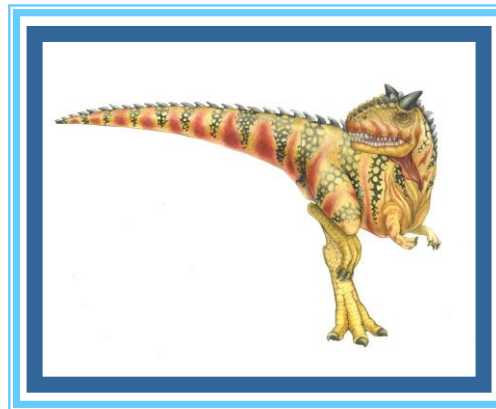
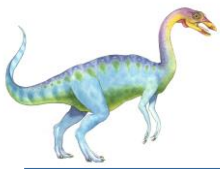


Chapter 6: I/O Systems





Chapter 12: I/O Systems

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





Objectives

- Explore the structure of an operating system's I/O subsystem
- Discuss the principles and complexities of I/O hardware
- Explain the performance aspects of I/O hardware and software

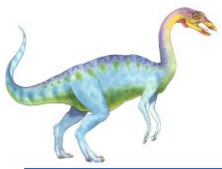




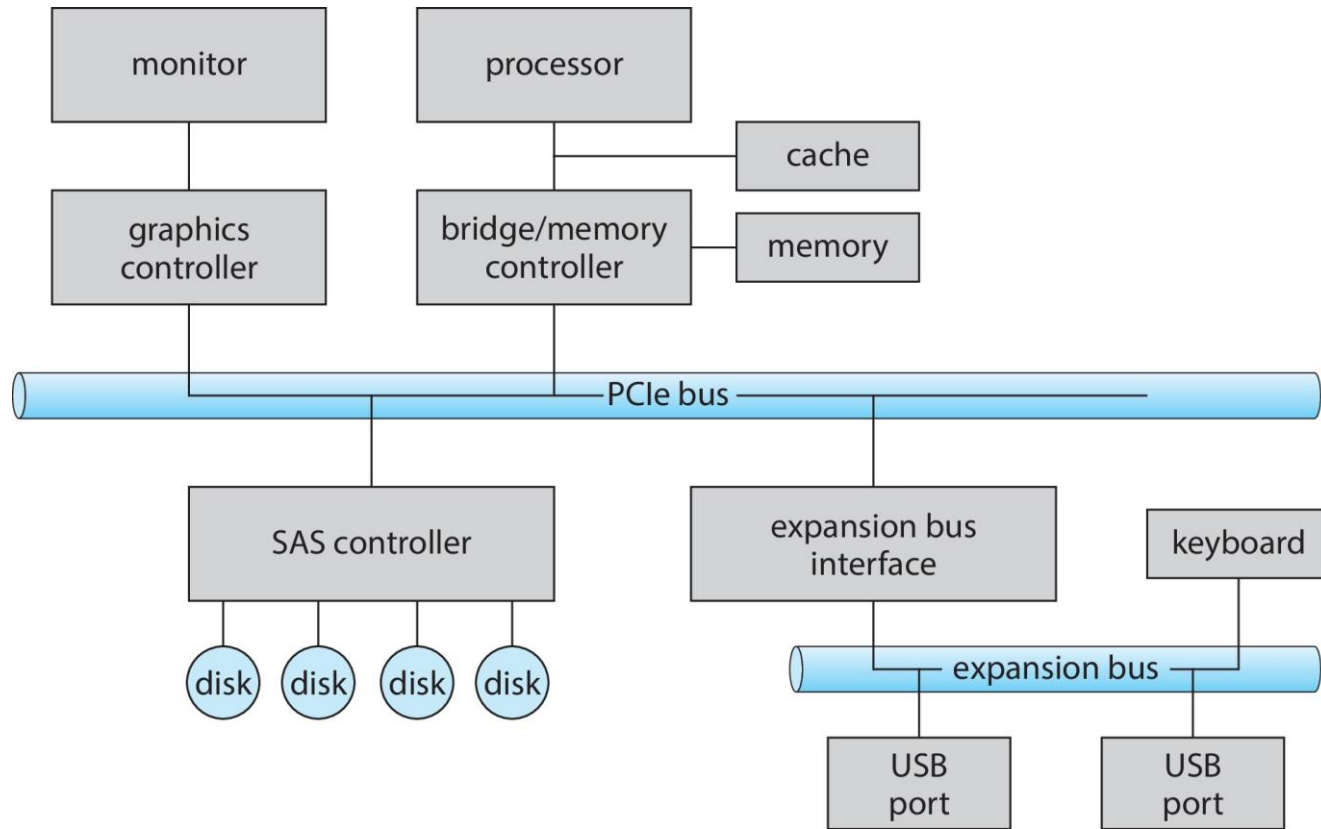
Overview

- 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





A Typical PC Bus Structure





I/O Hardware (Cont.)

- Devices have addresses, used by
 - Direct I/O instructions
 - **Memory-mapped I/O**
 - ▶ Device data and command registers mapped to processor address space
 - ▶ Especially for large address spaces (graphics)





Polling

- For each byte of I/O
 1. Read busy bit from status register until 0
 2. Host sets read or write bit and if write copies data into data-out register
 3. Host sets command-ready bit
 4. Controller sets busy bit, executes transfer
 5. Controller clears busy bit, error bit, command-ready bit when transfer done
- 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





