CSE150
Operating Systems
Lecture 18

I/O Sytems

Demand Paging (Review)

- Replacement policies
  - FIFO: Place pages on queue, replace page at end
  - MIN: Replace page that will be used farthest in future
  - LRU: Replace page used farthest in past
- Clock Algorithm: Approximation to LRU
  - Arrange all pages in circular list
  - Sweep through them, marking as not “in use”
  - If page not “in use” for one pass, than can replace
- N^{th}-chance clock algorithm: Another approx LRU
  - Give pages multiple passes of clock hand before replacing
- Second-Chance List algorithm: Yet another approx LRU
  - Divide pages into two groups, one of which is truly LRU and managed on page faults.
- Thrashing: a process is busy swapping pages in and out
  - Process will thrash if working set doesn’t fit in memory
  - Need to swap out a process
- Working Set:
  - Set of pages touched by a process recently

How to get from Kernel→User

- What does the kernel do to create a new user process?
  - Allocate and initialize process control block
  - Read program off disk and store in memory
  - Allocate and initialize translation map
    - » Point at code in memory so program can execute
    - » Possibly point at statically initialized data
  - Run Program:
    - » Set machine registers
    - » Set hardware pointer to translation table
    - » Set processor status word for user mode
    - » Jump to start of program
- How does kernel switch between processes?
  - Same saving/restoring of registers as before
  - Save/restore hardware pointer to translation map

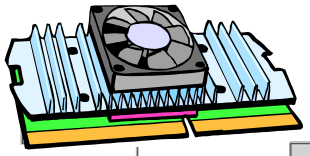
User→Kernel (Exceptions: Traps and Interrupts)

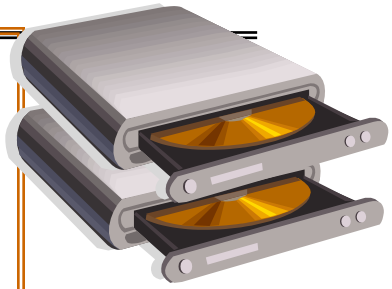
- System call instr. causes a synchronous exception (or “trap”)
  - In fact, often called a software “trap” instruction
- Other sources of *Synchronous Exceptions:*
  - Divide by zero, Illegal instruction, Bus error (bad address, e.g. unaligned access)
  - Segmentation Fault (address out of range)
  - Page Fault
- Interrupts are *Asynchronous Exceptions*
  - Examples: timer, disk ready, network, etc….
  - Interrupts can be disabled, traps cannot!
- SUMMARY – On system call, exception, or interrupt:
  - Hardware enters kernel mode with interrupts disabled
  - Saves PC, then jumps to appropriate handler in kernel
  - For some processors (x86), processor also saves registers, changes stack, etc.

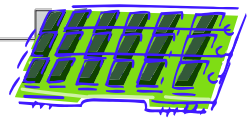
Today

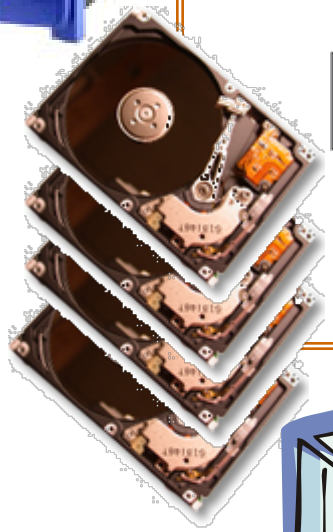
- I/O Systems
  - Hardware Access
  - Device Drivers

