# Reminders:–

- **Project 4 due *Friday, February 24***
  - Late penalties described in syllabus on InstructAssist

- **Any project submitted after 4 PM on Tuesday, Feb 27 *will not be graded!***

- **Any project not demonstrated by 6 PM on Thursday, March 1, *will not be graded***

- **Schedule your demos ASAP!**
  - If not enough slots, be sure to ask for more *early!*

# Input and Output

Professor Hugh C. Lauer

CS-3013 — Operating Systems

**While a typical Linux or Windows distribution includes support for > 50 file systems …**

**… these systems are called upon to support 1000's of I/O devices and subsystems**

# The I/O subsystem

- The largest, most complex subsystem in an OS
- Most lines of code
- Highest rate of code changes
- Where OS engineers most likely to work
- Difficult to test thoroughly

- Make-or-break issue for any system
  - Big impact on performance and perception
  - Bigger impact on acceptability in market

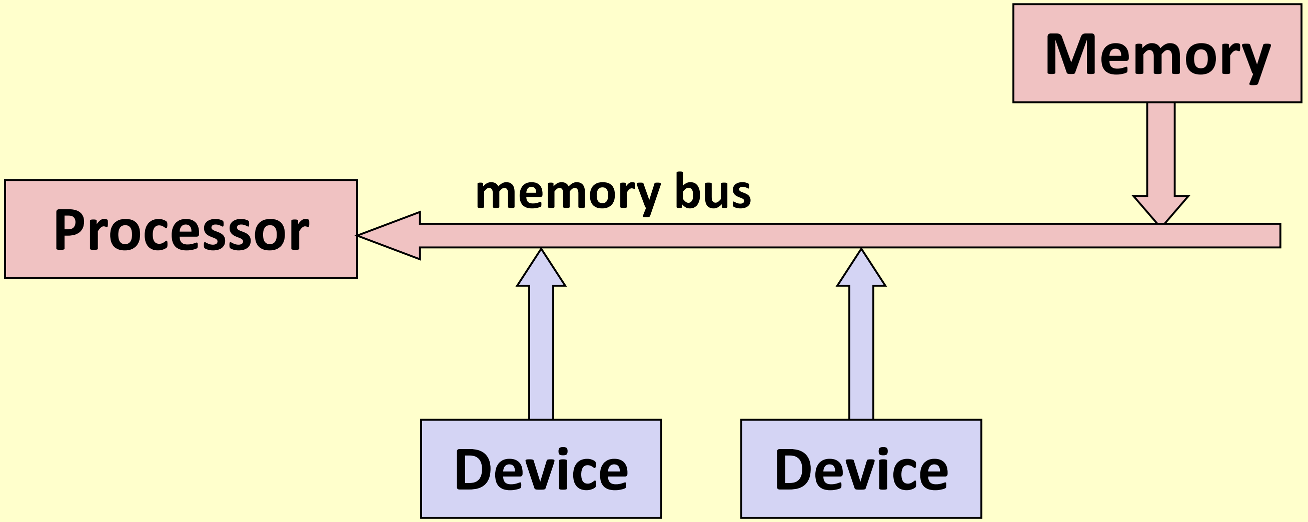
# Overview

- What is I/O?
- Principles of I/O hardware
- Principles of I/O software
- Methods of implementing input-output activities
- Organization of device drivers
- Specific kinds of devices

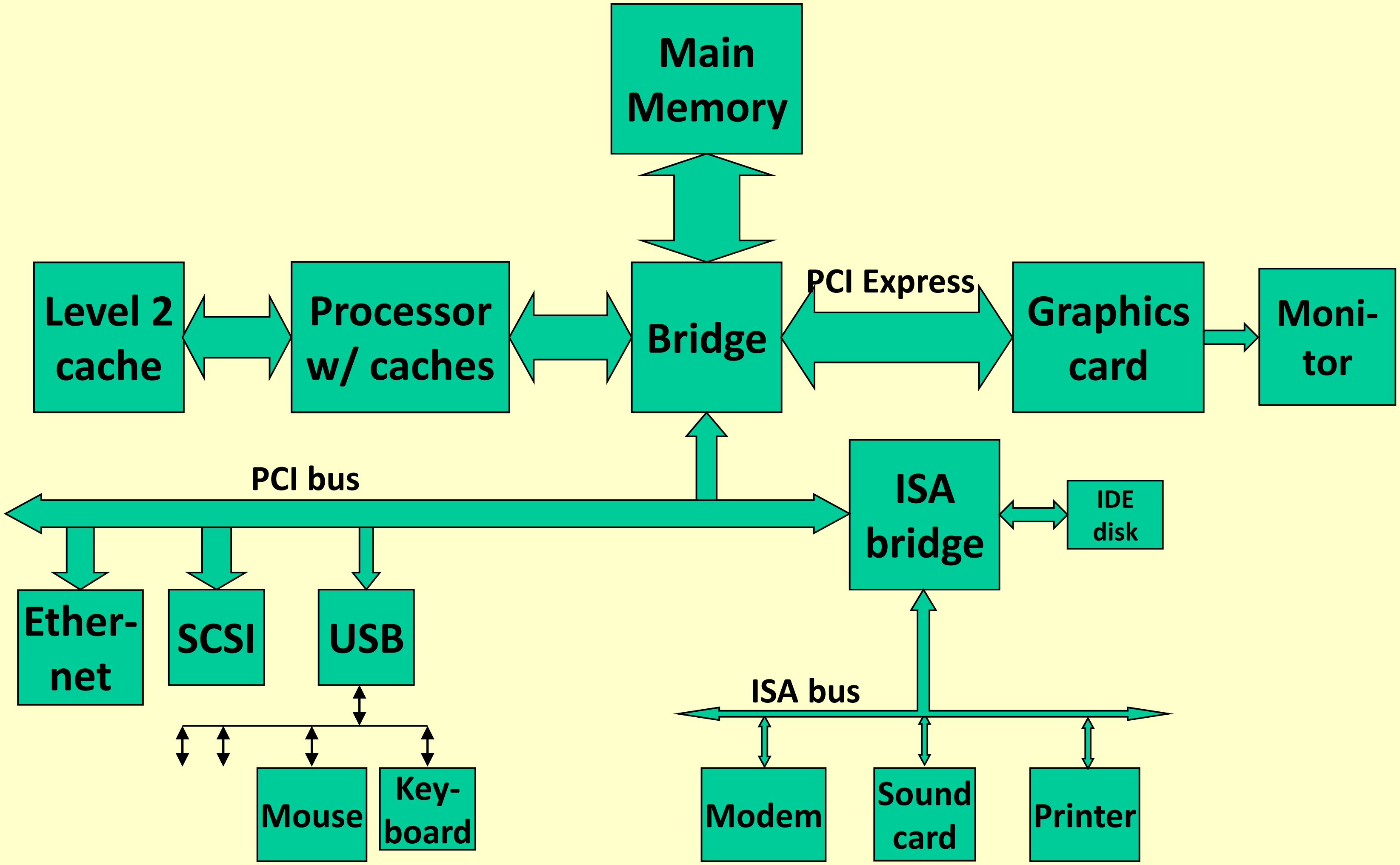
(OSTEP, §36-38)