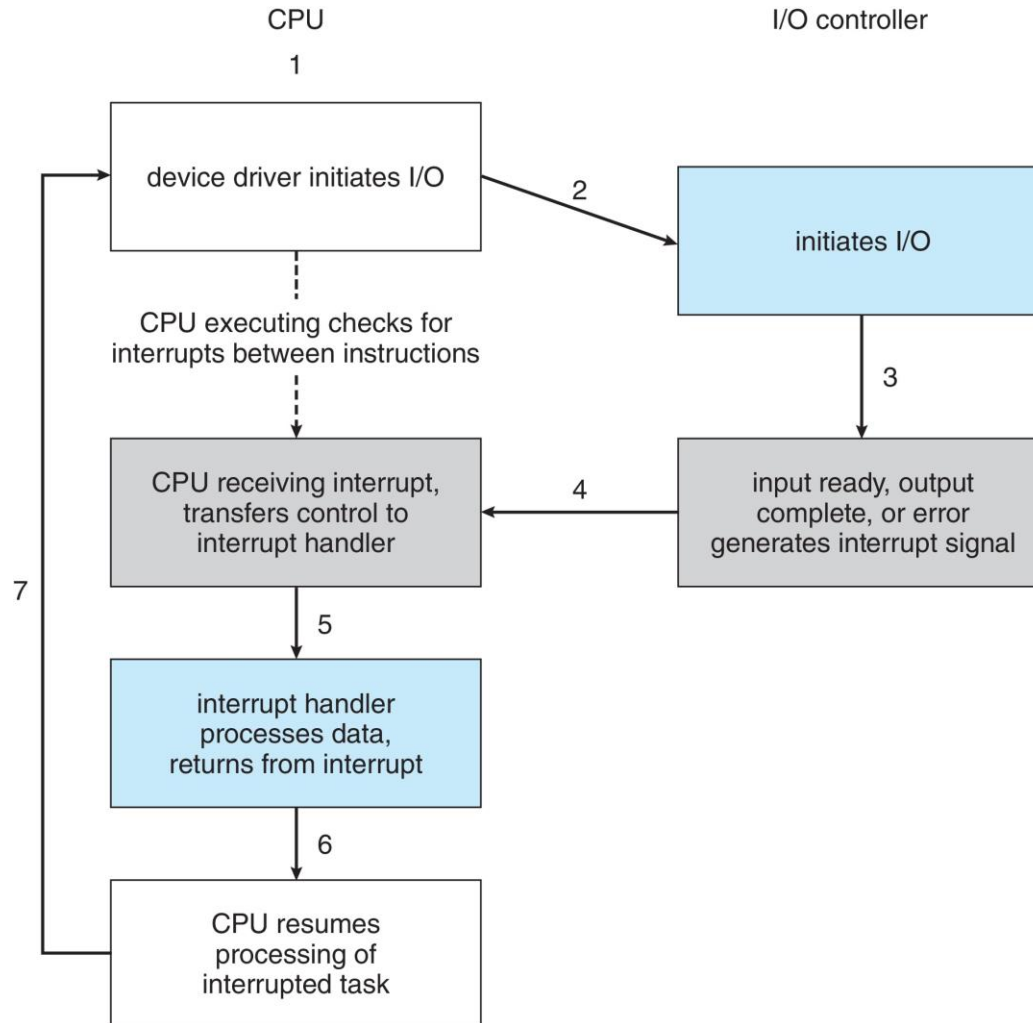
Interrupts

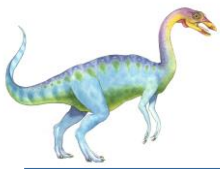
- Polling can happen in 3 instruction cycles
 - Read status, logical-and to extract status bit, branch if not zero
 - How to be more efficient if non-zero infrequently?
- CPU **Interrupt-request line** triggered by I/O device
 - Checked by processor after each instruction
- **Interrupt handler** receives interrupts
 - **Maskable**(يمكن تجاهلها) to ignore or delay some interrupts
- **Interrupt vector** to dispatch interrupt to correct handler
 - Context switch at start and end
 - Based on priority
 - Some **nonmaskable**(لا يمكن تجاهلها)
 - Interrupt chaining if more than one device at same interrupt number