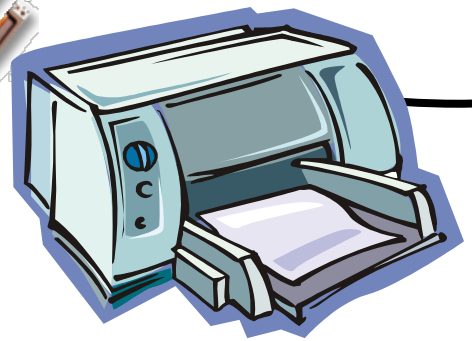
Modern I/O Systems

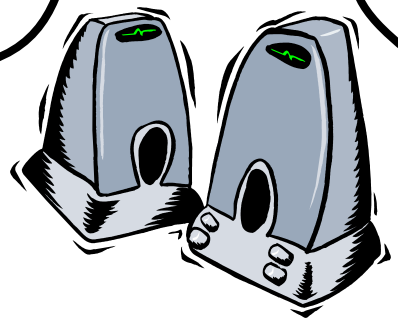












monitor

graphics controller

cache

bridge/memory controller

SCSI bus

disk

disk

disk

disk

SCSI controller

PCI bus

IDE disk controller

disk

disk

disk

disk

expansion bus interface

keyboard

expansion bus

parallel port

serial port

What is the Role of I/O?

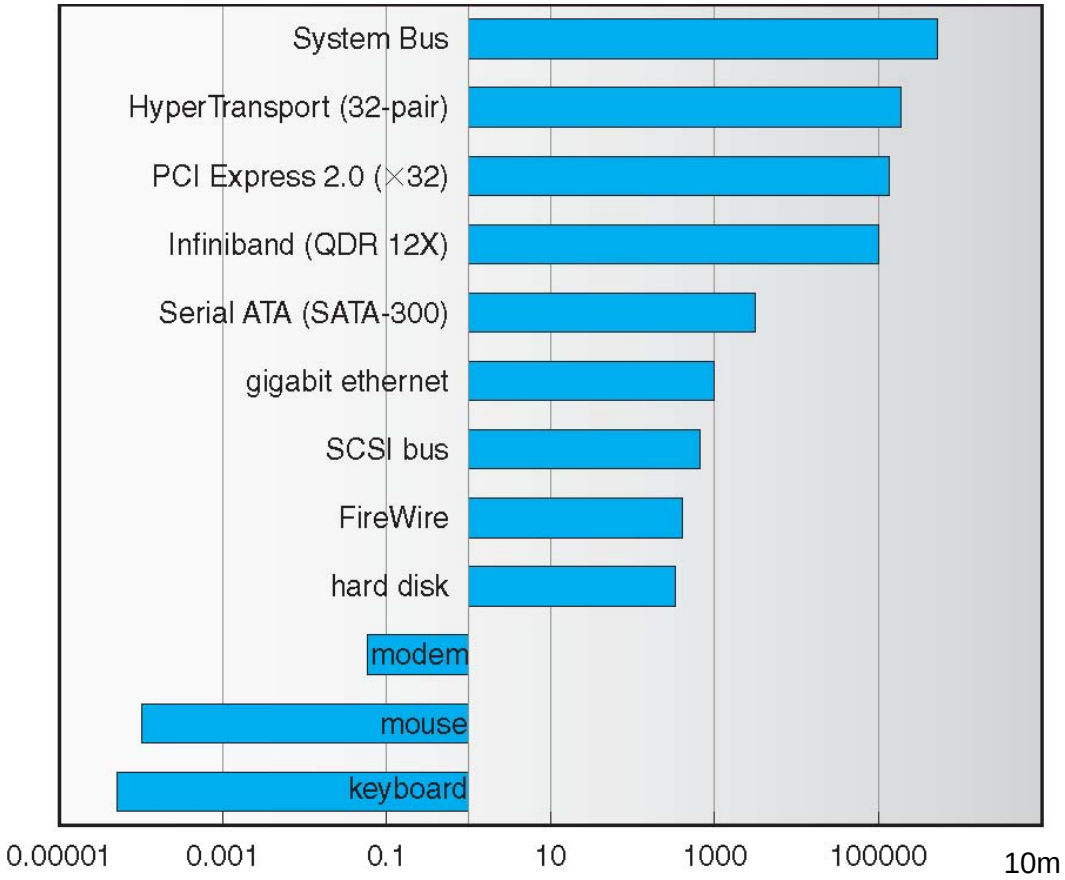
- Without I/O, computers are useless (disembodied brains?)
- But… thousands of devices, each slightly different
  - How can we standardize the interfaces to these devices?
- Devices unreliable: media failures and transmission errors
  - How can we make them reliable???
- Devices unpredictable and/or slow
  - How can we manage them if we don't know what they will do or how they will perform?

