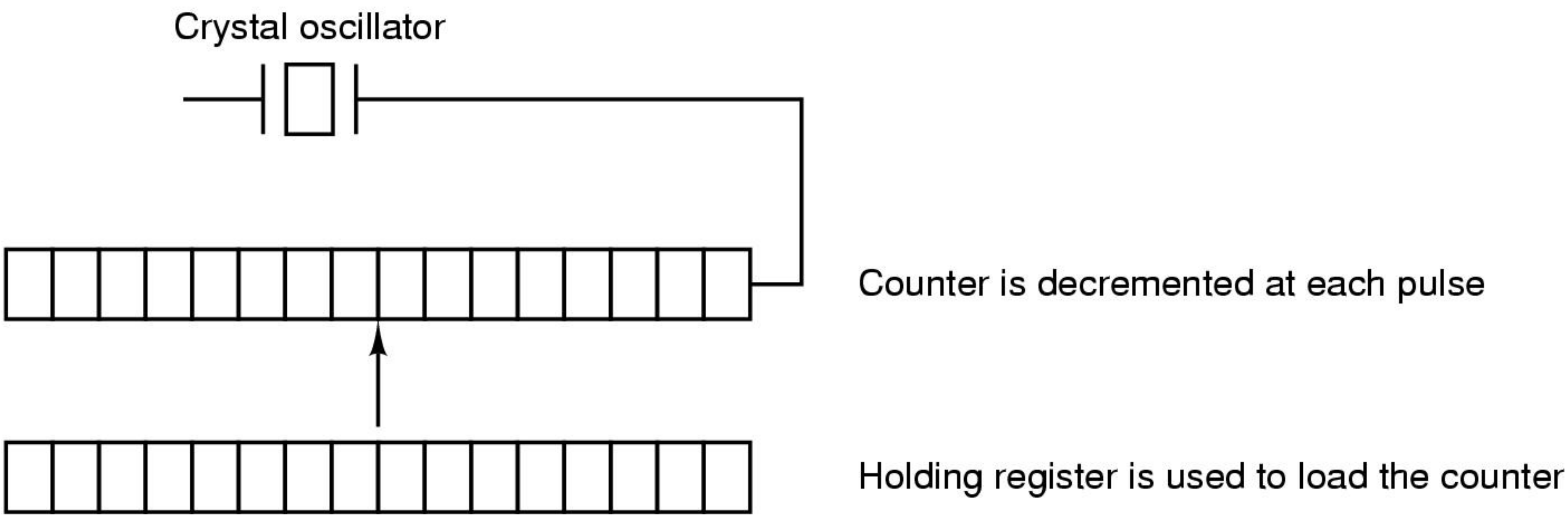
Communicating across the I/O layers



Some example I/O devices

- Timers
- Terminals
- Graphical user interfaces
- Network terminals

Programmable clocks



- **One-shot mode:**
  - Counter initialized then decremented until zero
  - At zero a single interrupt occurs
- **Square wave mode:**
  - At zero the counter is reinitialized with the same value
  - Periodic interrupts (called “clock ticks”) occur

Time

- **500 MHz Crystal (oscillates every 2 nanoseconds)**
- **32 bit register overflows in 8.6 seconds**
  - So how can we remember what the time is?

- **Backup clock**
  - Similar to digital watch
  - Low-power circuitry, battery-powered
  - Periodically reset from the internet
  - UTC: Universal Coordinated Time
  - Unix: Seconds since Jan. 1, 1970
  - Windows: Seconds since Jan. 1, 1980

