COT 4600 Operating Systems Fall 2009

Dan C. Marinescu

Office: HEC 439 B

Office hours: Tu-Th 3:00-4:00 PM

Lecture 23

- Attention: project phase 4 – due Tuesday November 24
  - Final exam – Thursday December 10 4-6:50 PM
- Last time:
  - Performance Metrics (Chapter 5)
  - Random variables
  - Elements of queuing theory
- Today:
  - Methods to diminish the effect of bottlenecks: batching, dallying, speculation
  - I/O systems; the I/O bottleneck
  - Multi-level memories
  - Memory characterization
  - Multilevel memories management using virtual memory
  - Adding multi-level memory management to virtual memory
- Next Time:
  - Scheduling

Methods to diminish the effect of bottlenecks

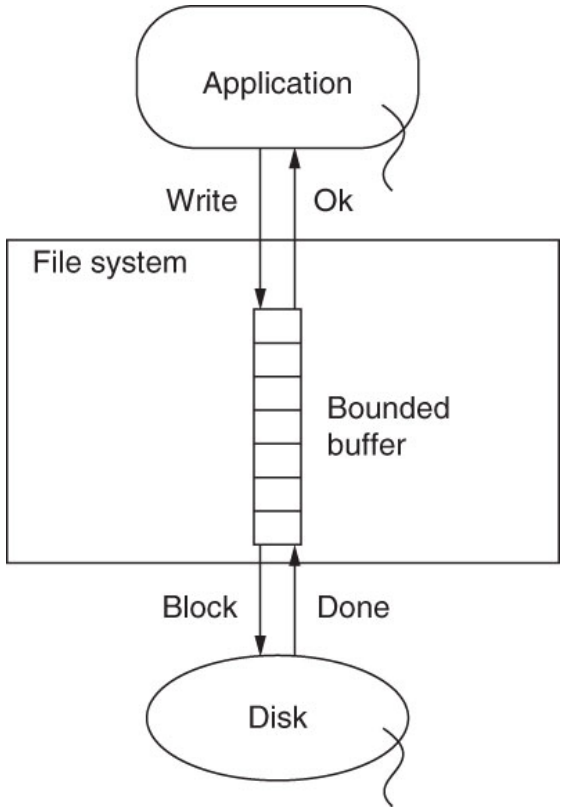
- Batching → perform several requests as a group to avoid the setup overhead of doing one at a time
  - f: fixed delay
  - v: variable delay
  - n: the number of items
  - n (f + v) versus f+nv
- Dallying → delay a request on the chance that it will not be needed or that one could perform batching.
- Speculation→ perform an operation in advance of receiving the request.
  - speculative execution of branch instructions
  - speculative page fetching rather than on demand
- All increase complexity due to concurrency.

The I/O bottleneck

- An illustration of the principle of incommensurate scaling → CPU and memory speed increase at a faster rate than those of mechanical I/O devices limited by the laws of Physics.
- Example: hard drives
  - The average seek time (AST): AST = 8 msec
  - average rotation latency (ARL): rotation speed: 7200 rotation/minute →120 rotations /second (8.33 msec/rotation) → ARL =4.17 msec
  - A typical 400 Gbyte disk
    - 16,383 cylinders → 24 Mbyte/cylinder
    - 8 two-sided platters → 16 tracks/cylinder →24/16 MBytes/track→ 1.5 Mbyte/track
  - The maximum rate transfer rate of the disk drive is:

    120 revolutions/sec x 1.5 Mbyte/track=180 Mbyte/sec
  - The bus transfer rates (BTR):
    - ATA3 bus → 3 Gbytes/sec
    - IDE bus 66 Mbyte/sec. This is the bottleneck!!
  - The average time to read a 4 Kbyte block:

    AST+ARL+4 /180 = 8 + 4.17 + 0.02 = 12.19 msec
  - The throughput: 328 Kbytes/sec.


Application
Write
Ok
File system
Bounded
buffer
Block
Done
Disk

I/O bottleneck

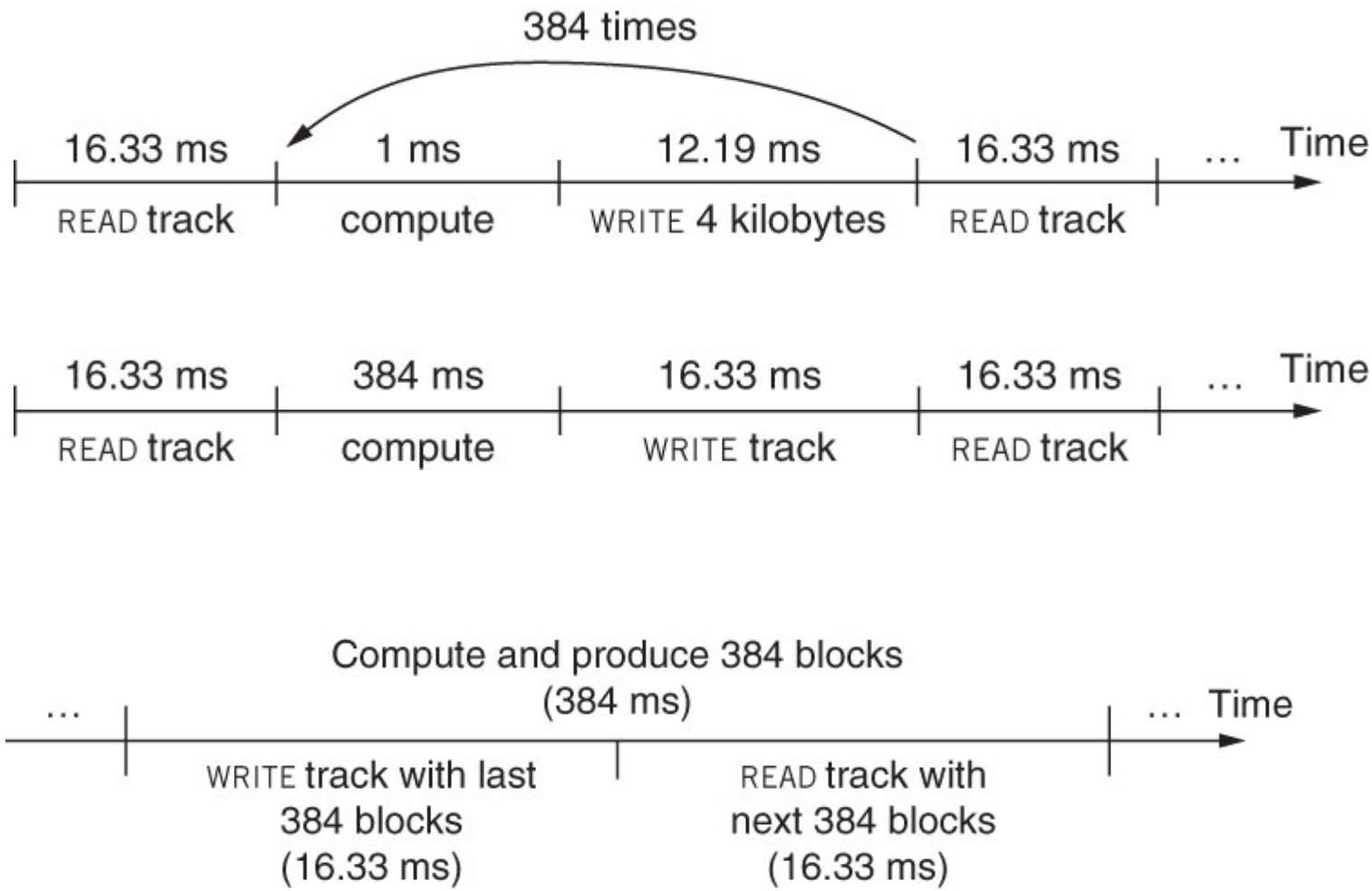
- If the application consists of a loop: (read a block of data, compute for 1 msec, write back) and if
  - the block are stored sequentially on the disk thus we can read a full track at once (speculative execution of the I/O)
  - we have a write-though buffer so that we can write a full track at one (batching)

  then the execution time can be considerably reduced.
- The time per iteration: read time + compute time + write time
- Initially: 12.19 + 1 + 12.19 = 25.38 msec
- With speculative reading of an entire track and overlap of reading and writing
  - Read an entire track of 1.5 Mbyte → reads the data for 384=1,500/4 iterations
  - The time for 384 iterations:

