I/O

CS 416: Operating Systems Design

Department of Computer Science
Rutgers University

http://www.cs.rutgers.edu/~vinodg/416

I/O Devices

So far we have talked about how to abstract and manage CPU and memory

Computation "inside" computer is useful only if some results are communicated "outside" of the computer

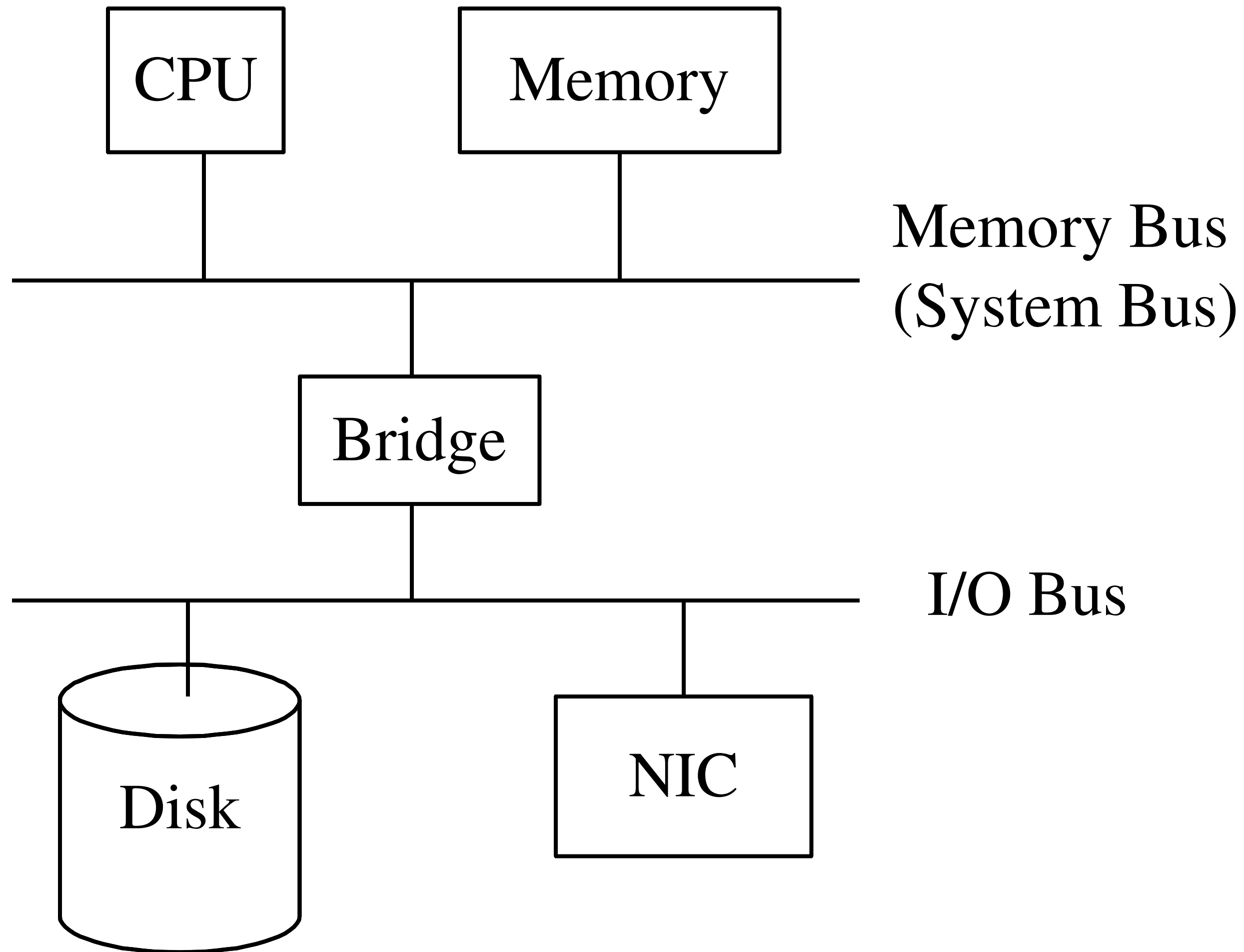
I/O devices are the computer's interface to the outside world (I/O ≡ Input/Output)

- Example devices: display, keyboard, mouse, speakers, network interface, and disk

