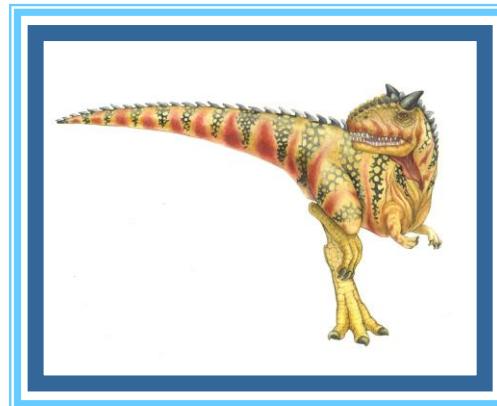
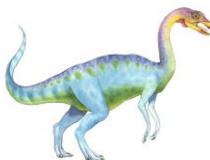


# Chapter 10: Virtual Memory



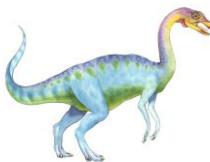


# Chapter 10: Virtual Memory

---

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples





# Objectives

---

- Define virtual memory and describe its benefits
- Illustrate how pages are loaded into memory using demand paging
- Apply the FIFO, optimal, and LRU page-replacement algorithms
- Describe the working set of a process, and explain how it is related to program locality
- Describe how Linux, Windows 10, and Solaris manage virtual memory





# Background

---

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at the same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs run at the same time
    - ▶ Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory -> each user program runs faster



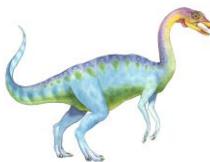


# Virtual memory

---

- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes





# Virtual memory (Cont.)

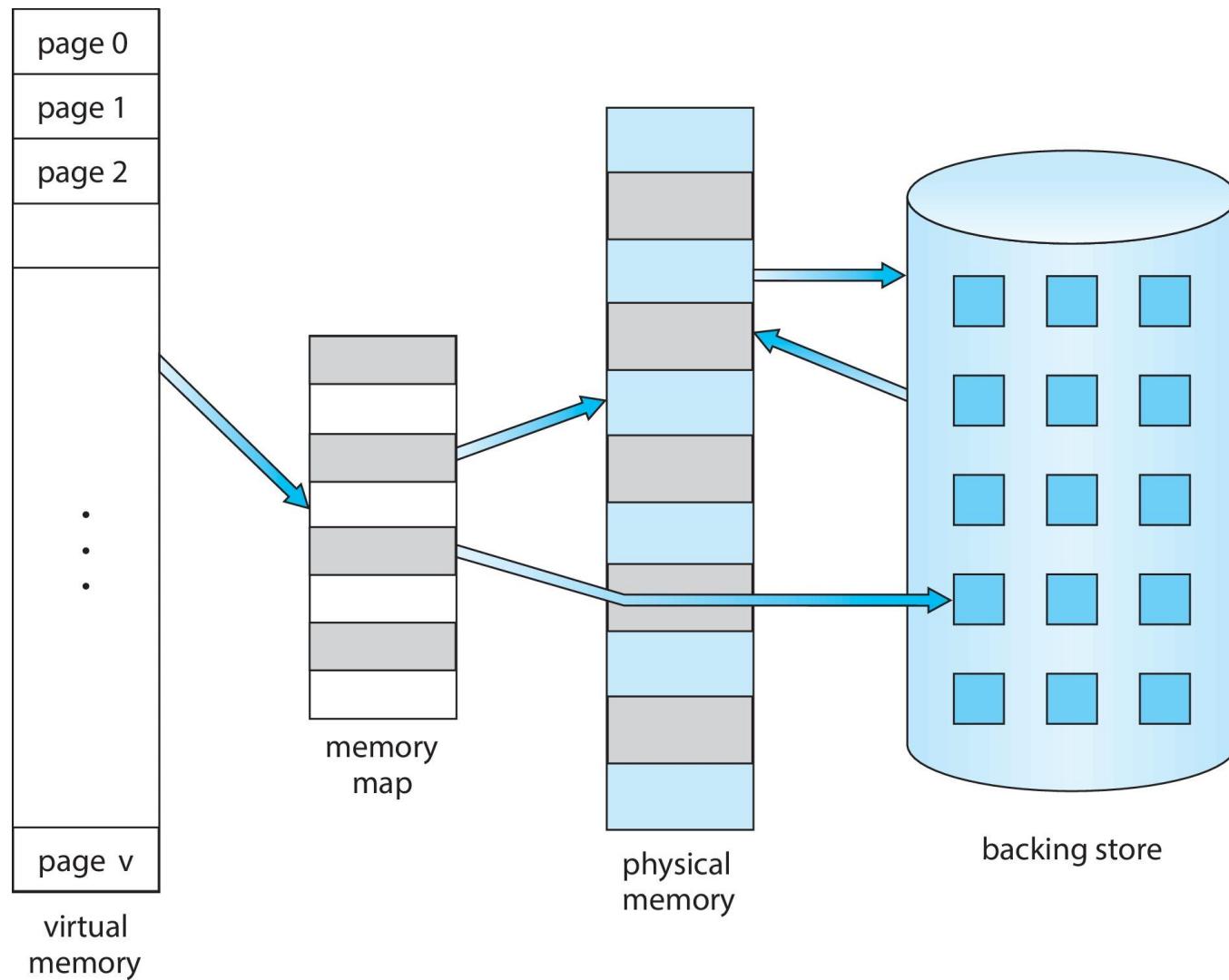
---

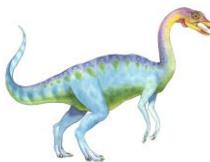
- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation





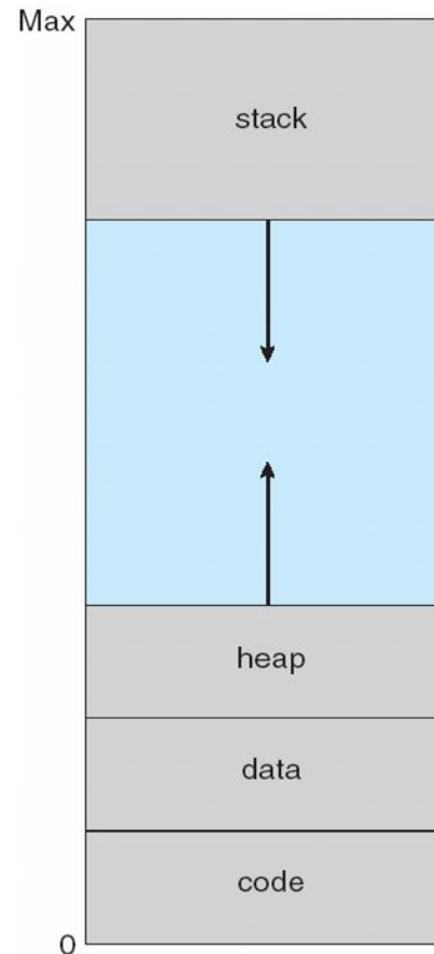
# Virtual Memory That is Larger Than Physical Memory