    Fixed delay: average seek time + 1 rotational delay: 8 + 8.33 msec= 16.33 msec

    Variable delay: 384(compute time + data transfer time)= 384(1+12.19)= 5065 msec

    Total time: 16.33 +5,065= 5,081 msec


384 times
16.33 ms
READ track
1 ms
compute
12.19 ms
WRITE 4 kilobytes
16.33 ms
READ track
…
Time
16.33 ms
READ track
384 ms
compute
16.33 ms
WRITE track
16.33 ms
READ track
…
Time
Compute and produce 384 blocks
(384 ms)
…
WRITE track with last
384 blocks
(16.33 ms)
READ track with
next 384 blocks
(16.33 ms)
…
Time

Disk writing strategies

- Keep in mind that buffering data before writing to the disk has implications; if the system fails then the data is lost.
- Strategies:
  - Write-through → write to the disk before the *write* system call returns to the user application
  - User-controlled write through a *force* call.
  - At the time the file is closed
  - After a predefined number of *write* calls or after a pre-defined time.

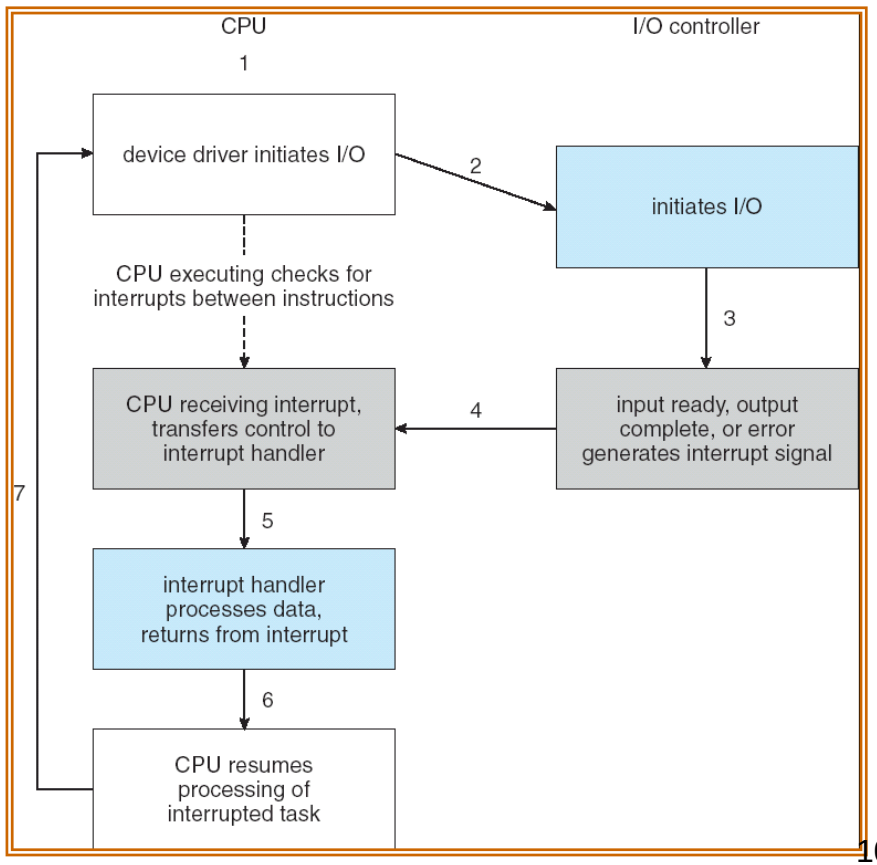
Communication among asynchronous sub-systems: polling versus interrupts

- Polling➔ periodically checking the status of an I/O device
- Interrupt ➔ deliver data or status information when status information immediately .
- Intel Pentium 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

Interrupts: used for I/O and for exceptions

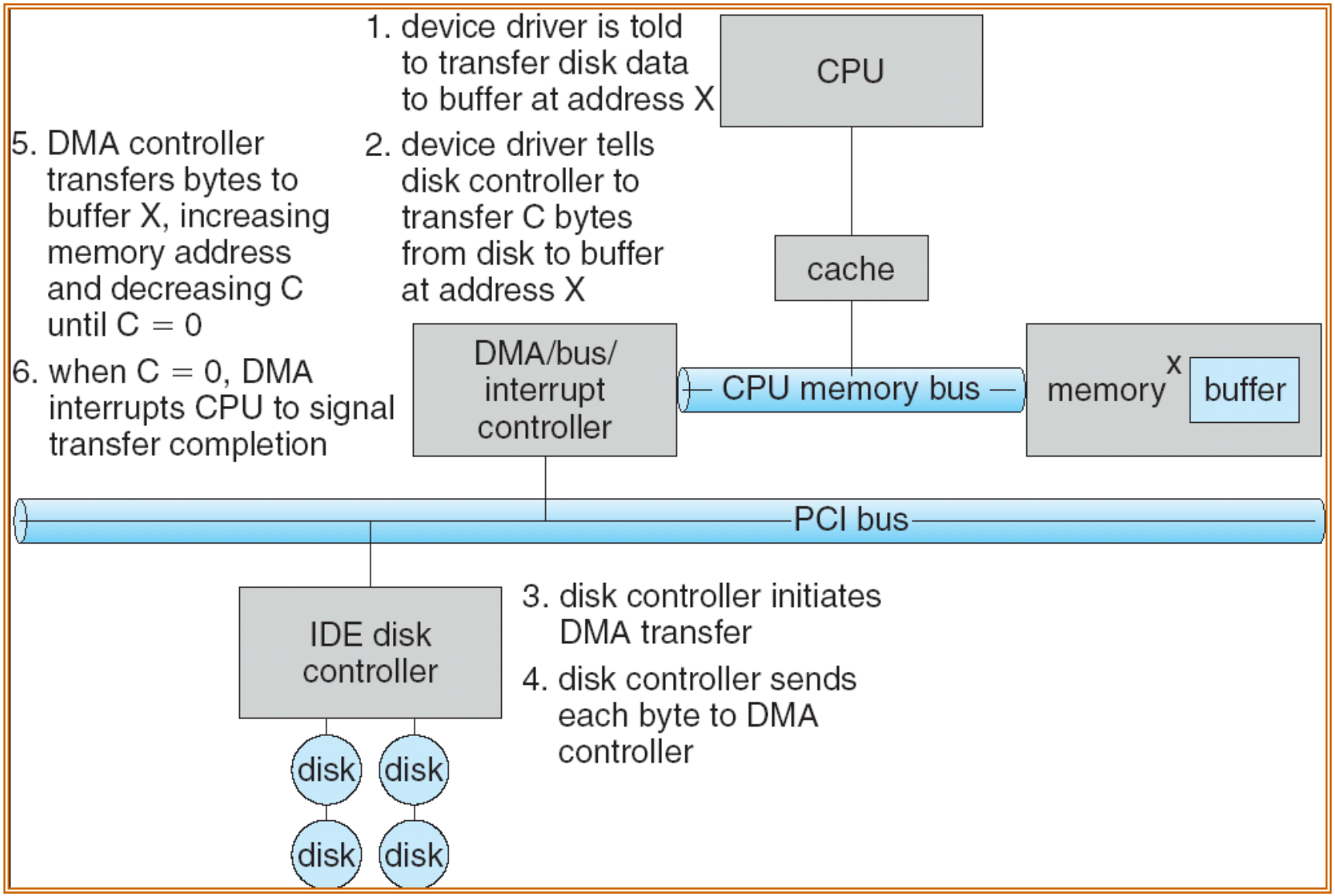
- CPU Interrupt-request line ➔ triggered by I/O device
- Interrupt handler receives interrupts
- To mask an interrupt ➔ ignore or delay some interrupts
- Interrupt vector to dispatch interrupt to correct handler
  - Based on priority
  - Some non-maskable