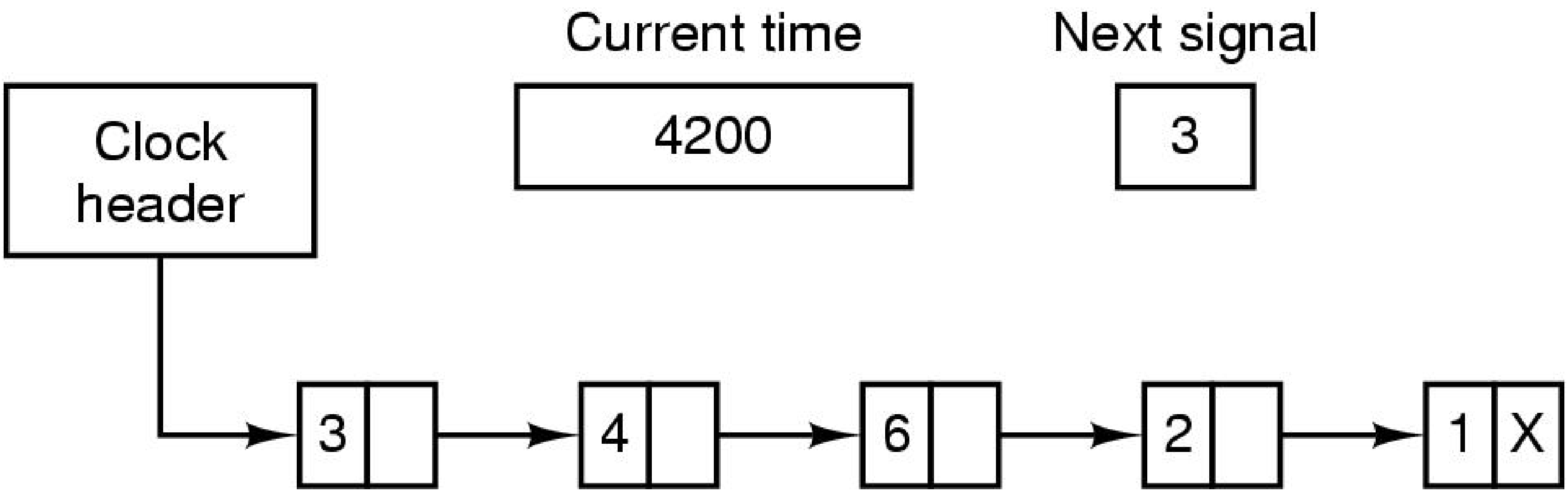
Goals of clock software

- **Maintain time of day**
  - Must update the time-of-day every tick
- **Prevent processes from running too long**
- **Account for CPU usage**
  - Separate timer for every process
  - Charge each tick to the current process
- **Handling the "Alarm" syscall**
  - User programs ask to be sent a signal at a given time
- **Providing watchdog timers for the OS itself**
  - E.g., when to spin down the disk
- **Doing profiling, monitoring, and statistics gathering**

Software timers

- **A process can ask for notification (alarm) at time T**
  - At time T, the OS will signal the process
- **Processes can "go to sleep until time T"**
- **Several processes can have active timers**
- **The CPU has only one clock**
  - Must service the "alarms" in the right order
- **Keep a sorted list of all timers**
  - Each entry tells when the alarm goes off and what to do then

