11 I/O Systems

chapter 12 of the book

- Overview
- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Transforming I/O Requests to Hardware Operations
- STREAMS
- Performance

The Central I/O Problem

The Speed Mismatch

```
CPU Cycle:      ~0.3 nanoseconds

RAM Access:     ~100 nanoseconds

SSD Access:     ~100,000 nanoseconds

Network Packet: ~1,000,000 nanoseconds
```

The Challenge: How do we manage I/O without wasting the CPU's incredible speed?

Two Key Questions:

1. **Synchronization:** How does the CPU know when a device is ready?
2. **Data Transfer:** How does data move efficiently between device and memory?

I/O Hardware Foundation: Buses

The Communication Highway System

Simplified PC Architecture (Historical):

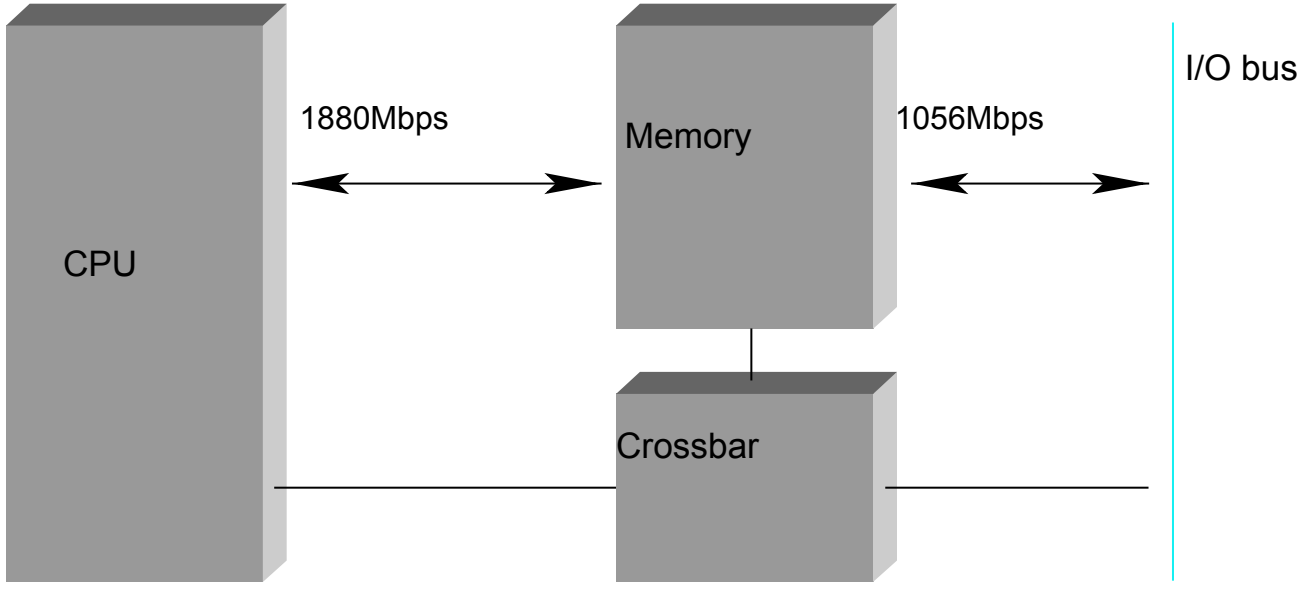
```
CPU --[Fast Bus]--> Memory
  |
[I/O Bridge]
  |
 [I/O Bus] --> Disk, Network, USB, etc.
```

Modern Architecture (Intel/AMD):

```
CPU (with Memory Controller & PCIe Lanes)
  |
  |-- Direct to GPU/NVMe (PCIe)
  |-- Direct to RAM
  |
[Platform Controller Hub]
  |
  |-- USB, SATA, Ethernet, Legacy Devices
```

Key Point: Modern CPUs have **integrated controllers** for high-speed direct connections.

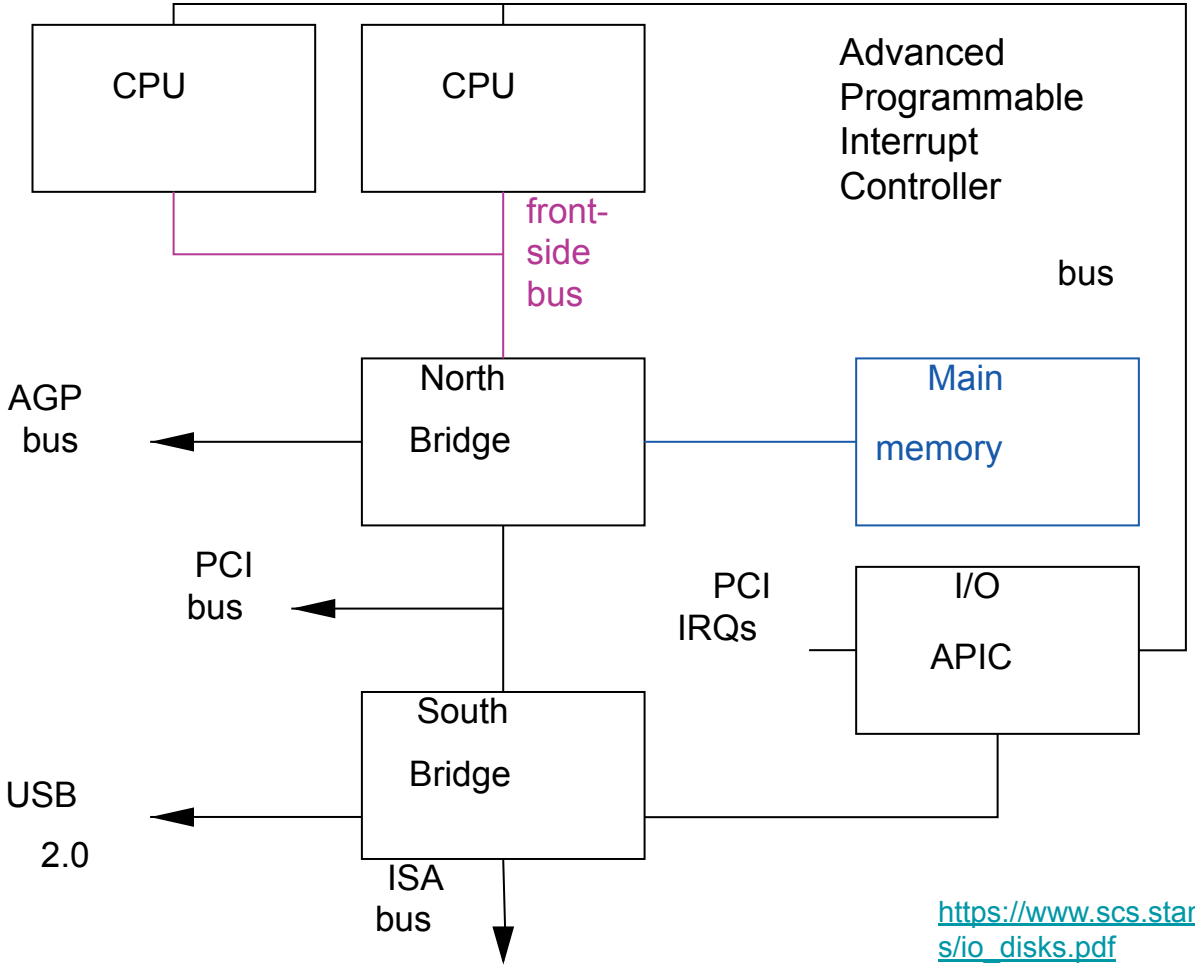
Memory and I/O buses



- CPU accesses physical memory over a bus
- Devices access memory over I/O bus with DMA
- Devices can appear to be a region of memory

https://www.scs.stanford.edu/24wi-cs212/notes/io_disks.pdf

Realistic ~2005 PC architecture



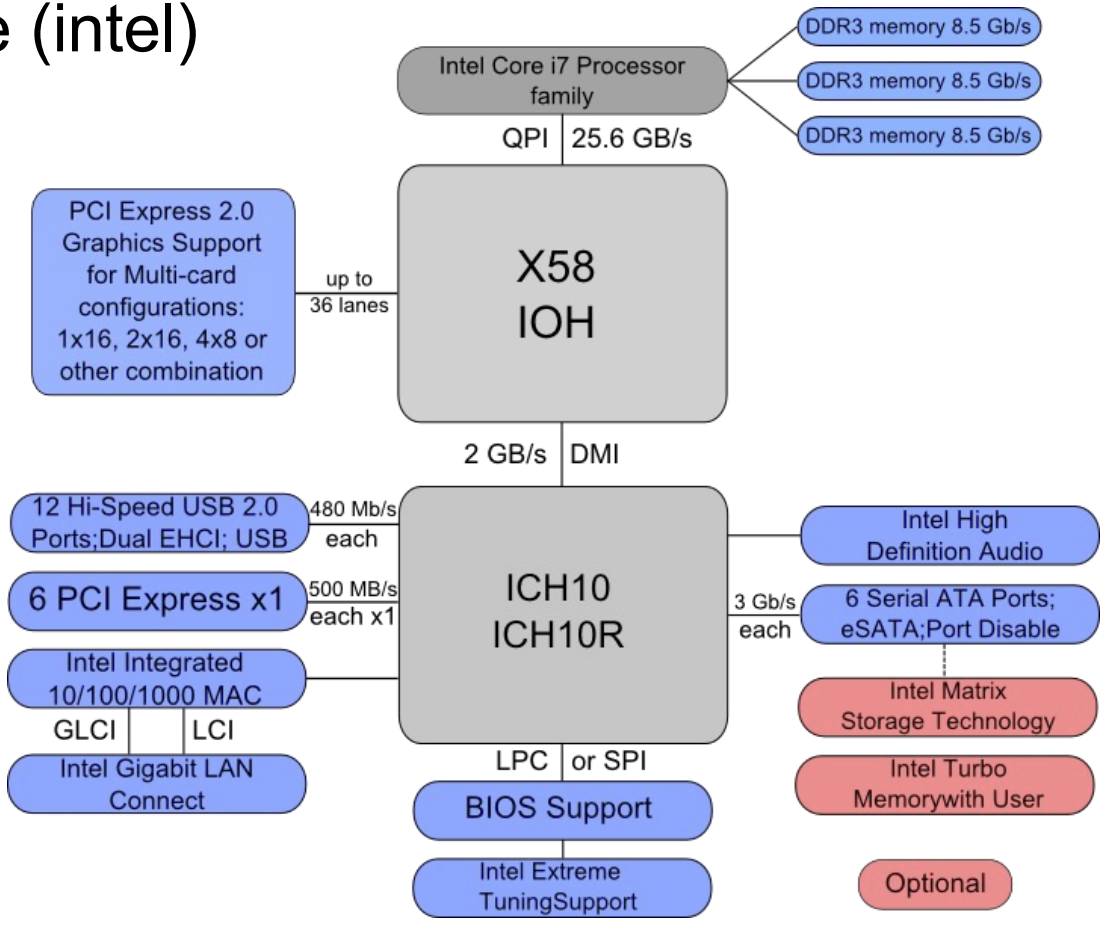
https://www.scs.stanford.edu/24wi-cs212/notes/io_disks.pdf

Modern PC architecture (intel)