- Treats disks as I/O devices rather than repositories for files

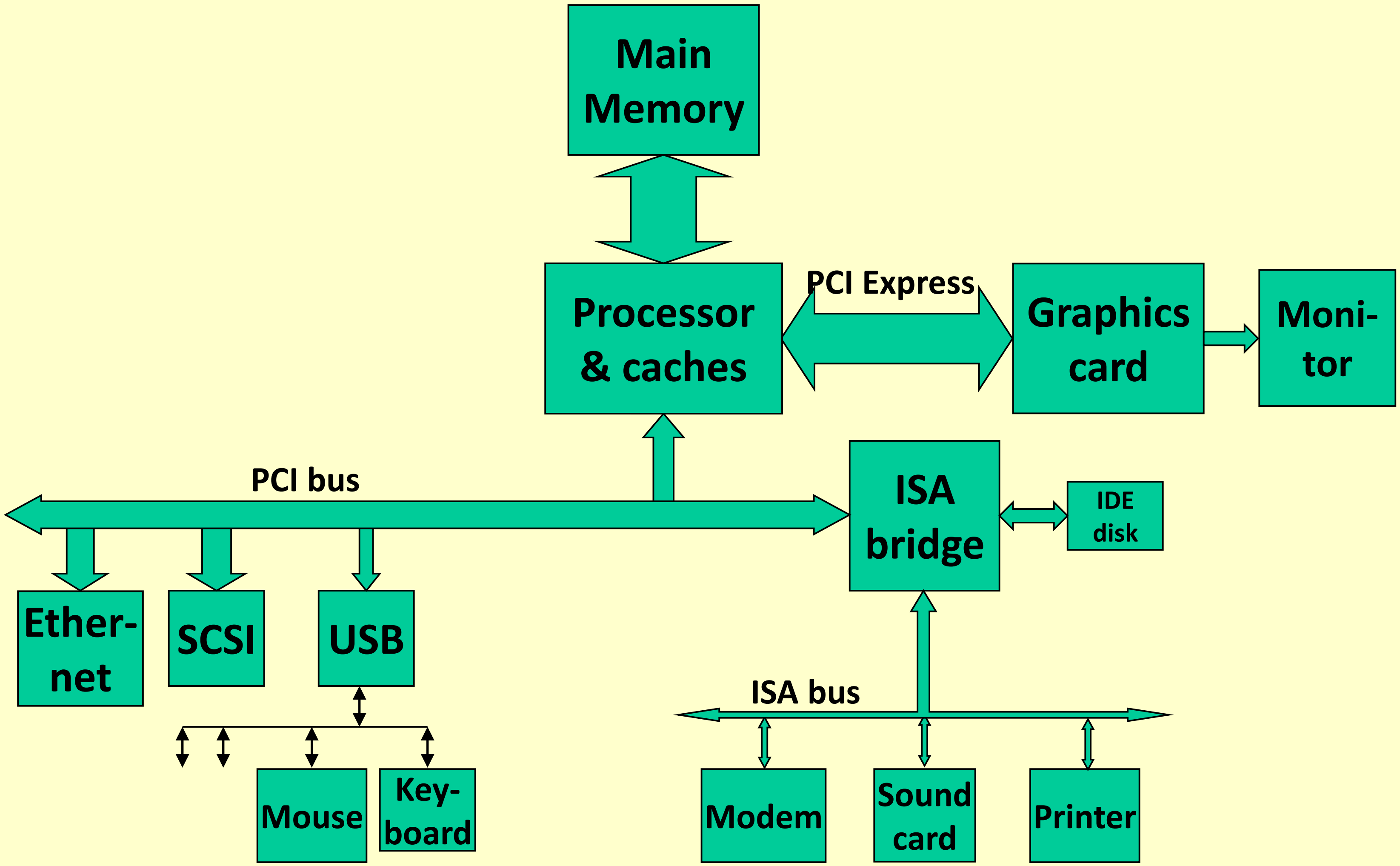
# Hardware organization (simple, naïve)



# Hardware organization (PC not long ago)



# Hardware organization (more recent PC)



# Kinds of I/O devices

- **Character (and sub-character) devices**
  - Mouse, character terminal, joy stick, some keyboards
- **Block transfer**
  - Disk, tape, CD, DVD
  - Network
- **Clocks**
  - Internal, external
- **Graphics**
  - GUI, games
- **Multimedia**
  - Audio, video
- **Other**
  - Sensors, cameras, controllers, touch screens, etc.