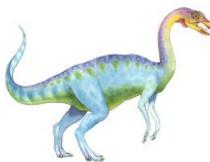




# Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
    - ▶ No physical memory needed until heap or stack grows to a given new page



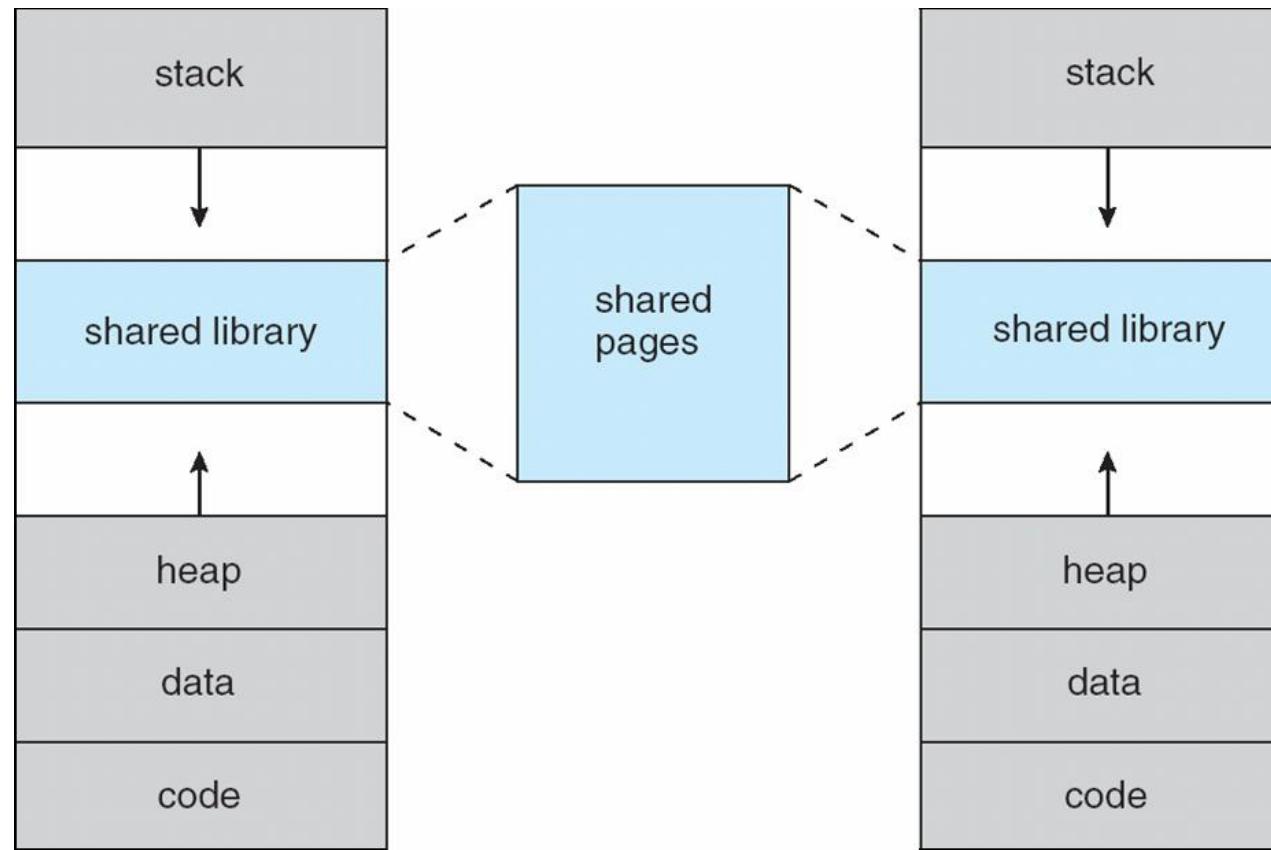


- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc.
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation





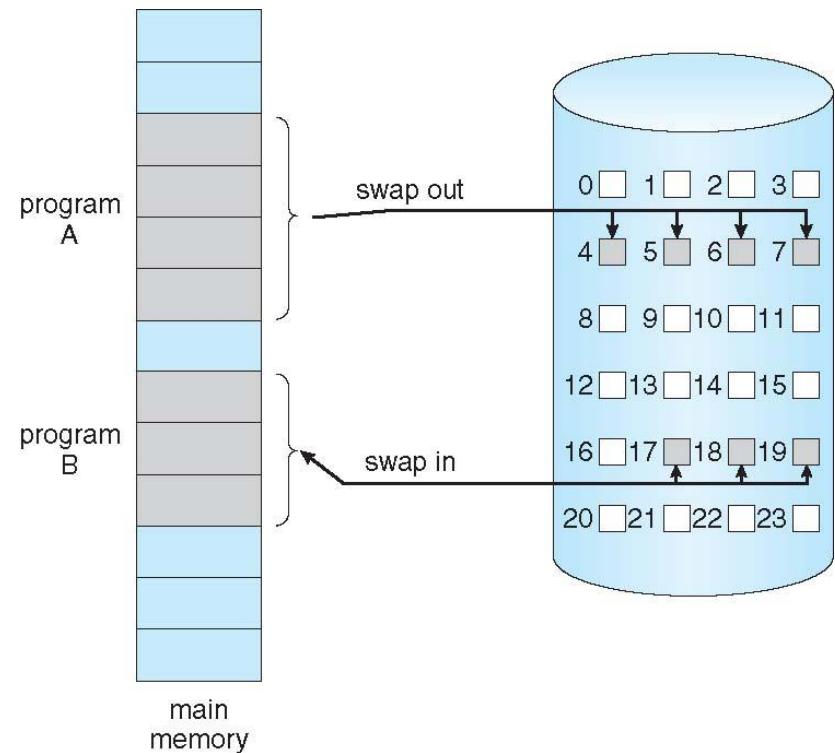
# Shared Library Using Virtual Memory





# Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory **only when it is needed**
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)



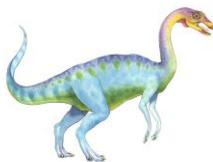


# Demand Paging

---

- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**





# Basic Concepts

---

- With swapping, pager guesses which pages will be used before swapping out again
  - Instead, pager brings in only those pages into memory
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
  - No difference from non demand-paging
- If page needed are not memory resident
  - Need to detect and load the page into memory from storage
    - ▶ Without changing program behavior
    - ▶ Without programmer needing to change code





# Valid-Invalid Bit

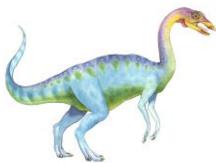
- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