IOH-I/O hub

ICH = I/O controller Hub

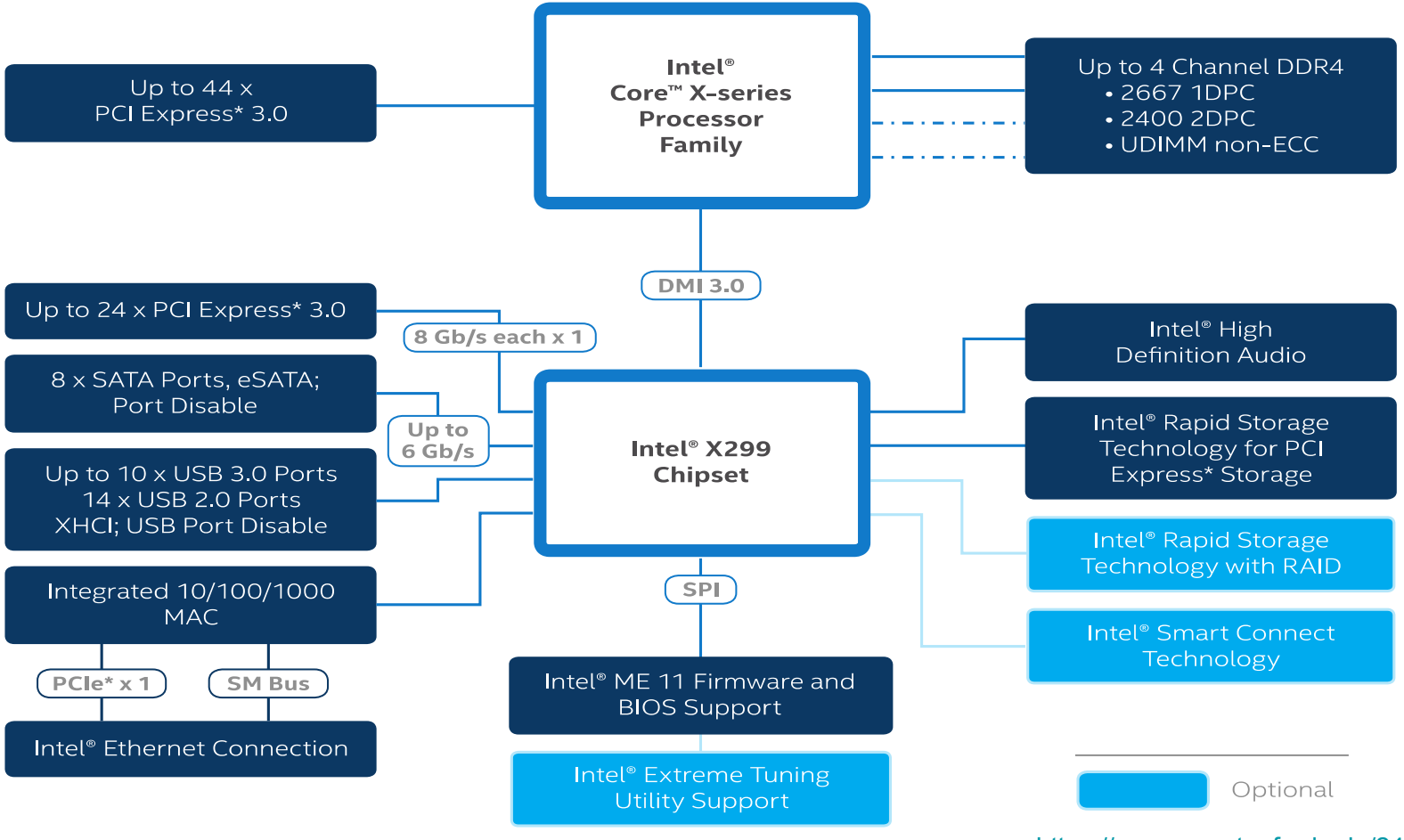
QPI = [Intel QuickPath Interconnect - Wikipedia]



https://www.scs.stanford.edu/24wi-cs212/notes/io_disks.pdf

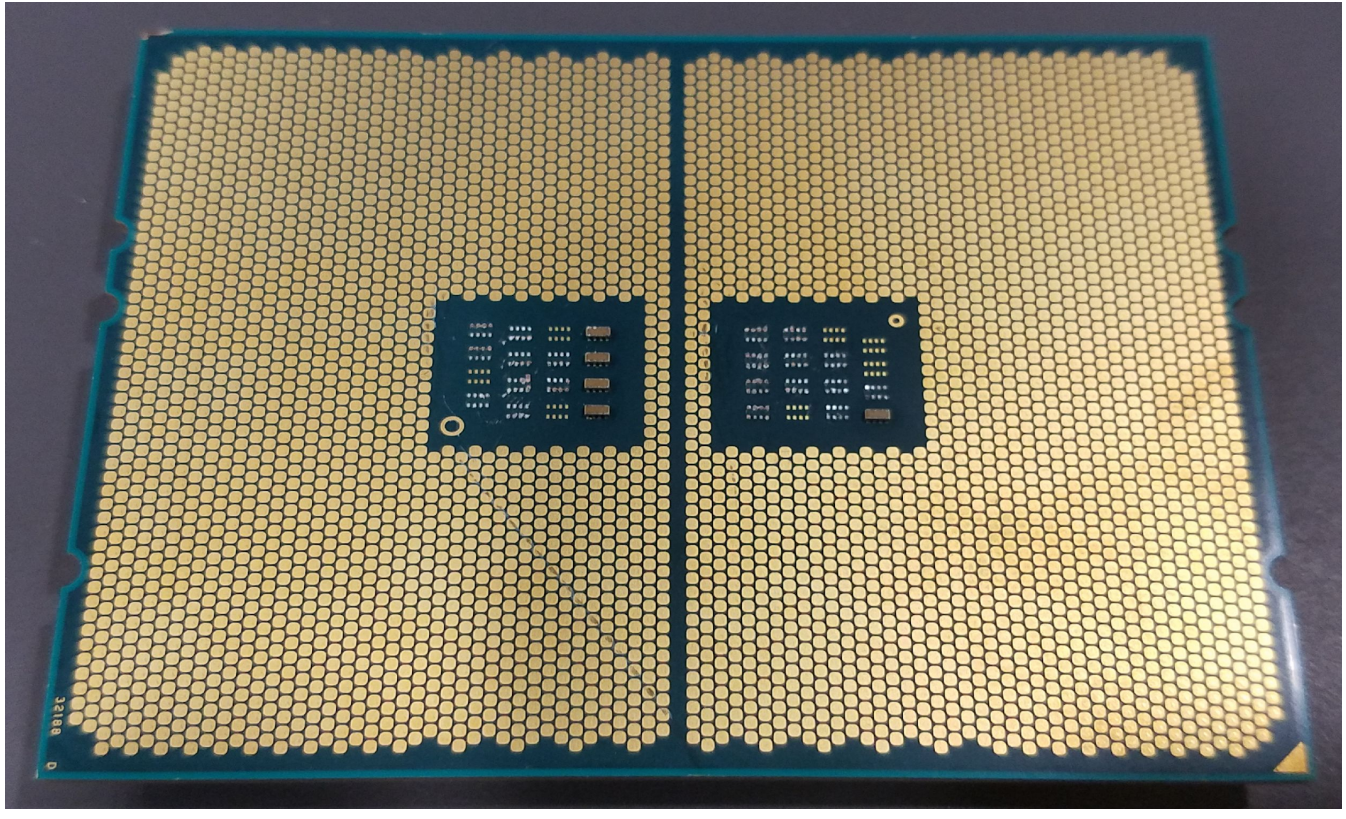
https://en.wikipedia.org/wiki/Intel_X58

CPU now entirely subsumes IOH [intel]



https://www.scs.stanford.edu/24wi-cs212/notes/io_disks.pdf

AMD EPYC is essentially an SoC



4094 pins: both memory controller and 128 lanes PCIe

directly on chip!

What is memory

SRAM – Static RAM

- Like two NOT gates circularly wired input-to-output
- 4–6 transistors per bit, actively holds its value
- Very fast, used to cache slower memory

DRAM – Dynamic RAM

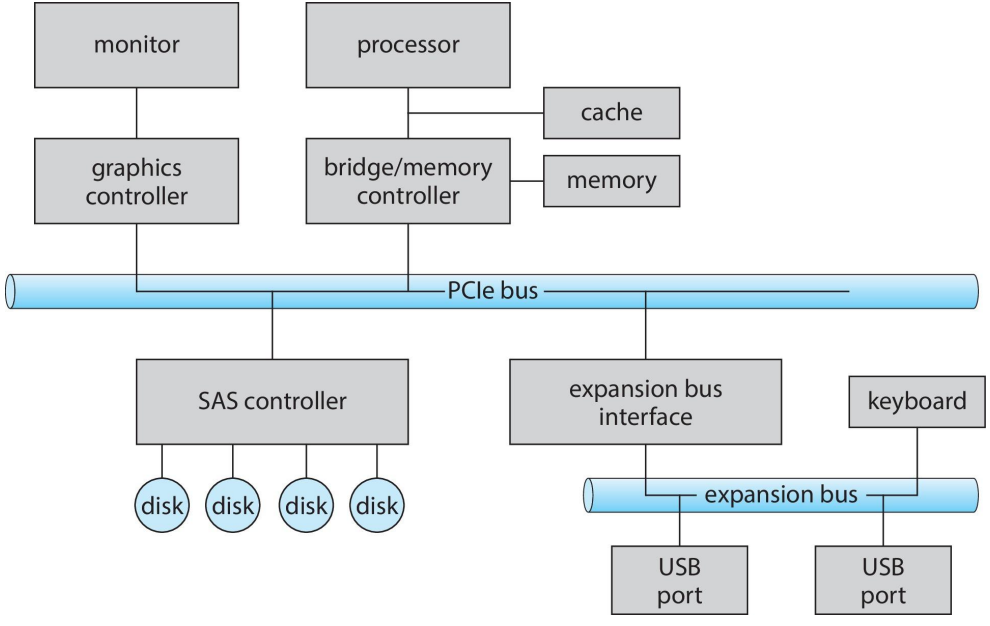
- A capacitor + gate, holds charge to indicate bit value
- 1 transistor per bit – extremely dense storage
- Charge leaks – need slow comparator to decide if bit 1 or 0
- Must rewrite charge after reading, and periodically refresh

VRAM – “Video RAM”

- - Dual ported DRAM, can write while someone else reads

What is I/O bus? E.g., PCI:A Typical PC Bus Structure

- Incredible variety of I/O devices
  - Storage
  - Transmission
  - Human-interface
- Common concepts – signals from I/O devices interface with computer
  - **Port** – connection point for device
  - **Bus** - **daisy chain** or shared direct access
    - **PCI** bus common in PCs and servers, PCI Express (**PCIe**)
    - **expansion bus** connects relatively slow devices
    - **Serial-attached SCSI** (**SAS**) common disk interface



How Devices Connect: Controllers & Adapters

The "Translators" Between Bus and Device

```
Device (e.g., Disk)
  |
Controller / Host Adapter
  |
Bus (e.g., PCIe, SATA)
  |
CPU/Memory
```

Controller (**host adapter**) – electronics that operate port, bus, device

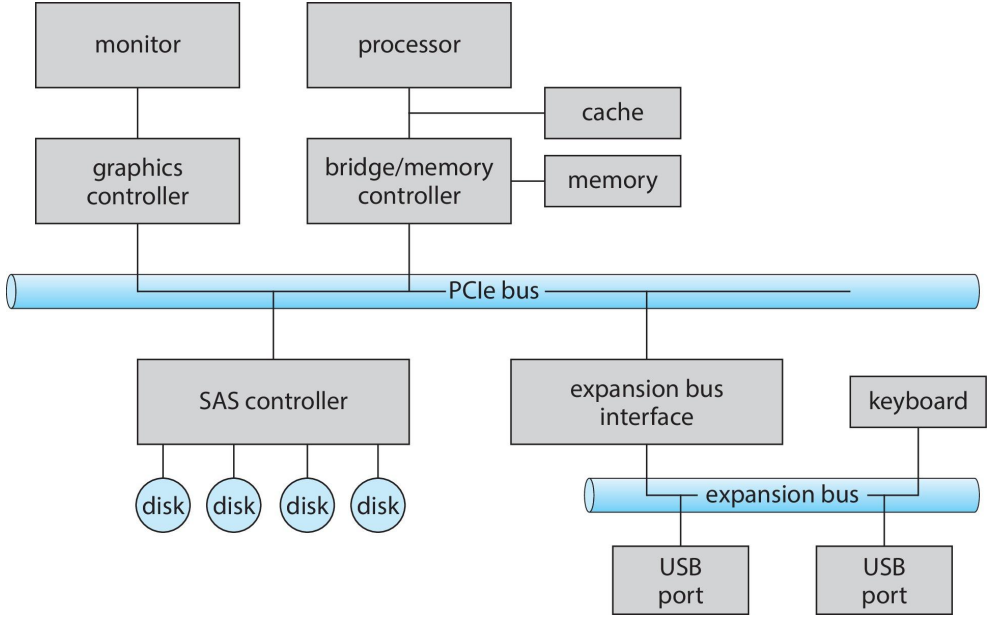
- Sometimes integrated
- Sometimes separate circuit board (host adapter)
- Can have its own processor, microcode, private memory, bus controller, etc.
    - Some talk to per-device controller with bus controller, microcode, memory, etc.