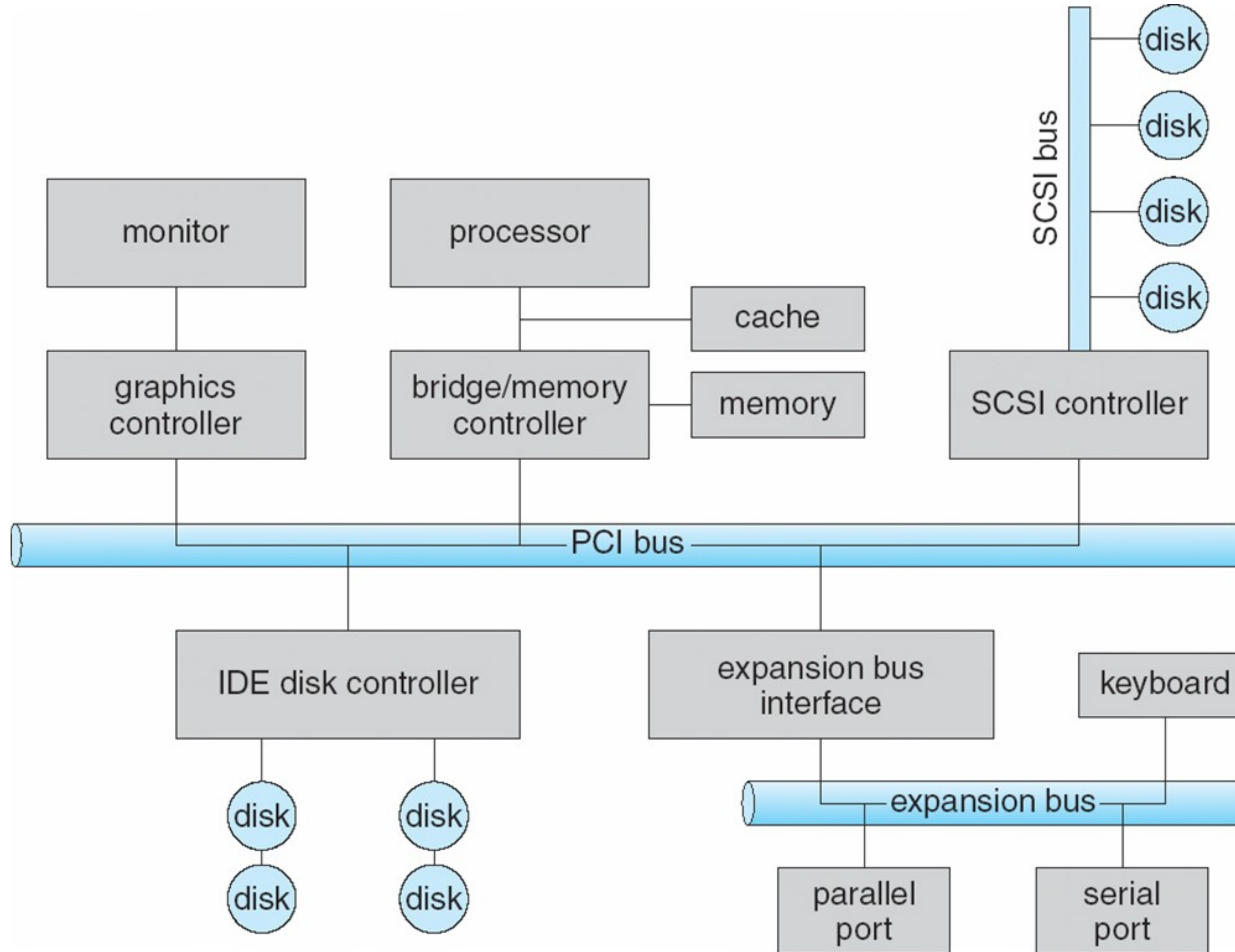
I/O Hardware

- Incredible variety of I/O devices
- Common concepts
  - **Port**
  - **Bus** (**daisy chain** or shared direct access)
  - **Controller** (**host adapter**)
- I/O instructions control devices
- Devices have addresses, used by
  - Direct I/O instructions
  - **Memory-mapped** I/O

Basic Computer Structure



A Typical PC Bus Structure



Polling

Determines state of device

- command-ready
- busy
- Error

Busy-wait cycle to wait for I/O from device

Interrupts

CPU **Interrupt-request line** triggered by I/O device

Interrupt handler receives interrupts

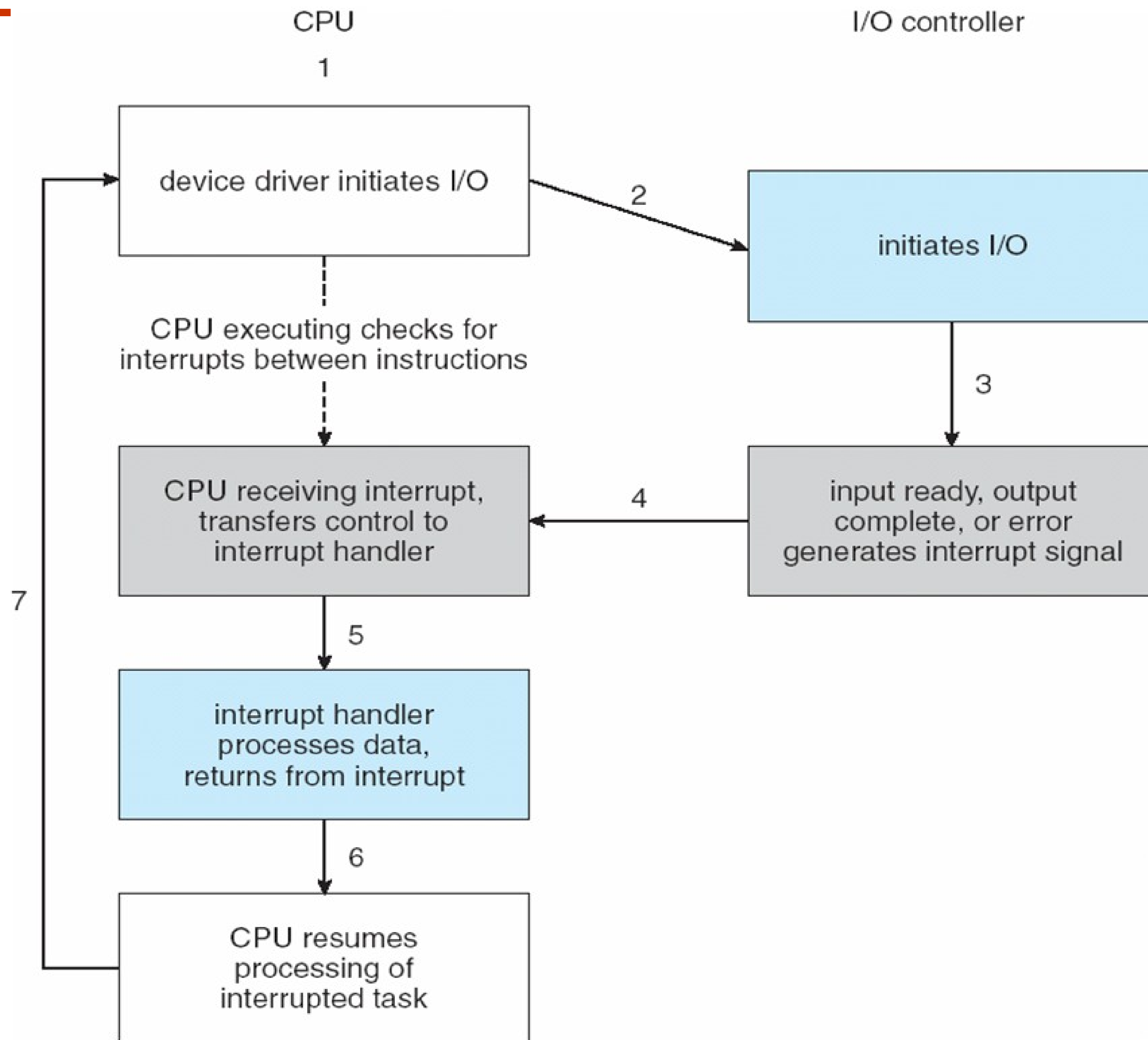
Maskable to ignore or delay some interrupts

Interrupt vector to dispatch interrupt to correct handler

- Based on priority
- Some **nonmaskable**

Interrupt mechanism also used for exceptions

Interrupt-Driven I/O Cycle



OS: Abstractions and Access Calls

OS virtualizes a wide range of devices into a few simple abstractions:

- Storage
  - Hard drives, tapes, CD drives
- Networking
  - Ethernet, radio, serial line
- Multimedia
  - DVD, camera, microphone

OS provides consistent calls to access the abstractions

- Otherwise, programming is too hard and unsafe

User/OS Interface

The same interface is used to access devices (like disks and network cards) and more abstract resources (like files)

4 main calls:

open()

close()

read()

write()

Semantics depend on the type of the device. Devices vary in terms of transfer modes (block, char, network); access methods (sequential, random); transfer schedules (synchronous, asynchronous). A synchronous device is one that performs transfers with predictable times. We will talk more about these later.