# Controlling an I/O device

- **A function of host Processor architecture**
  - Two approaches:– Special instructions vs. memory-mapped
- **Special I/O instructions**
  - Opcode to stop, start, query, etc.
  - Separate I/O address space
  - Kernel mode only
- **Memory-mapped I/O control registers**
  - Each register has a physical memory address
  - Writing to data register is output
  - Reading from data register is input
  - Writing to control register causes action
  - Can be mapped to kernel or user-level virtual memory

# Character device (example)

- ***Data register:***
  - Register or address where data is read from or written to
  - Very limited capacity (at most a few bytes)
- ***Action register:***
  - When writing to register, causes a physical action
  - Reading from register yields zero
- ***Status register:***
  - Reading from register provides information
  - Writing to register is no-op

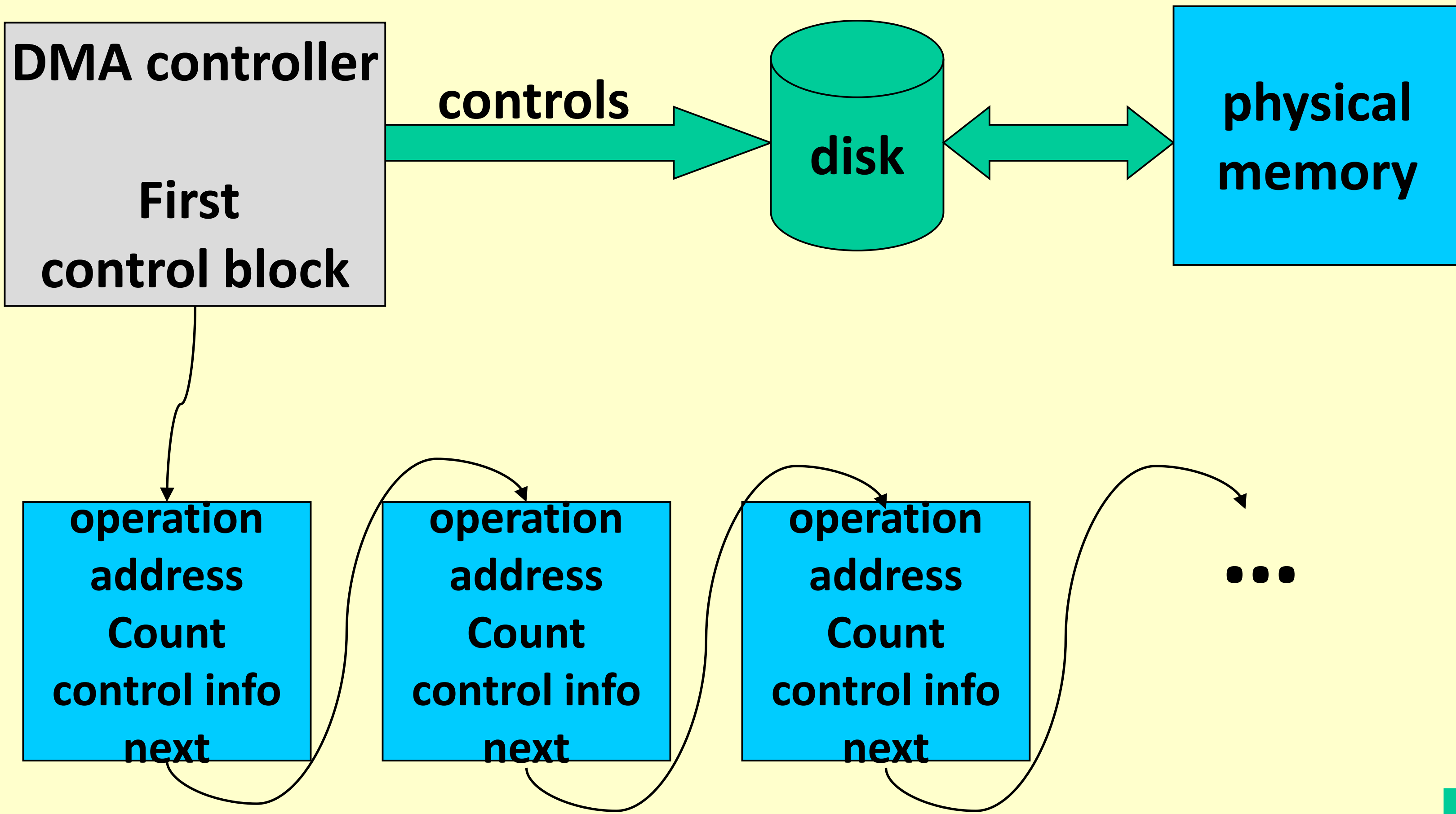
# Block transfer device (example)

- ***Buffer address register:***
  - Points to area in *physical* memory to read or write data

    *or*
- **Addressable *buffer* for data**
  - E.g., network cards, modern hard drives **Keyboard/mouse**
- ***Action register:***
  - When writing to register, initiates a physical action or data transfer
  - Reading from register yields zero
- ***Status register:***
  - Reading from register provides information
  - Writing to register is no-op

# DMA (Direct Memory Access)

- **Block devices to autonomously read from and/or write to main memory**
  - (Usually) physical addresses
  - Compete with processor(s) for access to bus, memory
- **Transfer address**
  - Points to location in physical memory
- **Action register:**
  - Initiates a reading of control block chain to start actions
- **Status register:**
  - Reading from register provides information


DMA controller
First
control block
controls
disk
physical
memory
operation
address
Count
control info
next
operation
address
Count
control info
next
operation
address
Count
control info
next
…

# Questions?

# Overview

- What is I/O?
- Principles of I/O hardware
- Principles of I/O software
- Methods of implementing input-output activities
- Organization of device drivers
- Specific kinds of devices

(OSTEP, §36-38)

# Requirements of I/O software

- ***Efficiency* – Do not allow I/O operations to become system bottleneck**
  - Especially slow devices
- ***Device independence* – isolate OS and application programs from device specific details and peculiarities**
- ***Uniform naming* – support a way of naming devices that is scalable and consistent**
- ***Error handling* – isolate the impact of device errors, retry where possible, provide uniform error codes**
  - Errors are abundant in I/O
- **…**