Controller Responsibilities:

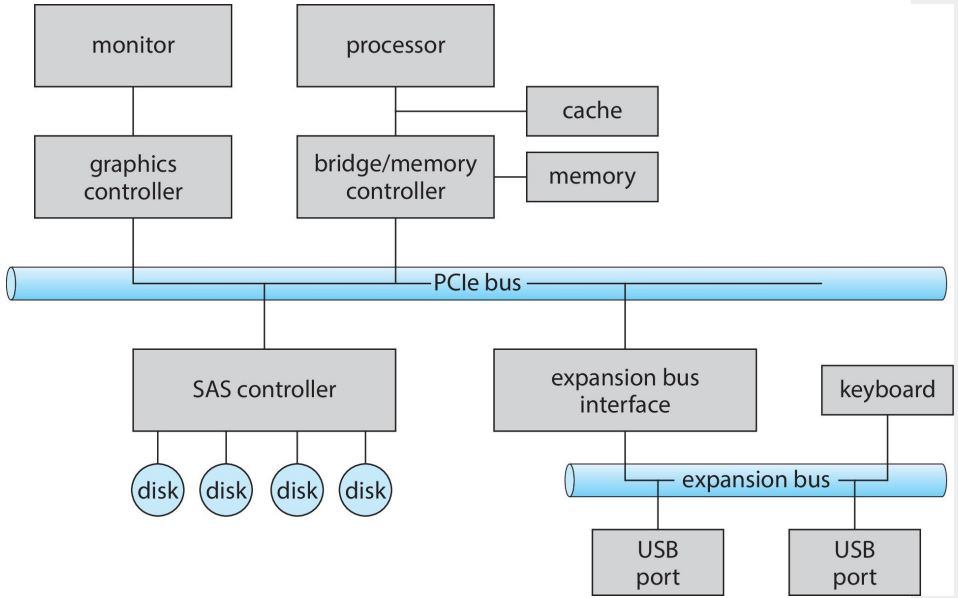
- Owns registers for communication
- Translates high-level commands to device-specific operations
- Example: Disk controller, NIC (Network Interface Card)

Device drivers

- I/O management is a major component of operating system design and operation
  - Important aspect of computer operation
  - I/O devices vary greatly
  - Various methods to control them
  - Performance management
  - New types of devices frequent
- Ports, busses, device controllers connect to various devices
- **Device drivers** encapsulate device details
  - Present uniform device-access interface to I/O subsystem



Communicating with Devices



- communicating with devices **through registers**
  - Data-in register, data-out register, status register, control register
  - Typically 1-4 bytes, or FIFO buffer

- Two Methods for Register Access
  - Method 1: Port-Mapped I/O (PMIO)
  - Method 2: Memory-Mapped I/O (MMIO)
    - a certain portion of the processor's address space is mapped to the device

communicating with devices through registers

data-in register is read by the host to get input from the device.

data-out register is written by the host to send output.

status register has bits read by the host to ascertain the status of the device, such as idle, ready for input, busy, error, transaction complete, etc.

control register has bits written by the host to issue commands or to change settings of the device such as parity checking, word length, or full- versus half-duplex operation.

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Device I/O Port Locations on PCs (partial)

Method 1: Port-Mapped I/O (PMIO)

The "Special Instructions" Approach

- Dedicated I/O address space separate from memory
- Special CPU instructions: in and out
- Limited to 65,536 ports (16-bit addressing)

x86 I/O instructions

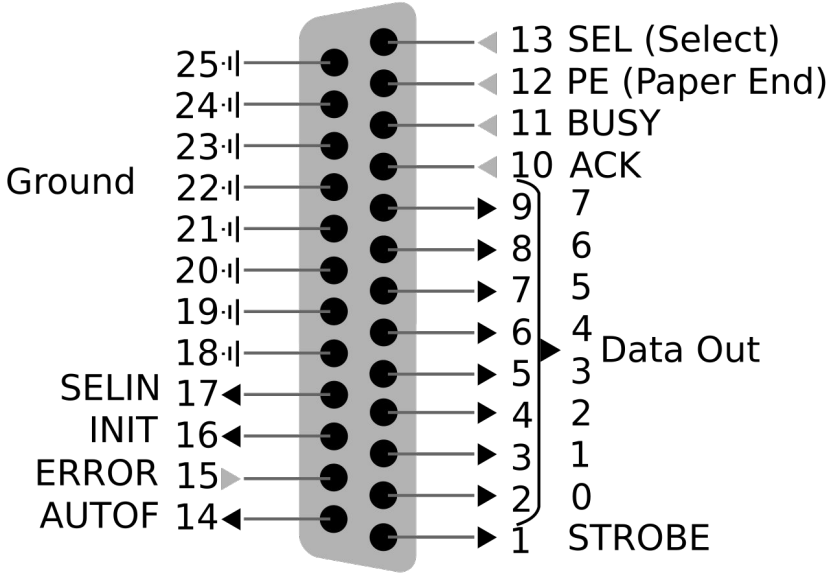
```
// Read from port (byte)
static inline uint8_t inb(uint16_t port) {
    uint8_t data;
    asm volatile("inb %w1, %b0" : "=a"(data) : "Nd"(port));
    return data;
}

// Write to port (byte)
static inline void outb(uint16_t port, uint8_t data) {
    asm volatile("outb %b0, %w1" : : "a"(data), "Nd"(port));
}

// Read multiple words (16-bit)
static inline void insw(uint16_t port, void *addr, size_t cnt) {
    asm volatile("rep insw" : "+D"(addr), "+c"(cnt) : "d"(port));
  }
```

https://man7.org/linux/man-pages/man2/outb.2.html
https://www.scs.stanford.edu/24wi-cs212/notes/io_disks.pdf

Port Address Map (PC Example):

Address Range	Device
0x378-0x37F	Parallel Port 1 (LPT1)
0x3F8-0x3FF	Serial Port 1 (COM1)
0x1F0-0x1F7	Primary IDE Controller
0x200-0x20F	Game Controller

Example LPT1



Simple hardware has three control registers:

Bit-to-pin mapping for the Standard Parallel Port (SPP):

Address		MSB							LSB
	Bit:	7	6	5	4	3	2	1	0
Base (Data port)	Pin:	9	8	7	6	5	4	3	2
Base+1 (Status port)	Pin:	~11	10	12	13	15			
Base+2 (Control port)	Pin:					~17	16	~14	~1

~ indicates a hardware inversion of the bit.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

read/write data register (port 0x378)

$\overline{\text{BSY}}$	$\overline{\text{ACK}}$	PAP	OFON	$\overline{\text{ERR}}$	–	–	–

read-only status register (port 0x379)

–	–	–	IRQ	DSL	$\overline{\text{INI}}$	ALF	STR

read/write control register (port 0x37a)

Writing a byte to parallel port

```
/* Send one byte to a parallel port printer */
void sendbyte(uint8_t byte) {
    /* Step 1: Wait until printer is ready (BUSY=1) */
    while ((inb(0x379) & 0x80) == 0)  // Check bit 7 (BUSY)
        delay();

    /* Step 2: Put data on output pins */
    outb(0x378, byte);
    /* Step 3: Pulse STROBE line (tell printer: "Data ready!") */
    uint8_t ctrlval = inb(0x37a);
    outb(0x37a, ctrlval | 0x01);  // Set STROBE=1
    delay();
    outb(0x37a, ctrlval);         // Set STROBE=0
 }
```

Parallel port - OSDev Wiki
https://www.scs.stanford.edu/24wi-cs212/notes/io_disks.pdf

IDE disk driver

```
void IDE_ReadSector(int disk, int off, void *buf){
   outb(0x1F6, disk == 0 ? 0xE0 : 0xF0); // Select Drive
   IDEWait();
   outb(0x1F2, 1);          // Read length (1 sector = 512 B)
   outb(0x1F3, off);        // LBA(linear block address) low
   outb(0x1F4, off >> 8);   // LBA mid
   outb(0x1F5, off >> 16); // LBA high
   outb(0x1F7, 0x20);       // Read command
   insw(0x1F0, buf, 256);   // Read 256 words
}
void IDEWait() {
   // Discard status 4 times
   inb(0x1F7);   inb(0x1F7);   inb(0x1F7);   inb(0x1F7);
   // Wait for status BUSY flag to clear
   while ((inb(0x1F7) & 0x80) != 0)
       ;
}
```

https://www.scs.stanford.edu/24wi-cs212/notes/io_disks.pdf

Summary of I/O instructions

- in/out instructions slow and clunky
  - Instruction format restricts what registers you can use
  - Only allows 2^16 different port numbers
  - Per-port access control turns out not to be useful (any port access allows you to disable all interrupts)
-

Alternative use **memory mapped I/O**

- Devices can achieve same effect with physical addresses, e.g.:

```
volatile int32_t *device_control
= (int32_t *) (0xc0100 + PHYS_BASE);
*device_control = 0x80;
int32_t status = *device_control;
```

  - OS must map physical to virtual addresses, ensure non-cacheable
- Assign physical addresses at boot to avoid conflicts. PCI:
  - Slow/clunky way to access configuration registers on device
  - Use that to assign ranges of physical addresses to device

Method 2: Memory-Mapped I/O (MMIO)

- Device registers mapped into physical memory address space
  - **The "Memory as Interface" Approach**
- Accessed using normal memory read/write instructions
- No special in/out instructions needed
- suitable for devices which must move large quantities of data quickly
  - such as graphics cards.

- This can be used in combination with traditional registers:
  - For example, graphics cards still use registers for control information such as setting the video mode.
- A potential problem exists,
  - what if a process is allowed to write directly to the address space used by a memory-mapped I/O device.

- **Note this is NOT the same as DMA**

```c
// Device registers appear as memory locations
volatile uint32_t *device_control = (uint32_t*)(0xF0000000);
volatile uint32_t *device_status = (uint32_t*)(0xF0000004);
volatile uint32_t *device_data = (uint32_t*)(0xF0000008);

// Write to control register (just like writing to memory)
*device_control = 0x80;  // Send "ENABLE" command

// Read from status register (just like reading memory)
while ((*device_status & 0x01) == 0) {
    // Wait for READY bit
}

// Read data
uint32_t data = *device_data;
```

Modern NVMe SSD (MMIO)

```
void NVMe_ReadBlock(uint64_t lba, void *buffer) {
    // Command submission (MMIO)
    volatile NVMe_Registers *regs =
        (NVMe_Registers*)NVME_BASE_ADDRESS;

    // Build command in memory (not shown)
    // ...

    // Ring doorbell (MMIO write)
    regs->doorbell = command_id;

    // Wait for completion (MMIO read)
    while (!(regs->status & COMPLETION_BIT));

    // Data already in memory via DMA
  }
```

PMIO vs MMIO: Side-by-Side Comparison

Feature	Port-Mapped I/O (PMIO)	Memory-Mapped I/O (MMIO)
Address Space	Separate I/O space (64K)	Part of main memory space
CPU Instructions	Special: `in`, `out`	Normal: `load`, `store`
Protection	I/O privilege level (Ring 0)	Page table permissions
Caching	Usually uncacheable	Can be cacheable (with care!)
Performance	Historically slower	Faster with cache hits
CPU Complexity	Needs I/O instructions	Simpler CPU design
Example Use	Legacy PC devices (LPT, COM)	Modern devices (GPU, PCIe)
Architecture Support	x86 (has both)	ARM, RISC-V (MMIO only)

The Synchronization Problem

How the CPU knows when devices are ready

The Challenge:

- CPU runs at nanosecond scale
- Devices operate at microsecond to millisecond scale
- How to coordinate without wasting CPU cycles?

Polling vs Interrupts

How should driver synchronize with card?

- **Device driver provides several entry points to kernel**
  - **Reset, ioctl, output, interrupt, read, write, strategy . . .**

- **How should driver synchronize with card?**
  - **E.g., Need to know when transmit buffers free or packets arrive**
  - **Need to know when disk request complete**

- **Two fundamental approach**
  - **pooling**
  - **interrupt driven devices**

https://www.scs.stanford.edu/24wi-cs212/notes/io_disks.pdf

Option 1: POLLING (Ask Repeatedly)

CPU: "Are you ready? Are you ready? Are you ready?"

Analogy: Constantly checking your watch while waiting for a bus

Option 2: INTERRUPTS (Wait for Notification)

CPU: "Do other work. Device: 'Ping!' when ready."

Analogy: Sitting on a bench reading a book, looking up when the bus arrives

Approach 1: Polling (Busy-Wait)

The "Keep Asking" Method

How Polling Works:

1. **CPU repeatedly reads device status register**
2. **Waits for specific bit to change (e.g., READY=1, BUSY=0)**
3. **When ready, CPU performs data transfer**

- Example from IDE driver:

```
while ((inb(0x1F7) & 0x80) != 0);
```

Approach 1: Polling (Busy-Wait)

```
void IDEWait() {
    // Poll: Keep reading status until BUSY=0
    while ((inb(0x1F7) & 0x80) != 0) {
        // Just wait... doing nothing useful
    }
}

void ReadFromDisk() {
    outb(0x1F7, READ_COMMAND);  // Send read command

    IDEWait();  // Poll until disk ready

    // Now read the data
    insw(0x1F0, buffer, 256);
  }
```

Step 1 is **busy-wait** cycle to wait for I/O from device

- Reasonable if device is fast
- But inefficient if device slow
- CPU switches to other tasks?
    - But if miss a cycle data overwritten / lost

When Polling Works Well

Polling is **EFFICIENT** when:

Device is very fast (response time < switching to another task)

- Predictable timing (know exactly when to check)
- Simple implementation needed (no interrupt handler setup)
- Real-time systems where you can't tolerate interrupt latency
- High-performance networking (modern NICs with busy-polling)

```
// Checking a flag in shared memory (very
fast)
while (!(*device_flag)) {
    // May be empty or very short wait
}
process_data(*device_buffer);
```

⚠️ CRITICAL DISTINCTION ⚠️:

Linux poll()/select() syscalls for sockets = User-space blocking calls that wait efficiently

Busy-wait polling = Kernel driver spinning in a loop checking device

Similar idea but these are COMPLETELY DIFFERENT mechanisms!