Examples: disk = block, random, synchronous; tape = block, sequential, synchronous; keyboard = char, sequential, asynchronous; display = char, random, synchronous.

Application I/O Interface

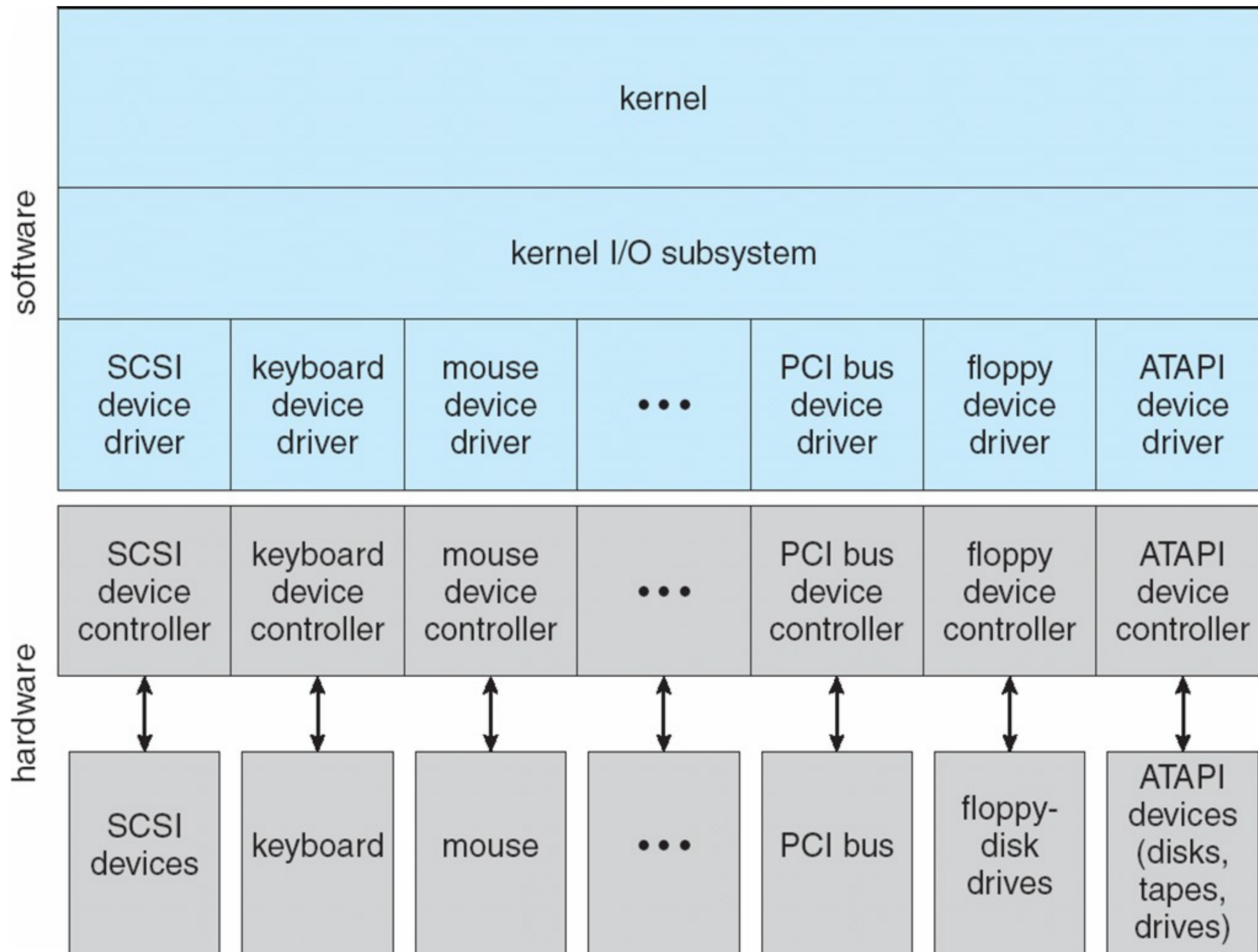
I/O system calls encapsulate device behaviors in generic classes

Device-driver layer hides differences among I/O controllers from kernel

Devices vary in many dimensions

- **Character-stream** or **block**
- **Sequential or random-access**
- **Sharable or dedicated**
- **Speed of operation**
- **read-write, read only,** or **write only**

A Kernel I/O Structure



Unix I/O Calls

```
fileHandle  = open(pathName, flags, mode)
```

A file handle is a small integer, valid only within a single process, to operate on the device or file

Pathname: a name in the file system. In unix, devices are put under /dev. E.g. /dev/ttya is the first serial port, /dev/sda the first SCSI drive

Flags: blocking or non-blocking …

Mode: read only, read/write, append …

```
errorCode = close(fileHandle)
```

Kernel will free the data structures associated with the device

Unix I/O Calls

```
byteCount = read(fileHandle, buf, count)
```

Read at most count bytes from the device and put them in the byte buffer buf. Bytes placed from 0th byte.

Kernel can give the process less bytes, user process must check the byteCount to see how many were actually returned.

A negative byteCount signals an error (value is the error type)

```
byteCount = write(fileHandle, buf, count)
```

Write at most count bytes from the buffer buf

Actual number written returned in byteCount

A negative byteCount signals an error

Unix I/O Example

What's the correct way to write 1000 bytes?

Calling

```
ignoreMe = write(fileH, buffer, 1000);
```

works most of the time. What happens if cannot accept 1000 bytes right now?

Disk is full

How do we do this right?