# Principles of I/O software (continued)

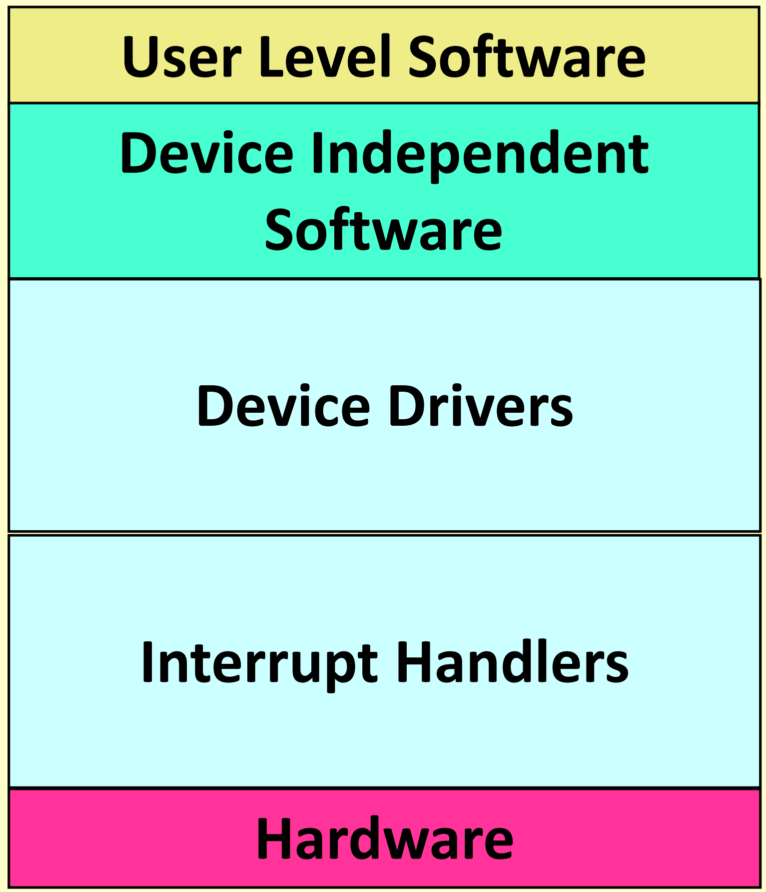
- *…*
- *Buffering* – provide uniform methods for storing and copying data between physical memory and the devices
- *Uniform data transfer modes* – synchronous and asynchronous, read, write, ..
- *Controlled device access* – sharing and transfer modes
- *Uniform driver support* – specify interfaces and protocols that drivers must adhere to

# I/O software "stack"

I/O API & libraries

Device Dependent – schedulable

Device Dependent – as quick as possible




(Rest of the OS)

# Three common ways of I/O

- Programmed I/O
- Interrupt-Driven I/O
- I/O using DMA

# Programmed I/O (polling)

- **Used when device and controller are relatively quick to process an I/O operation**
- **Device driver**
  - Gains access to device
  - Initiates I/O operation
  - Loops testing for completion of I/O operation (busy wait)
  - If there are more I/O operations, repeat
- **Used in following kinds of cases**
  - Service interrupt time > Device response time
  - Device has no interrupt capability
  - Embedded systems where Processor has nothing else to do

# Programmed I/O example — bitmapped keyboard & mouse

- **Keyboard & mouse buttons implemented as 128-bit read-only register**
  - One bit for each key and mouse button
  - *0 = "up"; 1 = "down"*
- **Mouse position implemented as pair of counters**
  - One increment per unit of motion in each of *x* and *y* directions
- **Periodic clock interrupt (e.g., every 10 msec)**
  - Reads keyboard register, compares to previous copy
  - Determines key & button transitions up or down
  - Decodes transition stream to form character and button sequence
  - Reads and compares mouse counters to form motion sequence

**Advantage:– key positions & mouse behavior are programmed entirely in software**

# Other programmed I/O examples

- **Check status of device**
- **Read from disk or boot device at boot time**
  - No OS present, hence no interrupt handlers
  - Needed for bootstrap loading of the inner portions of kernel
- **External sensors or controllers**
  - Real-time control systems

# Questions about programmed I/O?

# Interrupt-driven I/O

- **Interrupts occur on I/O events**
  - operation completion
  - Error or change of status
- **Data transferred by interrupt handler**
  - Reading or writing device registers

**Processor participates in *every* byte or word transferred!**

# Interrupt request lines (IRQs)

- **Every device is assigned an *IRQ***
  - Used when raising an interrupt
  - Interrupt handler can identify the interrupting device

- **Assigning IRQs**
  - In older and simpler hardware, physical wires and contacts on device or bus
  - In most modern PCs, etc., assigned dynamically at boot time

# Handling interrupts (Linux style)

- **Terminology**
  - *Interrupt context* – kernel operating not on behalf of any process
  - *Process context* – kernel operating on behalf of a particular process
  - *User context* – process executing in user-space virtual memory

- ***Interrupt Service Routine* (ISR), also called *Interrupt Handler***
  - The function that is invoked when an interrupt is raised
  - Identified by IRQ
  - Operates on *Interrupt stack* (as of Linux kernel 2.6)
    - *One* interrupt stack per processor or core
    - Approx 4-8 kbytes
    - *All handlers share same stack on processor!*

- **…**

# Handling interrupts (Linux style – continued)

Definitions!

- …
- ***Top half*** **– does minimal, time-critical work necessary**
  - Acknowledge interrupt, reset device, copy buffer or registers, etc.
  - Interrupts (usually) disabled on current processor
- ***Bottom half*** **– the part of the ISR that can be deferred to more convenient time**
  - Completes I/O processing; does most of the work
  - Interrupts enabled (usually)
  - Communicates with processes
  - Possibly in a kernel thread (or even a user thread!)