Frame #	valid-invalid bit
	v
	v
	v
	i
...	
	i
	i

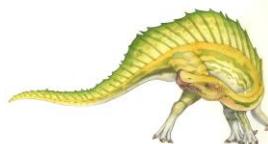
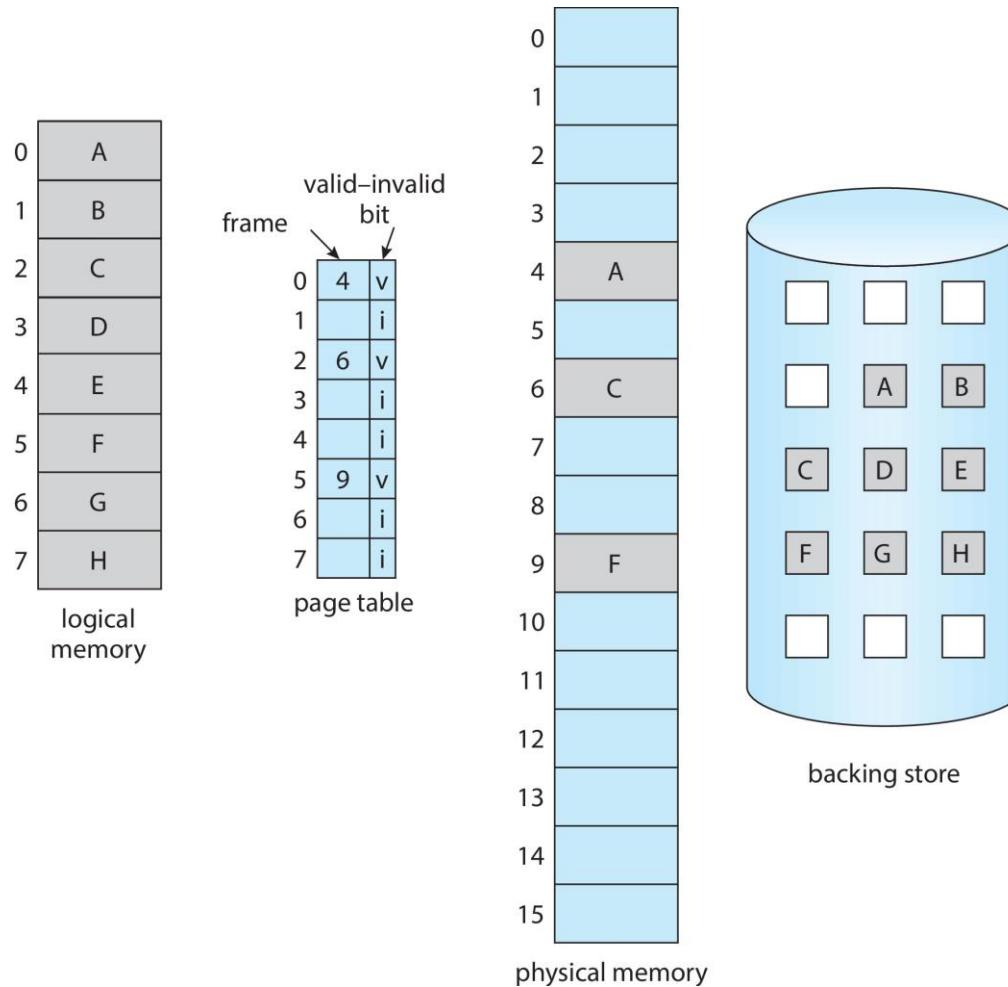
page table

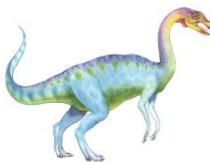
- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault





