Chapter 13: I/O Systems

- Device Management Objectives
- Device Characterization
- Device Interface
  - Control
  - Data Transfer
- Kernel I/O Subsystem
  - Device Independent Services
    - Buffering
    - Streams
  - Device Drivers
    - Data Structures

Device Management Objectives

- Abstraction from details of physical devices
- Uniform Naming that does not depend on HW details
- Serialization of I/O-operations by concurrent applications
- Protection of standard-devices against unauthorized accesses
- Buffering, if data from/to a device cannot be stored in the final destination
- Error Handling of sporadic device errors
- Virtualizing physical devices via memory and time multiplexing (e.g., pty, RAM disk)

Characteristics of I/O Devices

- **Block devices** include disk drives
  - Commands include read, write, seek
  - Raw I/O or file-system access
  - Memory-mapped file access possible
- **Character devices** include keyboards, mice, serial ports
  - Commands include `get, put`
  - Libraries layered on top allow line editing
- **Network devices** vary enough from block and character devices to have own interface
  - Unix and Windows include socket interface
    - Separates network protocol from network operation
    - Includes `select` functionality

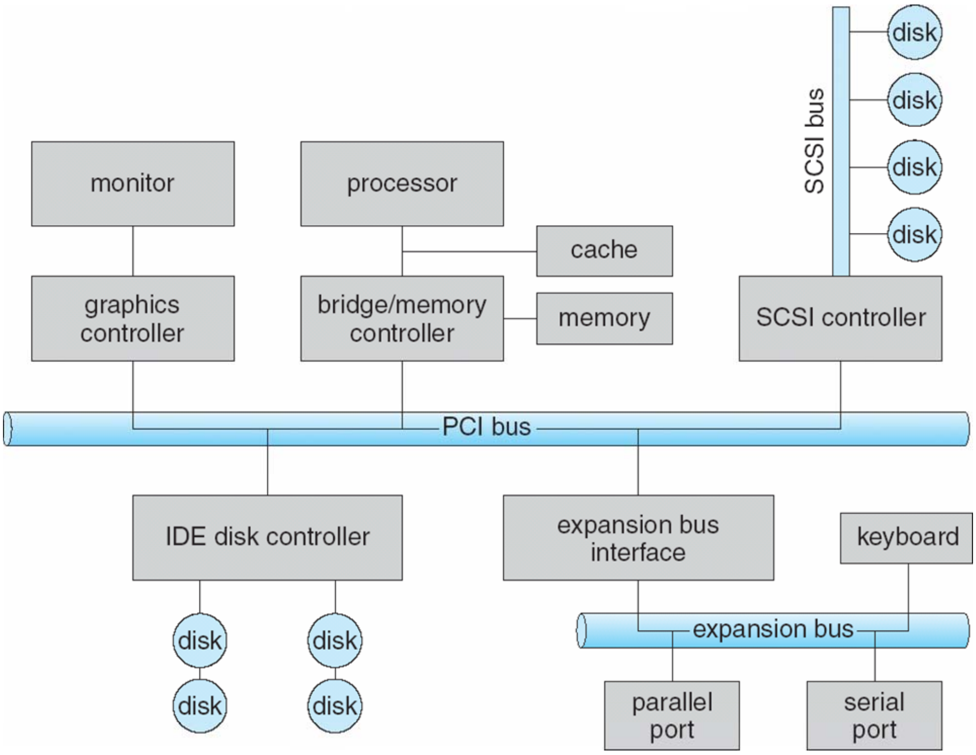
Characteristics of I/O Devices

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

Device Speed

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

A Typical PC Bus Structure



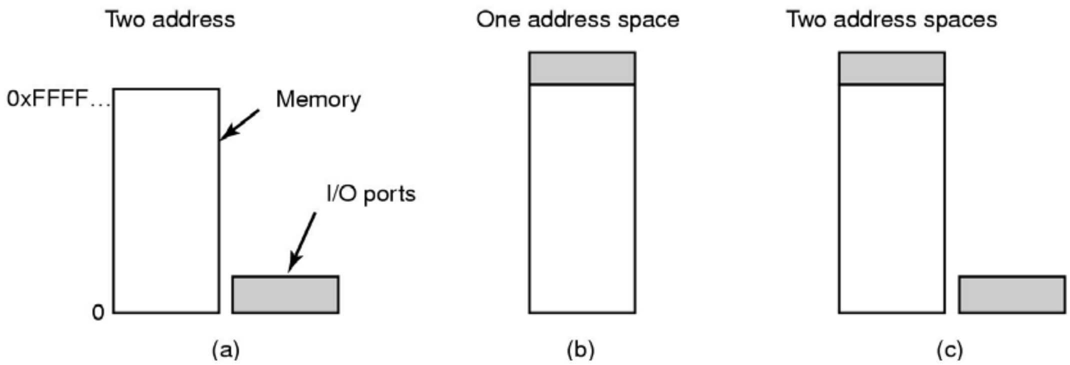
I/O Hardware

- Common components
  - **Controller**
  - **Port** (external connection point)
  - **Bus** (daisy chain or shared direct access)
- Devices have addresses, used by
  - Direct I/O instructions (e.g., to access x86 I/O ports)
  - Memory-mapped I/O
- Device addresses typically point to
  - Status register
  - Control register
  - Data-in register
  - Data-out register

Device I/O Port Locations on PCs (partial)