Direct Memory Access (DMA)

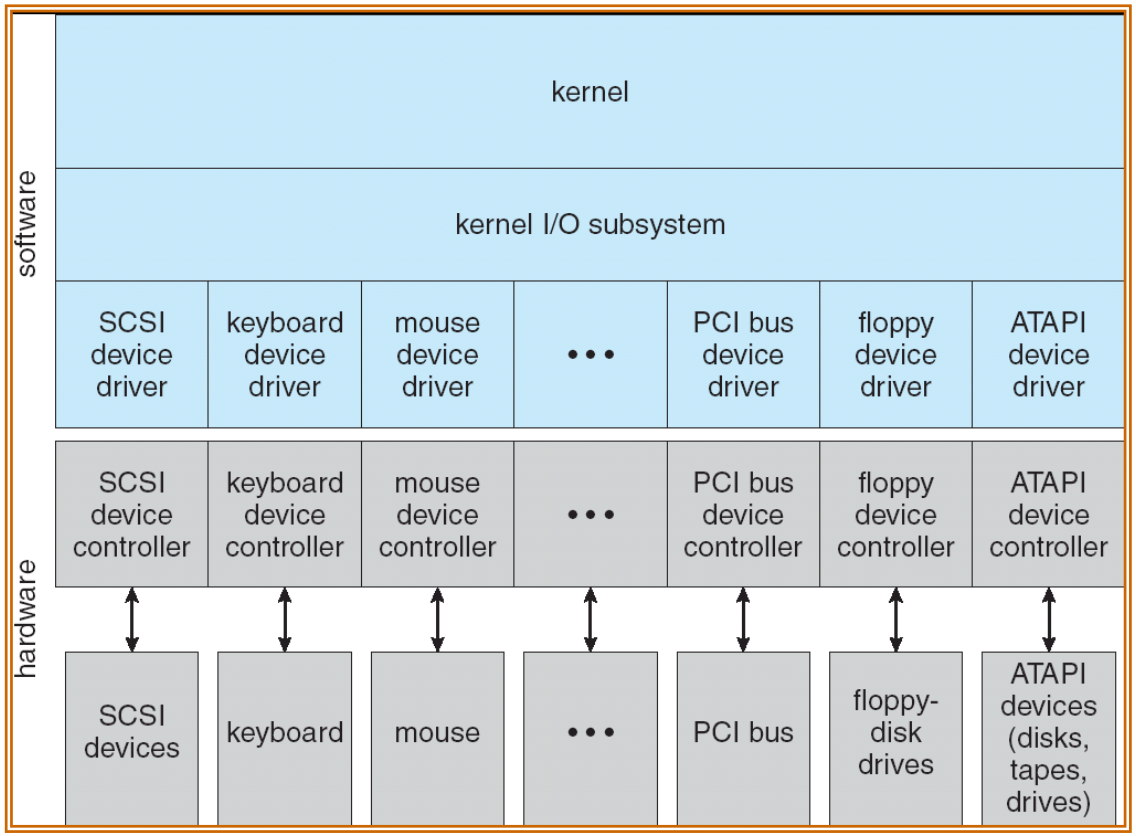
- DMA ➔ Bypasses CPU to transfer data directly between I/O device and memory; it allows subsystems within the computer to access system memory for reading and/or writing independently of CPU:
  - disk controller,
  - graphics cards,
  - network cards,
  - sound cards, GPUs (graphics processors),
  - also used for intra-chip data transfer in multi-core processors,.
- Avoids programmed I/O for large data movement
- Requires DMA controller

DMA Transfer



Device drivers and I/O system calls

- Multitude of I/O devices
  - Character-stream or block
  - Sequential or random-access
  - Sharable or dedicated
  - Speed of operation
  - Read-write, read only, or write only
- Device-driver layer hides differences among I/O controllers from kernel:
- I/O system calls encapsulate device behaviors in generic classes



Block and Character Devices

- Block devices (e.g., disk drives, tapes)
  - Commands e.g., *read, write, seek*
  - Raw I/O or file-system access
  - Memory-mapped file access possible

- Character devices (e.g., keyboards, mice, serial ports)
  - Commands e.g., `get, put`
  - Libraries allow line editing

Network Devices and Timers

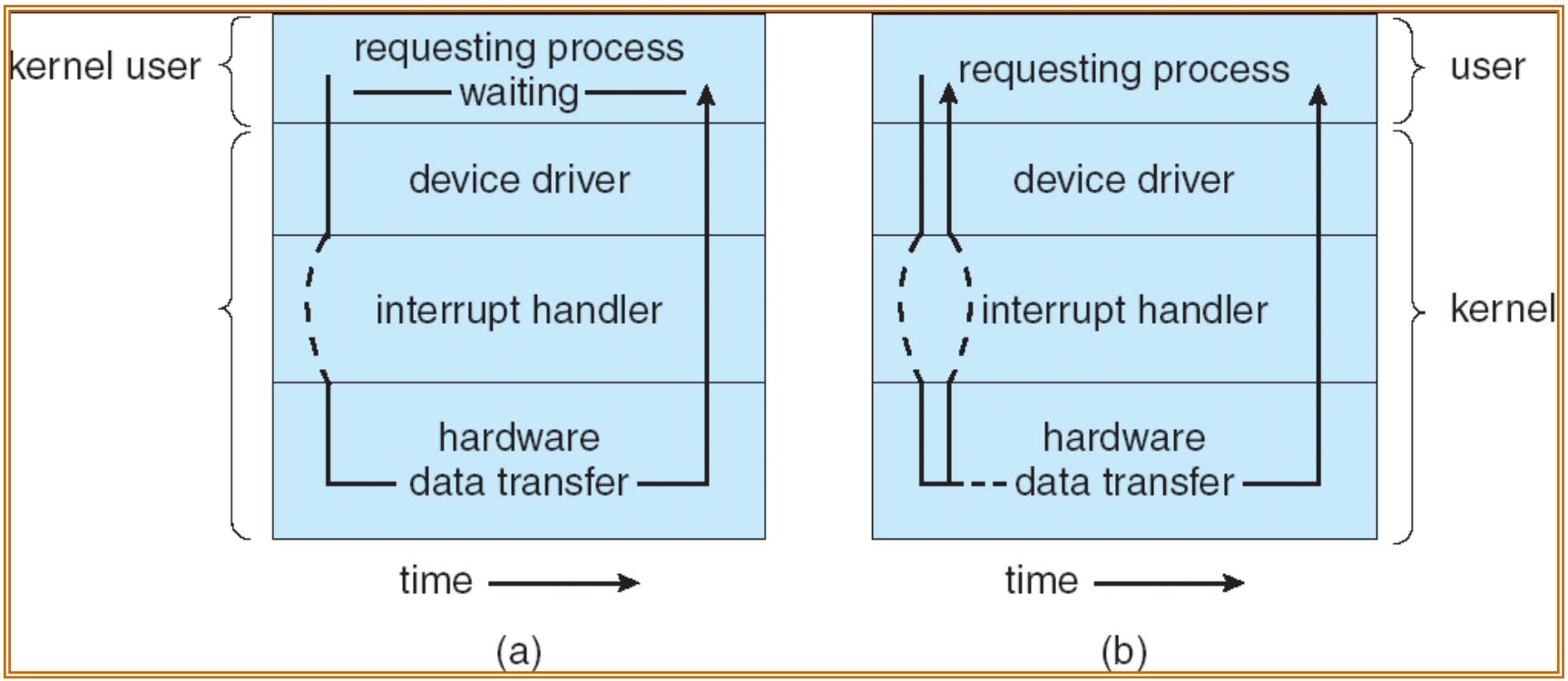
- Network devices
  - Own interface different from bloc or character devices
  - Unix and Windows NT/9*x*/2000 include socket interface
    - Separates network protocol from network operation
    - Includes `select` functionality
  - Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)