# Page Table When Some Pages Are Not in Main Memory





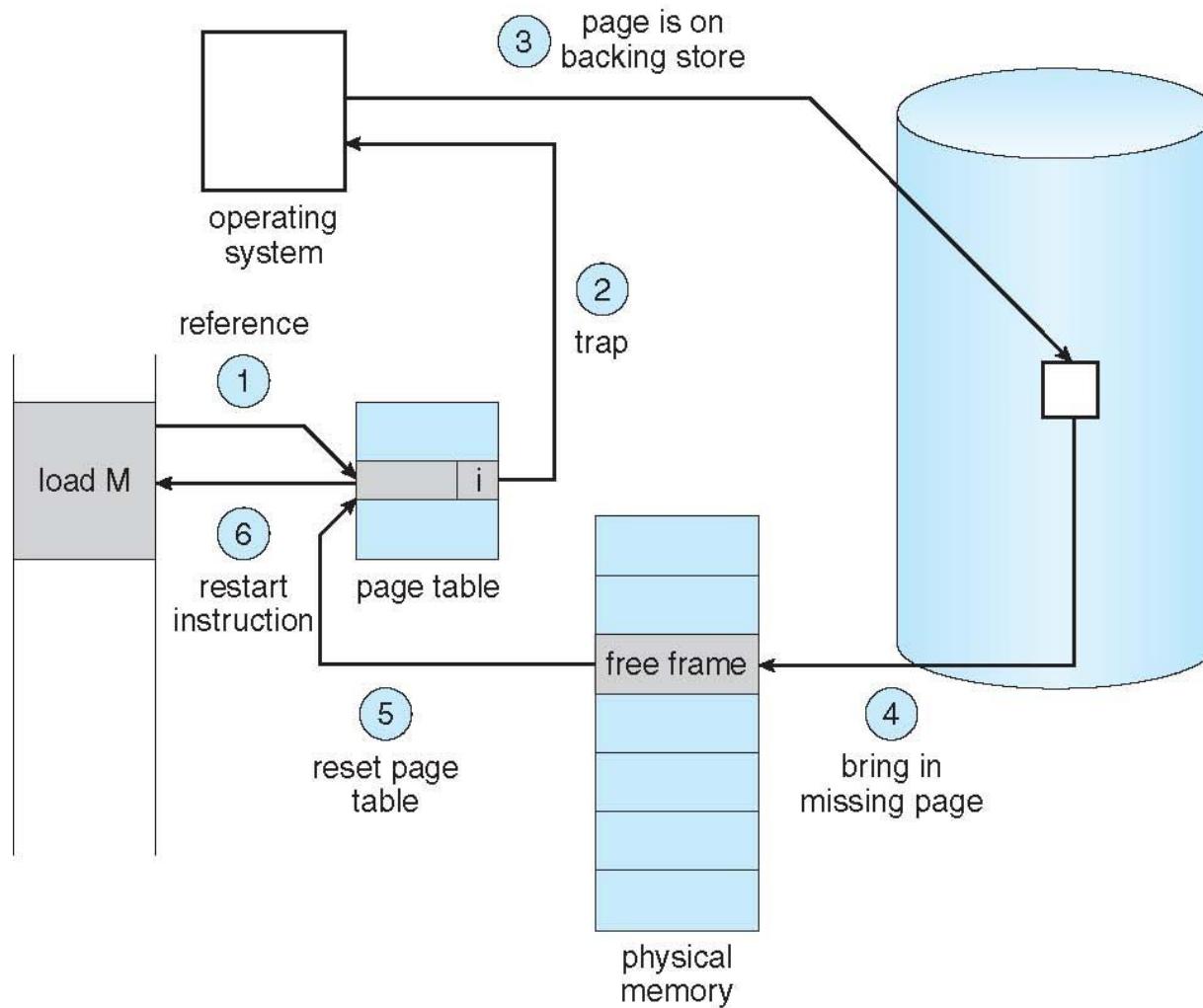
# Steps in Handling Page Fault

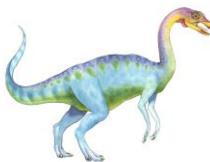
1. If there is a reference to a page, first reference to that page will trap to OS
  - Page fault
2. OS looks at another table to decide:
  - Invalid reference  $\Rightarrow$  abort
  - Just not in memory
3. Find free frame
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory  
Set validation bit = **V**
6. Restart the instruction that caused the page fault





# Steps in Handling a Page Fault (Cont.)





# Aspects of Demand Paging

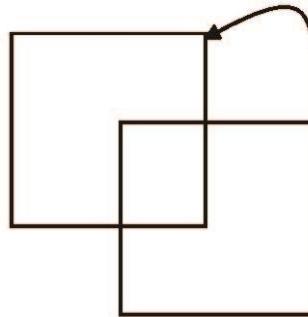
- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart





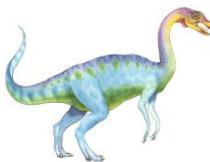
# Instruction Restart

- Consider an instruction that could access several different locations
  - Block move



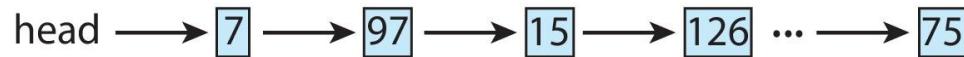
- Auto increment/decrement location
- Restart the whole operation?
  - ▶ What if source and destination overlap?





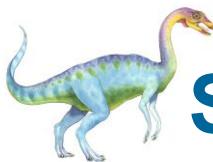
# Free-Frame List

- When a page fault occurs, the OS must bring the desired page from secondary storage into main memory
- Most OS maintain a **free-frame list** -- a pool of free frames for satisfying such requests



- OS typically allocates free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed-out before being allocated
- When a system starts up, all available memory is placed on the free-frame list

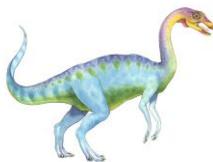




# Stages in Demand Paging – Worst Case

1. Trap to the OS
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
  - a) Wait in a queue for this device until the read request is serviced
  - b) Wait for the device seek and/or latency time
  - c) Begin the transfer of the page to a free frame

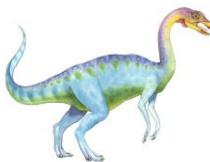




# Stages in Demand Paging (Cont.)

6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then  
resume the interrupted instruction

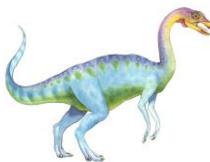




# Performance of Demand Paging

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- Page Fault Rate  $0 \leq p \leq 1$ 
  - if  $p = 0$  no page faults
  - if  $p = 1$ , every reference is a fault
- Effective Access Time (EAT)
$$\text{EAT} = (1 - p) \times \text{memory access} + p \times (\text{page fault overhead} + \text{swap page out} + \text{swap page in})$$



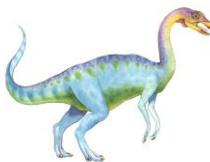


# Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $EAT = (1 - p) \times 200 + p (8 \text{ milliseconds})$ 
$$= (1 - p) \times 200 + p \times 8,000,000$$
$$= 200 + p \times 7,999,800$$
- If one access out of 1,000 causes a page fault, then  
 $EAT = 8.2 \text{ microseconds}$ 

This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - $220 > 200 + 7,999,800 \times p$ 
$$20 > 7,999,800 \times p$$
  - $p < .0000025$
  - < one page fault in every 400,000 memory accesses





# Demand Paging Optimizations

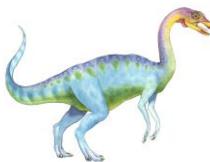
- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix





- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - ▶ Pages not associated with a file (like stack and heap) – **anonymous memory**
    - ▶ Pages modified in memory but not yet written back to the file system
- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)