Operational Parameters for I/O

- Data granularity: Byte vs. Block
  - Some devices provide single byte at a time (*e.g.,* keyboard)
  - Others provide whole blocks (*e.g.,* disks, networks, etc.)

- Transfer mechanism: Polling vs. Interrupts
  - Some devices require continual monitoring
  - Others generate interrupts when they need service

E.g. Device-Transfer Rates Mb/s (Sun Enterprise 6000)



- Device Rates vary over many orders of magnitude
  - System better be able to handle this wide range
  - Better not have high overhead/byte for fast devices!
  - Better not waste time waiting for slow devices

The Goal of the I/O Subsystem

- Provide uniform interfaces, despite wide range of different devices
  - This code works on many different devices:

```
FILE fd = fopen(“/dev/something”,“rw”);
for (int i = 0; i < 10; i++) {
  fprintf(fd, “Count %d\n”, i);
}
close(fd);
```

  - How? Code that controls devices (“device driver”) implements standard interface

Want Standard Interfaces to Devices

- Block Devices: *e.g.,* disk drives and USB drives.
  - Access blocks of data
  - Commands include `open()`, `read()`, `write()`, `seek()`
  - Raw I/O or file-system access
  - Memory-mapped file access possible
- Character/Byte Devices: *e.g.,* keyboards, mice, serial ports.
  - Single characters at a time
  - Commands include `get()`, `put()`
- Network Devices: *e.g.,* Ethernet, Wireless, Bluetooth
  - Different enough from block/character to have own interface
  - Unix and Windows include socket interface
    - » Separates network protocol from network operation

How Does User Deal with Timing?

- Blocking Interface: "Wait"
  - When request data (*e.g.*, `read()` system call), put process to sleep until data is ready
  - When write data (*e.g.*, `write()` system call), put process to sleep until device is ready for data
- Non-blocking Interface: "Don't Wait"
  - Returns quickly from read or write request with count of bytes successfully transferred to kernel
  - Read may return nothing, write may write nothing
- Asynchronous Interface: "Tell Me Later"
  - When requesting data, take pointer to user's buffer, return immediately; later kernel fills buffer and notifies user
  - When sending data, take pointer to user's buffer, return immediately; later kernel takes data and notifies user