- Timers
  - Provide current time, elapsed time, timer
  - Programmable interval timer ➔for timings, periodic interrupts
  - `ioctl` (on UNIX) covers odd aspects of I/O such as clocks and timers

Blocking and non-blocking I/O

- Blocking ➔ process suspended until I/O completed
  - Easy to use and understand
  - Insufficient for some needs
- Non-blocking ➔ I/O call returns control to the process immediately
  - User interface, data copy (buffered I/O)
  - Implemented via multi-threading
  - Returns quickly with count of bytes read or written
- Asynchronous ➔ process runs while I/O executes
  - I/O subsystem signals process when I/O completed

Synchronous/Asynchronous I/O



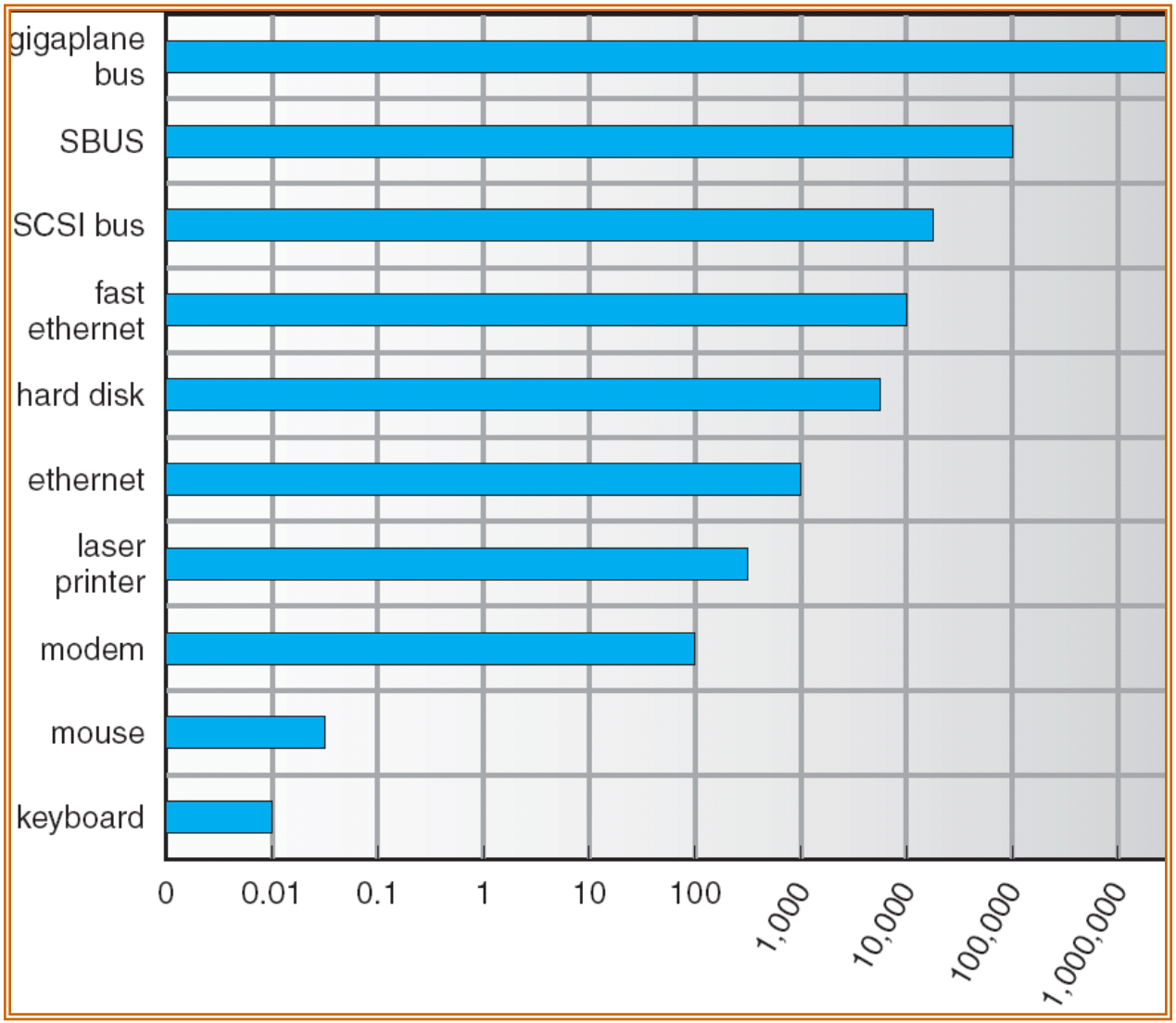
Synchronous

Asynchronous

Kernel I/O Subsystem

- Scheduling
  - Some I/O request ordering using per-device queue
  - Some OSs try fairness
- Buffering – store data in memory while transferring to I/O device.
  - To cope with device speed mismatch or transfer size mismatch
  - To maintain “copy semantics”

Sun Enterprise 6000 Device-Transfer Rates

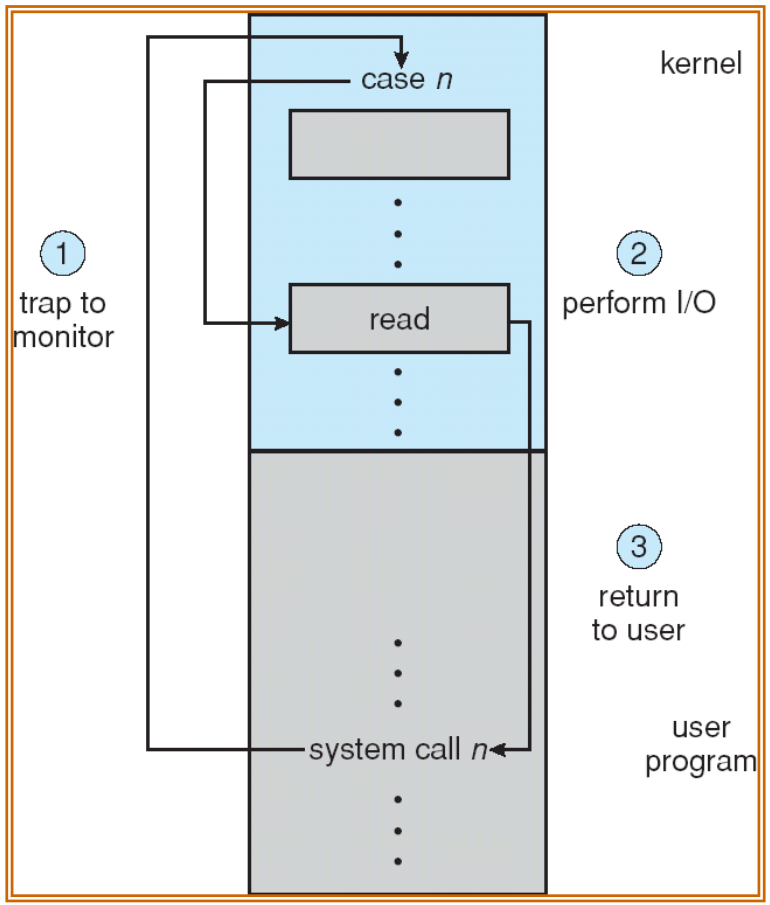


Kernel I/O Subsystem and Error Handling

- Caching ➔ fast memory holding copy of data
  - Always just a copy
  - Key to performance
- Spooling ➔ holds output for a device that can serve only one request at a time (e.g., printer).
- Device reservation ➔ provides exclusive access to a device
  - System calls for allocation and de-allocation
  - Possibility of deadlock
- Error handling:
  - OS can recover from disk read, device unavailable, transient write failures
  - When I/O request fails error code.
  - System error logs hold problem reports