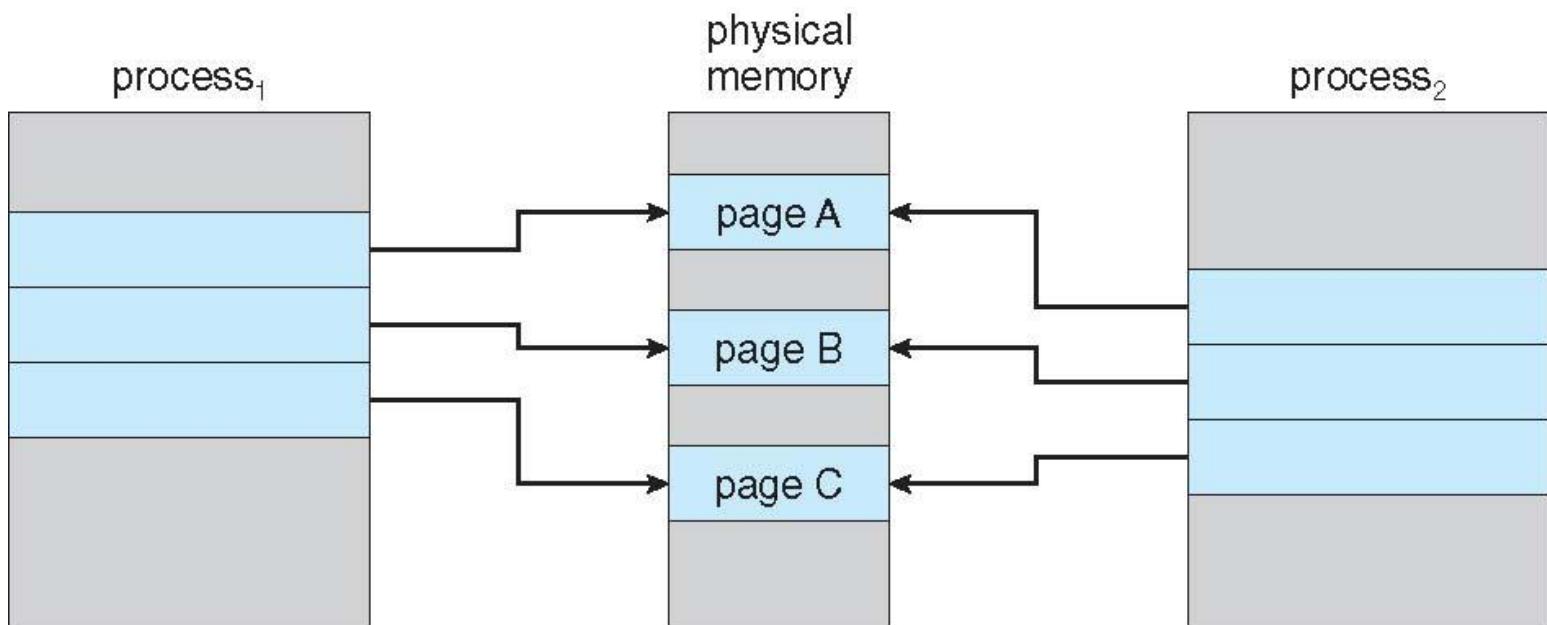
# Copy-on-Write

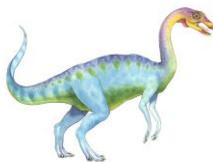
- **Copy-on-Write** (COW) allows both parent and child processes to initially **share** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied



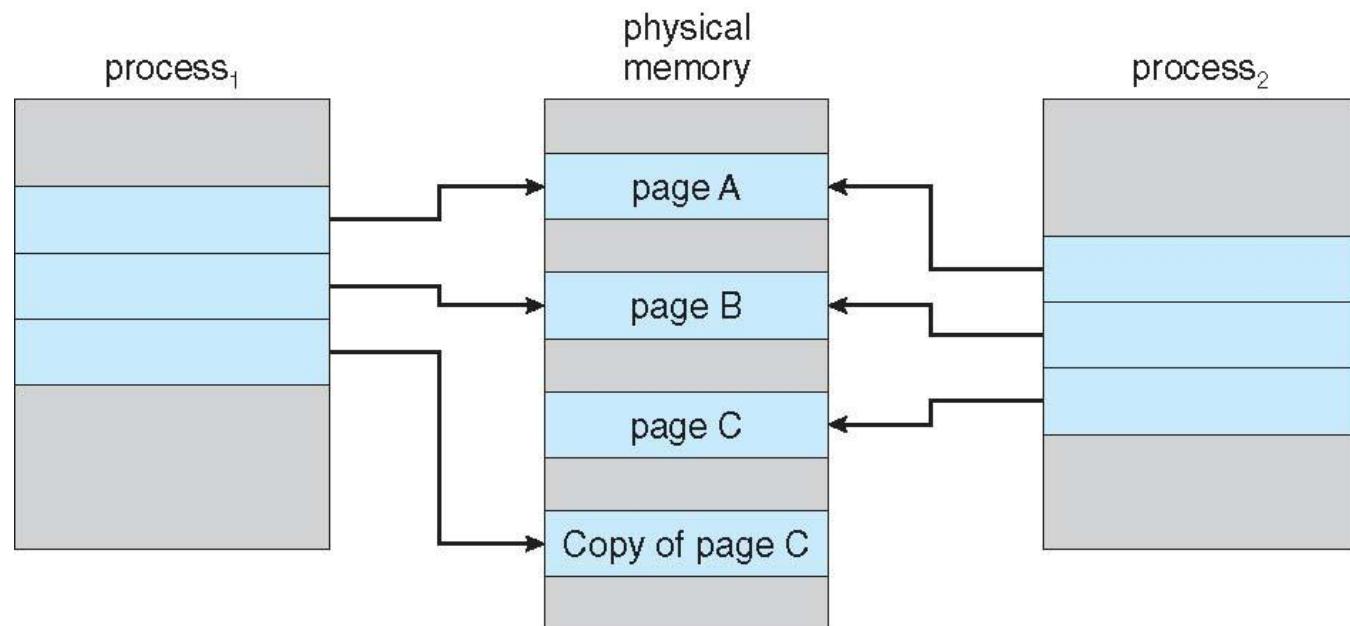


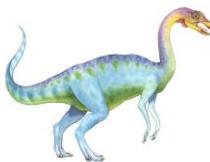
# Before Process 1 Modifies Page C





# After Process 1 Modifies Page C





- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
    - ▶ Don't want to have to free a frame as well as other processing on page fault
  - Why zero-out a page before allocating it?
- vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call exec()
  - Very efficient

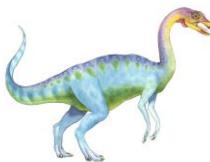




# What Happens if There is no Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- **Page replacement** – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times