Interrupt-Driven I/O Cycle





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
```






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
- OS writes DMA command block into memory
 - Source and destination addresses
 - Read or write mode
 - Count of bytes
 - Writes location of command block to DMA controller
 - Bus mastering of DMA controller – grabs bus from CPU
 - ▶ **Cycle stealing** from CPU but still much more efficient
 - When done, interrupts to signal completion
- Version that is aware of virtual addresses can be even more efficient - **DVMA**





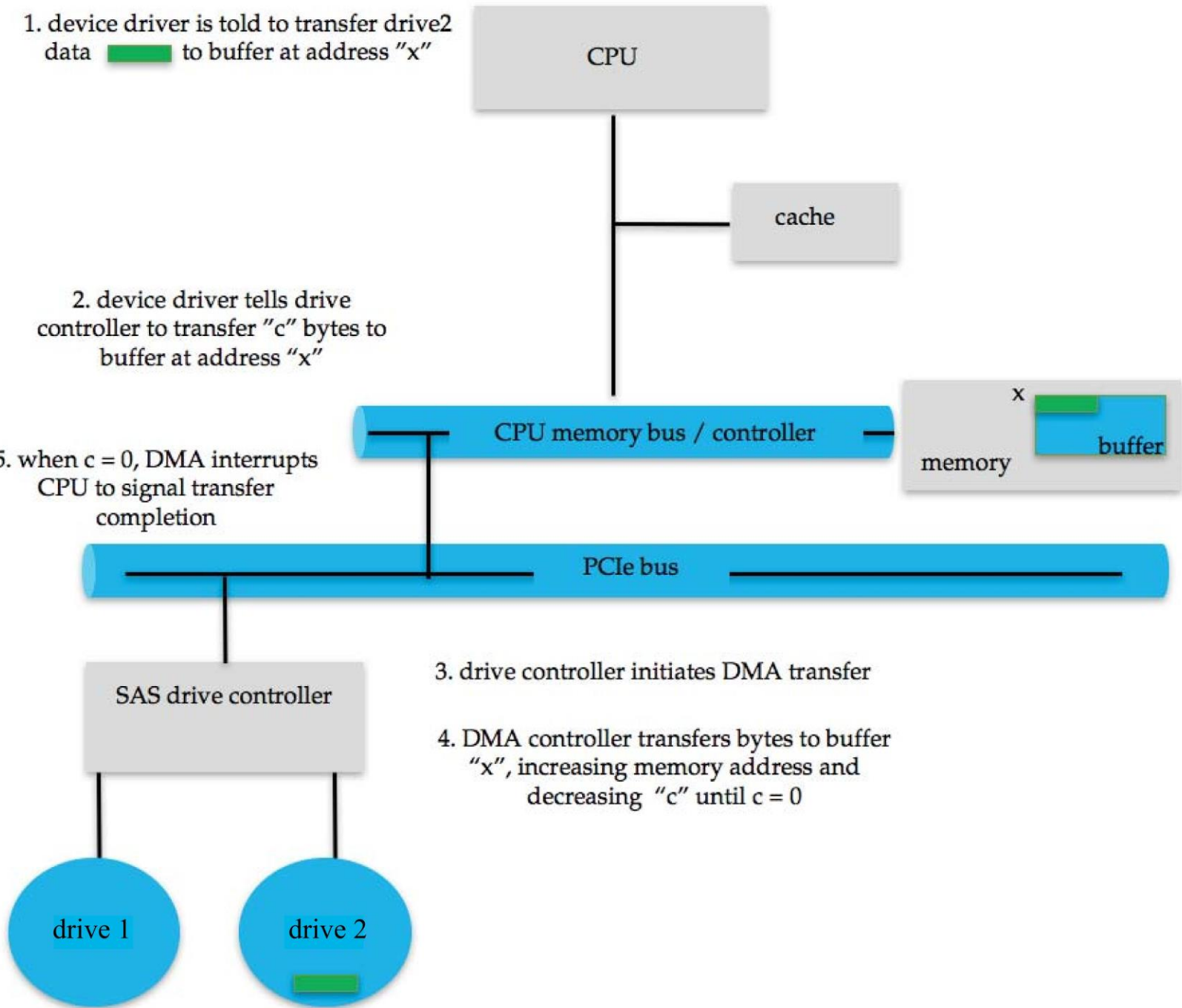
1. device driver is told to transfer drive2 data  to buffer at address "x"