# Interrupt-driven I/O example — software time-of-day clock

- **Interrupt occurs at fixed intervals**
  - 50 or 60 Hz
  - 1 kHz in more modern processors
- **Service routine (*top half*):–**
  - Adds one *tick* to clock counter
- **Service routine (*bottom half*):–**
  - Checks list of *soft timers*
  - Notifies tasks of any expired timers

**Note that this looks a lot like programmed I/O**

- **Except for bottom-half processing**

# Other interrupt-driven I/O examples

- **Very "slow" character-at-a-time terminals**
  - Mechanical printers (15 characters/second)
  - Some keyboards (one character/keystroke)
    - Command-line completion in many Unix systems
  - Game consoles
  - Serial modems
  - Many I/O devices in "old" computers
    - Paper tape, punched cards, etc.

# Programmed I/O *vs.* interrupt-driven I/O

- **Programmed I/O**
  - A process or thread pro-actively works the device, gets or puts the data, does everything else.

- **Interrupt-driven I/O**
  - Device operates autonomously, let's processor know when it is done or ready

- **Both**
  - *Processor participates in transfer of every byte or word.*

# DMA interrupt handler

**Device transfers data itself**

- **Service Routine – *top half* (interrupts disabled)**
  - Does as little work as possible and returns
    - (Mostly) notices completion of one transfer, starts another
    - (Occasionally) checks for status
    - Setup for more processing in bottom half

- **Service Routine – *bottom half* (interrupts enabled)**
  - Compiles control blocks from I/O requests
  - Manages & pins buffers, translates virtual to physical addresses
  - Posts completion of transfers to requesting applications
    - Unpin and/or release buffers
  - Possibly in a kernel thread

# DMA examples

- Disks, CD-ROM readers, DVD readers
- Ethernet & wireless "modems"
- Tape and bulk storage devices
- Common themes:–
  - Device controller has space to buffer a (big) block of data
  - Controller has intelligence to update physical addresses and transfer data
  - Controller (often) has intelligence to interpret a sequence of control blocks without processor help
- ***Processor does not touch data during transfer!***

**Not GUIs**

# Buffering

- **DMA devices need memory to read from, write to**
  - Must be contiguous pages
  - (Usually) physical addresses
- ***Double buffering***
  - One being filled (or emptied) by device
  - Other being emptied (or filled) by application
  - Special case of producer-consumer with *n = 2*

**Producer-Consumer:– see OSTEP §30.2**

# Digression:– error detection and correction

- **Most data storage and network devices have hardware error detection and correction**
- **Redundancy code added during writing**
  - Parity: detects 1-bit errors, not 2-bit errors
  - Hamming codes
    - Corrects 1-bit errors, detects 2-bit errors
  - Cyclic redundancy check (CRC)
    - Detects errors in string of 16- or 32-bits
    - Reduces probability of undetected errors to very, very low
- **Check during reading**
  - Report error to device driver
- **Error recovery:– one of principal responsibilities of a device driver!**

# Questions?

# Overview

- What is I/O?
- Principles of I/O hardware
- Principles of I/O software
- Methods of implementing input-output activities
- **Organization of device drivers**
- Specific kinds of devices

(OSTEP, §36-38)

# Device Drivers

- Organization
- Uniform interfaces to OS
- Uniform buffering strategies
- Hide device idiosyncrasies

# Device Drivers

- **Device Drivers are dependent on both the OS & device**
- ***OS dependence***
  - Meet the interface specs of the device independent layer
  - Use the facilities supplied by the OS – buffers, error codes, etc.
  - Accept and execute OS commands – e.g. *read*, *open*, etc.
- ***Device Dependent***
  - Actions during Interrupt Service routine
  - Translate OS commands into device operations
    - E.g read block *n* becomes a series of setting and clearing and interpreting device registers or interfaces
  - Note that some device drivers have layers
    - Strategy or policy part to optimize arm movement or do retries; plus a mechanism part the executes the operations