Unix I/O Example

How do we do this right?

```
written = 0;  target = 1000;
while (written < target) {
    bytes = write(fileH, buffer+written, target-written);
    if ( bytes < target-written ) {
        if ( bytes > 0 )
            written += bytes;
        puts (“Fix problem and type something\n”);  getchar();
    }
}
```

I/O Semantics

From this basic interface, two different dimensions to how I/O is processed:

- blocking vs. non-blocking vs. asynchronous
- buffered vs. unbuffered

The OS tries to support as many of these dimensions as possible for each device

The semantics are specified during the open() system call

Blocking vs. Non-blocking vs. Asynchronous I/O

Blocking – process is blocked until all bytes in the `count` field are read or written

E.g., for a network device, if the user wrote 1000 bytes, then the OS would only unblock the process after the write() call completes.

+ Easy to use and understand

-- If the device just can't perform the operation (e.g., you unplug the cable), what to do? Give up an return the successful number of bytes.

Non-blocking – the OS only reads or writes as many bytes as is possible without blocking the process

+ Returns quickly

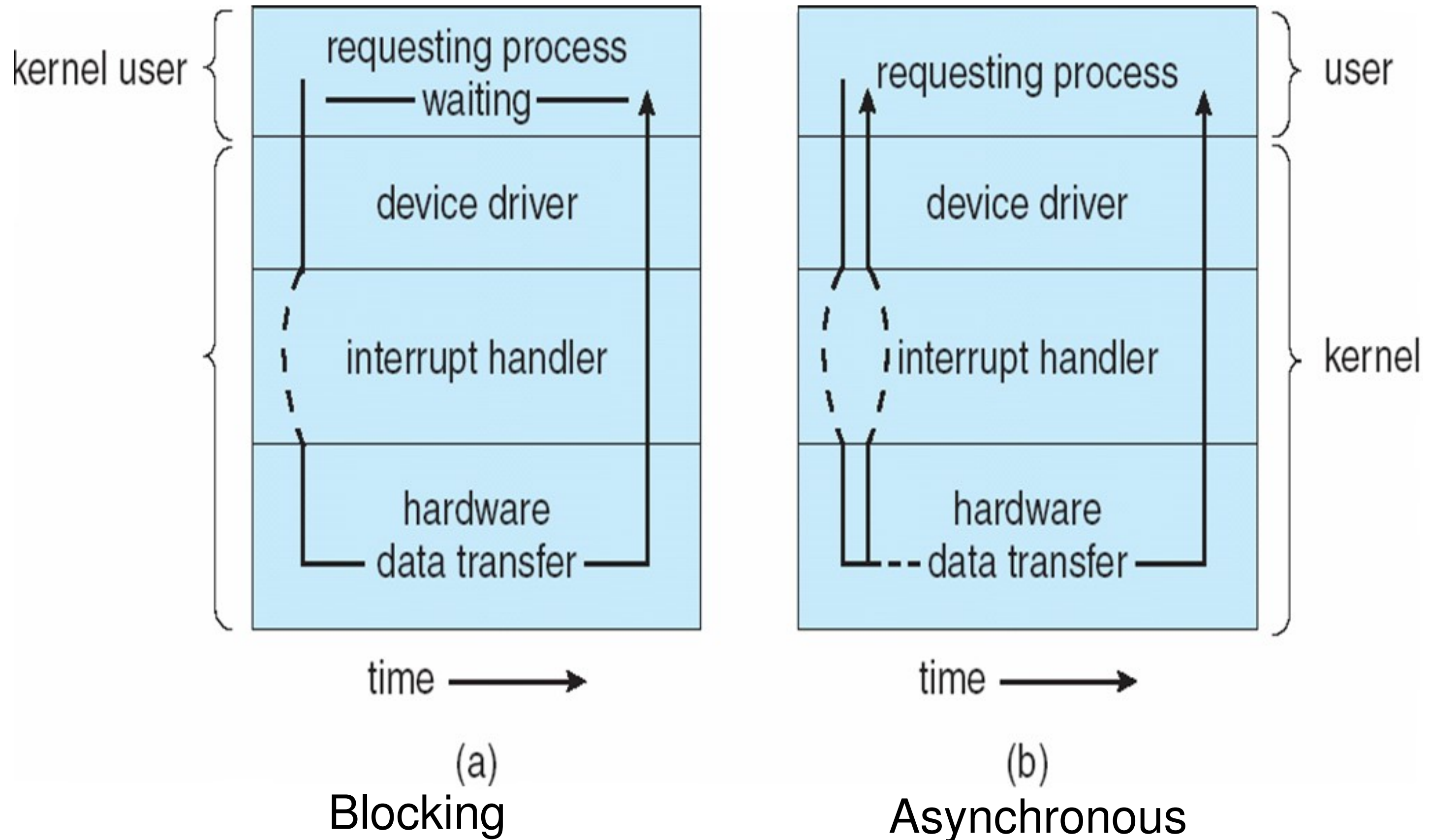
-- More work for the programmer (but really good for robust programs)

Blocking vs. Non-blocking vs. Asynchronous I/O

Asynchronous – similar to non-blocking I/O. The I/O call returns immediately, without waiting for the operation to complete. I/O subsystem signals the process when I/O is done. Same advantages and disadvantages of non-blocking I/O.

Difference between non-blocking and asynchronous I/O: a non-blocking read() returns immediately with whatever data available; an asynchronous read() requests a transfer that will be performed in its entirety, but that will complete at some future time.

Blocking vs. Asynchronous



Blocking

Asynchronous

Buffered vs. Unbuffered I/O

Sometimes we want the ease of programming of blocked I/O without the long waits if the buffers on the device are small.

Buffered I/O allows the kernel to make a copy of the data and adjust to different device speeds.

write(): allows the process to write bytes and continue processing

read(): as device signals data is ready, kernel places data in the buffer. When process calls read(), the kernel just makes a copy.

Why not use buffered I/O?

-- Extra copy overhead

-- Delays sending data

Handling Multiple I/O Streams

If we use blocking I/O, how do we handle > 1 device at a time?

Example: a small program that reads from a serial line and outputs to a tape and is also reading from a tape and outputting to a serial line

Structure the code like this?

```
while (TRUE) {
    read(tape 1); // block until tape is ready
    write(serial line 1); // send data to serial line
    read(serial line 2);
    write(tape 2);
}
```

Could use non-blocking I/O, but huge waste of cycles if data is not ready.