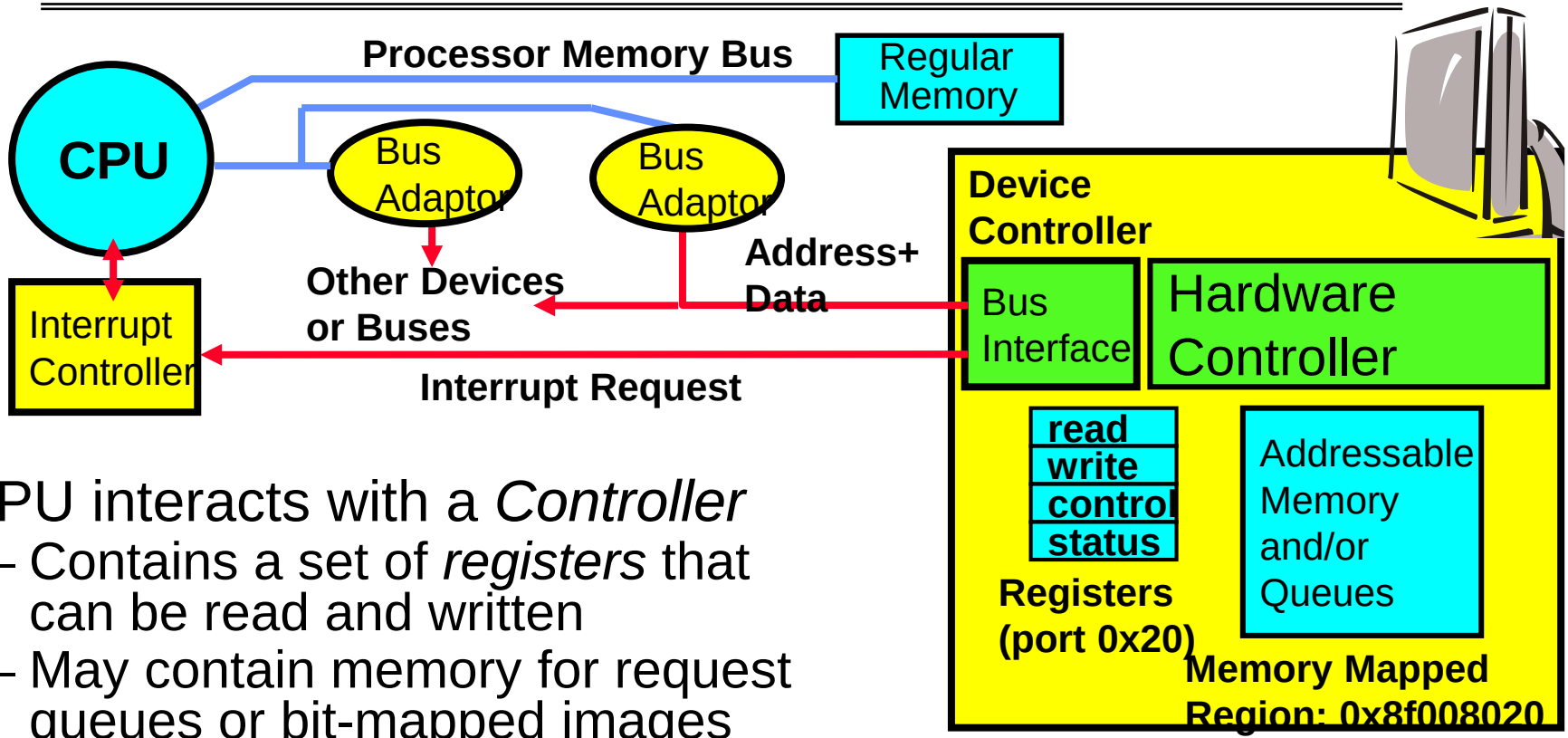
Kernel vs User-level I/O

- Both are popular/practical for different reasons:
  - Kernel-level drivers for critical devices that must keep running, e.g. display drivers.
    - » Programming is a major effort, correct operation of the rest of the kernel depends on correct driver operation.
  - User-level drivers for devices that are non-threatening, e.g USB devices in Linux (libusb).
    - » Provide higher-level primitives to the programmer, avoid every driver doing low-level I/O register tweaking.
    - » The multitude of USB devices can be supported.
    - » New drivers don't have to be compiled for each version of the OS, and loaded into the kernel.

Kernel vs User-level Programming Styles

- Kernel-level drivers
  - Have a much more limited set of resources available:
    - » Only a fraction of libc routines typically available.
    - » Memory allocation (e.g. Linux kmalloc) much more limited in capacity and required to be physically contiguous.
    - » Should avoid blocking calls.
- User-level drivers
  - Similar to other application programs but:
    - » Will be called often – should do its work fast, or postpone it – or do it in the background.
    - » Can use threads, blocking operations (usually much simpler) or non-blocking or asynchronous.