2. device driver tells drive controller to transfer "c" bytes to buffer at address "x"

5. when $c = 0$, DMA interrupts CPU to signal transfer completion

3. drive controller initiates DMA transfer

4. DMA controller transfers bytes to buffer "x", increasing memory address and decreasing "c" until $c = 0$





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





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





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

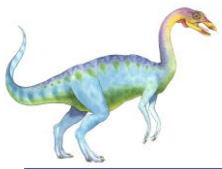




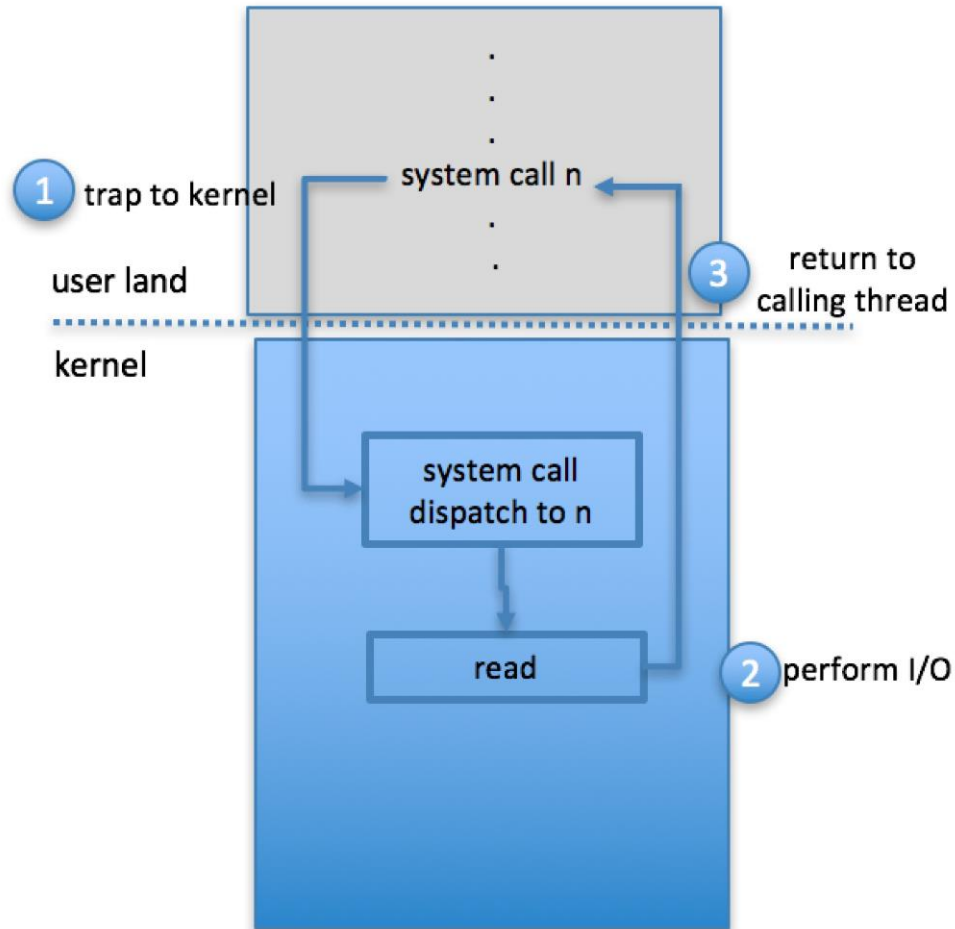
I/O Protection

- 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





Power Management

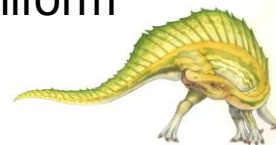
- Not strictly domain of I/O, but much is I/O related