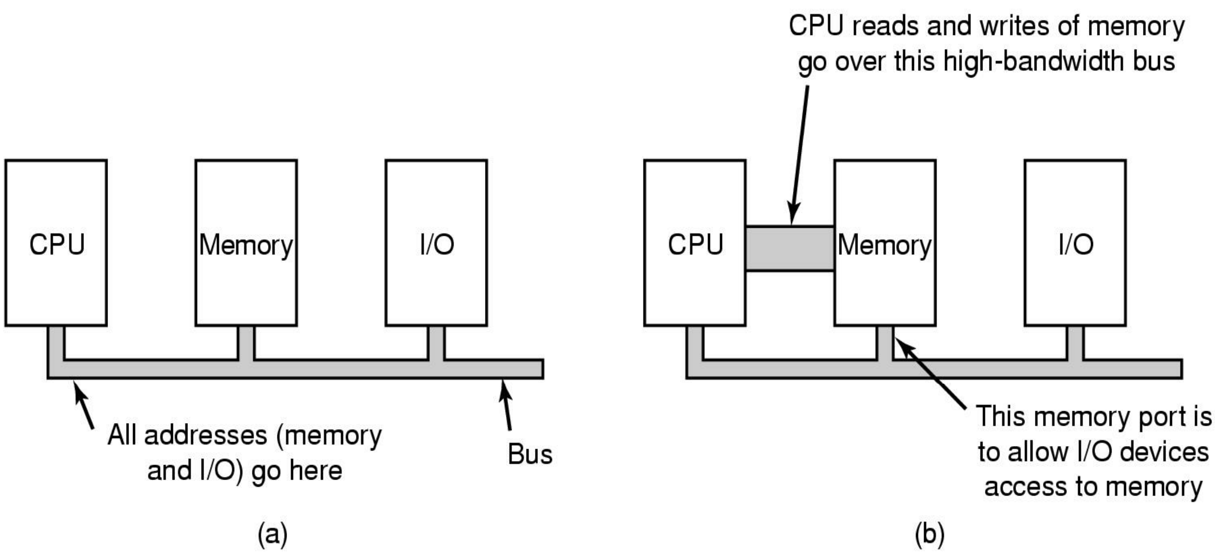
I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Memory-Mapped I/O (1)



- Separate I/O-address space and memory address space
  - MOV R0, 4 // <4> → R0
  - IN R0, 4 // <port 4> →R0
- Memory-mapped I/O // 1 common physical AS
- Hybrid (Pentium) // part of I/O space in memory
  // part in an extra address space

Memory-Mapped I/O (2)



- (a) Single-bus architecture
- (b) Dual-bus memory architecture

Techniques for I/O-Management

- Programmed I/O
  - thread is busy-waiting for the I/O-operation to complete, processor cannot be used else where
- Interrupt-driven I/O
  - I/O-command is issued
  - processor continues executing instructions
  - I/O-device sends an interrupt when I/O-command is done
- Direct Memory Access (DMA)
  - DMA module controls exchange of data between main memory and I/O device
  - processor interrupted after entire block has been transferred
  - bypasses CPU to transfer data directly between I/O device and memory

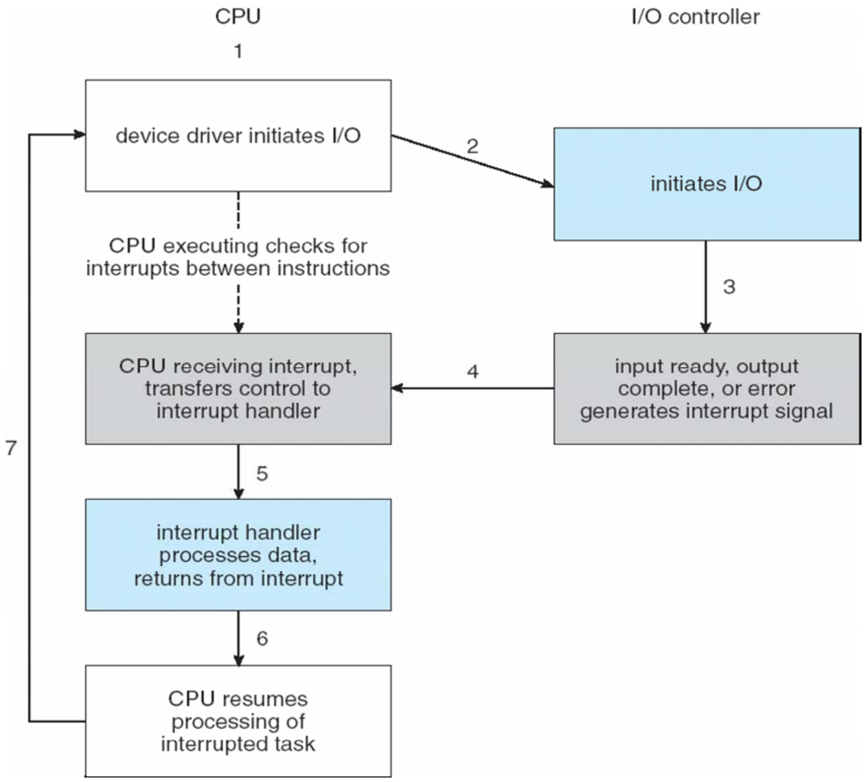
Polling vs. Interrupts

- **Polling** determines state of device with busy-wait cycle to wait for I/O from device
  - command-ready
  - busy
  - Error
- CPU **Interrupt-request line** triggered by I/O device
  - **Interrupt handler** receives interrupts
  - **Maskable** to ignore or delay some interrupts
    - Some **nonmaskable**
  - Interrupt vector to dispatch interrupt to correct handler based on priority
  - Can be executed at almost any time
    - Raise (complex) concurrency issues in the kernel
  - Interrupt mechanism also used for exceptions