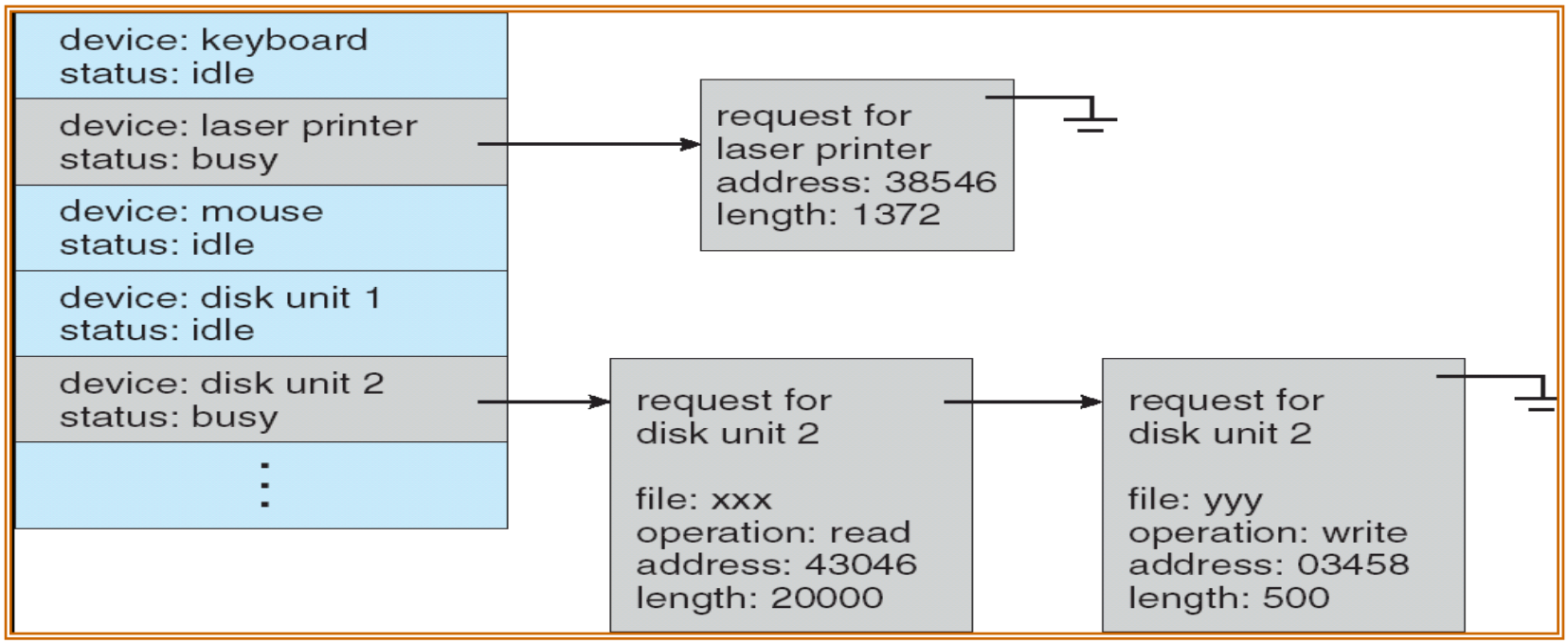
I/O Protection

- I/O instructions are priviledged
- Users make system calls



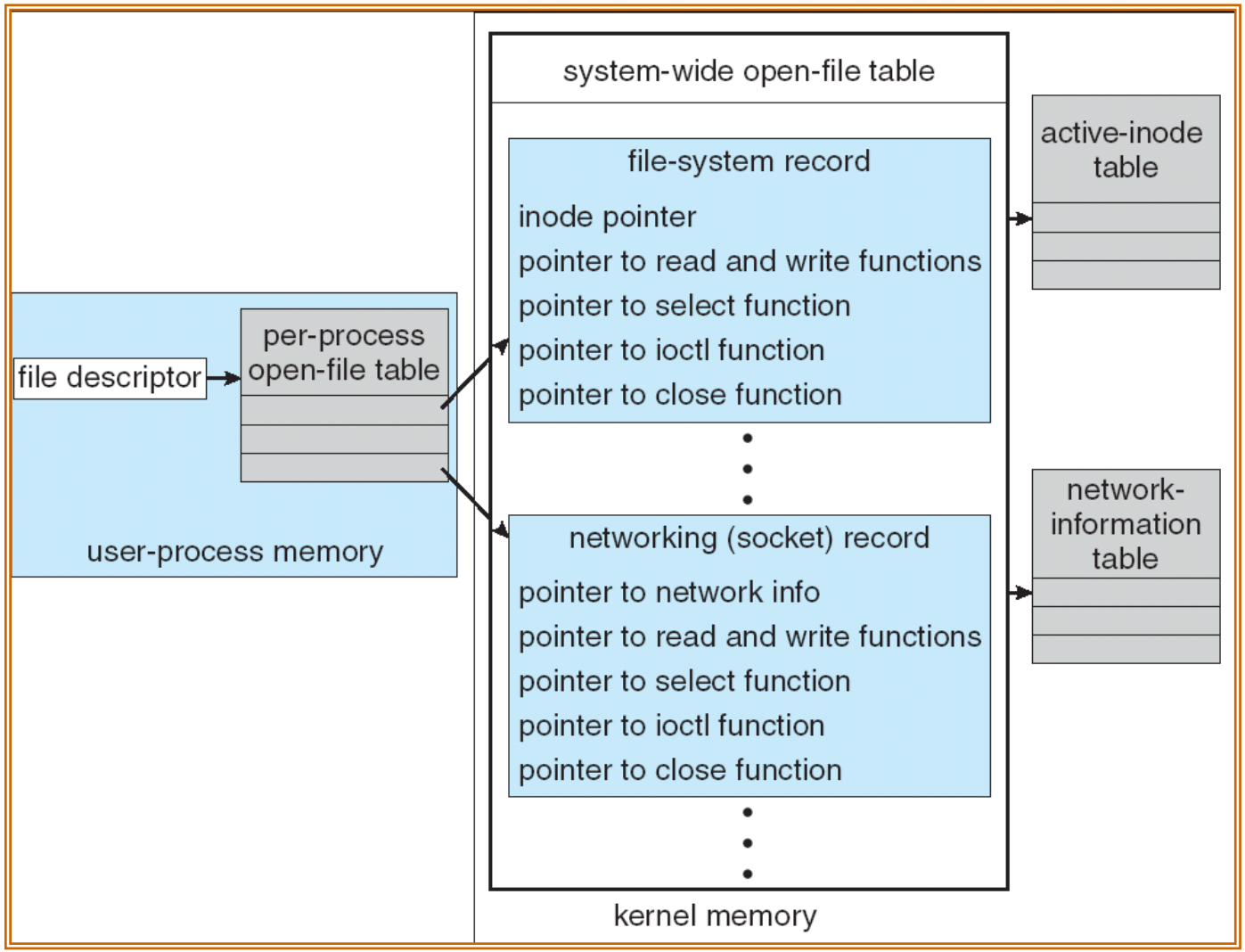
Kernel Data Structures for I/O handling

- Kernel keeps state info for I/O components, including open file tables, network connections, device control blocs