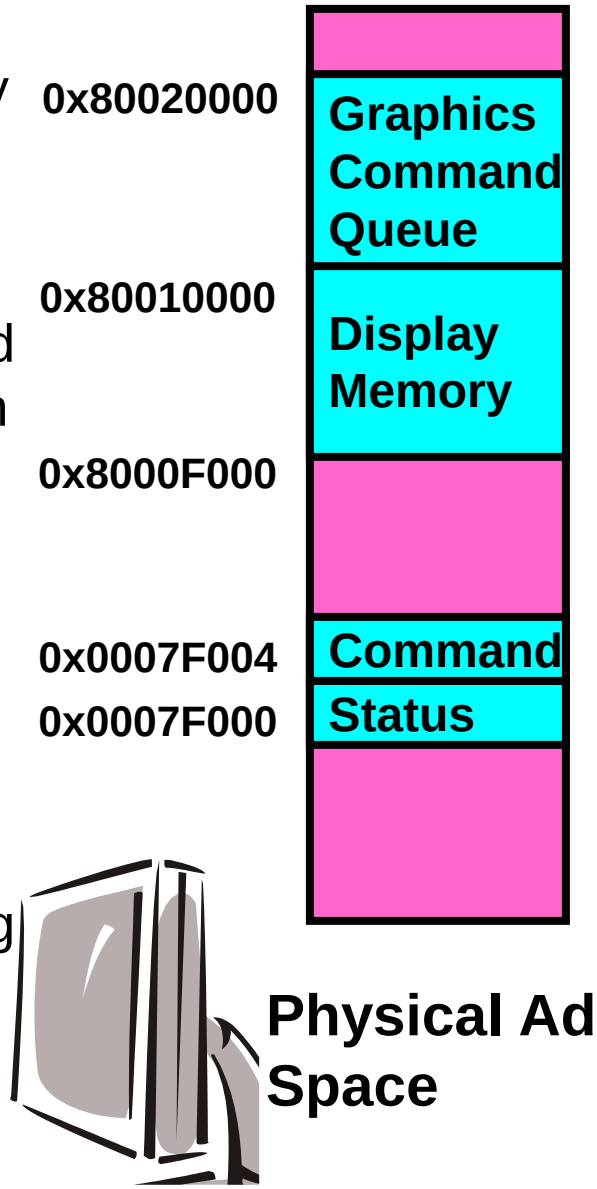
How Does the Processor Talk to Devices?



- CPU interacts with a *Controller*
  - Contains a set of *registers* that can be read and written
  - May contain memory for request queues or bit-mapped images
- Regardless of the complexity of the connections and buses, processor accesses registers in two ways (IA):
  - I/O instructions: in/out instructions (e.g., Intel's 0x21,AL)
  - Memory mapped I/O: load/store instructions
    - » Registers/memory appear in physical address space
    - » I/O accomplished with load and store instructions