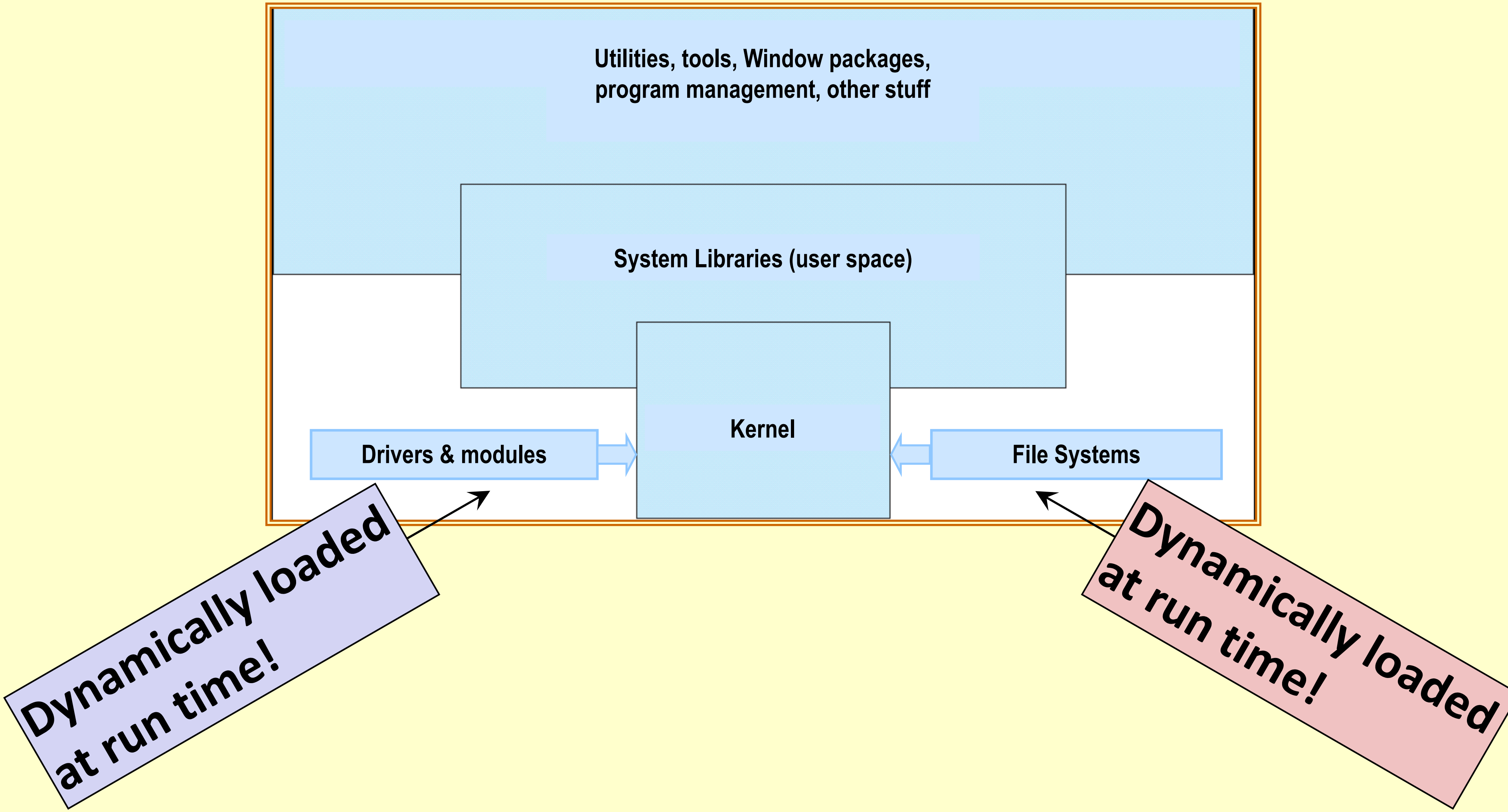
# OS Responsibility to Device Driver

- **Uniform API**
  - *Open*, *Close*, *Read*, *Write*, *Seek* functions
  - **`ioctl`** function as escape mechanism
- **Buffering**
  - Kernel functions for allocating, freeing, mapping, pinning buffers
- **Uniform naming**
  - **`/dev/(type)(unit)`**
    - type defines driver; unit says which device
- **Other**
  - Assign interrupt level (IRQ)
  - Protection (accessibility by application, user-space routines)
  - Error reporting mechanism

# Abstract Overview

- **Think of I/O subsystem as a C++ or Java-style abstract class**
  - Uniform interface in form of specific operations (methods) and services
  - Uniform state information
- **Each I/O driver implements a subclass**
  - Own methods for uniform interface
  - Additional state info & methods for specific needs
- **However, no Java or *C++* support in kernel**
  - Must implement everything in long-hand in C
  - Just like with file systems!

# Operating System Organization



# Installing Device Drivers

- **Classic Unix**
  - Create and compile driver to *.o* file
  - Edit and re-compile device table to add new device
  - Re-link with *.o* files for OS kernel ⇒ new boot file
- **Classic Macintosh**
  - Submit to Apple for verification, approval, and inclusion
- **MS-DOS and Windows**
  - Dynamic driver loading and installation
  - Special driver-level debuggers available; *open device environment*
  - Certification program for trademarking
- **Linux**
  - Dynamic driver loading and installation
  - *Open device environment*

**Reason why Windows won battle of the desktop**

# Dynamic Device Configuration

- **At boot time:–**
  1. Probe hardware for inventory of devices & addresses
  2. Map devices to drivers (using table previously created)
  3. Load necessary drivers into kernel space, register in interrupt vector (.sys files in Windows)
- **Run time:–**
  1. Detect interrupt from newly added device
  2. Search for driver, or ask user; add to table
  3. Load into kernel space, register in interrupt vector

# Allocating and Releasing Devices

- **Some devices can only be used by one application at a time**
  - CD-ROM recorders
  - GUI interface
- **Allocated at *Open*() time**
- **Freed at *Close*() time**