Polling: Disadvantages

- Can’t use CPU for anything else while polling

```
CPU: "READY?" → Device: "BUSY"

CPU: "READY?" → Device: "BUSY"

CPU: "READY?" → Device: "BUSY"

... (repeats thousands of times) ...

CPU: "READY?" → Device: "READY!"

CPU: Reads data
```

```
// 99.9% of these checks are
unnecessary!
while (!device_ready) {
    // Wasting billions of
cycles
}
```

Quantitative Example:

```
Disk read takes ~10ms (10,000,000 nanoseconds)

CPU checks status every ~10ns (1 billion times while waiting!)

99.999% of CPU time wasted just checking status!
```

Polling: Disadvantages

- Poor Responsiveness: Can't handle other tasks while waiting
- Power Inefficiency: CPU stays active, generating heat
- Scalability Issues: What if waiting for 10 devices?
- Schedule poll in future?
  - High latency to receive packet or process disk block bad for response time

```
// Horrible approach!
while (!device1_ready &&
!device2_ready && ...) {
    // Even worse waste
```

Approach 2: Interrupt-Driven I/O

- Asks card to interrupt CPU on events
  - Interrupts allow devices to notify the CPU
    - when they have data to transfer
    - or when an operation is complete

- This allows the CPU to perform other tasks when no I/O transfers need an immediate attention.

Interrupt handler runs at high priority

Asks card what happened:

- e.g. for NIC xmit buffer free, new packet

This is what most general-purpose OSes do

How Interrupts Work

CPU starts I/O operation, then does other work

Device completes operation, signals interrupt controller

CPU suspends current task, runs interrupt handler

Handler processes the data, returns to previous task

```
// 1. Driver sets up receive buffer
setup_receive_buffer(buffer);

// 2. Enable interrupts for this device
enable_device_interrupts();

// 3. CPU CONTINUES WITH OTHER WORK
process_user_requests();
calculate_results();
// ...

// LATER: Interrupt handler runs automatically
void network_interrupt_handler() {
    // 4. Handle the received packet
    process_packet(buffer);

    // 5. Return to what CPU was doing
}
```

Interrupt driven devices

CPU Interrupt-request line triggered by I/O device

- Checked(sensed) by processor after each instruction

A device's controller **raises** an interrupt by asserting a signal on the interrupt request line.

The CPU catches the interrupt and dispatches the interrupt handler.

- The CPU performs
  - a state save,
  - and transfers control to the interrupt handler routine at a fixed address in memory.

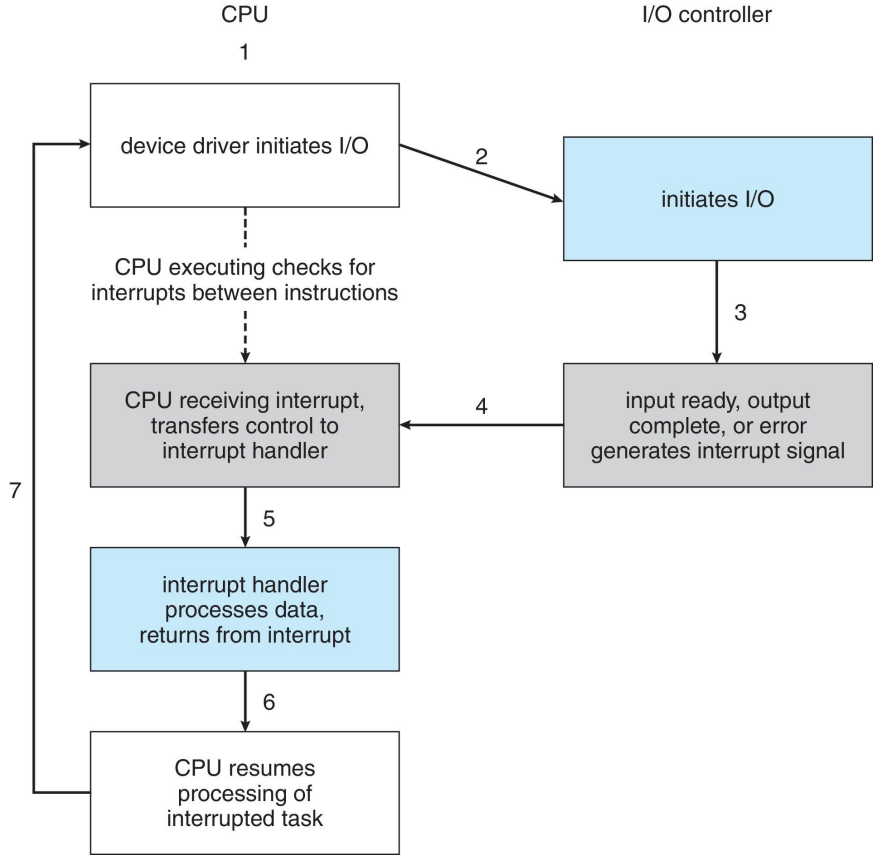
The interrupt handler clears the interrupt by servicing the device.

- The interrupt handler
  - determines the cause of the interrupt,
  - performs the necessary processing,
  - performs a state restore,
  - and executes a **return from interrupt** instruction to return control to the CPU.

Illustration of Interrupt-driven I/O cycle

Device → Interrupt Controller (APIC/IOAPIC) → CPU
When Device Completes Operation:

- Device asserts interrupt request line (IRQ)
- Interrupt Controller prioritizes, signals CPU
- CPU finishes current instruction
- CPU saves state (registers, program counter)
- CPU jumps to interrupt handler (via interrupt vector table)
- Handler processes the interrupt
- Handler clears interrupt at device
- CPU restores state, resumes interrupted task



The previous example does not deal with the following Issues in modern computing

1. The need to defer interrupt handling during critical processing,
2. The need to determine which interrupt handler to invoke, without having to poll all devices to see which one needs attention, and
3. The need for multi-level interrupts, so the system can differentiate between high- and low-priority interrupts for proper response.

https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/13_IOSystems.html

Managing Multiple Interrupt Sources

Modern Systems Have:

Many devices (keyboard, mouse, disk, network, USB, etc.)

All need to interrupt CPU

Need prioritization and management

Interrupt Controller Functions:

Prioritization: Critical interrupts first (keyboard over network)

Masking: Temporarily ignore non-critical interrupts

Routing: Send interrupt to correct CPU (in multi-core)

Vectoring: Tell CPU which handler to run

Interrupt controller on modern architectures

- two **Interrupt-request lines**
  - **Maskable** to ignore or delay some interrupts
  - **Non-maskable** for critical error conditions
- **Interrupt vector**
  - holds the addresses of routines prepared to process specific interrupts.
    - to dispatch interrupt to correct handler
  - Context switch at start and end
  - Based on priority
  - Some **nonmaskable**
  - **Interrupt chaining** if more than one device at same interrupt number

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Intel Pentium processor event-vector table.

x86 Interrupt Vector Table (Partial)

Vector	Purpose
0-31	CPU exceptions (divide by 0, page fault)
32	Timer interrupt
33	Keyboard
34	Cascade to second controller
44	PS/2 Mouse
46	IDE Primary Disk
...	...
128	System Call (software interrupt)

Reminder: Exceptions

- Interrupt mechanism also used for **exceptions**
  - Terminate process, crash system due to hardware error
- Page fault executes when memory access error
- System call executes via **trap** to trigger kernel to execute request
- Multi-CPU systems can process interrupts concurrently
  - If operating system designed to handle it
- Used for time-sensitive processing, frequent, must be fast

Costs of Interrupt-Driven I/O

1. Context Switch Overhead:
   - Save/restore CPU state: 100-1000 cycles
   - Cache pollution (different working set)
2. Interrupt Storm:
   - Fast device (network) can generate 10,000+ interrupts/sec
   - CPU spends all time handling interrupts
3. Priority Inversion:
   - Low-priority task holds lock
   - High-priority interrupt can't proceed

4. Latency Issues

Critical Task Running

↓

Network Interrupt! (must wait)

↓

Handler Runs (delayed)

↓

Critical Task Resumes (late!)

Latency

- Stressing interrupt management because even single-user systems manage hundreds or interrupts per second and servers hundreds of thousands
- For example, a quiet macOS desktop generated 23,000 interrupts over 10 seconds

```
Fri Nov 25 13:55:59                                  0:00:10
                     SCHEDULER     INTERRUPTS
---------------------------------------------
total_samples                13          22998

delays <  10 usecs           12          16243
delays <  20 usecs            1           5312
delays <  30 usecs            0            473
delays <  40 usecs            0            590
delays <  50 usecs            0             61
delays <  60 usecs            0            317
delays <  70 usecs            0              2
delays <  80 usecs            0              0
delays <  90 usecs            0              0
delays < 100 usecs            0              0
total  < 100 usecs           13          22998
```

on linux
$ cat /proc/interrupts

Hybrid Approach: Interrupt + Polling

Problem: High-speed network card interrupts for every packet (too many!)

Solution: NAPI (New API) in Linux

1. First packet: Use interrupt (wake up CPU)
2. Subsequent packets: Poll while processing
3. When queue empty: Re-enable interrupts

```
void network_handler() {
    // 1. Disable further interrupts
    disable_interrupts();

    // 2. Process ALL pending packets (poll)
    while (packets_in_queue()) {
        process_packet();
    }

    // 3. Re-enable interrupts for next burst
    enable_interrupts();
  }
```

Result: Fewer interrupts, better CPU utilization for bulk transfers.

Beyond Simple Interrupts

Problem with Traditional Model:

- Interrupt per I/O operation → overhead
- Especially bad for async I/O, NVMe, high-speed networking

Modern Solution: Completion Queues & Doorbells

1. CPU posts multiple requests to device (batch)
2. Device processes them asynchronously
3. Device writes completions to shared queue
4. Single interrupt when batch is done (or based on timer)

Example: NVMe SSD Interface

```
// 1. Submit 64 read requests (no waiting)
for (int i = 0; i < 64; i++) {
    post_read_request(queue, lba[i], buffer[i]);
}
ring_doorbell();  // Tell device: "Go!"

// 2. Do other work...

// 3. Later: Single interrupt for all 64 completions
void interrupt_handler() {
    while (completions_in_queue()) {
        process_completion();
    }
  }
```

How Operating Systems Handle Different Devices

Keyboard/Mouse (HID): Pure interrupts

- Low data rate, unpredictable timing
- Want immediate response to keypress

Hard Disk (SATA/NVMe): Interrupts with smart batching

- High latency (ms), high throughput
- OS groups nearby sectors, one interrupt per batch

Network (Gigabit+): Hybrid (NAPI/New API)

- 10,000+ packets/second possible
- Polling during burst, interrupts to wake up

GPU Commands: Often polling

- CPU sends command, polls for completion
- GPU may interrupt for major events (frame done)

USB Devices: Complex hierarchy

- Host controller uses interrupts
- Individual devices handled by host controller driver

Efficient Data Transfer - DMA

The Data Transfer Bottleneck

Even with interrupts, CPU moves every byte (Programmed I/O)

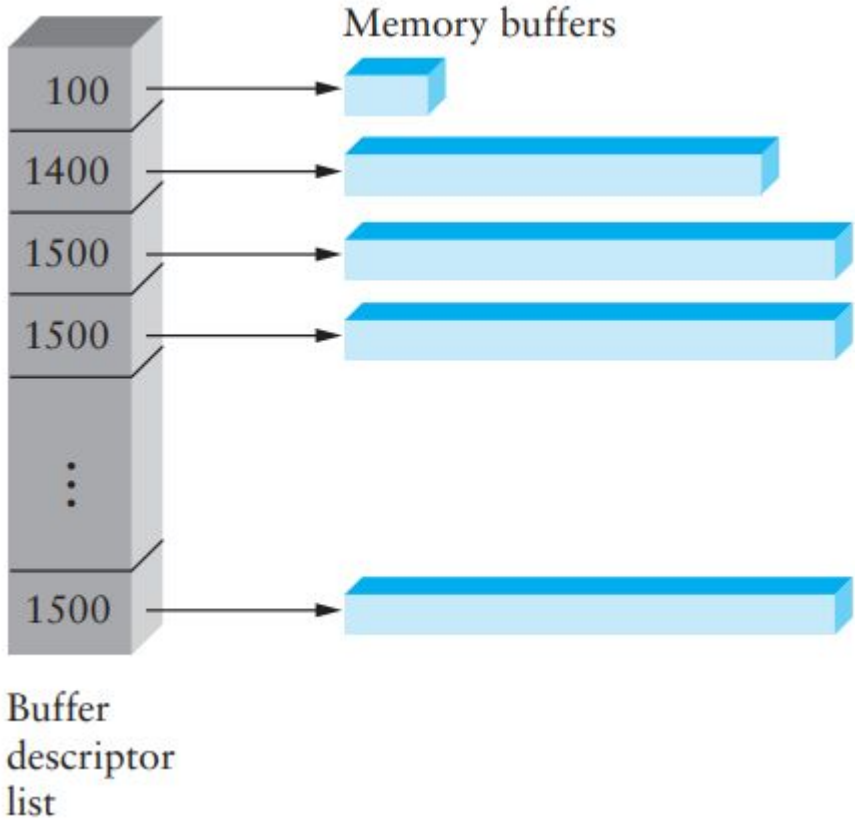
➔ Wasteful for large transfers (1MB disk read → 1M CPU operations)

★ CPU should compute, not copy data!