Intel Pentium Processor Event-Vector Table

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

Interrupt-Driven I/O Cycle



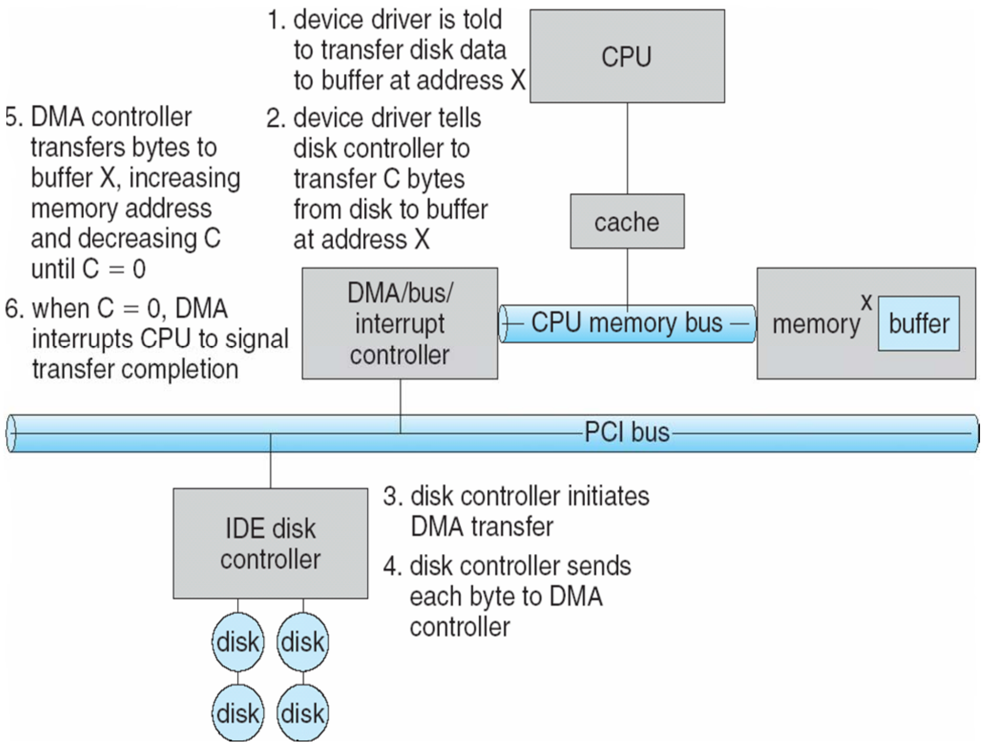
Steps for Handling an Interrupt

1. Save registers not already saved by HW-interrupt mechanism
2. Set up context (address space) for interrupt service procedure
   - Typically, handler runs in the context of the currently running process/task ⇒ not that expensive context switch
3. Set up stack for interrupt service procedure
   - Handler usually runs on the kernel stack of the current process/kernel-level thread
   - Handler cannot block, otherwise the unlucky interrupted process/kernel-thread would also be blocked, might lead to starvation or even to a deadlock
4. Acknowledge/mask interrupt controller, thus re-enable other interrupts

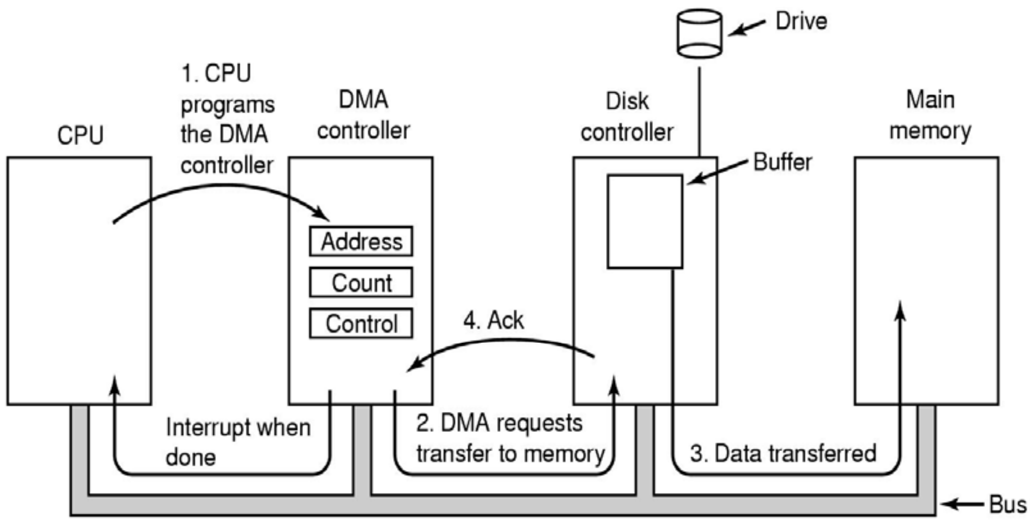
Steps for Handling an Interrupt II

5. Run interrupt service procedure
   - Acknowledges interrupt at device level
   - Figures out what caused the interrupt, e.g.
     - Received a network packet
     - Disk read has properly finished, …
   - If needed, it signals the blocked device driver
6. In some cases, we have to wake up a higher priority process/kernel level thread
   - Potentially schedule another process/kernel-level thread
   - Set up MMU context for process to run next
7. Load new/original process' registers
8. Return from Interrupt, start running new/original process