Solution: select System Call

totalFds = select(nfds, readfds, writefds, errorfds, timeout);

nfds: the range (0.. nfds) of file descriptors to check

readfds: bit map of fileHandles. User sets bit X to ask the kernel to check if fileHandle X ready for reading. Kernel returns a 1 if data can be read on the fileHandle.

writefds: bit map of fileHandles. User sets bit Y for writing to fileHandle Y. Kernel returns a 1 if data can be written to the fileHandle.

errorfds: bit map to check for errors

timeout: how long to wait for the select to complete

totalFds = number of set bits, negative number is an error

Getting Back to Device Types

Most OSs have three device types (in terms of transfer modes):

Character devices

- Used for serial-line types of devices (e.g., USB port)

Block devices

- Used for mass-storage (e.g., disks and CDROM)

Network devices

- Used for network interfaces (e.g., Ethernet card)

What you can expect from the read/write calls changes with each device type

Character Devices

Device is represented by the OS as an ordered stream of bytes

- bytes sent out to the device by the write system call
- bytes read from the device by the read system call

Byte stream has no "start", just open and start reading/writing

The user has no control over the read/write ratio, the sender process might write once 1000 bytes, and the receiver may have to call 1000 read calls each receiving 1 byte.

Block Devices

OS presents device as a large array of blocks

Each block has a fixed size (1KB - 8KB is typical)

User can read/write only in fixed-size blocks

Unlike other devices, block devices support random access

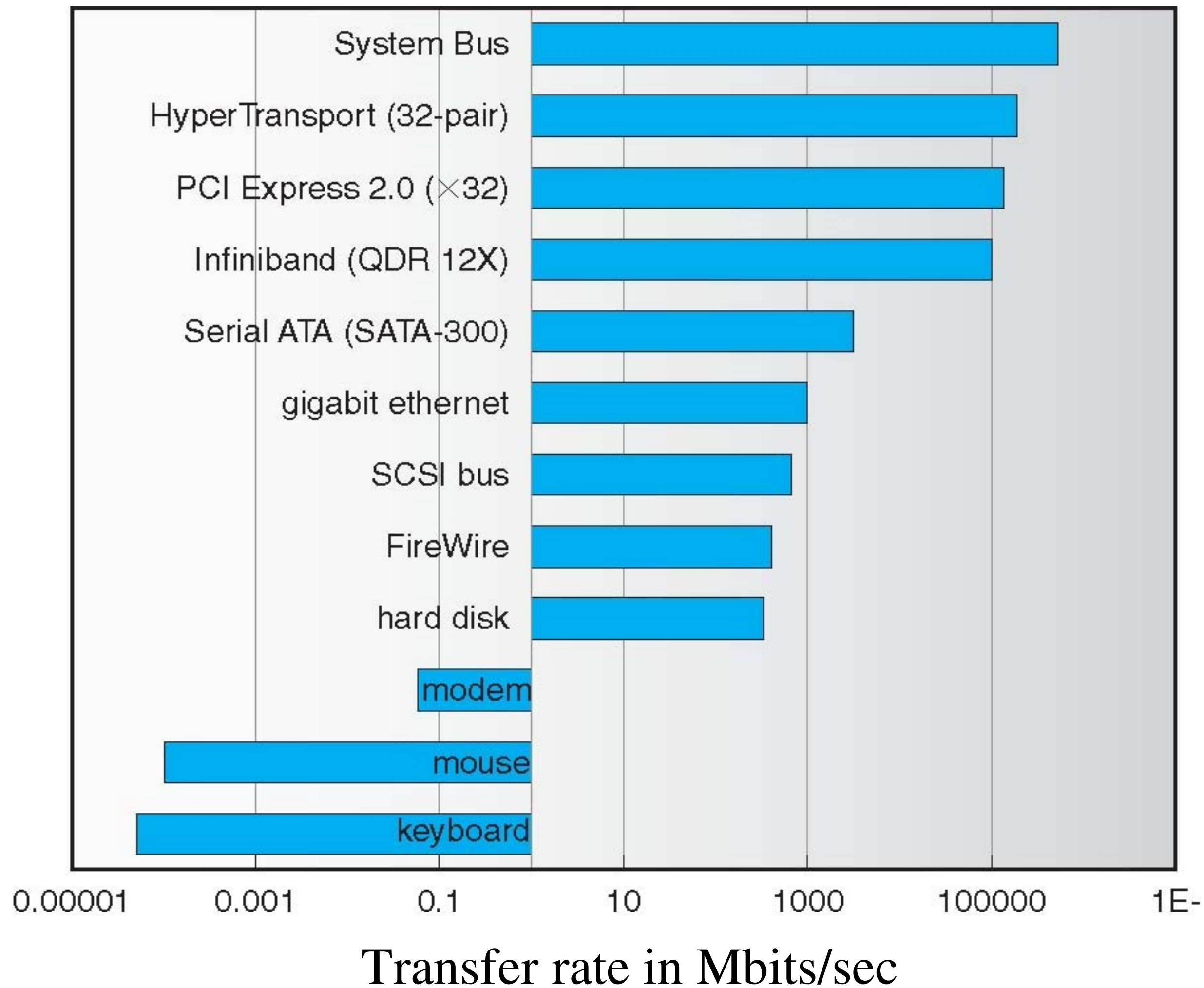
We can read or write anywhere in the device without having to 'read all the bytes first'

Network Devices

Like block-based I/O devices, but each write call either sends the entire block (packet), up to some maximum fixed size, or none.

On the receiver, the read call returns all the bytes in the block, or none.

Sun Enterprise 6000 Device Transfer Rates



Random Access: The File Pointer

For random access in block devices, OS adds a concept call the file pointer

A file pointer is associated with each open file or device, if the device is a block device

The next read or write operates at the position in the device pointed to by the file pointer

The file pointer points to bytes, not blocks

The Seek Call

To set the file pointer:

```
absoluteOffset = lseek(fileHandle, offset, from);
```

from specifies if the offset is absolute, from byte 0, or relative to the current file pointer position

The absolute offset is returned; negative numbers signal error codes

For devices, the offset should be a integral number of bytes.