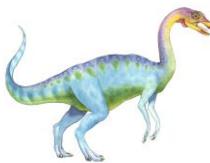


# Page Replacement

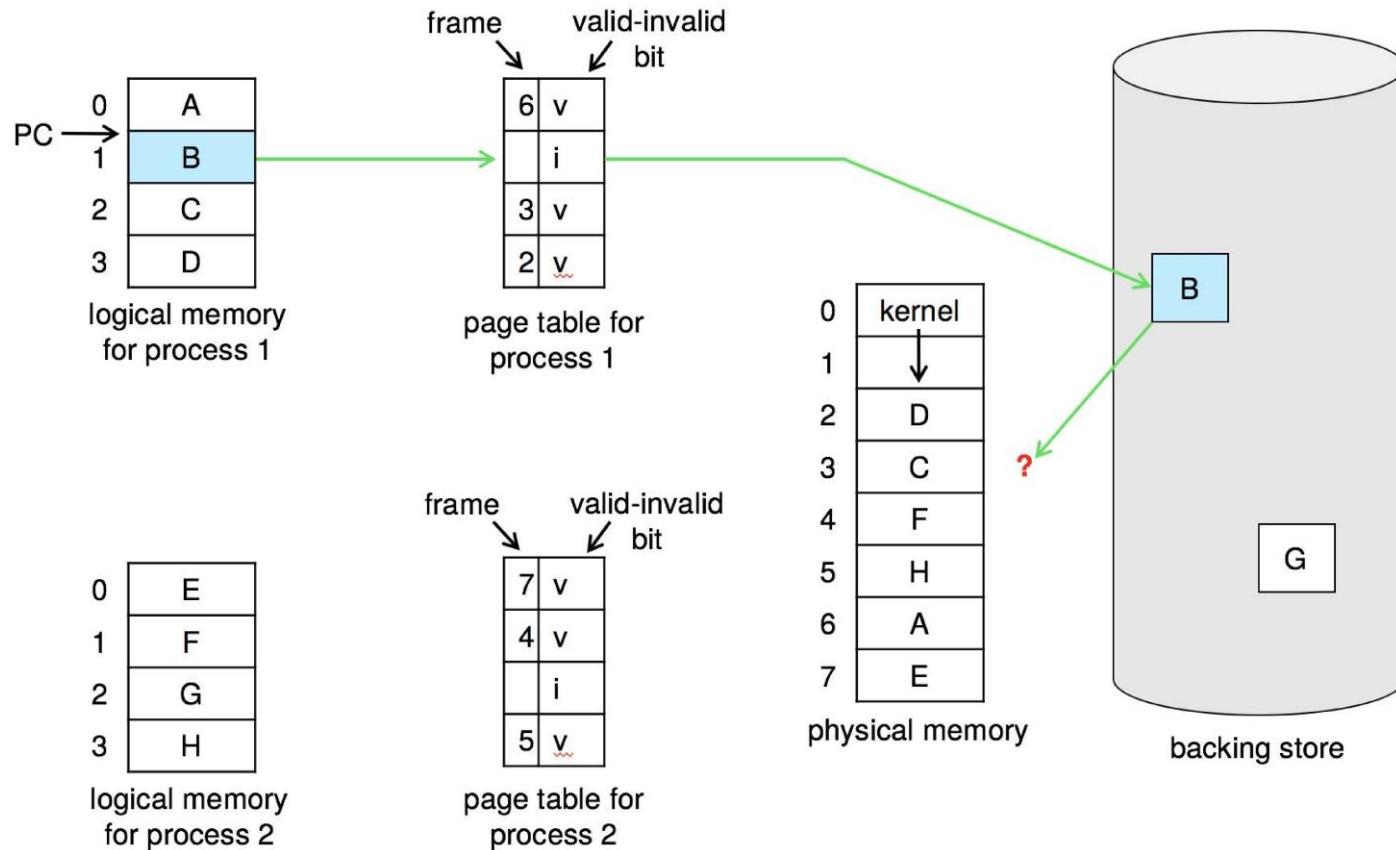
---

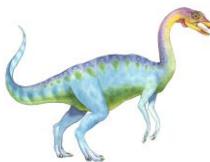
- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory





# Need For Page Replacement





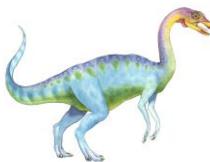
# Basic Page Replacement

---

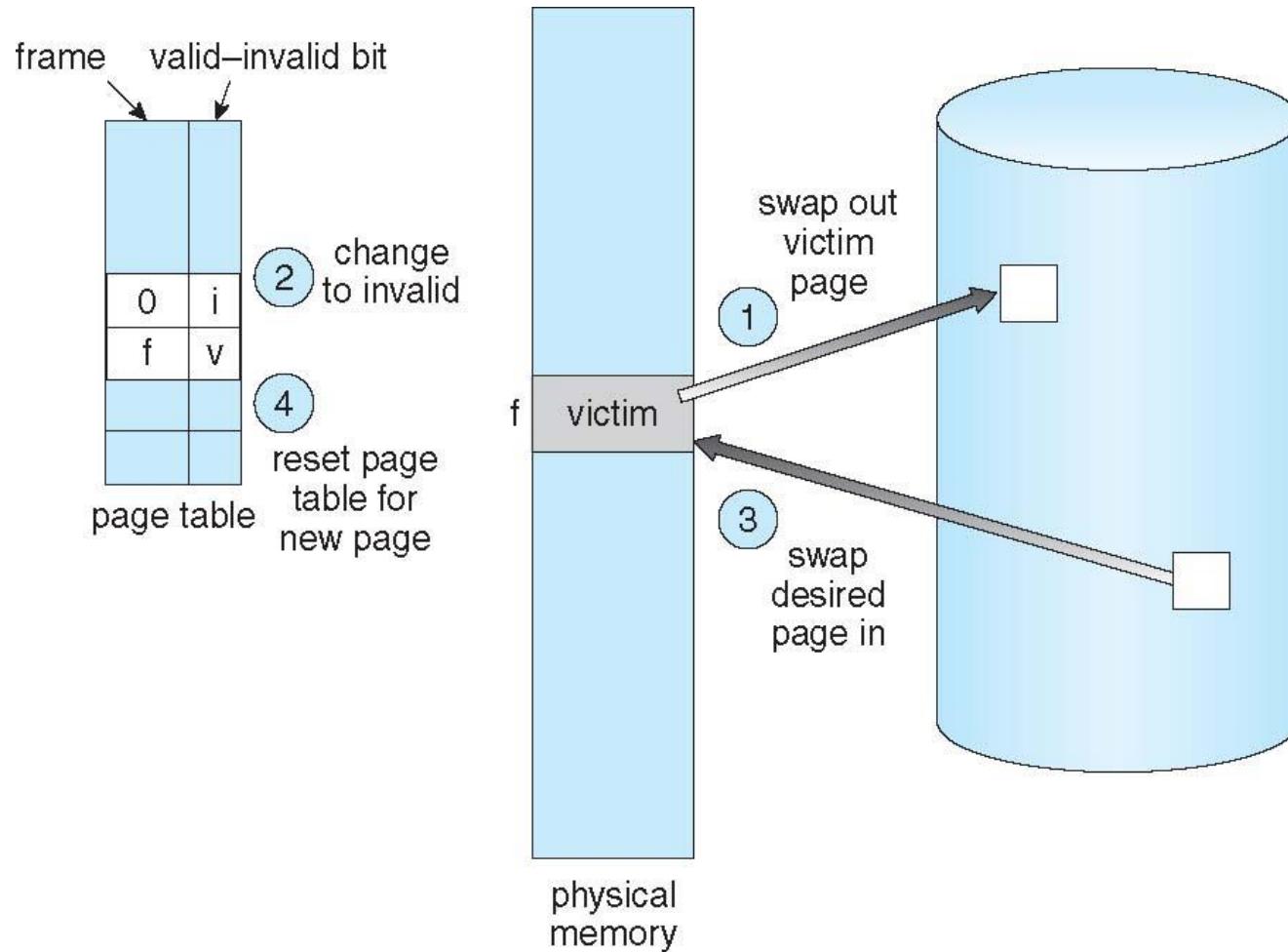
1. Find the location of the desired page on disk
2. Find a free frame:
  - If there is a free frame, use it
  - If there is no free frame, use a page replacement algorithm to select a **victim frame**
    - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

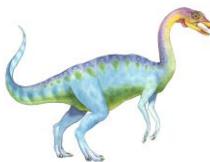
Note now potentially 2 page transfers for page fault – increasing EAT