Six Step Process to Perform DMA Transfer

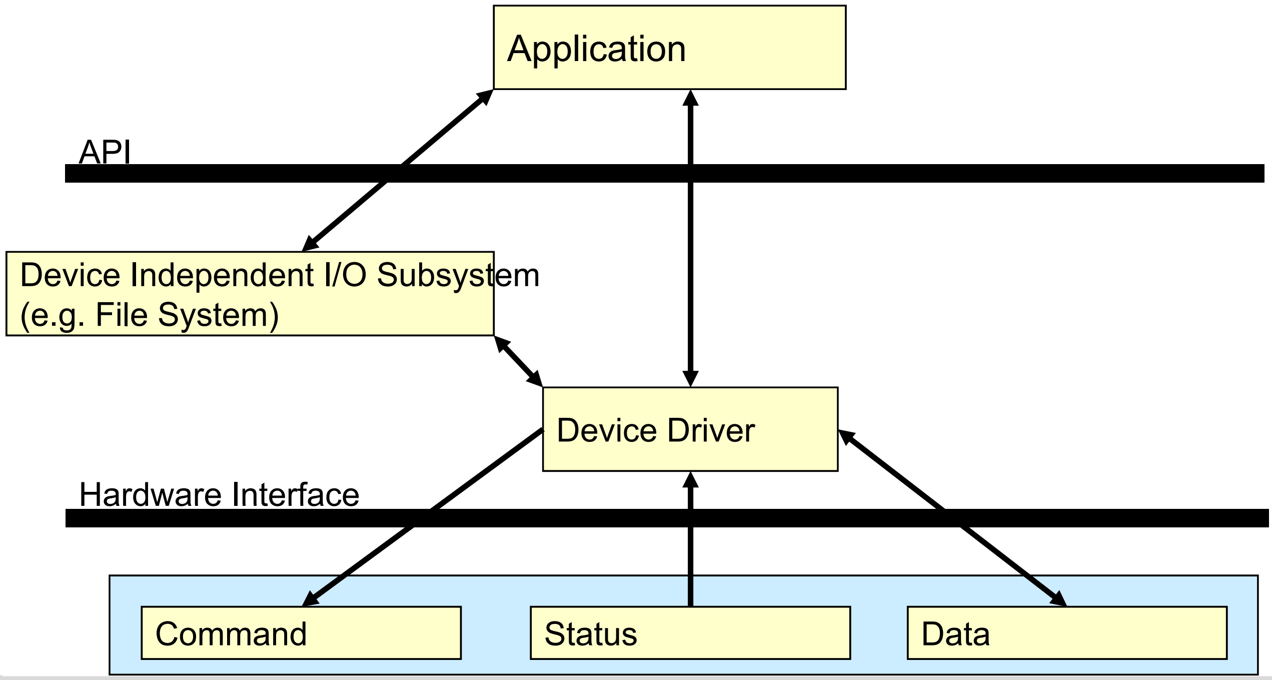


DMA Transfer with Fly-By Mode



- Word Mode (→ cycle stealing)
- Burst Mode

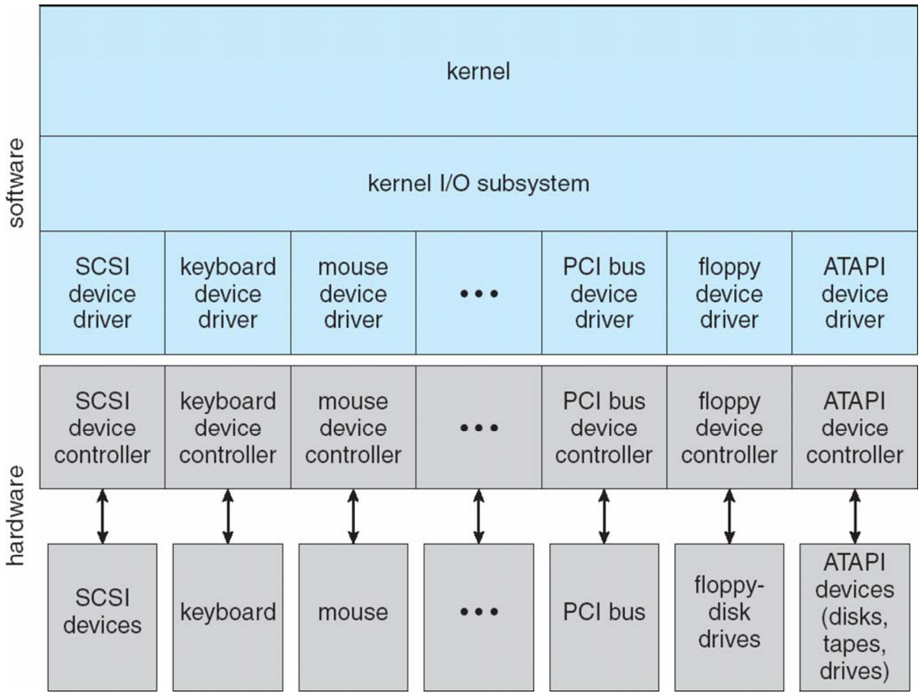
I/O System Organization



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



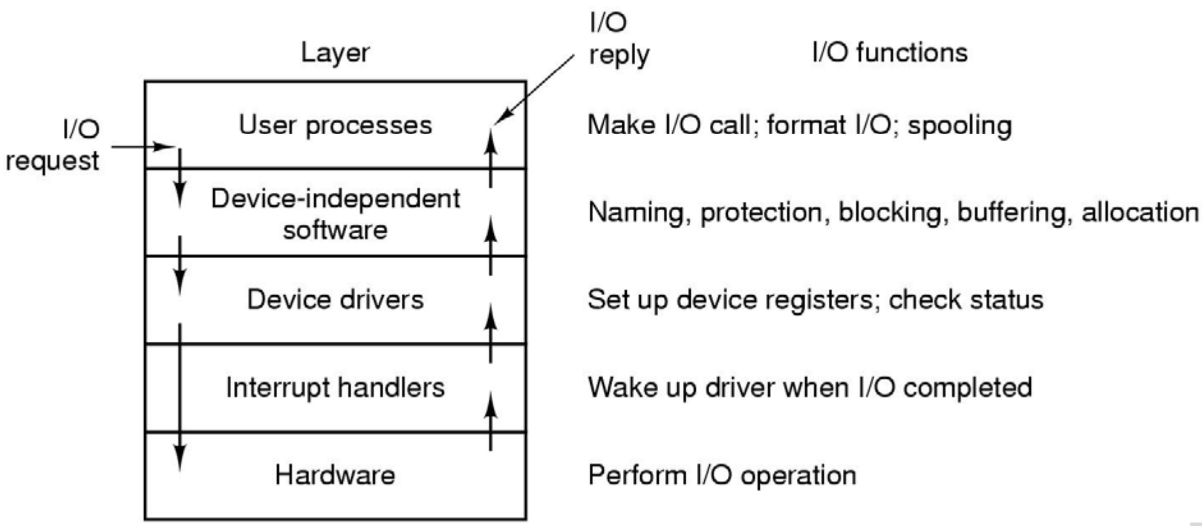
Kernel I/O Subsystem

- Scheduling
  - Some I/O request ordering via per-device queue
  - Some OSs try fairness
- 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”
- Error Handling
  - OS can recover from disk read, device unavailable, transient write failures
  - Most return an error number or code when I/O request fails
  - System error logs hold problem reports

Kernel I/O Subsystem