# Overview

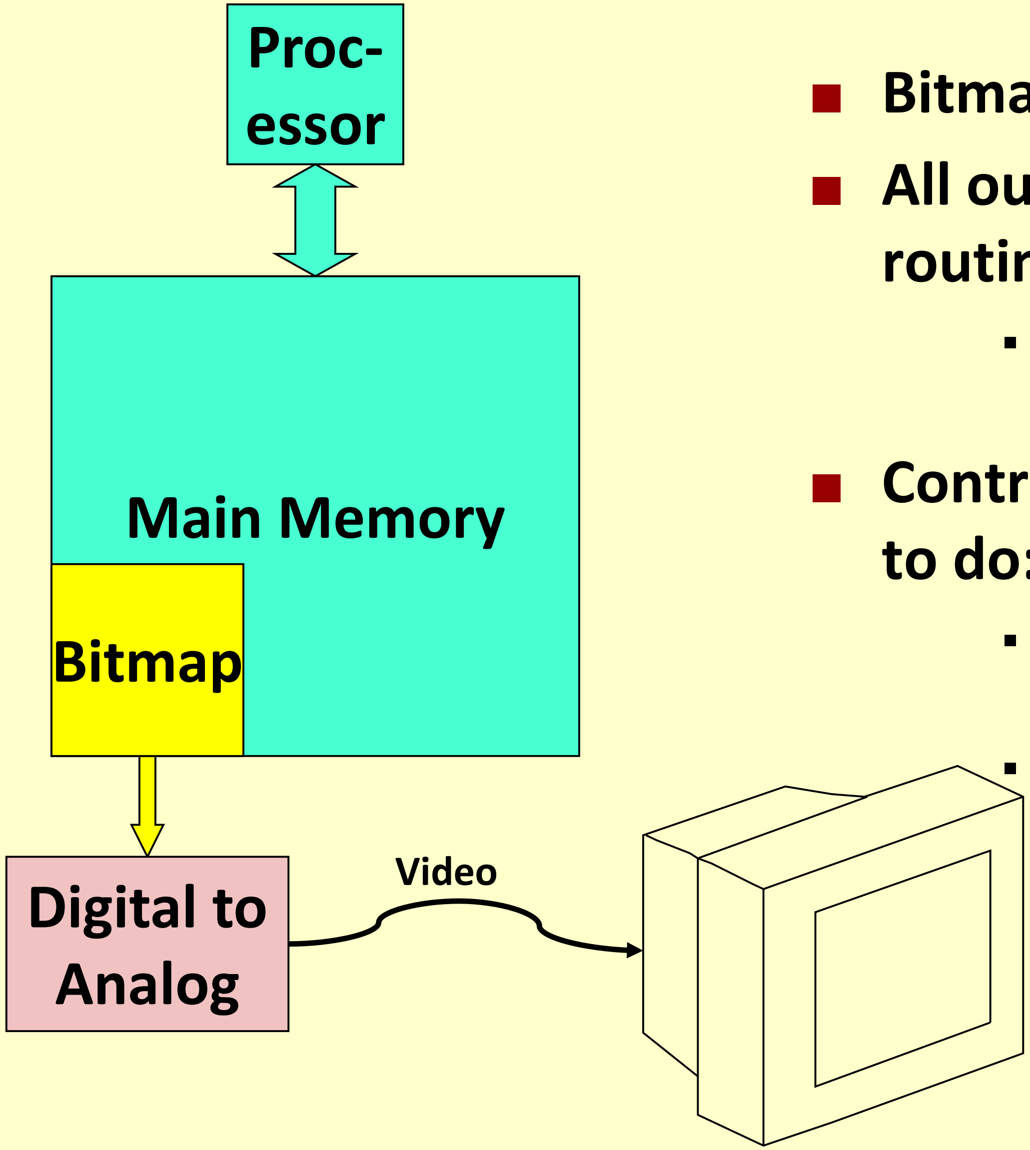
- What is I/O?
- Principles of I/O hardware
- Principles of I/O software
- Methods of implementing input-output activities
- Organization of device drivers
- **Specific kinds of devices**

(OSTEP, §36-38)

# A special kind of device —graphical user interface

- ***aka*, the *bitmapped* display**
  - In IBM language:– "all points addressable"
- **300K *pixels* to 2M *pixels***
- **Each pixel may be separated written**
- **Collectively, they create**
  - Windows
  - Graphics
  - Images
  - Videos
  - Games

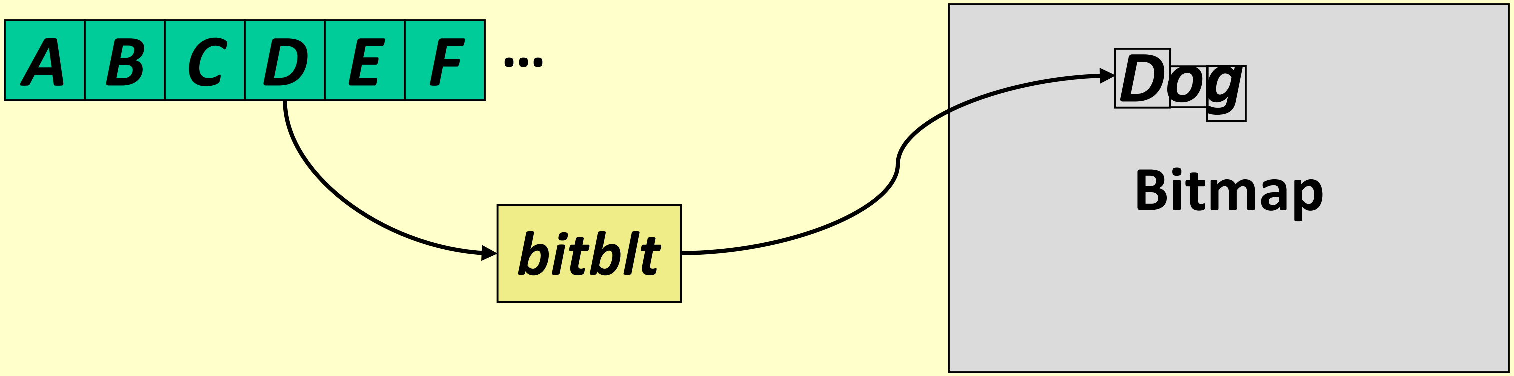
# GUI Device — early days




- Bitmap in main memory
- All output via library routines to bitmap
  - Entirely (or mostly) in user space
- Controller, an automaton to do:–
  - D-A conversion (digital to analog video)
  - 60+ Hz refresh rate
  - “clock” interrupt at top of each frame

# GUI Device — Displaying Text

- **Font: an array of bitmaps, one per character**
  - Designed to be pleasing to eye
- ***bitblt:* (*Bit*-oriented *Bl*ock *T*ransfer)**
  - An operation to copy a rectangular array of pixels from one bitmap to another



# GUI Device – Color

- ***Monochrome:* one bit per pixel**
    - *foreground* vs. *background*
- ***Color:* 2-32 bits per pixel**
- ***Direct* vs. *Color palette***
    - *Direct:* (usually) 8 bits each per Red, Green, Blue
    - *Palette:* a table of length $2^p$, for $p$-bit pixels

      Each entry (usually) 8 bits each for RGB

# GUI Device – Cursor

- **A small bitmap to overlay main bitmap**
- **Hardware support**
  - Substitute cursor bits during each frame
- **Software implementation**
  - *Bitblt* area under cursor to temporary bitmap
  - *Bitblt* cursor bitmap to main bitmap
  - Restore area under cursor from temporary bitmap
- **Very, very tricky!**
  - Timing is critical for smooth appearance
  - Best with double-buffered main bitmap

# GUI Device – Window

- **A virtual bitmap**
  - *size*, *position*, *clipping boundaries*
  - *font*, *foreground and background colors*
  - A list of operations needed to redraw contents
- **Operations to window itself:–**
  - *write*(), *refresh*()