Block Device Example

You want to read the 10th block of a disk

Each disk block is 4096 bytes long

```
fh = open(/dev/sda, , , );
pos = lseek(fh, 4096*9, 0);
if (pos < 0) error;
bytesRead = read(fh, buf, 4096);
if (bytesRead < 0) error;
…
```

Getting and Setting Device-Specific Information

Unix has an I/O control system call:

```
ErrorCode = ioctl(fileHandle, request, object);
```

request is a numeric command to the device

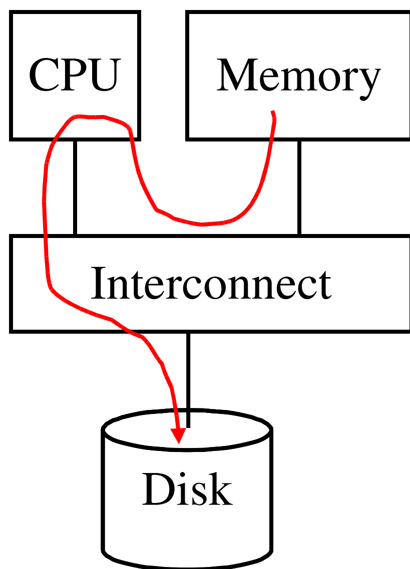
Can also pass an optional, arbitrary object to a device

The meaning of the command and the type of the object are device-specific

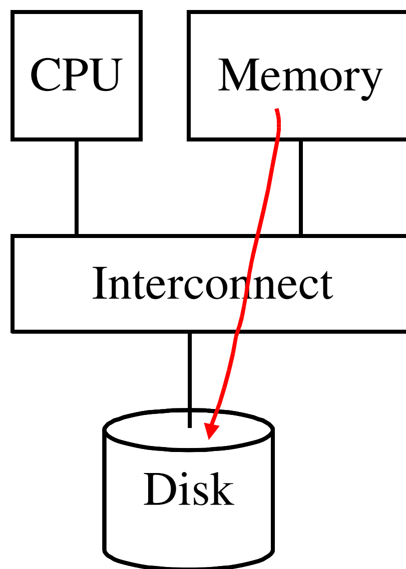
Programmed I/O vs. DMA

- Programmed I/O is ok for sending commands, receiving status, and communication of a small amount of data
- Inefficient for large amounts of data
  - Keeps CPU busy during the transfer
  - Programmed I/O ≡ memory operations ➜ slow
- Direct Memory Access
  - Device read/write directly from/to memory
  - Memory ➜ device typically initiated from CPU
  - Device ➜ memory can be initiated by either the device or the CPU

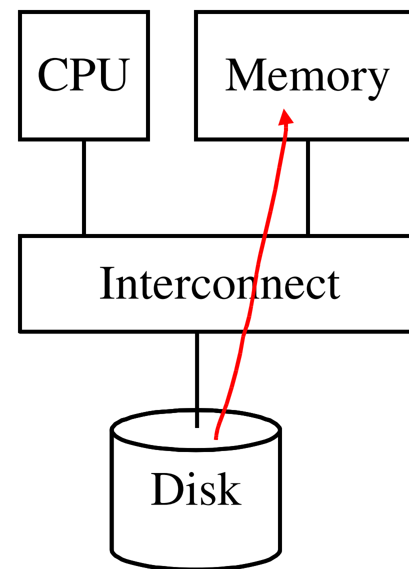
Programmed I/O vs. DMA



Programmed I/O



DMA



DMA
Device ➜ Memory
Problems?