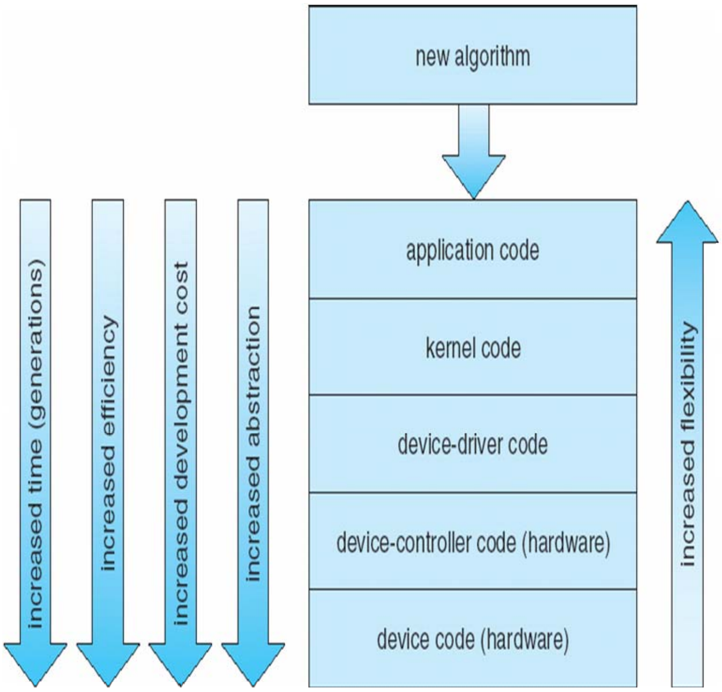
- Protection
  - User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions
  - I/O must be performed via system calls
    - Memory-mapped and I/O port memory locations must be protected too
- Spooling
  - Hold output for a device, if device can serve only one request at a time (i.e., Printing)
- Device reservation - provides exclusive access to a device
  - System calls for allocation and deallocation
  - Watch out for deadlock

I/O Software Summary

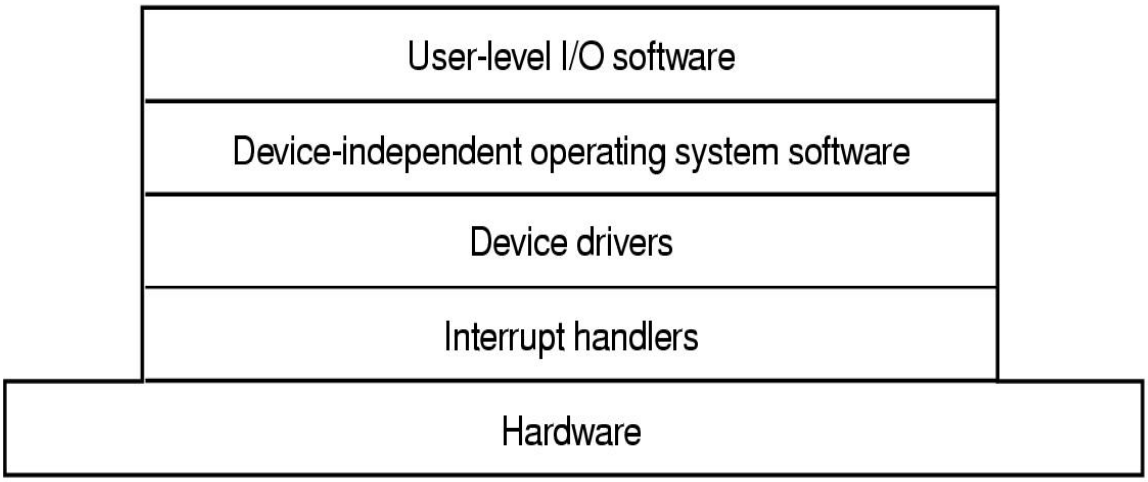


Layers of I/O system and main functions of each layer

Device-Functionality Progression



Layers of I/O Software System



Device-Independent I/O Software (1)