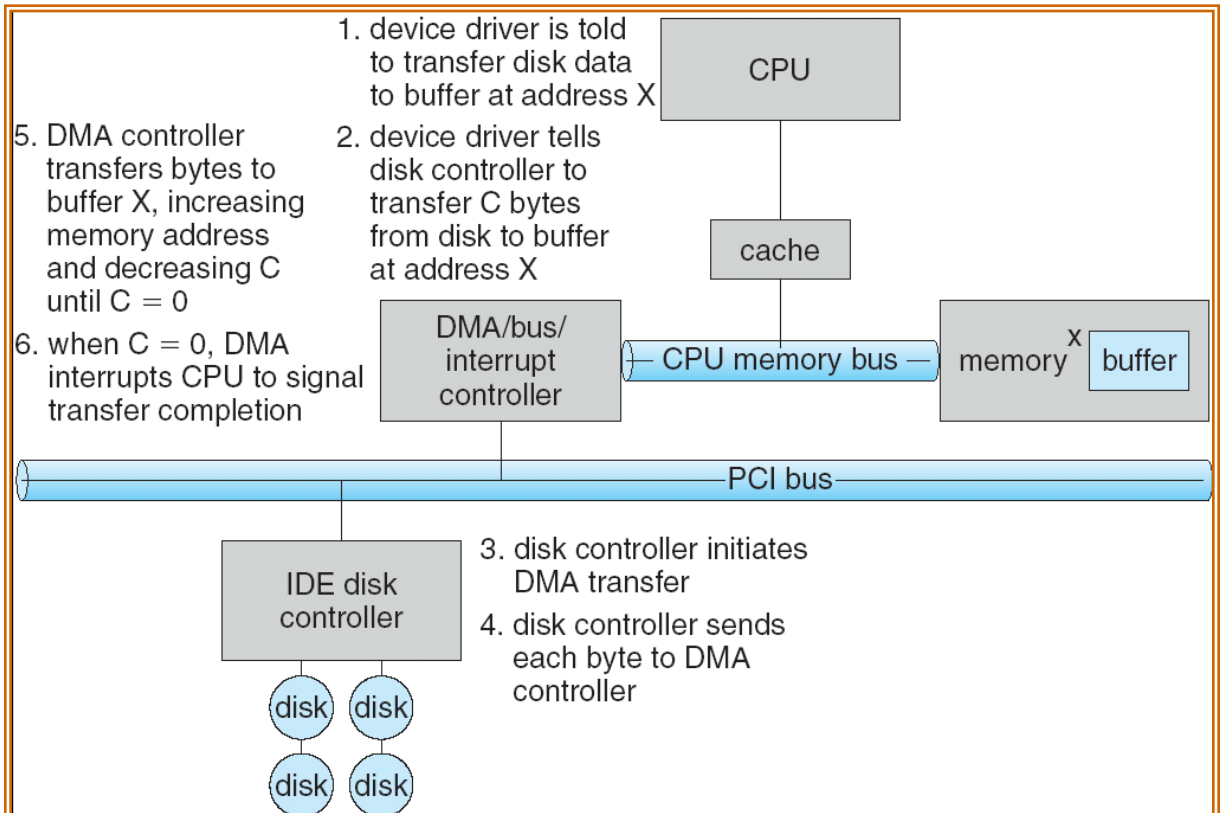
Example: Memory-Mapped Display Controller

- Memory-Mapped:
  - Hardware maps control registers and display memory into physical address space
    - » Addresses set by hardware jumpers or programming at boot time
  - Simply writing to display memory (also called the "frame buffer") changes image on screen
    - » Addr: 0x8000F000—0x8000FFFF
  - Writing graphics description to command-queue area
    - » Say enter a set of triangles that describe some scene
    - » Addr: 0x80010000—0x8001FFFF
  - Writing to the command register may cause on-board graphics hardware to do something
    - » Say render the above scene
    - » Addr: 0x0007F004
- Can protect with address translation