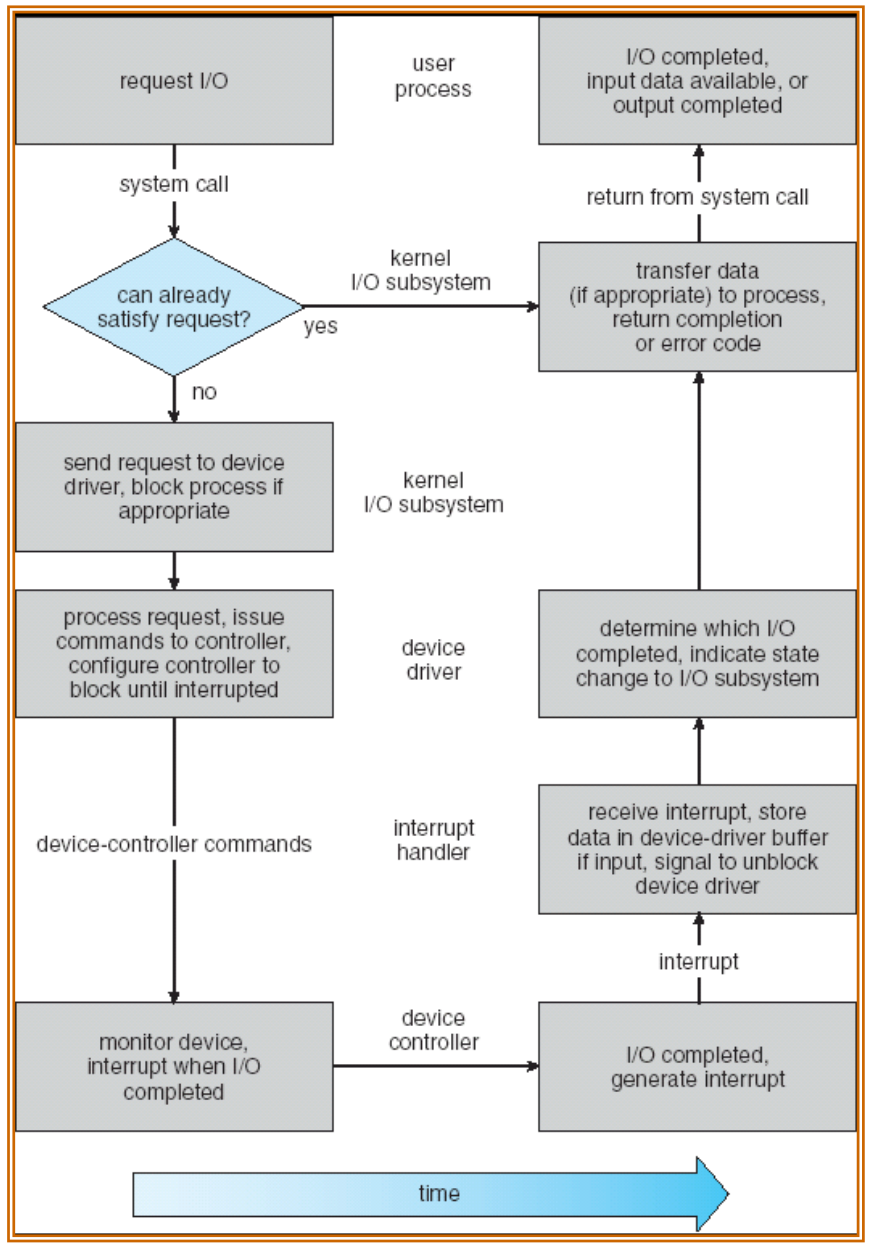
- Complex data structures to track buffers, memory allocation, “dirty” blocks
- Some use object-oriented methods and message passing to implement I/O

UNIX I/O Kernel Structure



Hardware Operations

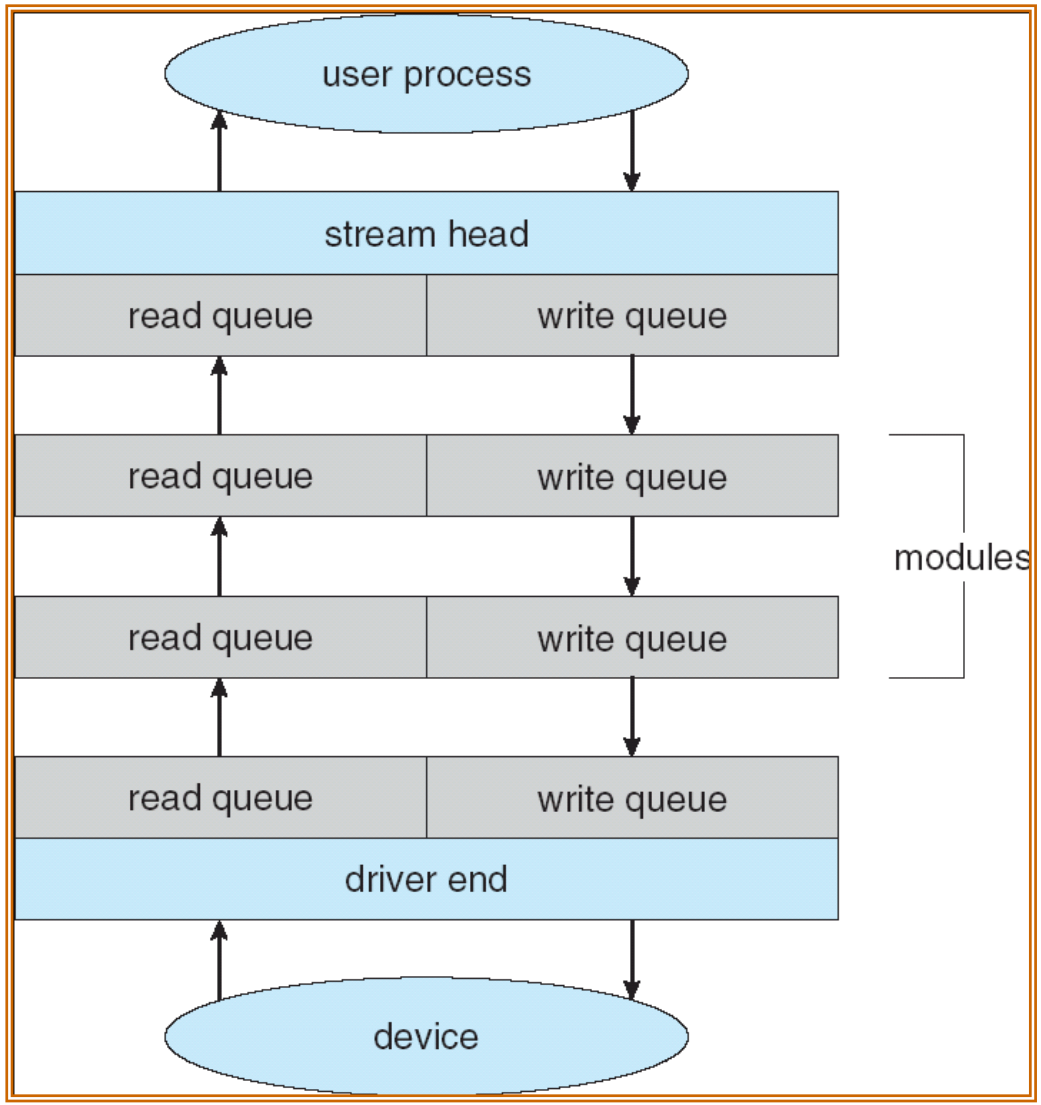
- Operation for reading a file:
  - Determine device holding file
  - Translate name to device representation
  - Physically read data from disk into buffer
  - Make data available to the process
  - Return control to process