- Computers and devices use electricity, generate heat, frequently require cooling
- OSes can help manage and improve use
 - Cloud computing environments move virtual machines between servers
 - ▶ Can end up evacuating whole systems and shutting them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





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





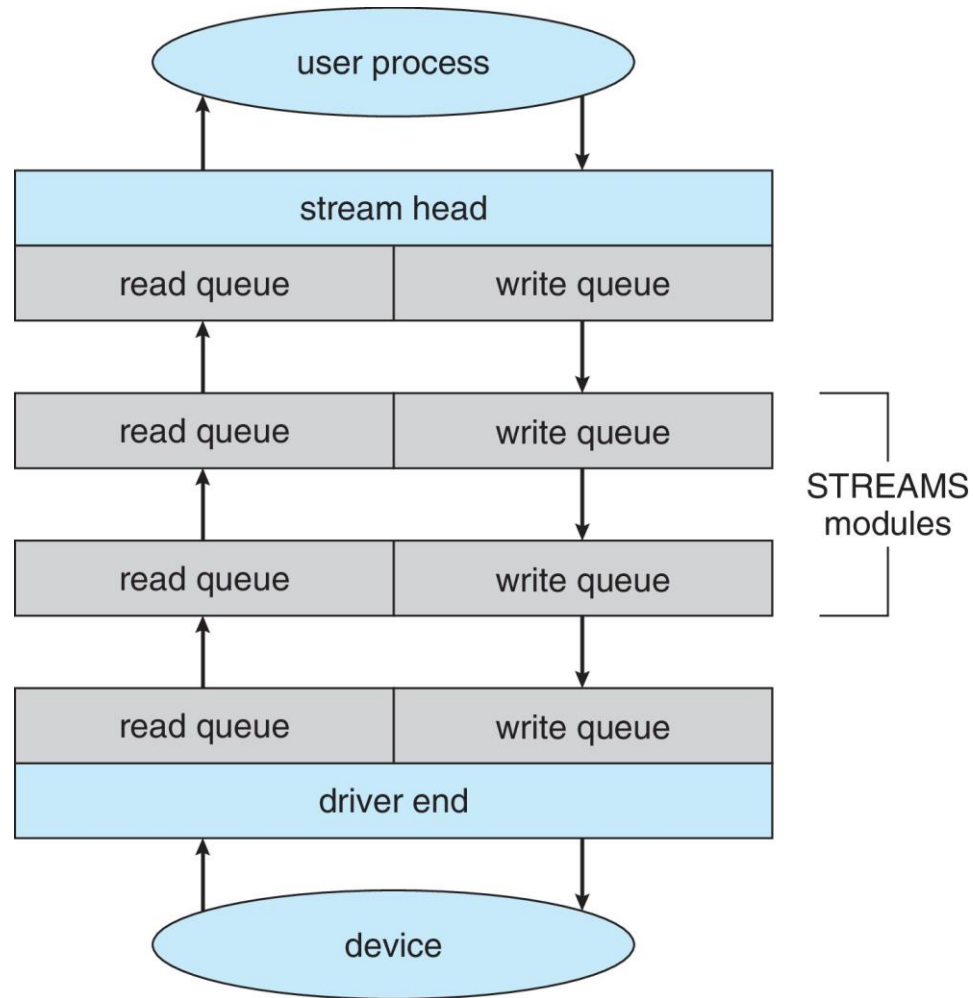
STREAMS

- **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
 - **Flow control** option to indicate available or busy
- Asynchronous internally, synchronous where user process communicates with stream head





The STREAMS Structure





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



End of Chapter 6