★ **Idea:** only use CPU to transfer control requests, not data

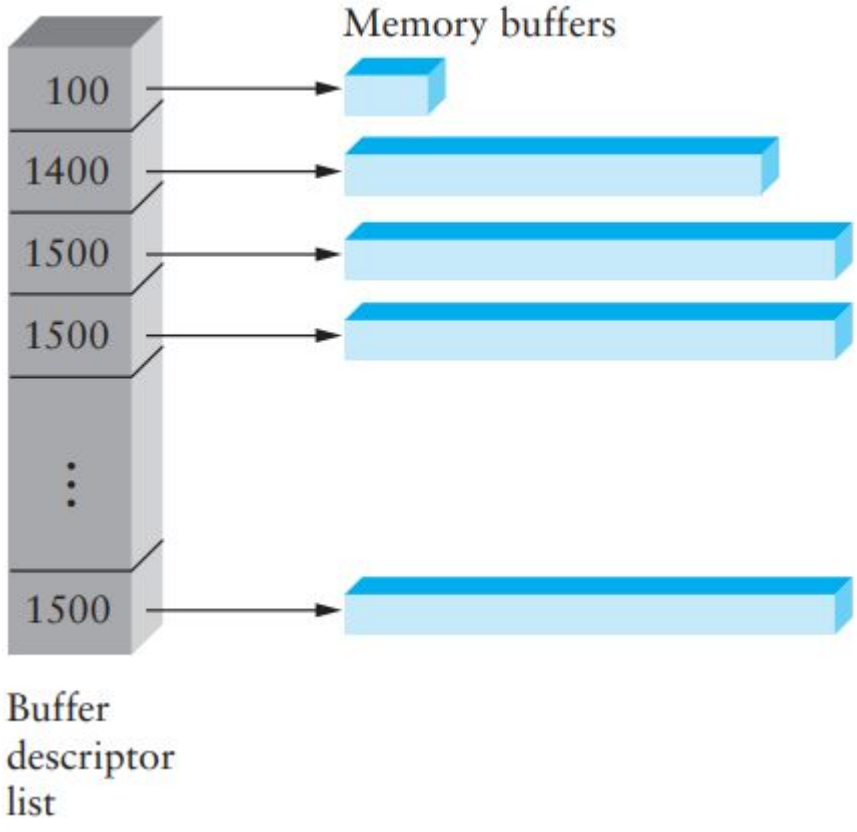
Direct memory access (DMA buffers)

- It is wasteful to tie up the CPU transferring data in and out of registers one byte at a time.
  - Instead this work can be off-loaded to a special processor, known as the Direct Memory Access, DMA, Controller.
- **Idea:** only use CPU to transfer control requests, not data
- Include list of buffer locations in main memory
  - Device reads list and accesses buffers through DMA
  - Descriptions sometimes allow for scatter/gather I/O



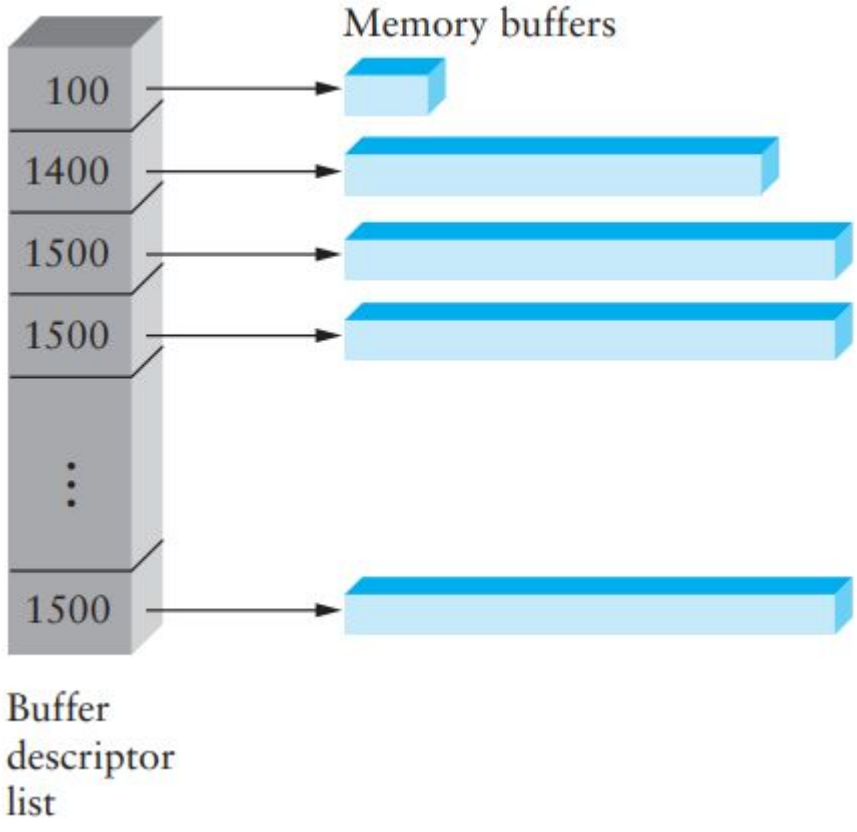
Direct Memory Access

- Used to avoid **programmed I/O** (one byte at a time) for large data movement
- Requires **DMA** controller
- Bypasses CPU to transfer data directly between I/O device and memory
- Version that is aware of virtual addresses can be even more efficient - **DVMA**
  - possible to transfer from one mem to another without main memory chips

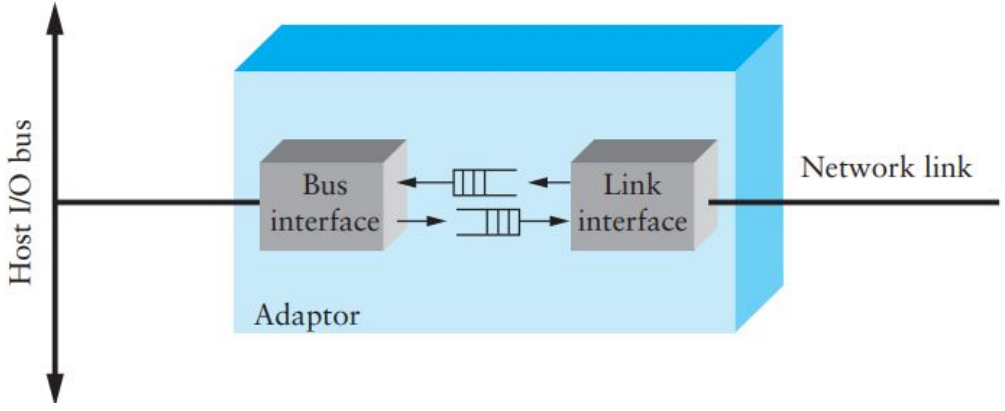


Direct Memory Access: How it works

- CPU programs DMA controller: source, destination, count
- DMA controller takes over bus **(cycle stealing)**
  - but still much more efficient
- Moves data directly between device and memory
- When done, interrupts to signal completion



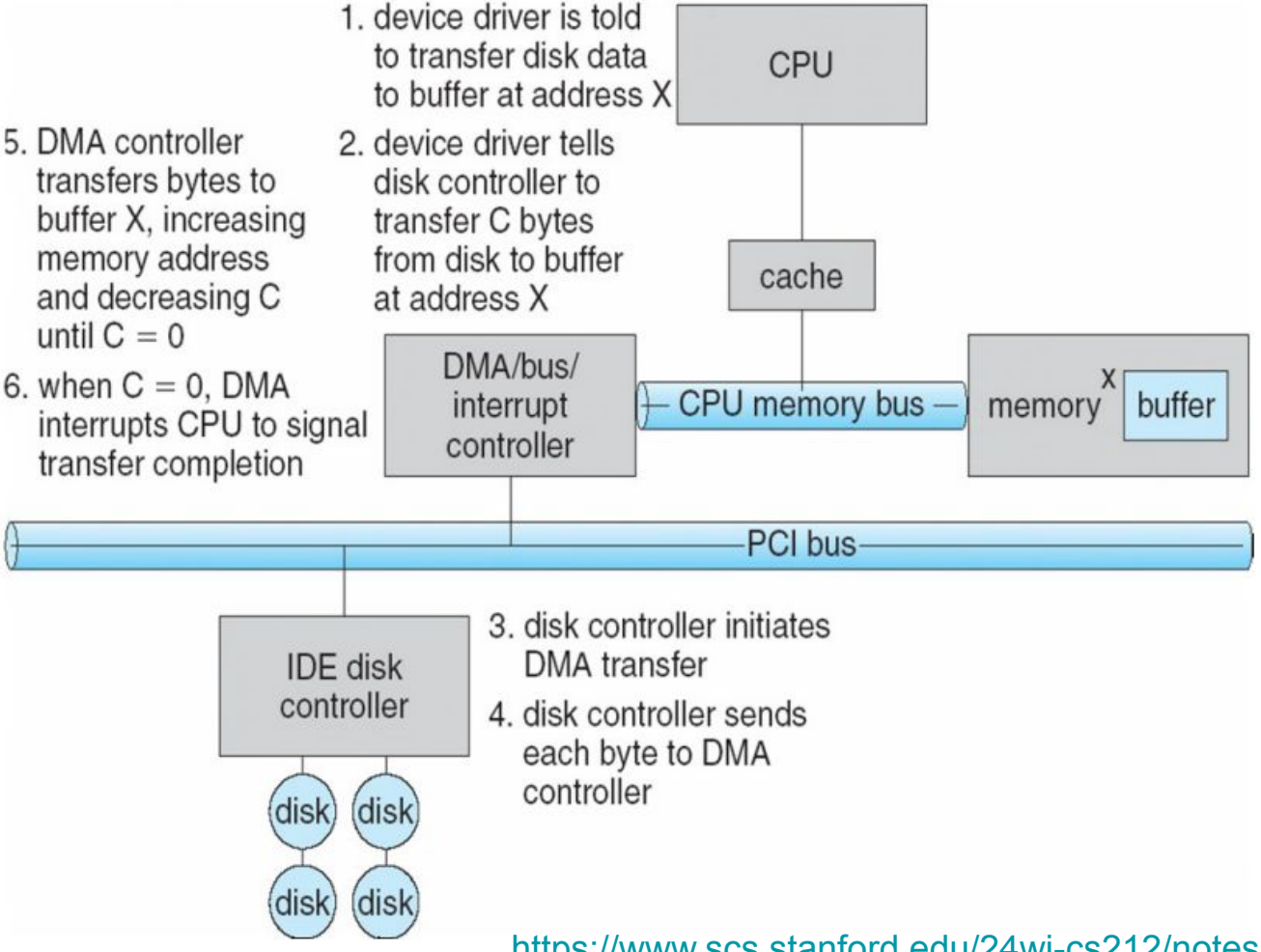
Example: Network Interface Card



- Link interface talks to wire/fiber/antenna
  - Typically does framing, link-layer CRC
- FIFOs on card provide small amount of buffering
- Bus interface logic uses DMA to move packets to and from buffers in main memory

https://www.scs.stanford.edu/24wi-cs212/notes/io_disks.pdf

Example: IDE disk read w. DMA



https://www.scs.stanford.edu/24wi-cs212/notes/io_disks.pdf

Summary: Putting It All together

Complete I/O stack: Application → System Call → Kernel Subsystem → Driver → Device

Efficient I/O = Interrupts (notification) + DMA (data transfer)