STREAMS in Unix

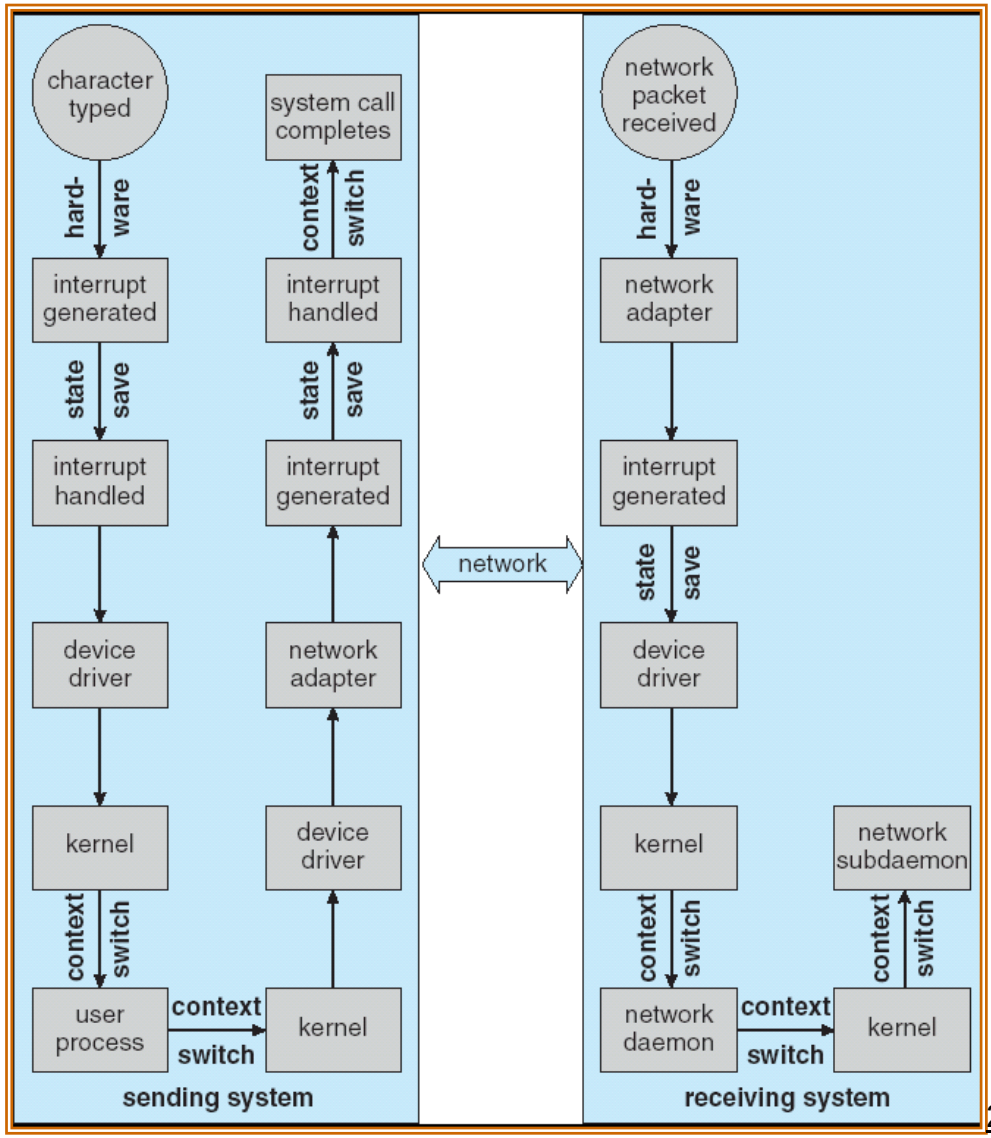
- STREAM ➔ a full-duplex communication channel between a user-level process and a device in Unix System V and beyond
- A STREAM consists of:
  - STREAM head interfaces with the user process
  - driver end interfaces with the device
  - zero or more STREAM modules between them.
- Each module contains a read queue and a write queue
- Message passing is used to communicate between queues

STREAMS


user process
stream head
read queue
write queue
read queue
write queue
read queue
write queue
read queue
write queue
modules
driver end
device

I/O ➔major factor in system performance:

- Execute
  - device driver,
  - *kernel I/O code*
  - Context switches
- Data copying
- Network traffic stressful

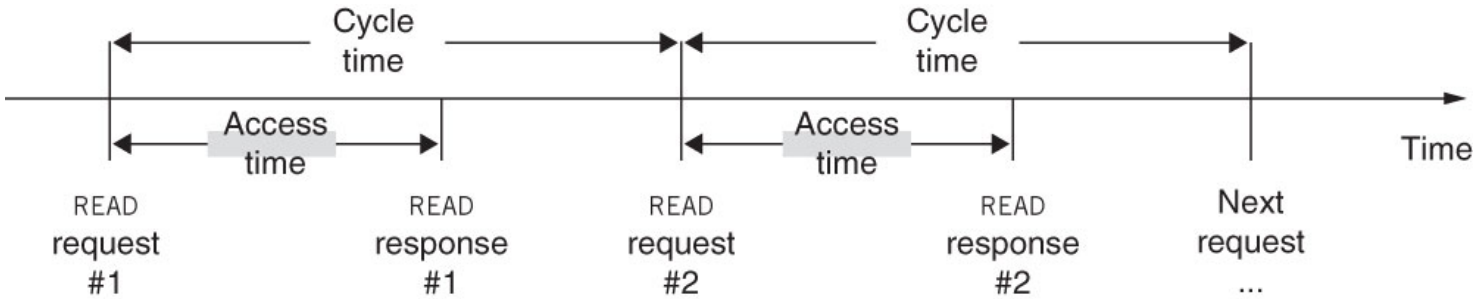


Improving Performance

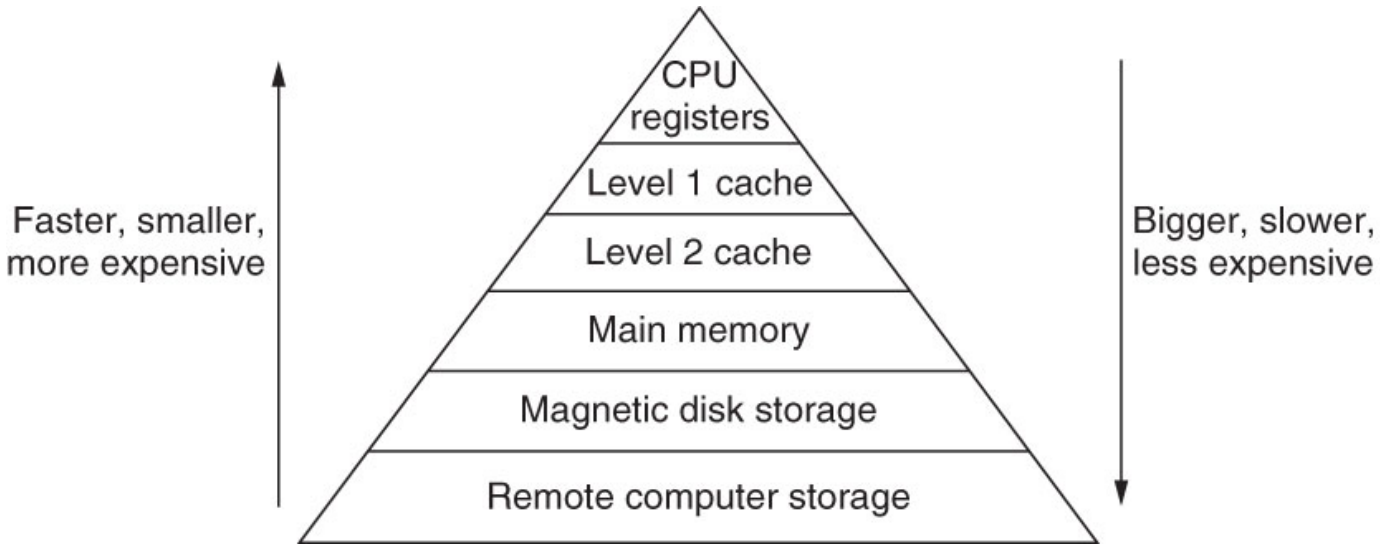
- Reduce number of context switches
- Reduce data copying
- Reduce interrupts by using large transfers, smart controllers, polling
- Use DMA
- Balance CPU, memory, bus, and I/O performance for highest throughput