- There is some commonality between drivers of similar classes $\Rightarrow$
  - Divide I/O software into device-dependent and device independent I/O software, e.g.
    - Buffer or buffer-cache management, i.e. provide a device-independent block size
    - Allocating and releasing dedicate devices
    - Error reporting to upper levels, i.e. all errors the driver cannot resolve

Device-Independent I/O Software (2)

Driver $\Leftrightarrow$ Kernel Interface

- Uniform interface to devices and kernel
  - Uniform device interface for kernel code
    - Allows different devices to be used in the same way, e.g. no need to rewrite your file-system when you are switching from IDE to SCSI or even to RAM disks
    - Allows internal changes of drivers without fearing of breaking kernel code
  - Uniform kernel interface for device code
    - Drivers use a defined interface to kernel service, e.g. kmalloc, install IRQ handler, etc.
    - Allows kernels to evolve without breaking device drivers

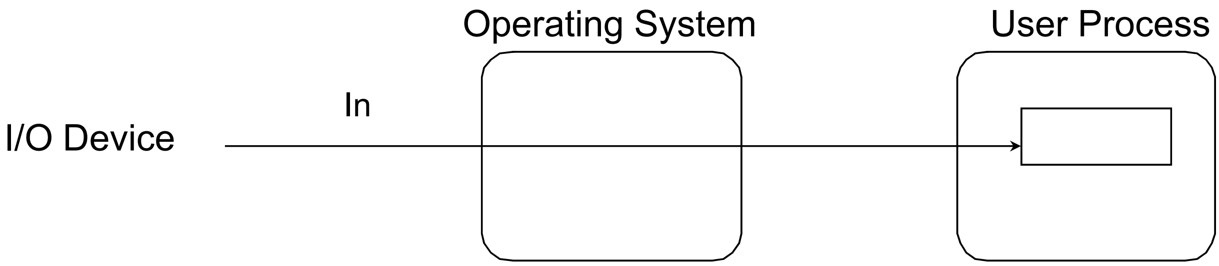
I/O Buffering

- Reasons for buffering
  - Otherwise threads must wait for I/O to complete before proceeding
  - Pages must remain in main memory during physical I/O
- Block-oriented
  - information is stored in fixed sized blocks
  - transfers are made a block at a time
  - used for disks and tapes
- Stream-oriented
  - transfer information as a stream of bytes
  - used for terminals, printers, communication ports, mouse, and most other devices that are not secondary storage

No Buffering

- Process reads/writes a device a byte/word at a time
  - Each individual system call adds significant overhead
  - Process must wait until every I/O is complete
  - Blocking/interrupt handling/unblocking adds to overhead
  - Many short CPU phases are inefficient, because
    - overhead induced by thread_switch
    - poor cache and TLB usage

User Level Buffering



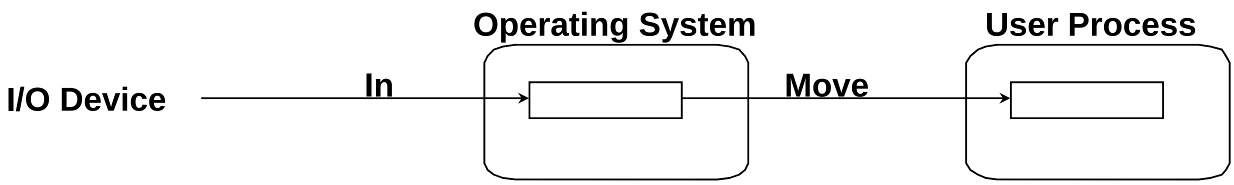
No buffering in OS

- Task specifies a memory buffer that incoming data is placed in until it fills
  - Filling can be done by interrupt service routine
  - Only one system_call and block/unblock per data buffer
    - More efficient than “NO BUFFERING”

User Level Buffering

- Issues
  - *What happens if buffer is currently paged out to disk?*
    - You may loose data while buffer is paged in
    - You could lock/pin this buffer (needed for DMA), however, you have to trust the application programmer, that she/he is not starting a denial of service attack
  - *Additional problems with writing?*
    - *When is the buffer available for re-use?*

Single Buffer



Single buffering

- User Process can process one block of data while next block is read in
- Swapping can occur since input is taking place in system memory, not user memory
- OS keeps track of assignment of system buffers to user processes

Single Buffer

- Stream-oriented
  - Buffer is an input line at time with carriage return signaling the end of the line
- Block-oriented
  - Input transfers made to system buffer
  - Buffer moved to user space when needed
  - Another block is read into system buffer