CPU involvement: Setup + brief interrupt handler

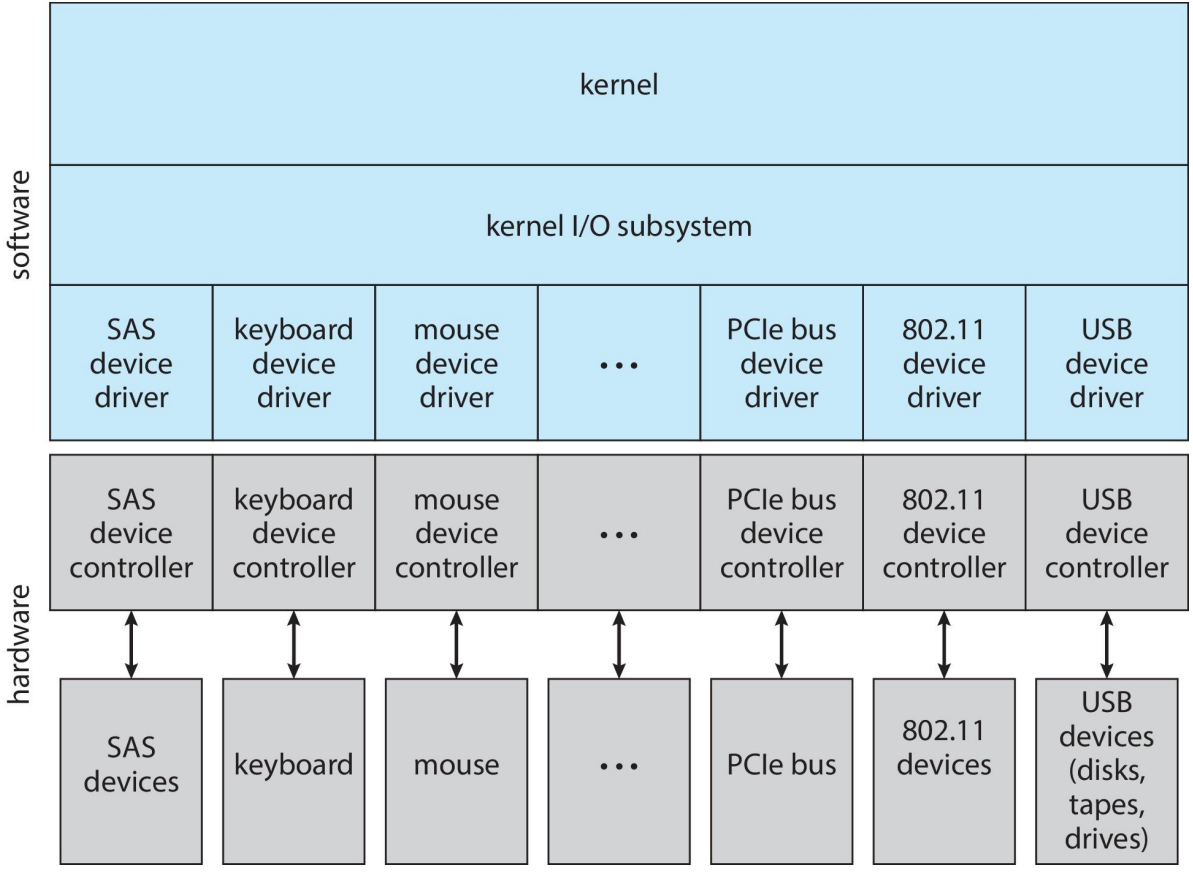
Higher-Level I/O Abstraction

- Application I/O Interface
- Kernel I/O Subsystem Services
- I/O Performance Considerations
- Special Topics
  - Power management
  - STREAMS
  - Life cycle of an I/O Requests

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
- New devices talking already-implemented protocols need no extra work
- Each OS has its own I/O subsystem structures and device driver frameworks
- Devices vary in many dimensions
  - **Character-stream** or **block**
  - **Sequential** or **random-access**
  - **Synchronous** or **asynchronous** (or both)
  - **Sharable** or **dedicated**
  - **Speed of operation**
  - **read-write**, **read only**, or **write only**

A Kernel I/O Structure



Characteristics of I/O Devices

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Characteristics of I/O Devices

- Subtleties of devices handled by device drivers
- Broadly I/O devices can be grouped by the OS into
  - **Block I/O**
  - **Character I/O (Stream)**
  - **Memory-mapped file access**
  - **Network sockets**

- For direct manipulation of I/O device specific characteristics, usually an escape / back door
  - Unix `ioctl()` call to send arbitrary bits to a device control register and data to device data register
- UNIX and Linux use tuple of “major” and “minor” device numbers to identify type and instance of devices (here major 8 and minors 0-4)

```
% ls -l /dev/sda*
```

```
brw-rw---- 1 root disk 8, 0 Mar 16 09:18 /dev/sda
brw-rw---- 1 root disk 8, 1 Mar 16 09:18 /dev/sda1
brw-rw---- 1 root disk 8, 2 Mar 16 09:18 /dev/sda2
brw-rw---- 1 root disk 8, 3 Mar 16 09:18 /dev/sda3
```

Block and Character Devices

- Block devices include disk drives
  - Commands include read, write, seek
  - **Raw I/O**, **direct I/O**, or file-system access
  - Memory-mapped file access possible
    - File mapped to virtual memory and clusters brought via demand paging
  - DMA
- Character devices include keyboards, mice, serial ports
  - Commands include `get()`, `put()`
  - Libraries layered on top allow line editing

Network Devices

- Varying enough from block and character to have own interface
- Linux, Unix, Windows and many others include **socket** interface
  - Separates network protocol from network operation
  - Includes `select()` functionality
- Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)

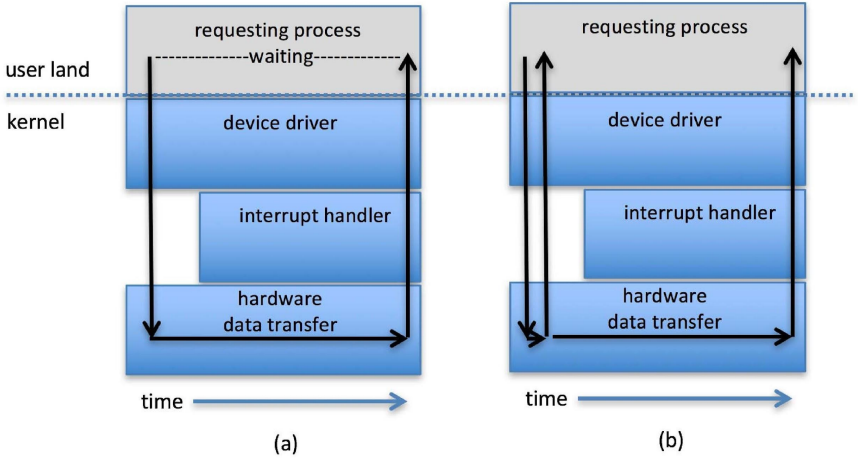
Clocks and Timers

- Provide current time, elapsed time, timer
- Normal resolution about 1/60 second
- Some systems provide higher-resolution timers
-

- **Programmable interval timer** used for timings, periodic interrupts
- `ioctl()` (on UNIX) covers odd aspects of I/O such as clocks and timers

Nonblocking and Asynchronous I/O

- **Blocking** - process suspended until I/O completed
  - Easy to use and understand
  - Insufficient for some needs
- **Nonblocking** - I/O call returns as much as available
  - User interface, data copy (buffered I/O)
  - Implemented via multi-threading
  - Returns quickly with count of bytes read or written
  - `select()` to find if data ready then `read()` or `write()` to transfer
- **Asynchronous** - process runs while I/O executes
  - Difficult to use
  - I/O subsystem signals process when I/O completed



Two I/O methods: (a) synchronous and (b) asynchronous.

Vectored I/O

- **Vectored I/O** allows one system call to perform multiple I/O operations
- For example, Unix `readve()` accepts a vector of multiple buffers to read into or write from
- This scatter-gather method better than multiple individual I/O calls
  - Decreases context switching and system call overhead
  - Some versions provide atomicity
    - Avoid for example worry about multiple threads changing data as reads / writes occurring

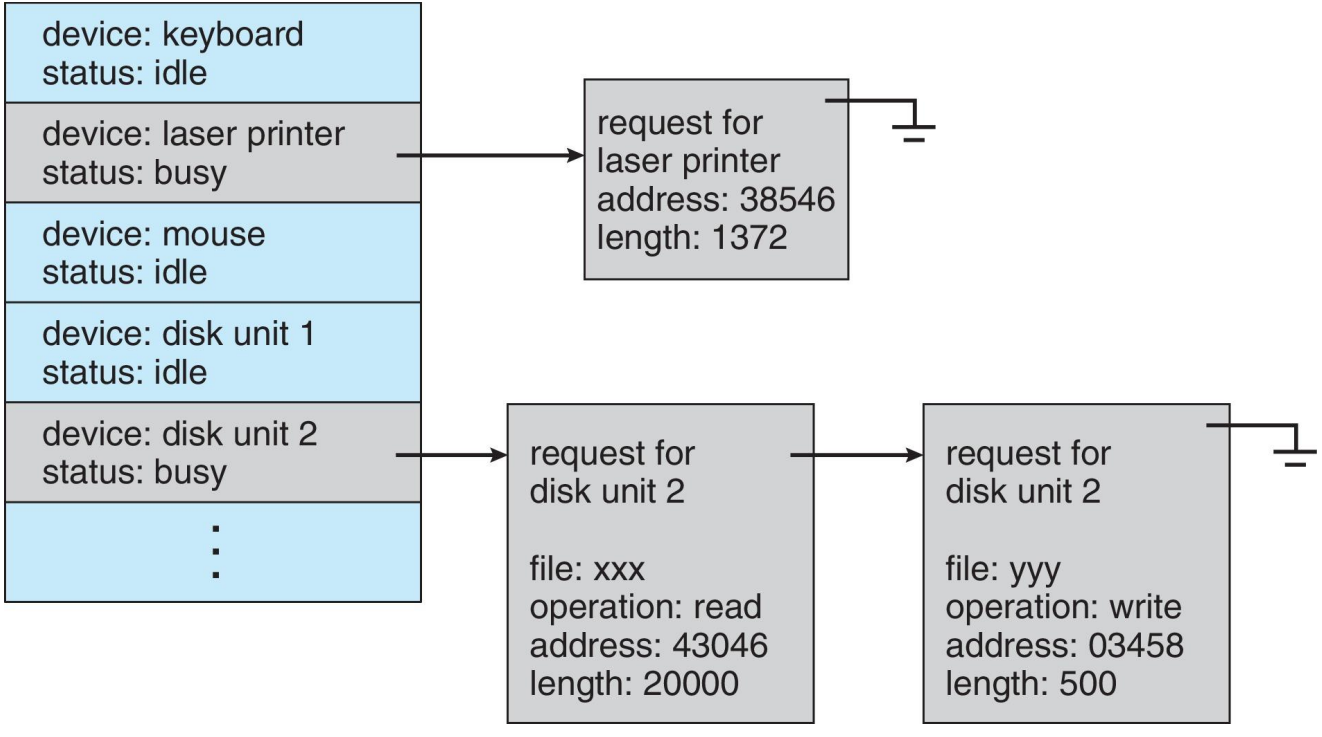
Kernel I/O Subsystem

- **I/O Scheduling**
  - Some I/O request ordering via per-device queue
  - Some OSs try fairness
  - Some implement Quality Of Service (i.e. IPQOS)
- **Buffering** - store data in memory while transferring between devices
  - To cope with device speed mismatch
  - To cope with device transfer size mismatch
  - To maintain “copy semantics”
  - **Double buffering** – two copies of the data
    - Kernel and user
    - Varying sizes
    - Full / being processed and not-full / being used
    - Copy-on-write can be used for efficiency in some cases