# Page Replacement



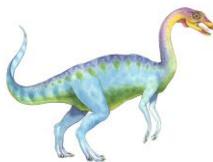


# Page and Frame Replacement Algorithms

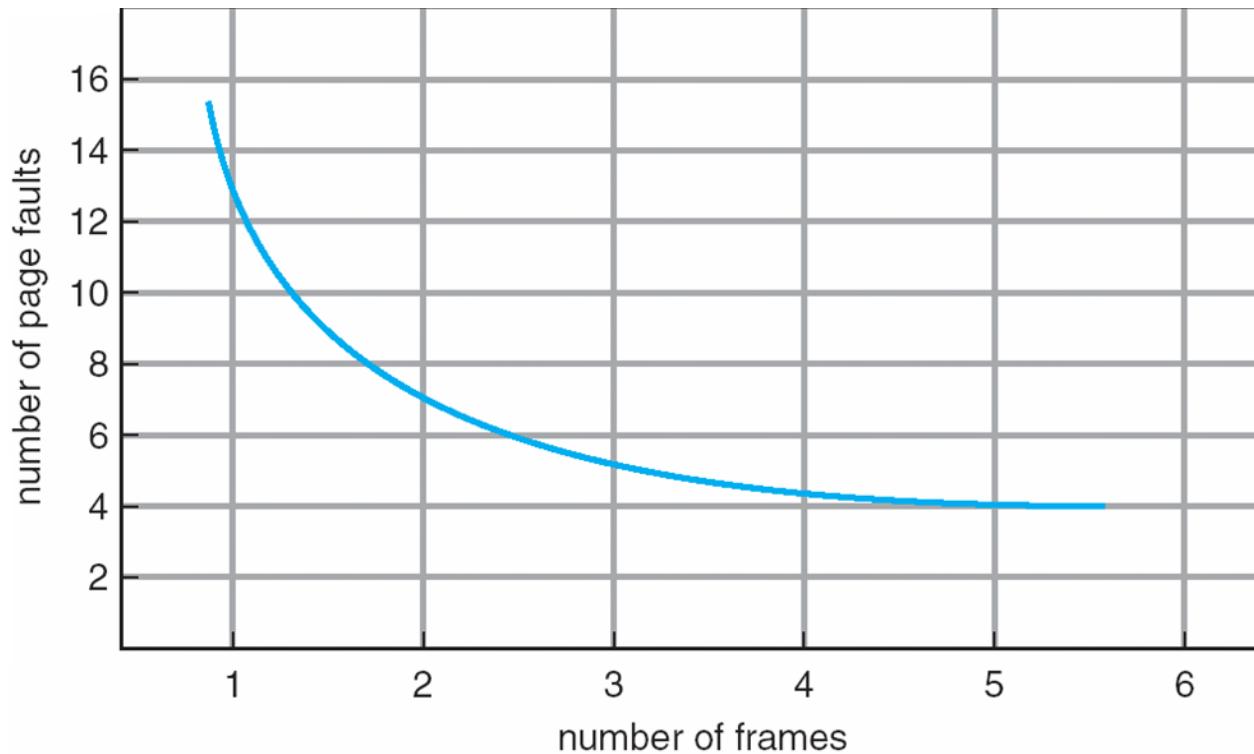
- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

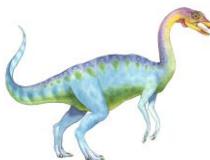
**7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**





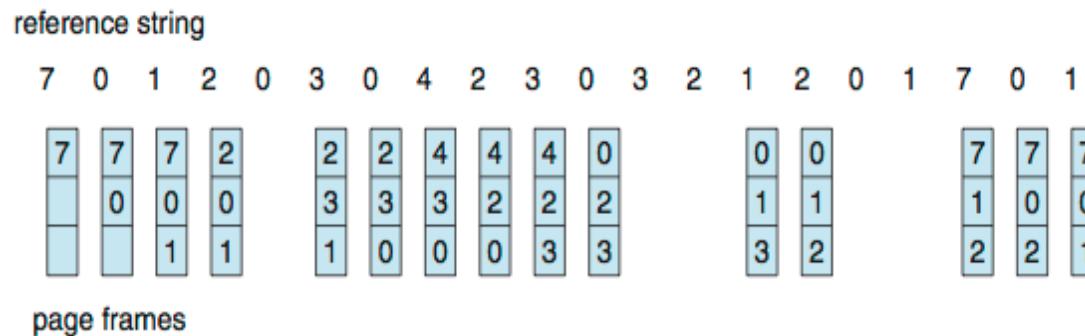
# Graph of Page Faults Versus the Number of Frames





# First-In-First-Out (FIFO) Algorithm

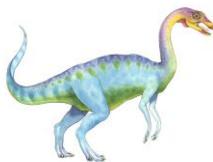
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