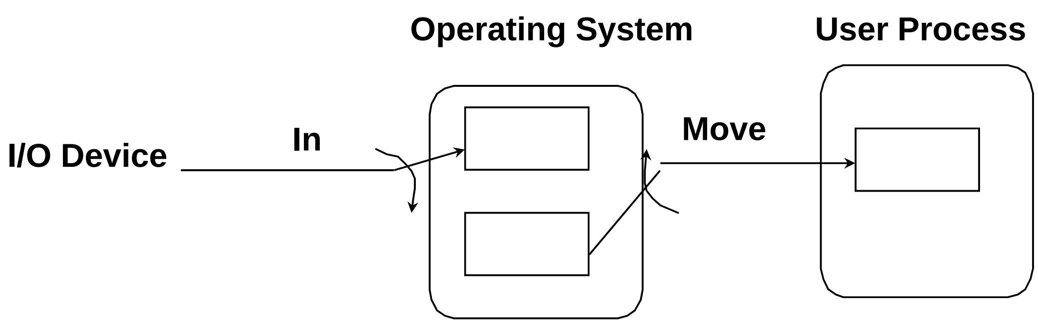
Single Buffer Speed Up

- Performance Model:
  - T = transfer time from device
  - C = copying time from system- to user-buffer
  - P = processing time of complete buffer content
  - Processing and transfer can be done in parallel
  - Potential speed up with single buffering:

$$\frac{T + P}{\max\{T, P\} + C}$$

- *What happens if system buffer is full, user buffer is swapped out, and more data is received?*
  - *Loose characters or drop network packets*

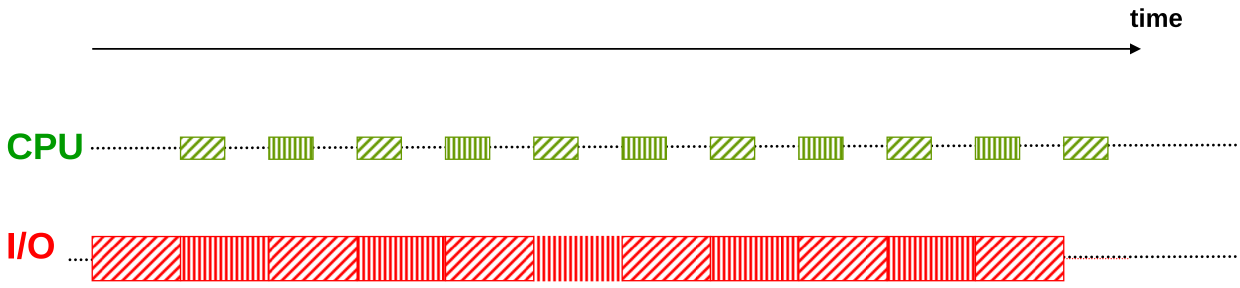
Double Buffer



- Use 2 system buffers instead of 1 (per user process)
- User process can write to or read from one buffer while the OS empties or fills the other buffer
- Speed up with double buffering:

$$\frac{T + P}{\max\{T, P+C\}}$$

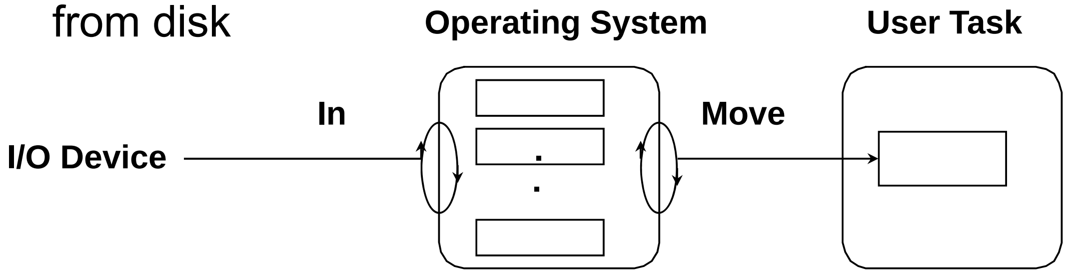
Timing Diagram for Double Buffering



Analysis: The slower I/O-device is busy the whole input-period, thus additional buffers are not needed (in this case).

Circular Buffering

- Double buffering may be insufficient for really bursty traffic situations:
  - Many writes between long periods of computations
  - Long periods of computations while receiving data
  - Might want to read ahead more than just a single block from disk



Single-, double-,and circular-buffering are all Bounded Buffer Producer-/Consumer Problems

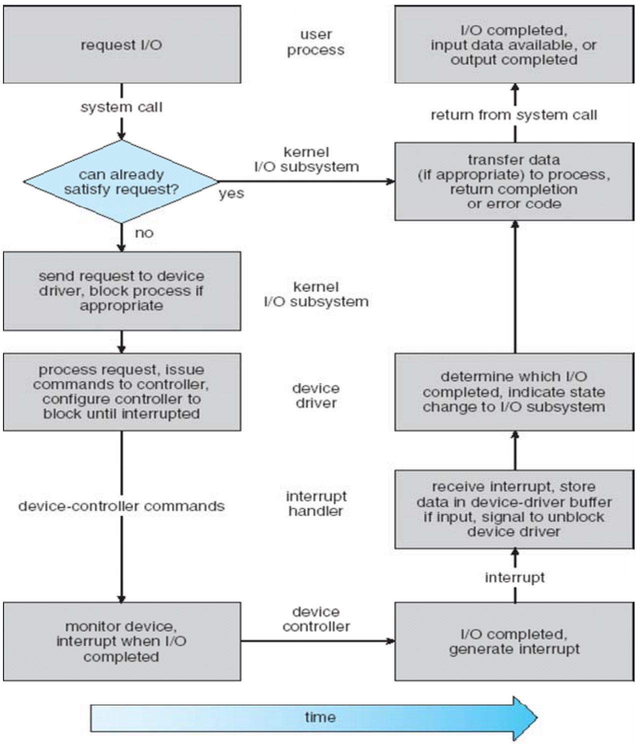
Device Driver

- Drivers classified into similar categories
  - Block devices and
  - Character (stream of data) devices
- OS defines standard (internal) interface to the different classes of devices
  - Device drivers job
    - Translate user request through device-independent standard interface, e.g. open, read, …, close) into appropriate sequence of device or controller commands (register manipulation)
    - Initialize HW at boot time
    - Shut down HW