Kernel I/O Subsystem

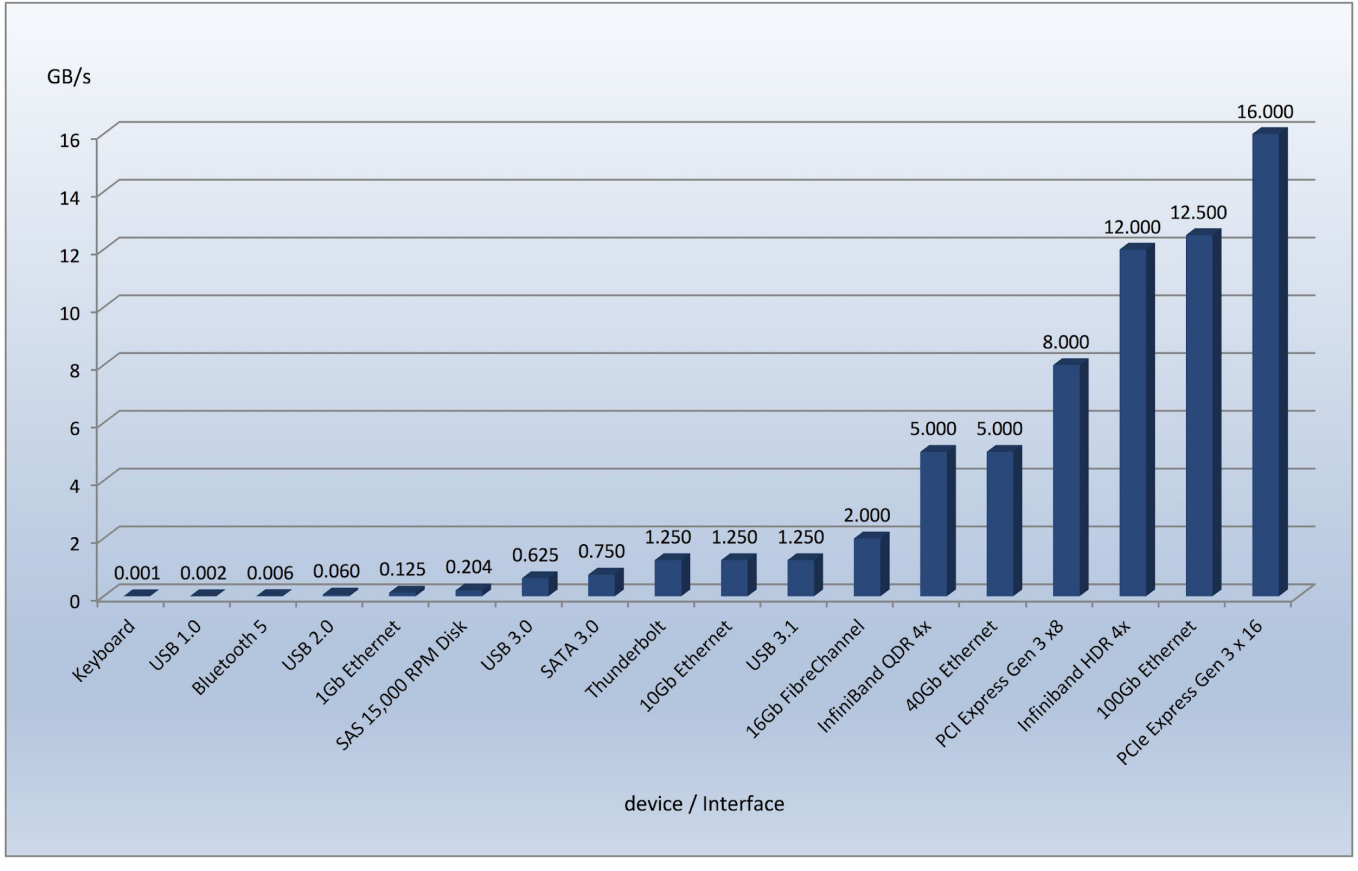
- **Caching** - faster device holding copy of data
  - Always just a copy
  - Key to performance
  - Sometimes combined with buffering
- **Spooling** - hold output for a device
  - If device can serve only one request at a time
  - i.e., Printing
- **Device reservation** - provides exclusive access to a device
  - System calls for allocation and de-allocation
  - Watch out for deadlock

Device-status Table



When a kernel supports asynchronous I/O, it must be able to keep track of many I/O requests at the same time.

Common PC and Data-center I/O devices and Interface Speeds



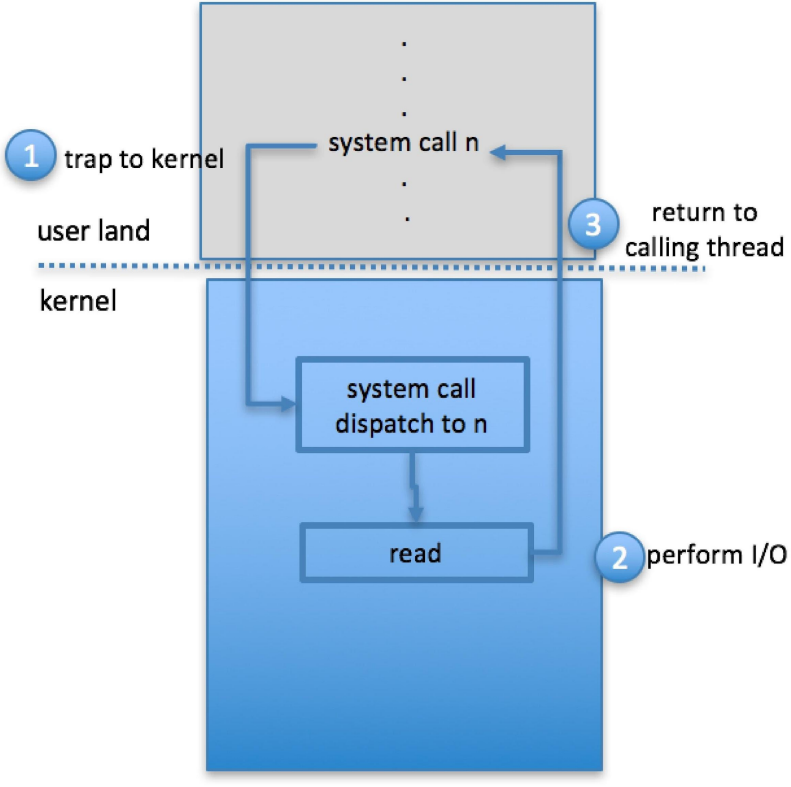
Error Handling

- OS can recover from disk read, device unavailable, transient write failures
  - Retry a read or write, for example
  - Some systems more advanced – Solaris FMA, AIX
    - Track error frequencies, stop using device with increasing frequency of retry-able errors
- Most return an error number or code when I/O request fails
- System error logs hold problem reports

I/O Protection

- User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions
  - All I/O instructions defined to be privileged
  - I/O must be performed via system calls
    - Memory-mapped and I/O port memory locations must be protected too

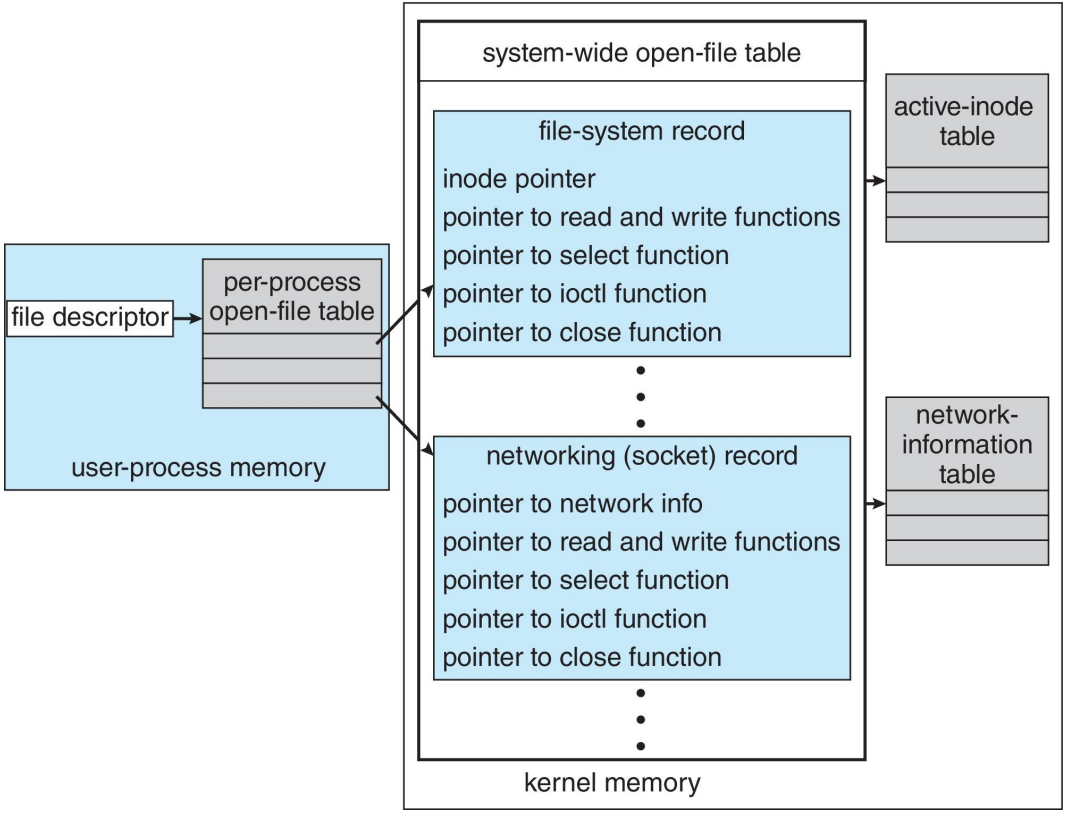
Use of a System Call to Perform I/O



Kernel Data Structures

- Kernel keeps state info for I/O components, including open file tables, network connections, character device state
- Many, many complex data structures to track buffers, memory allocation, “dirty” blocks
- Some use object-oriented methods and message passing to implement I/O
  - Windows uses message passing
    - Message with I/O information passed from user mode into kernel
    - Message modified as it flows through to device driver and back to process
    - Pros / cons?

UNIX I/O Kernel Structure



Power Management

- Not strictly domain of I/O, but much is I/O related
- Computers and devices use electricity, generate heat, frequently require cooling
- OSes can help manage and improve use
  - Cloud computing environments move virtual machines between servers
    - Can end up evacuating whole systems and shutting them down
- Mobile computing has power management as first class OS aspect

Power Management (Cont.)

- For example, Android implements
  - Component-level power management
    - Understands relationship between components
    - Build device tree representing physical device topology
    - System bus -> I/O subsystem -> {flash, USB storage}
    - Device driver tracks state of device, whether in use
    - Unused component – turn it off
    - All devices in tree branch unused – turn off branch
  - Wake locks – like other locks but prevent sleep of device when lock is held
  - Power collapse – put a device into very deep sleep
    - Marginal power use
    - Only awake enough to respond to external stimuli (button press, incoming call)

Power Management (Cont.)

- Modern systems use **advanced configuration and power interface** (**ACPI**) firmware providing code that runs as routines called by kernel for device discovery, management, error and power management

Kernel I/O Subsystem Summary

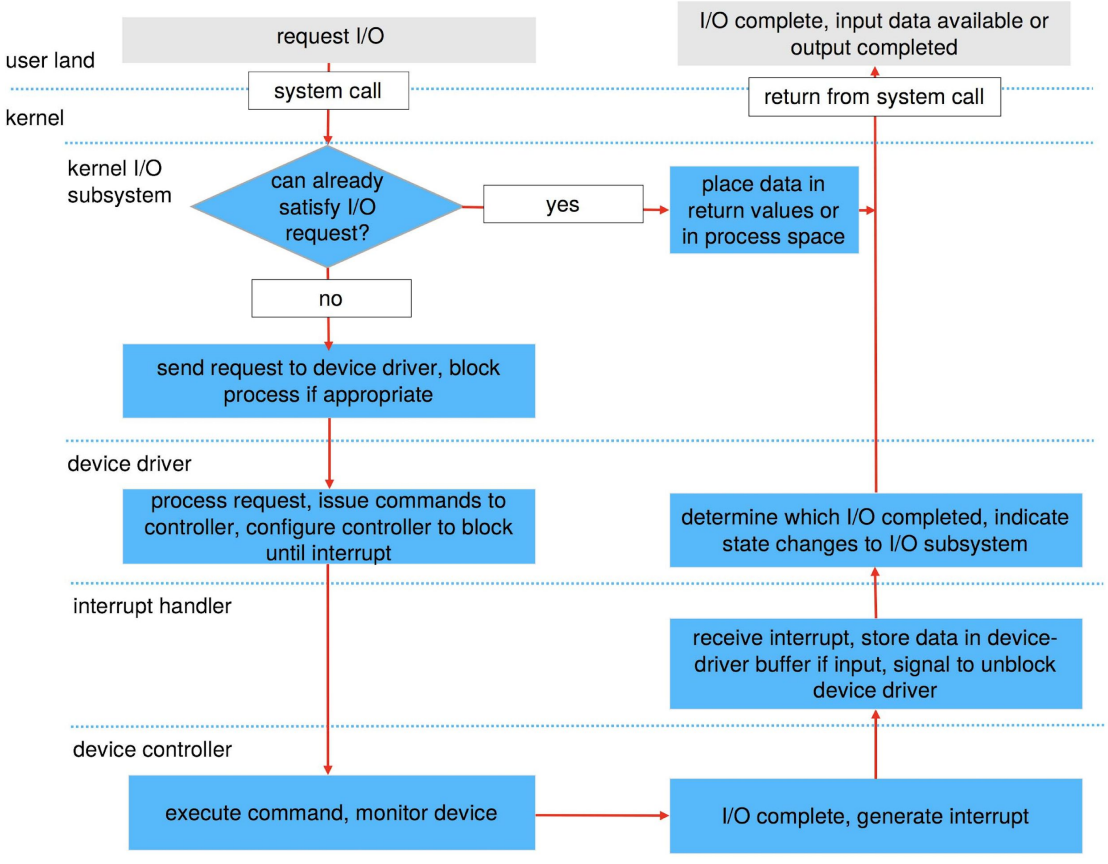
- In summary, the I/O subsystem coordinates an extensive collection of services that are available to applications and to other parts of the kernel
  - Management of the name space for files and devices
  - Access control to files and devices
  - Operation control (for example, a modem cannot seek())
  - File-system space allocation
  - Device allocation
  - Buffering, caching, and spooling
  - I/O scheduling
  - Device-status monitoring, error handling, and failure recovery
  - Device-driver configuration and initialization
  - Power management of I/O devices