Memory characterization

- Capacity
- Latency of random access memory:
  - Access time → time the data is available
  - Cycle time (> access time) → time when the next READ operation can be carried out (READ can be destructive)



  - Cost:
    - for random access memory cents/Mbyte
    - For disk storage: dollars/Gbyte
- Cell size → the number of bits transferred in a single READ or WRITE operation
- Throughput → Gbits/sec


Faster, smaller,
more expensive
CPU
registers
Level 1 cache
Level 2 cache
Main memory
Magnetic disk storage
Remote computer storage
Bigger, slower,
less expensive

Multi-level memories

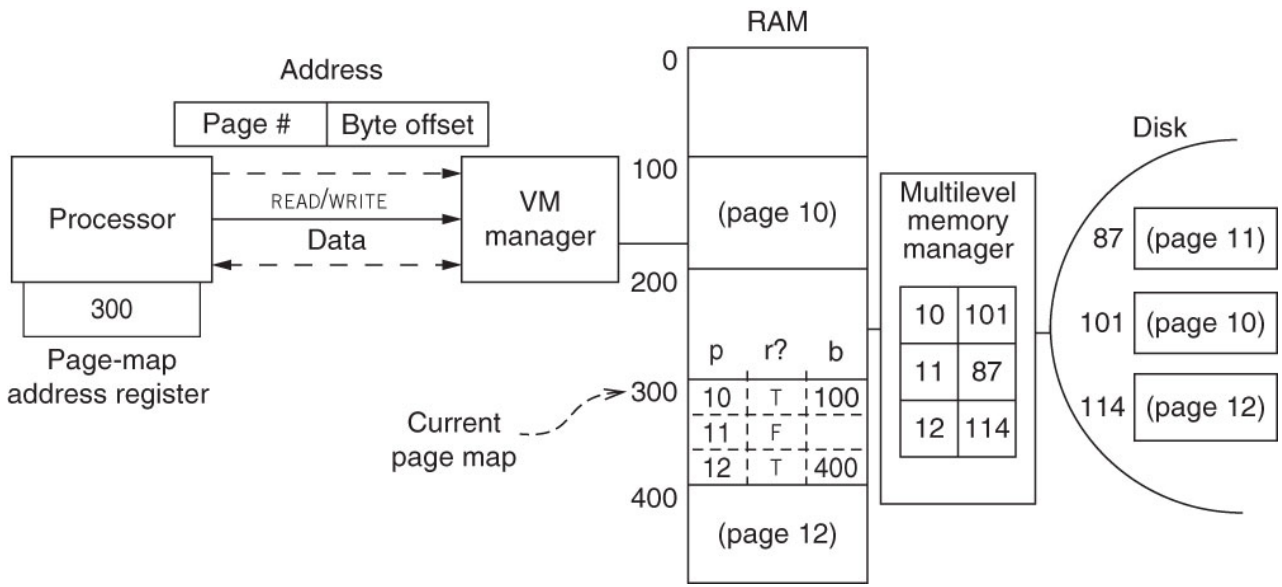
- In the following hierarchy the amount of storage and the access time increase at the same time
  - CPU registers
  - L1 cache
  - L2 cache
  - Main memory
  - Magnetic disk
  - Mass storage systems
  - Remote storage
- Memory management schemes ➔ where the data is placed through this hierarchy
  - Manual → left to the user
  - Automatic → based on memory virtualization
    - More effective
    - Easier to use

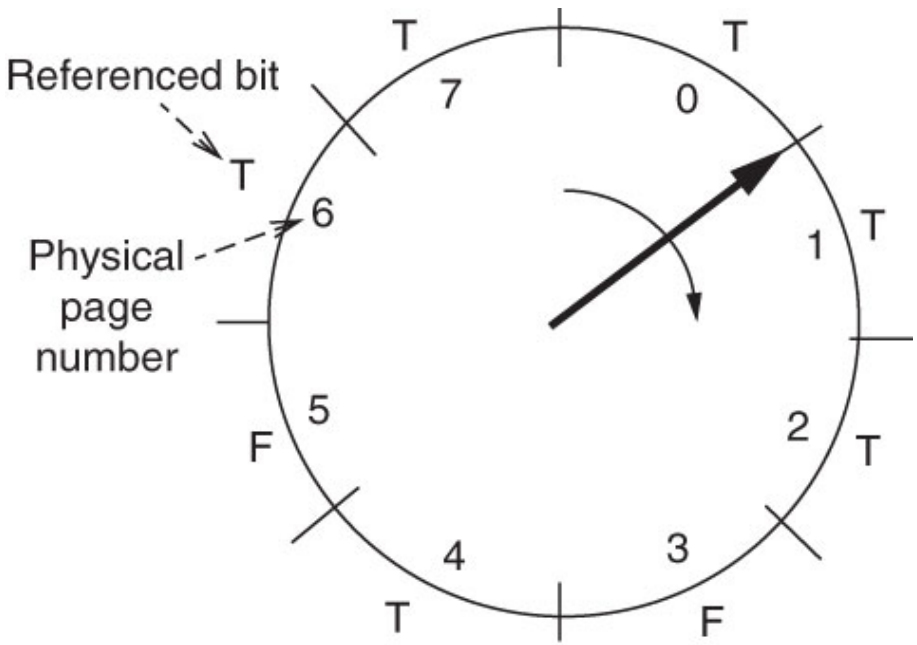
Other forms of memory virtualization

- Memory-mapped files → in UNIX *mmap*
- Copy on write → when several threads use the same data map the page holding the data and store the data only once in memory. This works as long all the threads only READ the data. If one of the threads carries out a WRITE then the virtual memory handling should generate an exception and data pages to be remapped so that each thread gets its only copy of the page.
- On-demand zero filled pages→ Instead of allocating zero-filled pages on RAM or on the disk the VM manager maps these pages without READ or WRITE permissions. When a thread attempts to actually READ or WRITE to such pages then an exception is generated and the VM manager allocates the page dynamically.
- Virtual-shared memory → Several threads on multiple systems share the same address space. When a thread references a page that is not in its local memory the local VM manager fetches the page over the network and the remote VM manager un-maps the page.

Multi-level memory management and virtual memory

- Two level memory system: RAM + disk
  - READ and WRITE from RAM → controlled by the VM manager
  - GET and PUT from disk → controlled by a multi-level memory manager
- Old design philosophy: integrate the two to reduce the instruction count
- New approach – modular organization
  - Implement the VM manager (VMM) in hardware
  - Implement the multi-level memory manager (MLMM) in the kernel in software. It transfers pages back and forth between RAM and the disk
- How it works:
  - VM attempts to translate the virtual memory address to a physical memory address
  - If the page is not in main memory VM generates a page-fault exception.
  - The exception handler uses a SEND to send to an MLMM port the page number
  - The SEND invokes ADVANCE which wakes up a thread of MLMM
  - The MMLM invokes AWAIT on behalf of the thread interrupted due to the page fault.
  - The AWAIT releases the processor to the SCHEDULER thread.
- The new approach leads to implicit I/O


RAM
Address
Page #
Byte offset
Processor
READ/WRITE
Data
VM manager
300
Page-map address register
0
100
200
300
400
(page 10)
p
r?
b
10
T
100
11
F
12
T
400
Current page map
(page 12)
Multilevel memory manager
10
101
11
87
12
114
Disk
87
(page 11)
101
(page 10)
114
(page 12)


Referenced bit
T
Physical page number
6
7
T
0
T
1
T
2
T
3
F
4
T
5
F