Device Driver

- After issue the command to the device, device either
  - completes immediately and the driver simply returns to the caller or it
  - processes request and the driver usually blocks waiting for an I/O (complete) interrupt signal
- Drivers are reentrant as they can be called by another process while a process is already blocked in the driver
  - Reentrant: code that can be executed by more than one thread (or CPU) at the same time
    - Manages concurrency using synch primitives

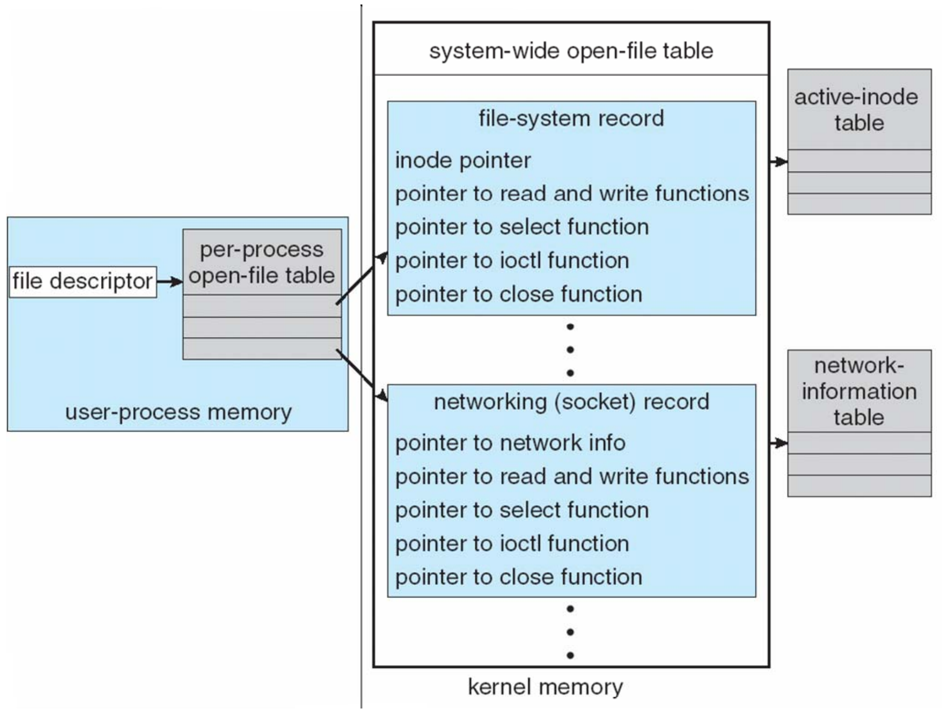
Life Cycle of An I/O Request



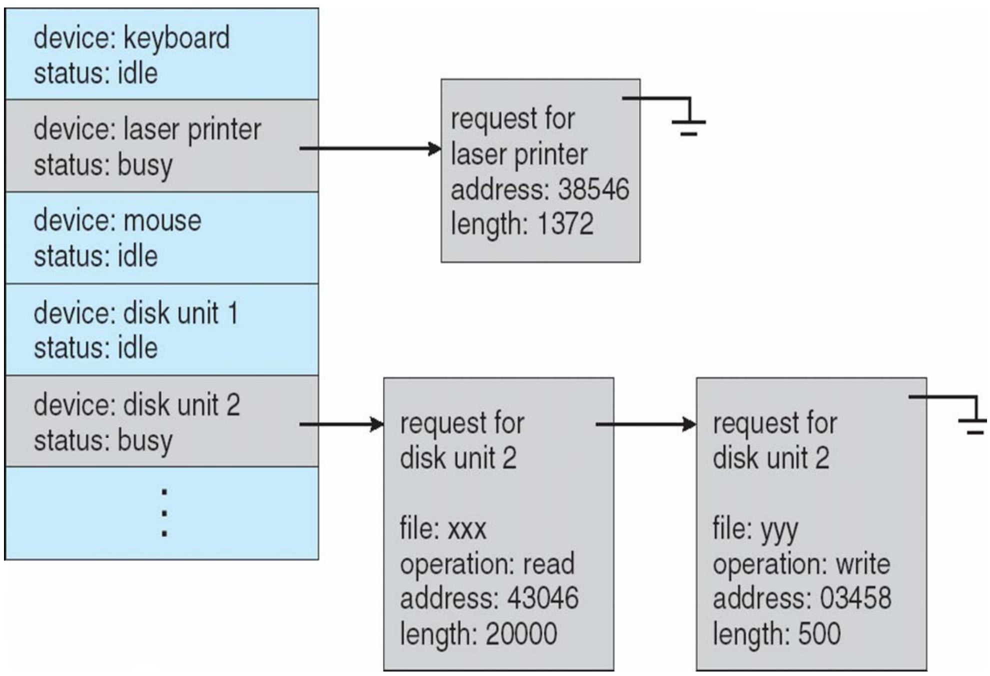
Kernel Data Structures

- 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

UNIX I/O Kernel Structure



Device-status Table



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