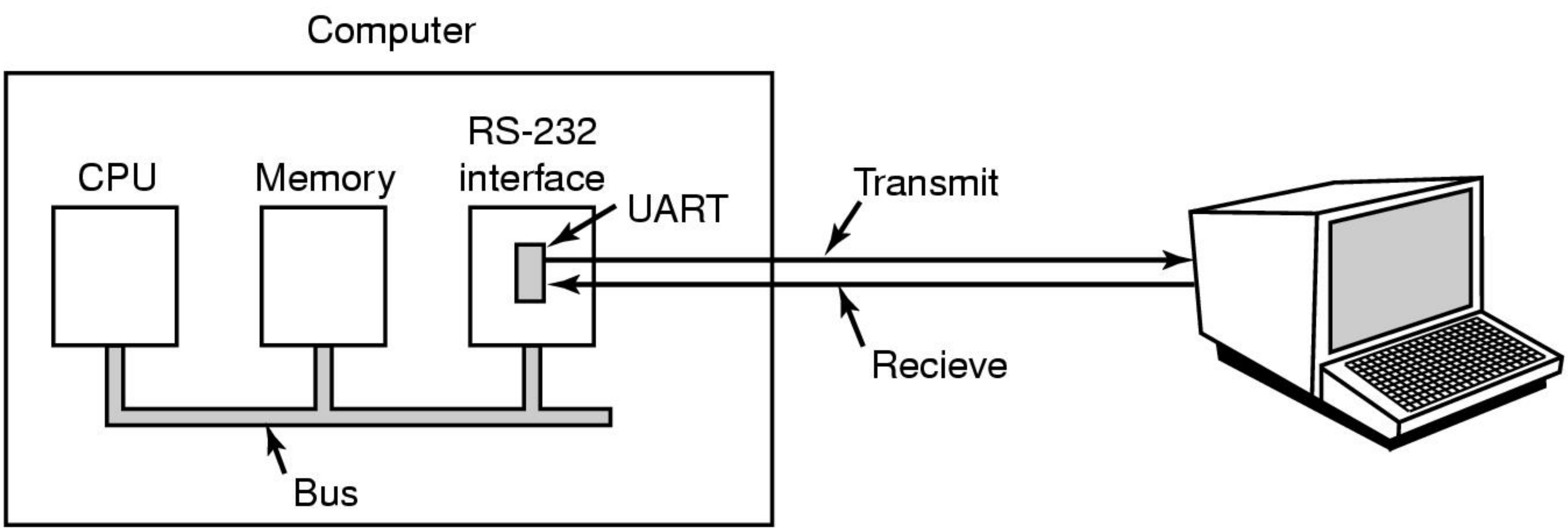
Software timers



- Alarms set for 4203, 4207, 4213, 4215 and 4216.
- Each entry tells how many ticks past the previous entry.
- On each tick, decrement the "NextSignal".
- When it gets to 0, then signal the process.

Character-oriented I/O

- RS-232 / Serial interface / Modem / Terminals / tty / COM
- Bit serial (9- or 25-pin connectors), only 3 wires used
- UART: Universal Asynchronous Receiver Transmitter
  - byte → serialize bits → wire → collect bits → byte



Terminals

- **56,000 baud = 56,000 bits per second = 7000 bytes / sec**
  - Each is an ASCII character code
- **Dumb terminals (CRTs / teletypes)**
  - Very few control characters
    - **newline, return, backspace**
- **Intelligent CRTs**
  - Also accept "escape sequences"
  - Reposition the cursor, clear the screen, insert lines, etc.
  - The standard "terminal interface" for computers
    - **Example programs: vi, emacs**

Input software

- **Character processing**
  - User types "`hella←o`"
  - Computer echoes as: "`hella←_←o`"
  - Program will see "hello"
- **Raw mode**
  - The driver delivers all characters to application
  - No modifications, no echoes
  - vi, emacs, the BLITZ emulator, password entry
- **Cooked mode**
  - The driver does echoing and processing of special chars.
  - "Canonical mode"

Special control characters (in "cooked mode")