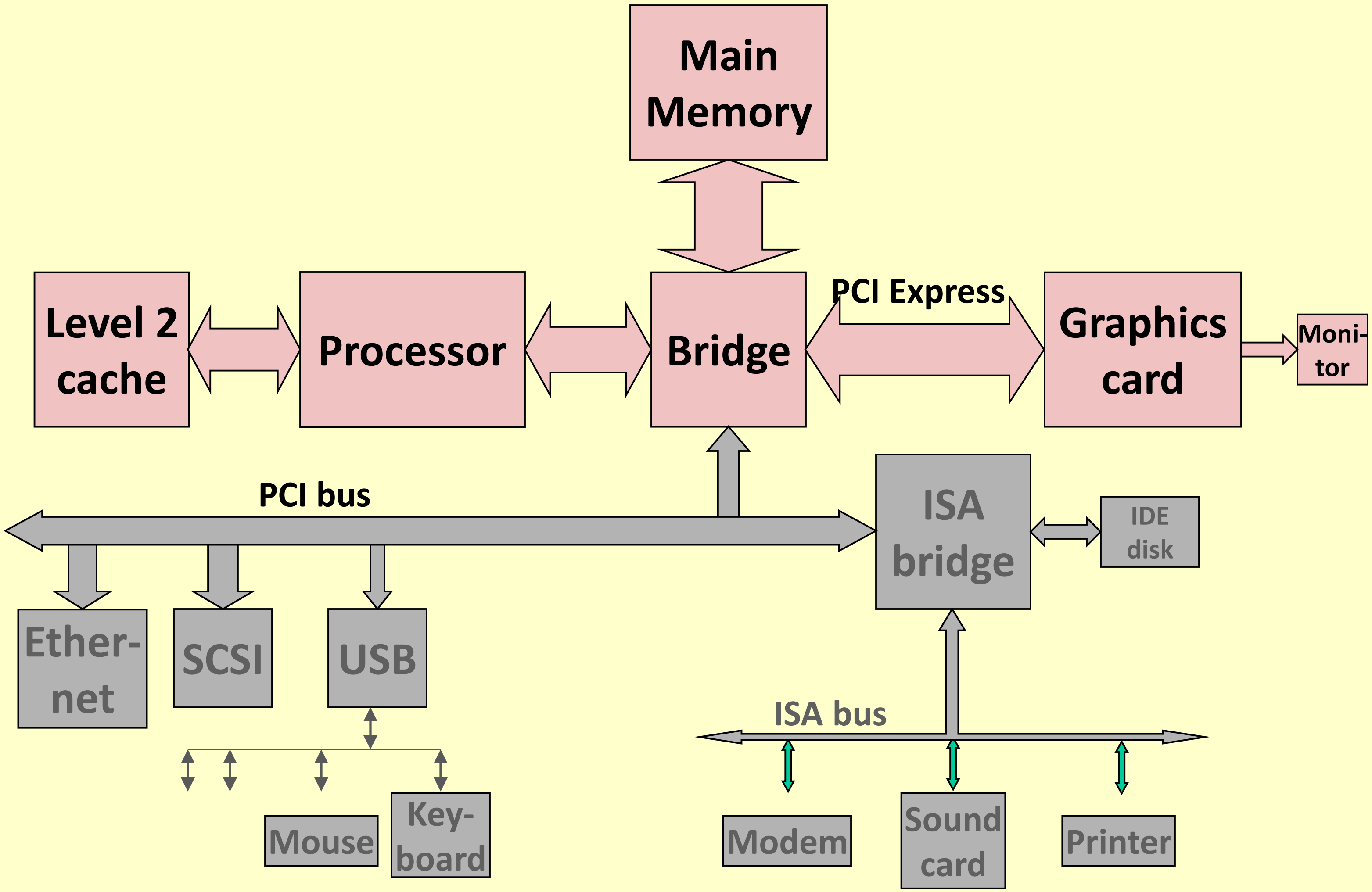
**Called by application to add/change information**

**Called by window manager to redraw current contents**

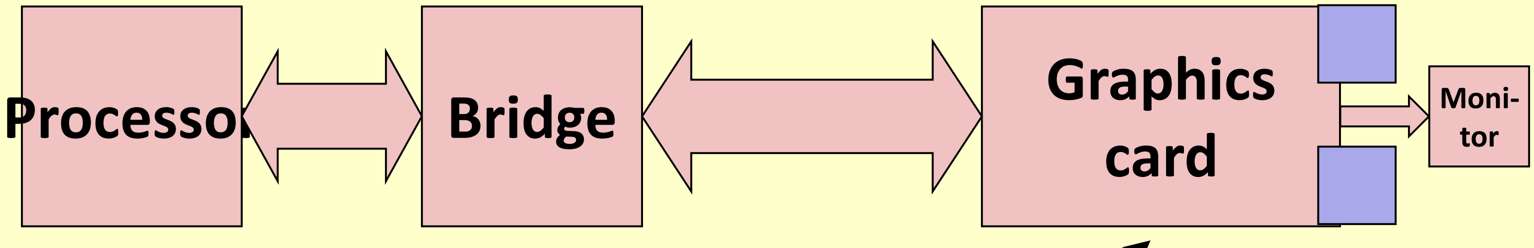
# GUI Device — Text Window

- **Character terminal emulated in a window**
    - RS-232 character set and controls
    - */dev/tty*
- **Operates like a character terminal with visible, partially obscured, or completely covered**

# Modern GUI devices



# Modern GUI Devices (continued)




- **Double-buffered bitmap in Graphics card**
  - Graphics and information written/drawn in *back* buffer
  - Monitor refreshes from main buffer (60+ Hz)
- ***Refresh* interrupt at start of every frame**
  - *Bitblt* to substitute cursor
- **Processor writes text, etc.**
- **Graphics processor (GPU) draws images, vectors, polygons**
- **Window manager orders redraw when necessary**

# Questions?

Reading Assignment

- OSTEP §37-39