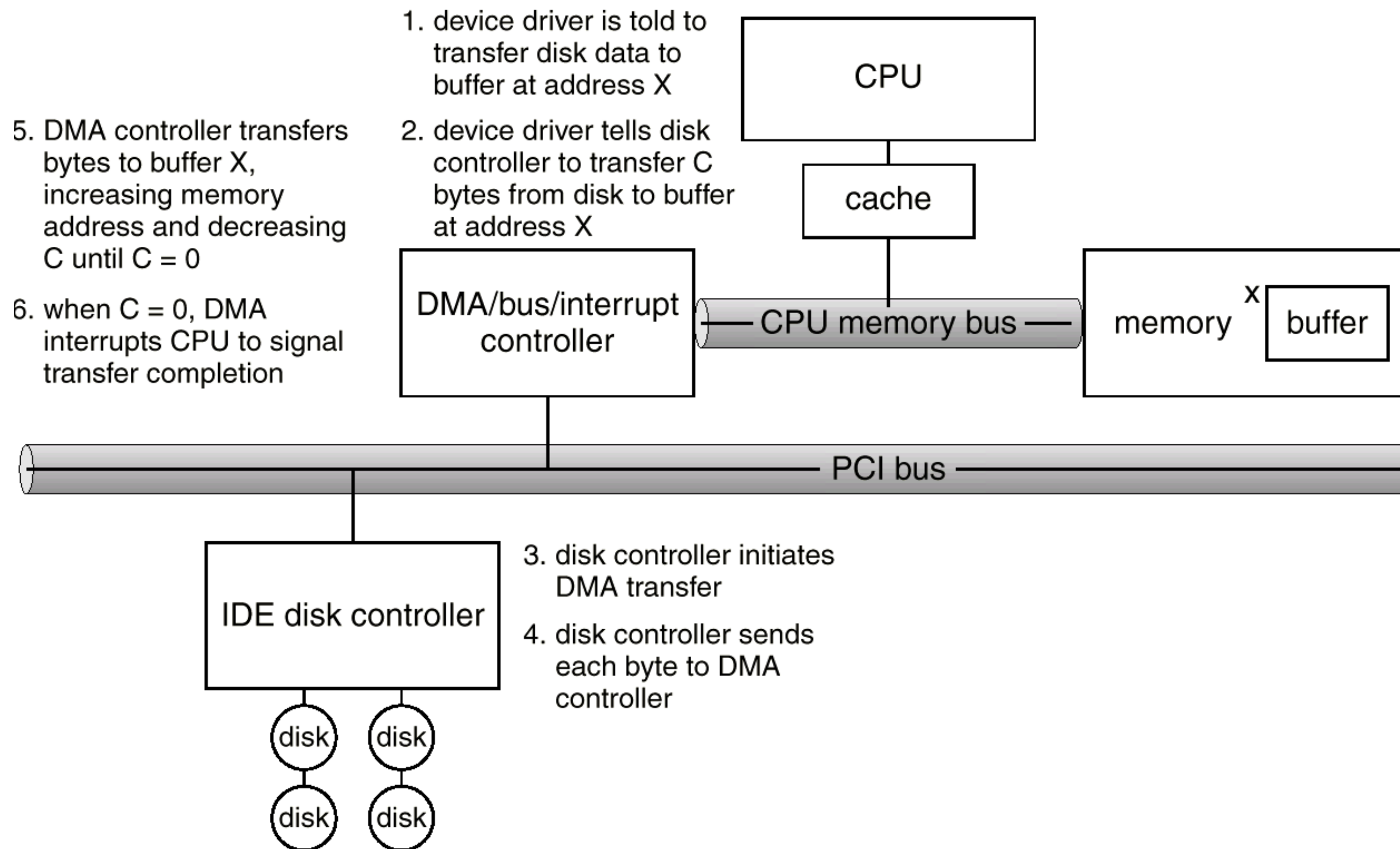
Direct Memory Access

Used to avoid programmed I/O for large data movements

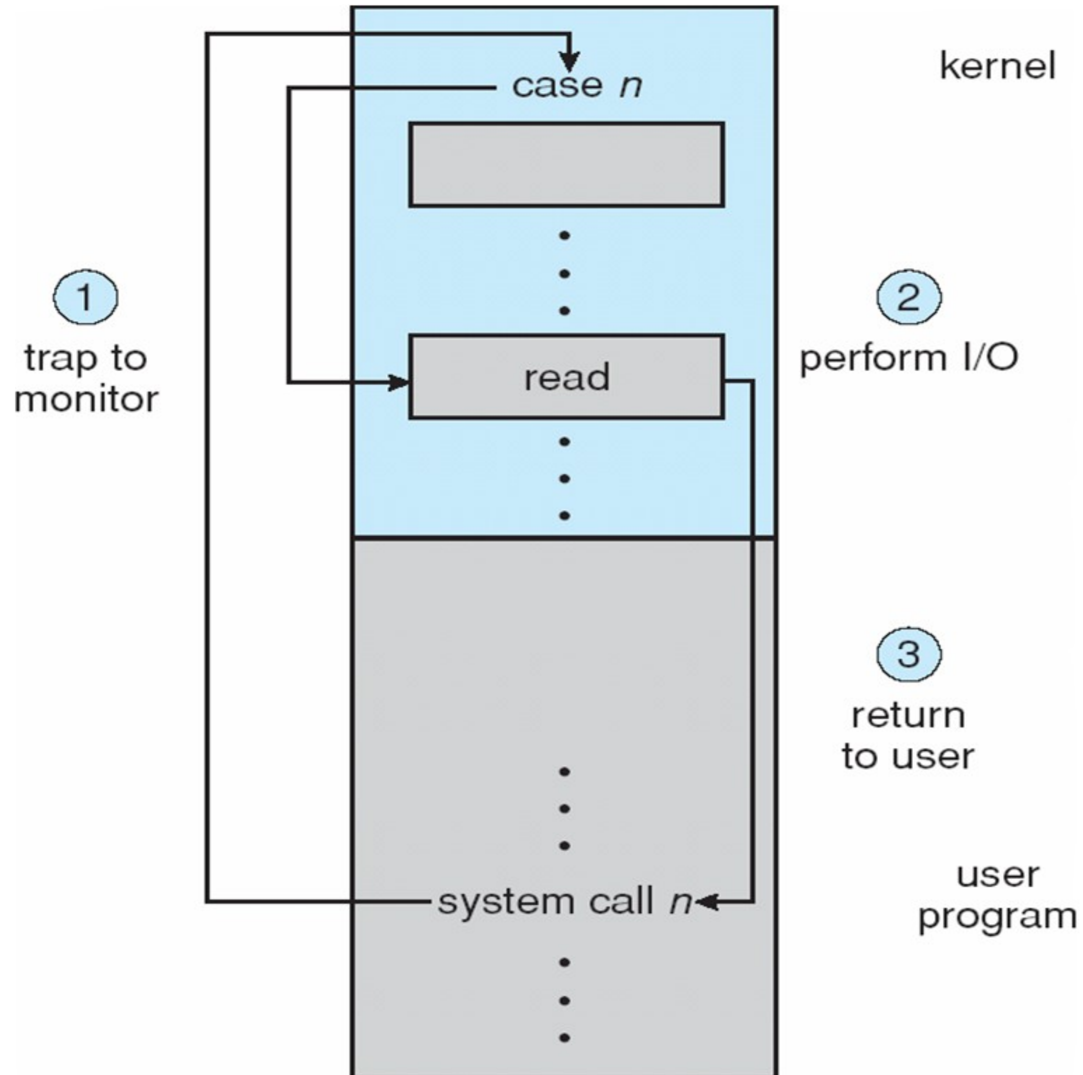
Requires DMA controller

Bypasses CPU to transfer data directly between I/O device and memory

Six Steps to Perform DMA Transfer



Use of a System Call to Perform I/O



I/O Requests to Hardware Operations

Consider reading a file from disk for a process:

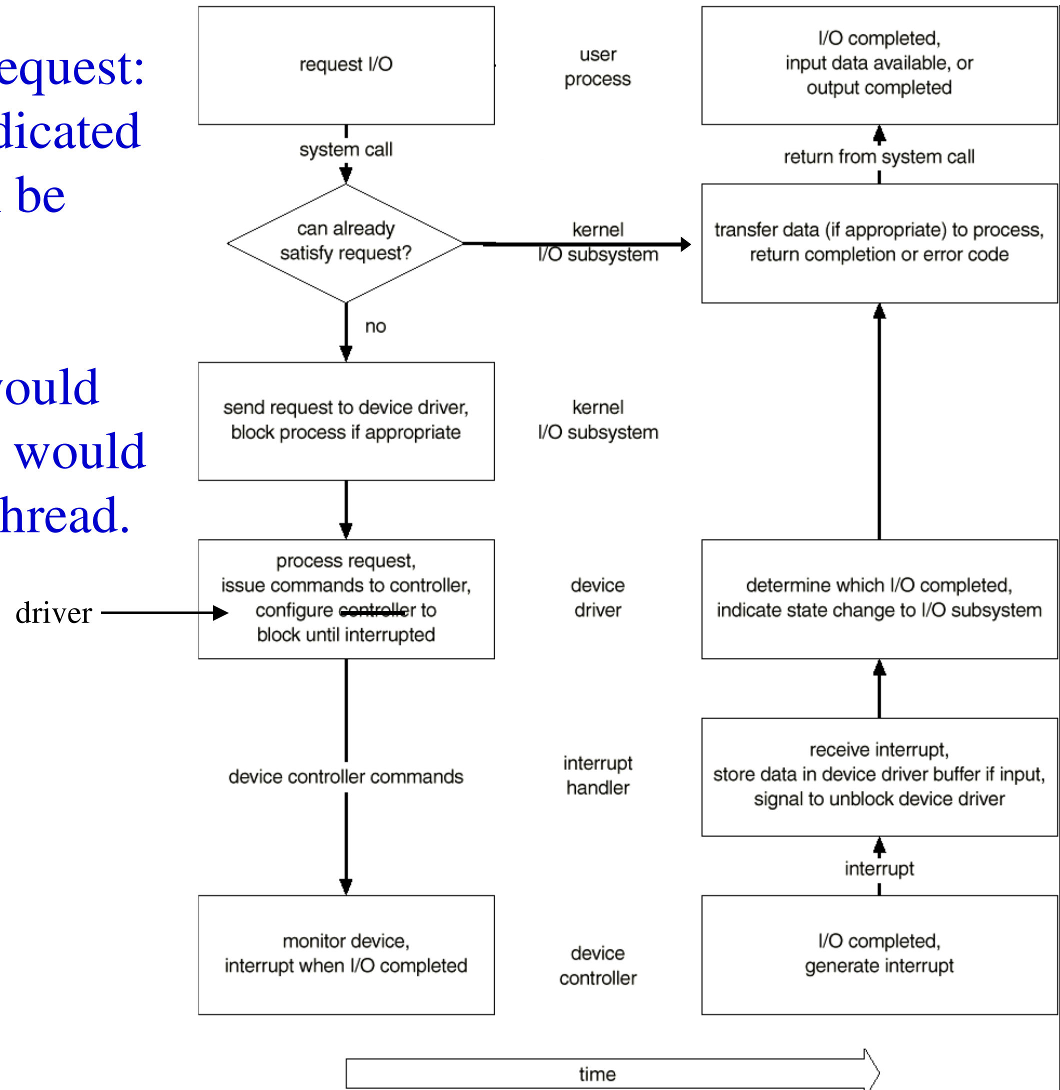
- Determine device holding file
- Translate name to device representation
- Physically read data from disk into buffer
- Make data available to requesting process
- Return control to process