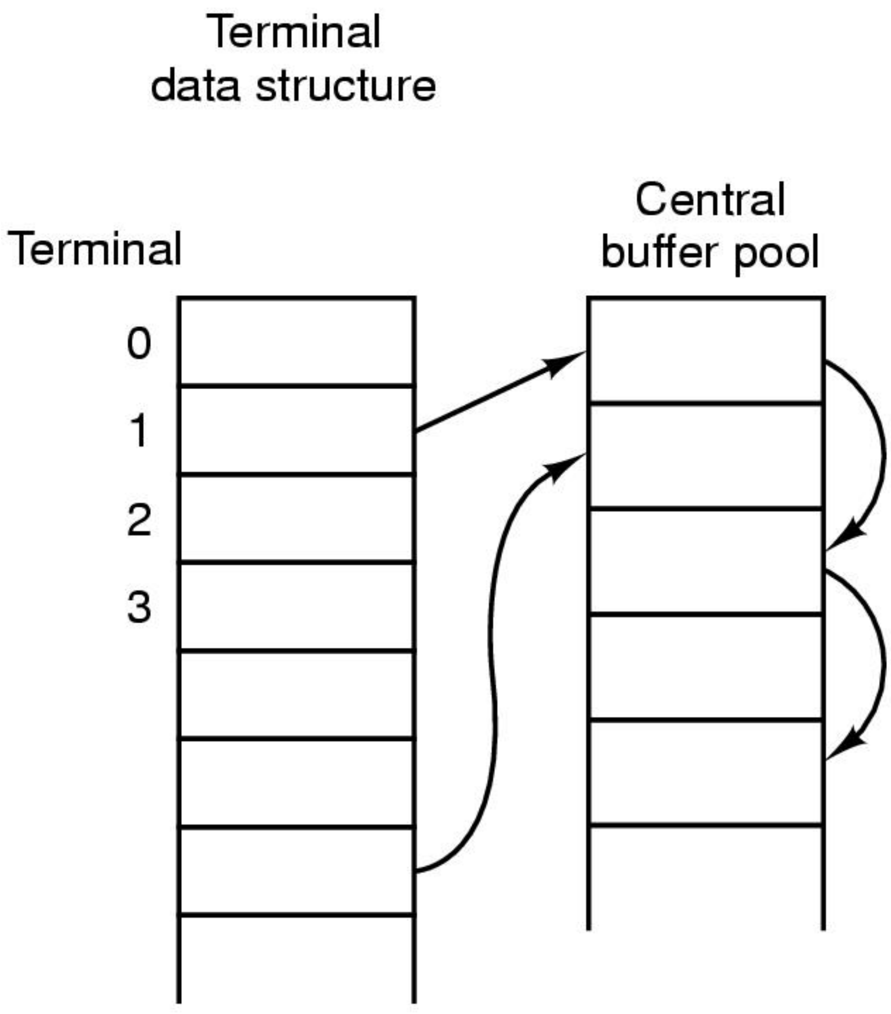
Character	POSIX name	Comment
CTRL-H	ERASE	Backspace one character
CTRL-U	KILL	Erase entire line being typed
CTRL-V	LNEXT	Interpret next character literally
CTRL-S	STOP	Stop output
CTRL-Q	START	Start output
DEL	INTR	Interrupt process (SIGINT)
CTRL-\	QUIT	Force core dump (SIGQUIT)
CTRL-D	EOF	End of file
CTRL-M	CR	Carriage return (unchangeable)
CTRL-J	NL	Linefeed (unchangeable)

Cooked mode

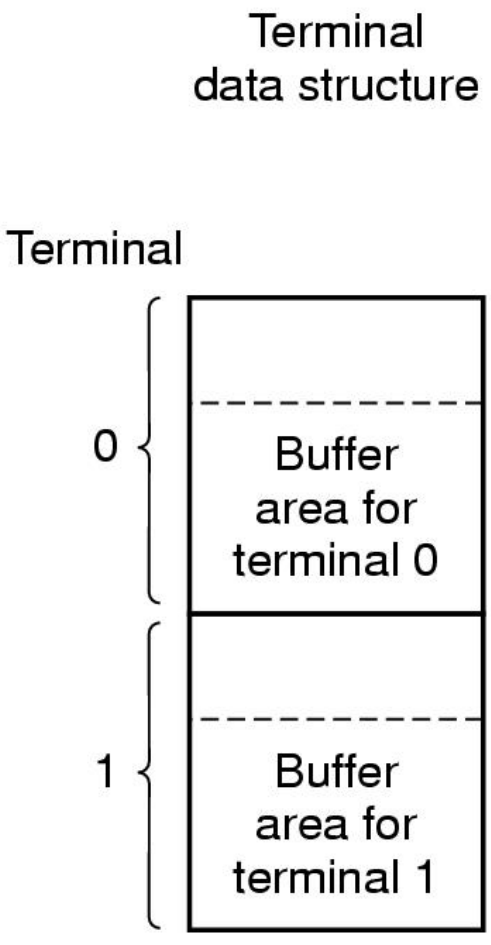
- **The terminal driver must...**
  - Buffer an entire line before returning to application
  - Process special control characters
    - **Control-C, Backspace, line-erase, tabs**
  - Echo the character just typed
  - Accommodate type-ahead
    - **Ie., it needs an internal buffer**