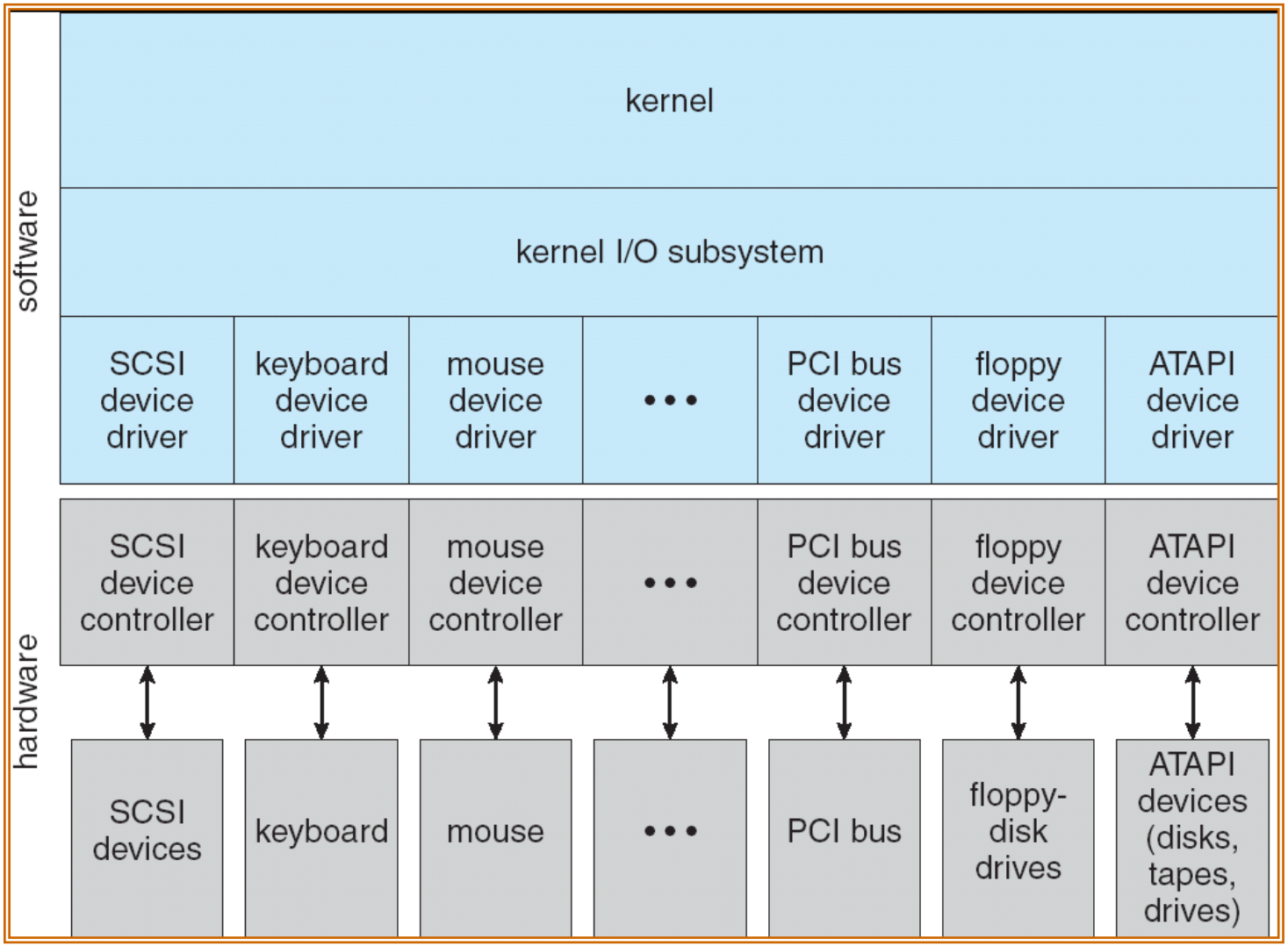


Transferring Data To/From Controller

- Programmed I/O: Each byte transferred via processor in/out or load/store
  - Pro: Simple hardware, easy to program
  - Con: Consumes processor cycles proportional to data size
- Direct Memory Access:
  - Give controller access to memory bus
  - Ask it to transfer data to/from memory directly and interrupt when done.



A Kernel I/O Structure



Device Drivers

- Device Driver: Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with the `ioctl()` system call
- Device Drivers typically divided into two pieces:
  - Top half: accessed in call path from system calls
    - » implements a set of standard, cross-device calls like `open()`, `close()`, `read()`, `write()`, `ioctl()`, `strategy()`
    - » This is the kernel's interface to the device driver
    - » Top half will *start* I/O to device, may put thread to sleep until finished
  - Bottom half: run as interrupt routine
    - » Gets input or transfers next block of output
    - » May wake sleeping threads if I/O now complete