- The upper levels of the I/O subsystem access devices via the uniform interface provided by the device drivers

Transforming 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

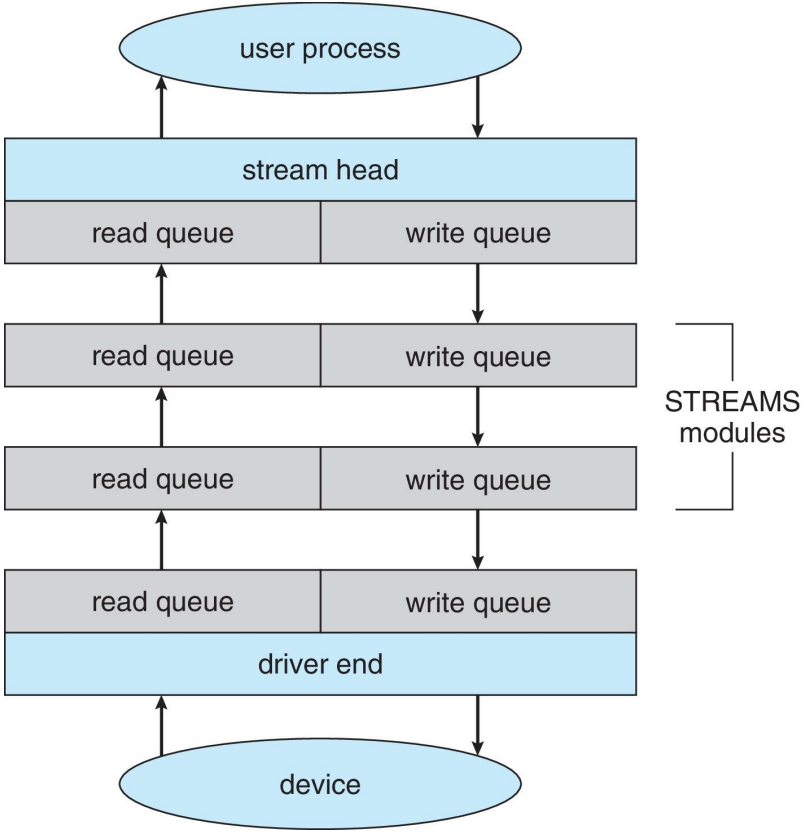
Life Cycle of An I/O Request



STREAMS

- **STREAM** – a full-duplex communication channel between a user-level process and a device in Unix System V and beyond
  - defines standard interfaces for char IO within kernel
- 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
  - **Flow control** option to indicate available or busy
- Asynchronous internally, synchronous where user process communicates with stream head

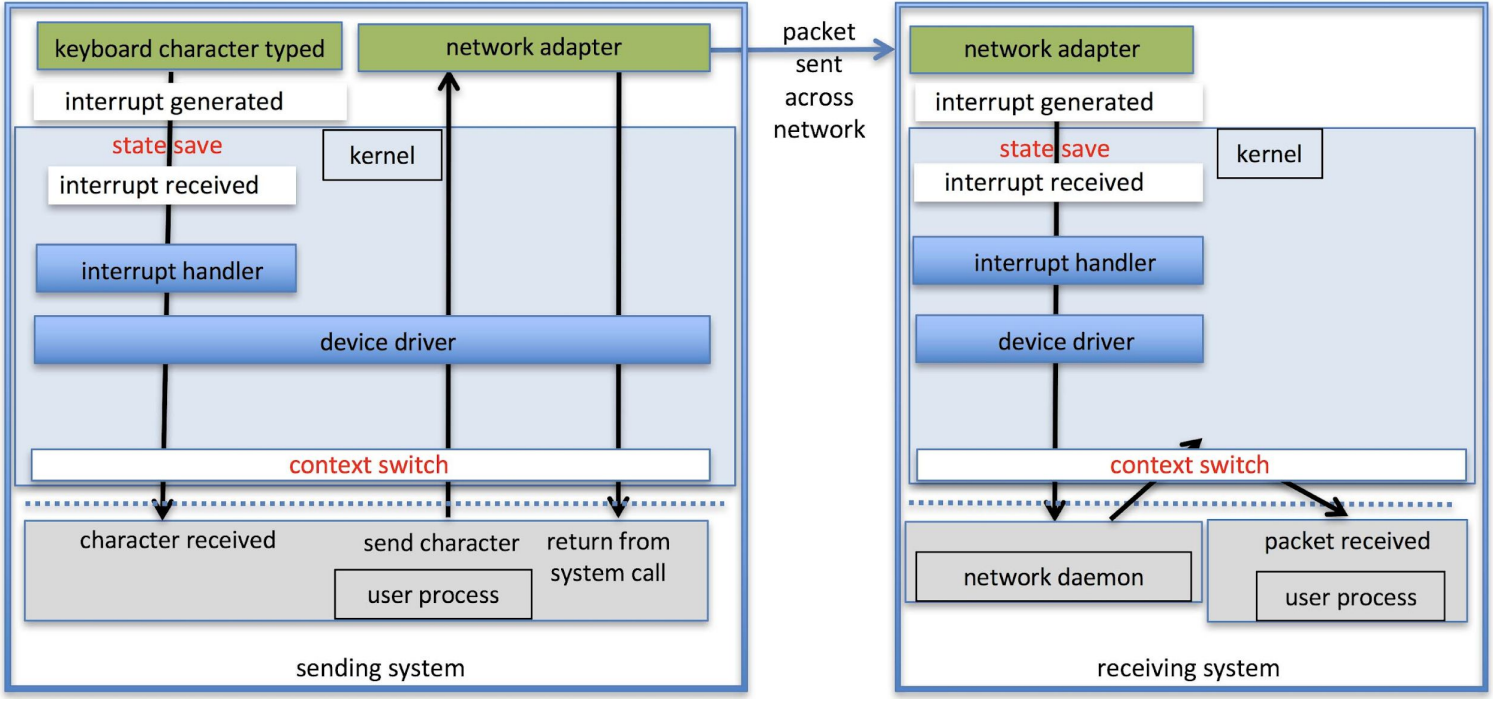
The STREAMS Structure in Unix



Performance

- I/O a major factor in system performance:
  - Demands CPU to execute device driver, kernel I/O code
  - Context switches due to interrupts
  - Data copying
  - Network traffic especially stressful

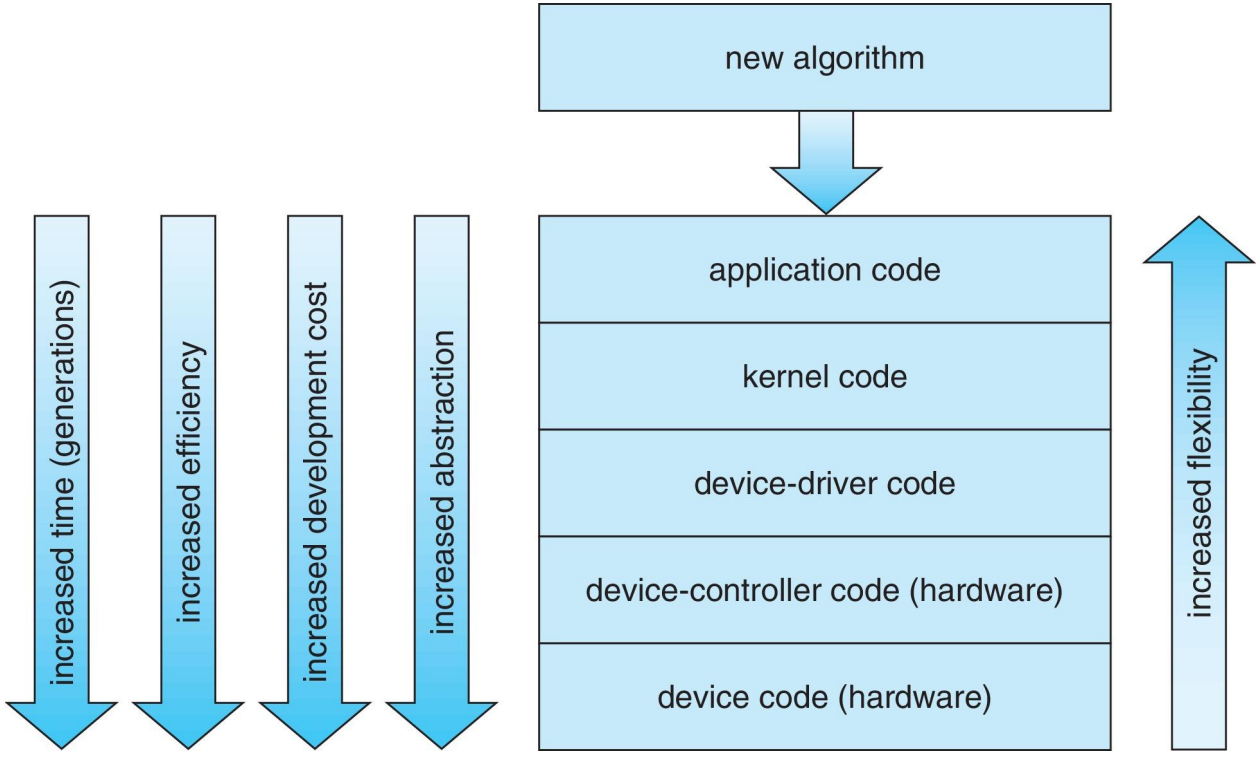
Intercomputer Communications



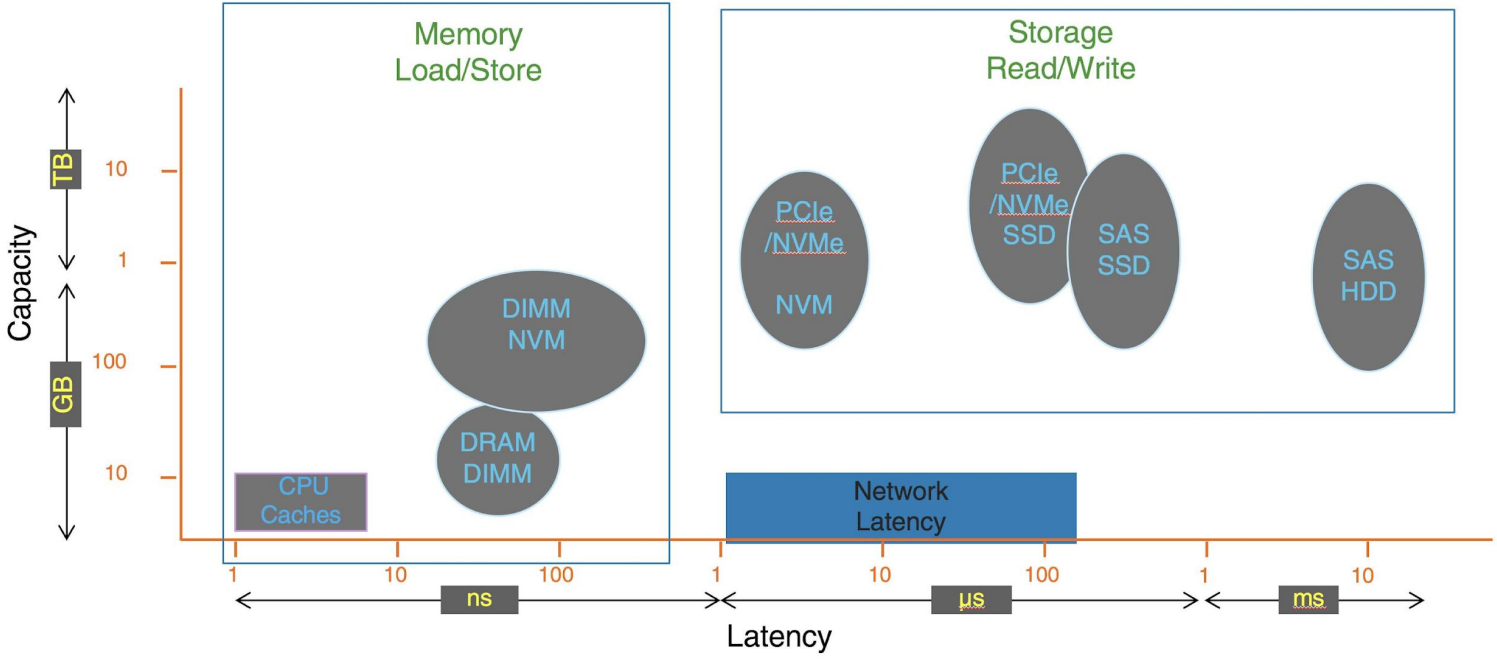
Improving Performance

- Reduce number of context switches
- Reduce data copying
- Reduce interrupts by using large transfers, smart controllers, polling
- Use DMA
- Use smarter hardware devices
- Balance CPU, memory, bus, and I/O performance for highest throughput
- Move user-mode processes / daemons to kernel threads

Device-Functionality Progression



I/O Performance of Storage (and Network Latency)



End of Chapter 12