- **Approach 1 (for computers with many terminals)**
  - Have a pool of buffers to use as necessary
- **Approach 2 (for single-user computer)**
  - Have one buffer (e.g., 500 bytes) per terminal

Central buffer pool vs. dedicated buffers



(a)



(b)

The end-of-line problem

- **NL "newline" (ASCII 0x0A, \n)**
  - Move cursor down one line (no horizontal movement)
- **CR "return" (ASCII 0x0D, \r)**
  - Move cursor to column 1 (no vertical movement)
- **"ENTER key"**
  - Behavior depends on the terminal specs
    - **May send CR, may send NL, may send both**
    - **Software must be device independent**

- **Unix, Macintosh:**
  - Each line (in a file) ends with a NL
- **Windows:**
  - Each line (in a file) ends with CR & NL

Control-D: EOF

- **Typing Control-D ("End of file") causes the read request to be satisfied immediately**
  - Do not wait for "enter key"
  - Do not wait for any characters at all
  - May return 0 characters

- **Within the user program**

```
count = Read (fd, buffer, buffSize)
if count == 0
    -- Assume end-of-file reached...
```

Outputting to a terminal

- The terminal accepts an "escape sequence"
- Tells it to do something special

ESCAPE: 0x1B

Example:

$\underbrace{\texttt{esc [3 ; 1 H}}_{\text{Move to position (3,1) on screen}}\ \underbrace{\texttt{esc [0 K}}_{\text{Erase the line}}\ \underbrace{\texttt{esc [1 M}}_{\text{Shift following lines up one}}$

- Each terminal manufacturer had a slightly different specification
  - *Makes device independent software difficult*
  - Unix "termcap" file
    - Database of different terminals and their behaviors.

ANSI escape sequence standard

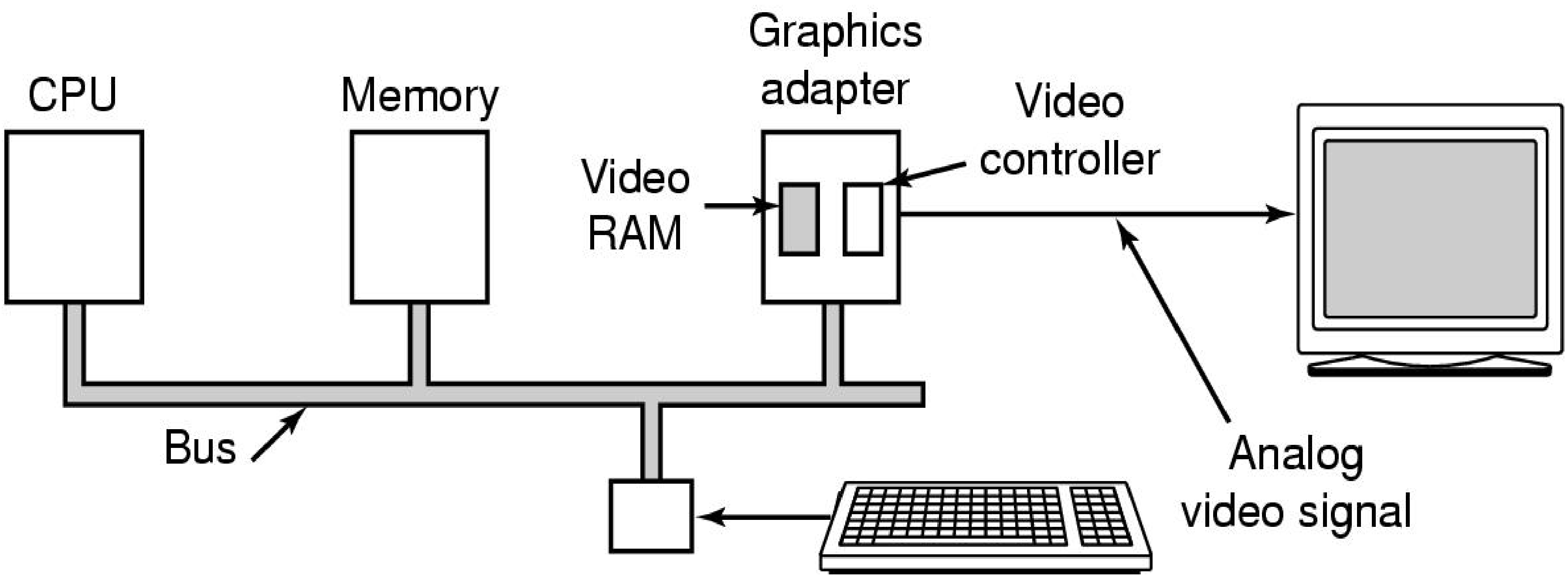
Escape sequence	Meaning
ESC [*n*A	Move up *n* lines
ESC [*n*B	Move down *n* lines
ESC [*n*C	Move right *n* spaces
ESC [*n*D	Move left *n* spaces
ESC [*m* ; *n*H	Move cursor to (*m*,*n*)
ESC [*s*J	Clear screen from cursor (0 to end, 1 from start, 2 all)
ESC [*s*K	Clear line from cursor (0 to end, 1 from start, 2 all)
ESC [*n*L	Insert *n* lines at cursor
ESC [*n*M	Delete *n* lines at cursor
ESC [*n*P	Delete *n* chars at cursor
ESC [*n*@	Insert *n* chars at cursor
ESC [*n*m	Enable rendition *n* (0=normal, 4=bold, 5=blinking, 7=reverse)
ESC M	Scroll the screen backward if the cursor is on the top line