Life Cycle of a Blocking I/O Request

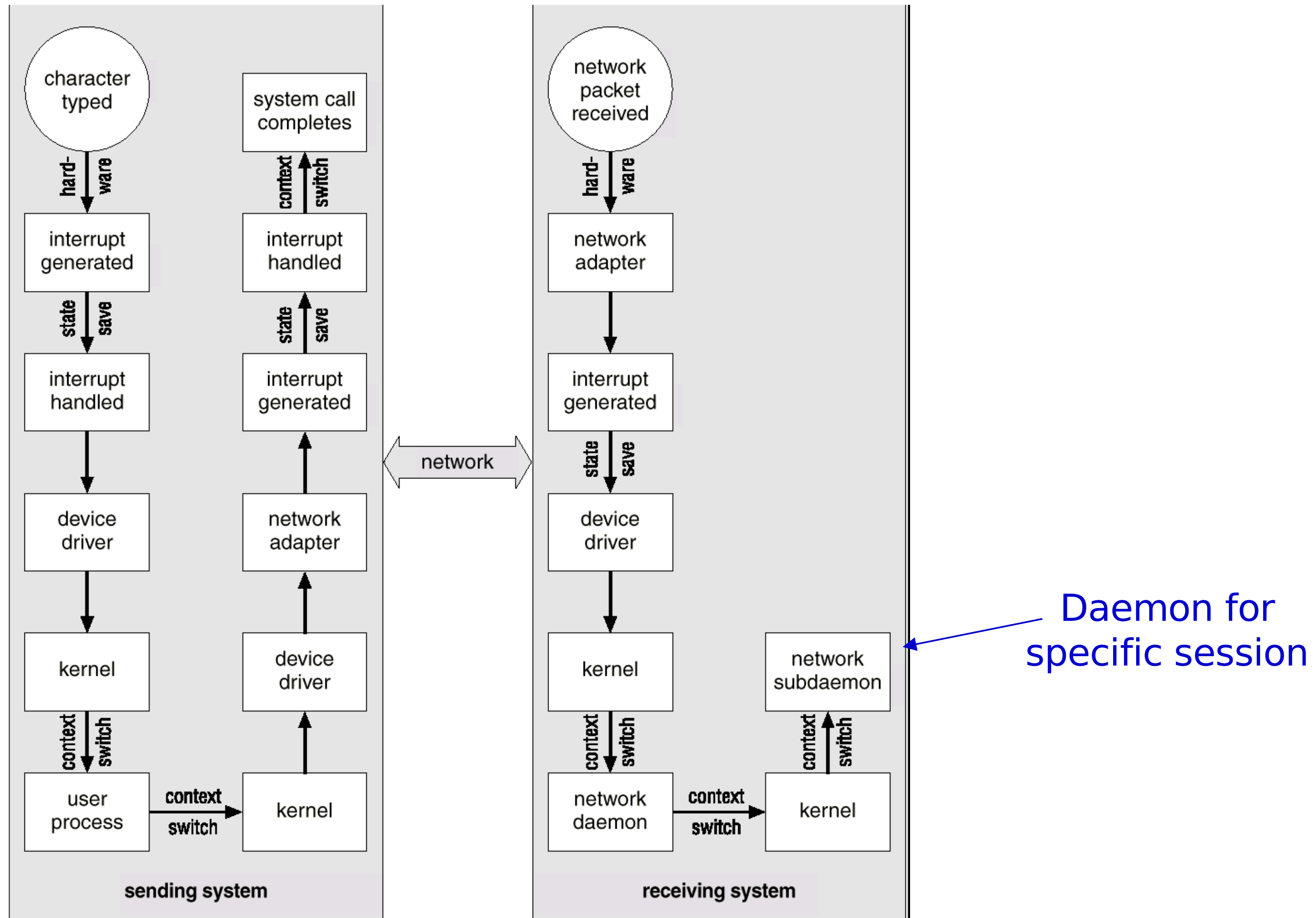
Naïve processing of blocking request: device driver executed by a dedicated kernel thread; only one I/O can be processed at a time.

More sophisticated approach would not block the device driver and would not require a dedicated kernel thread.

driver



Intercomputer Communications: Telnet



Performance

I/O a major factor in system performance

- Demands CPU to execute device driver, kernel I/O code
- State save/restore due to interrupts
- Data copying
- Disk I/O is extremely slow

Improving Performance

Reduce number of context switches

Reduce data copying

Reduce interrupts by using large transfers, smart controllers, polling

Use DMA

Balance CPU, memory, bus, and I/O performance for highest throughput

STREAMS

STREAM – a full-duplex communication channel between a user-level process and a device in Unix System V and beyond

A STREAM consists of:

- STREAM head interfaces with the user process
- driver end interfaces with the device

- zero or more STREAM modules between them.

Each module contains a **read queue** and a **write queue**

Message passing is used to communicate between queues

The STREAMS Structure