Life Cycle of An I/O Request

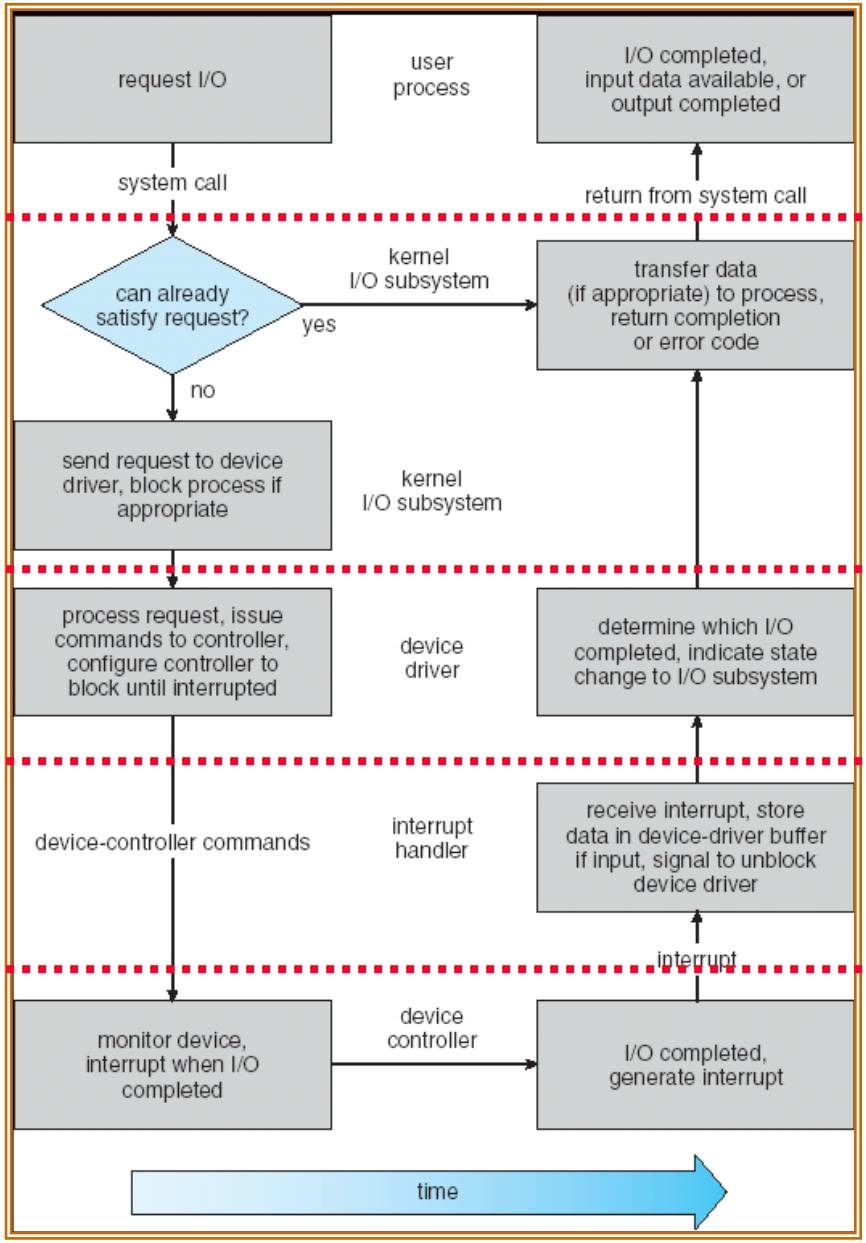
User Program

Kernel I/O Subsystem

Device Driver Top Half

Device Driver Bottom Half

Device Hardware



I/O Device Notifying the OS

- The OS needs to know when:
  - The I/O device has completed an operation
  - The I/O operation has encountered an error
- I/O Interrupt:
  - Device generates an interrupt whenever it needs service
  - Pro: handles unpredictable events well
  - Con: interrupts relatively high overhead
- Polling:
  - OS periodically checks a device-specific status register
    - » I/O device puts completion information in status register
  - Pro: low overhead
  - Con: may waste many cycles on polling if infrequent or unpredictable I/O operations
- Actual devices combine both polling and interrupts
  - For instance – High-bandwidth network adapter:
    - » Interrupt for first incoming packet
    - » Poll for following packets until hardware queues are empty

Summary

- I/O Devices Types:
  - Many different speeds (0.1 bytes/sec to GBytes/sec)
  - Different Access Patterns: block, char, net devices
  - Different Access Timing: Non-/Blocking, Asynchronous
- I/O Controllers: Hardware that controls actual device
  - CPU accesses thru I/O insts, ld/st to special phy memory
  - Report results thru interrupts or a status register polling
- Device Driver: Device-specific code in kernel