Graphical user interfaces (GUIs)

- Memory-mapped displays “bit-mapped graphics”
- Video driver moves bits into special memory region
  - Changes appear on the screen
  - Video controller constantly scans video ram
- Black and white displays
  - 1 bit = 1 pixel
- Color
  - 24 bits = 3 bytes = 1 pixels
    - red (0-255)
    - green (0-255)
    - blue (0-255)

*1280 * 854 * 3*
= 3 MB

Graphical user interfaces (GUIs)



Spare slides

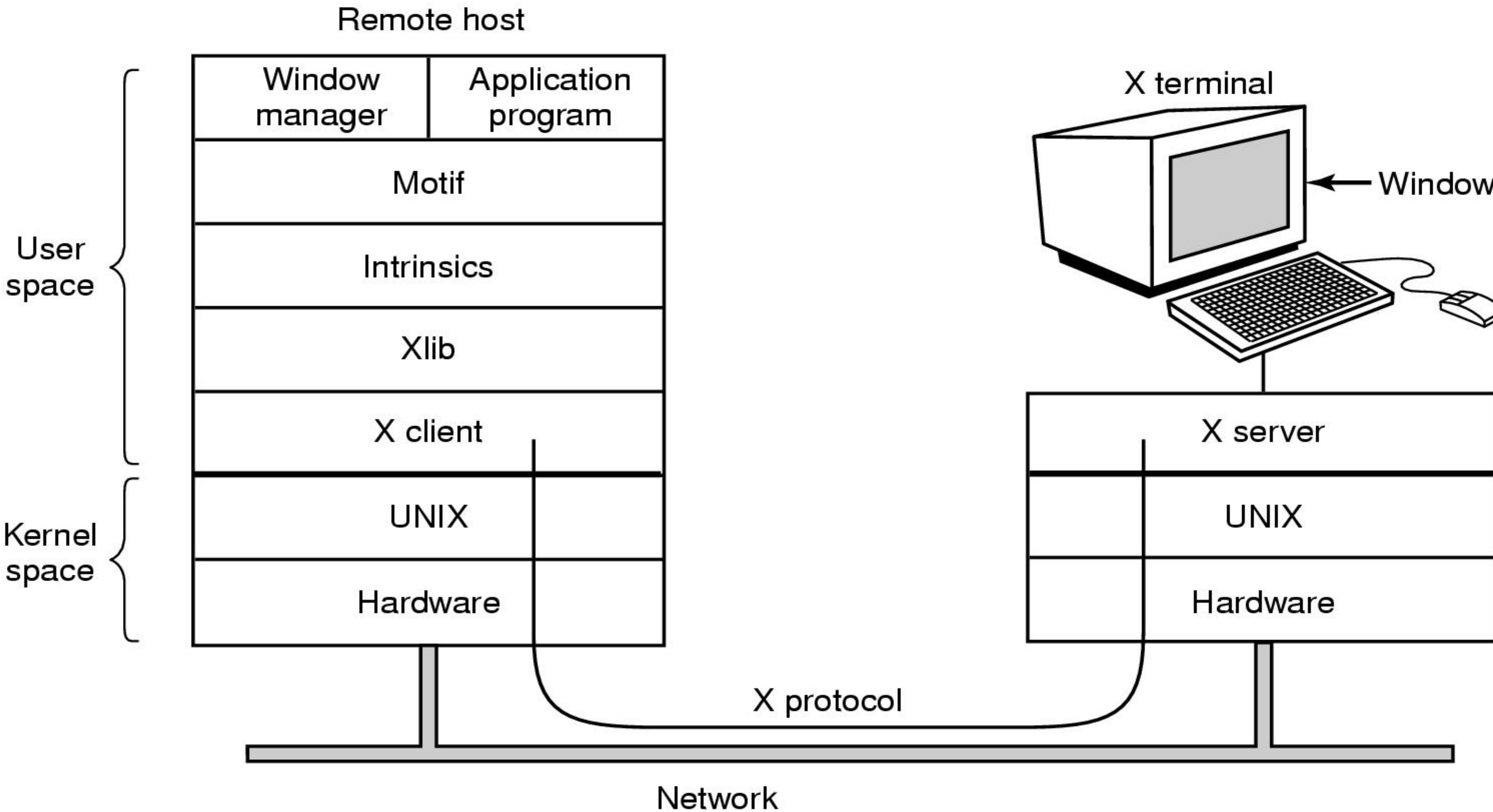
X Window System

- Client - Server
- Remote Procedure Calls (RPC)
- Client makes a call.
- Server is awakened; the procedure is executed.

- Intelligent terminals ("X terminals")

- The display side is the ***server***.
- The application side is the ***client***.
- The application (client) makes requests to the display server.
- Client and server are separate processes
  - (May be on the same or different machines)

X window system



X window system

- *X-Server*
  - Display text and geometric shapes, move bits
  - Collect mouse and keyboard status
- *X-Client*
  - Xlib
    - **library procedures; low-level access to X-Server**
  - Intrinsics
    - **provide "*widgets*"**
    - **buttons, scroll bars, frames, menus, etc.**
  - Motif
    - **provide a "look-and-feel" / style**
  - Window Manager
    - **Application independent functionality**
    - **Create & move windows**

The SLIM network terminal

- **Stateless Low-level Interface Machine (SLIM)**
  - Sun Microsystems
- ***Philosophy: Keep the terminal-side very simple!***
- **Back to “dumb” terminals”**
- **Interface to X-Server:**
  - 100’s of functions
- **SLIM:**
  - Just a few messages
  - The host tells which pixels to put where
  - The host contains all the intelligence

The SLIM network terminal

- **The SLIM Protocol**
  - from application-side (server)
  - to terminal (the “thin” client)

Message	Meaning
SET	Update a rectangle with new pixels
FILL	Fill a rectangle with one pixel value
BITMAP	Expand a bitmap to fill a rectangle
COPY	Copy a rectangle from one part of the frame buffer to another
CSCS	Convert a rectangle from television color (YUV) to RGB