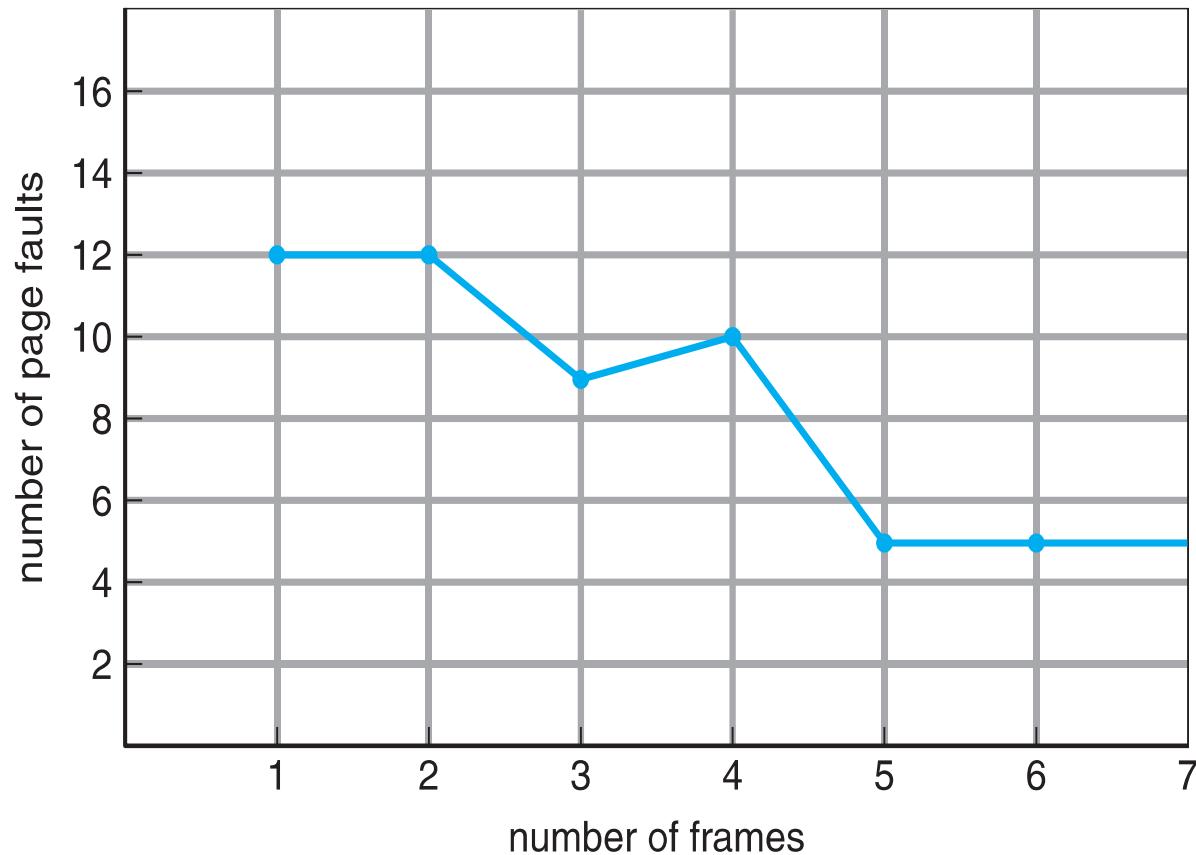
15 page faults

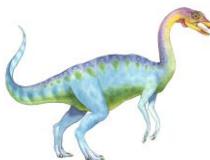
- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - ▶ **Belady's Anomaly**
- How to track ages of pages?
  - Just use a FIFO queue





# FIFO Illustrating Belady' s Anomaly



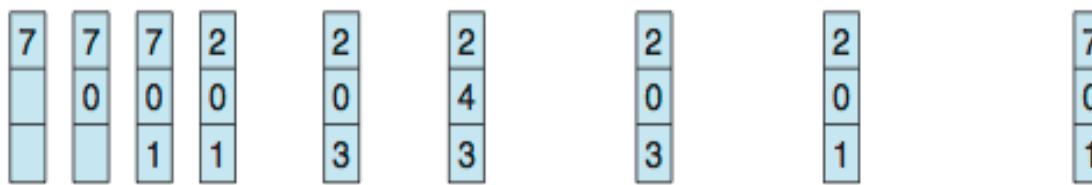


# Optimal Algorithm

- Replace page that **will** not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs

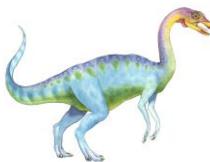
reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



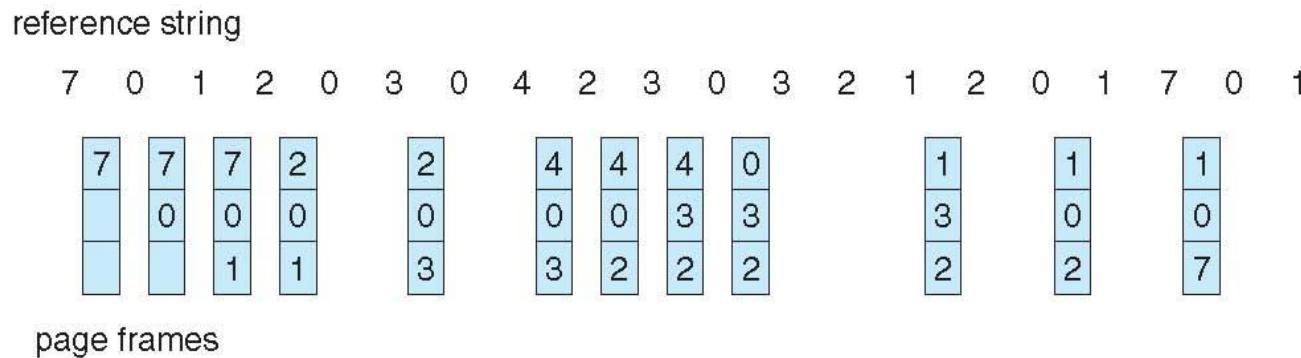
page frames





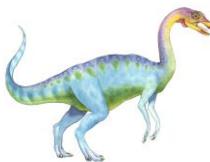
# Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that **has not been used in the most amount of time**
- Associate time of last use with each page



- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?





# LRU Algorithm (Cont.)

---

- **Counter** implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - ▶ Search through table needed
- **Stack** implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - ▶ move it to the top
    - ▶ requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement

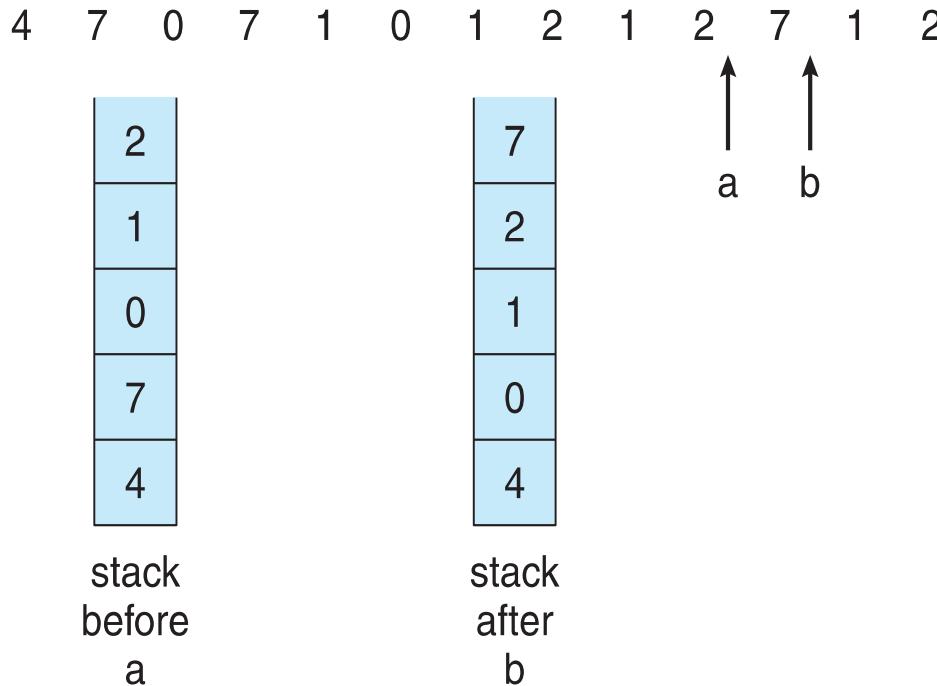


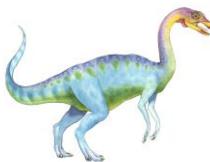


# LRU Algorithm (Cont.)

- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly
- Use of a Stack to Record Most Recent Page References

reference string



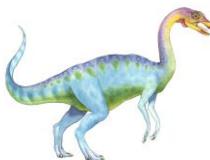


# LRU Approximation Algorithms

---

- LRU needs special hardware and still slow
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - ▶ We do not know the order, however



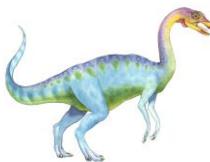


# LRU Approximation Algorithms (cont.)

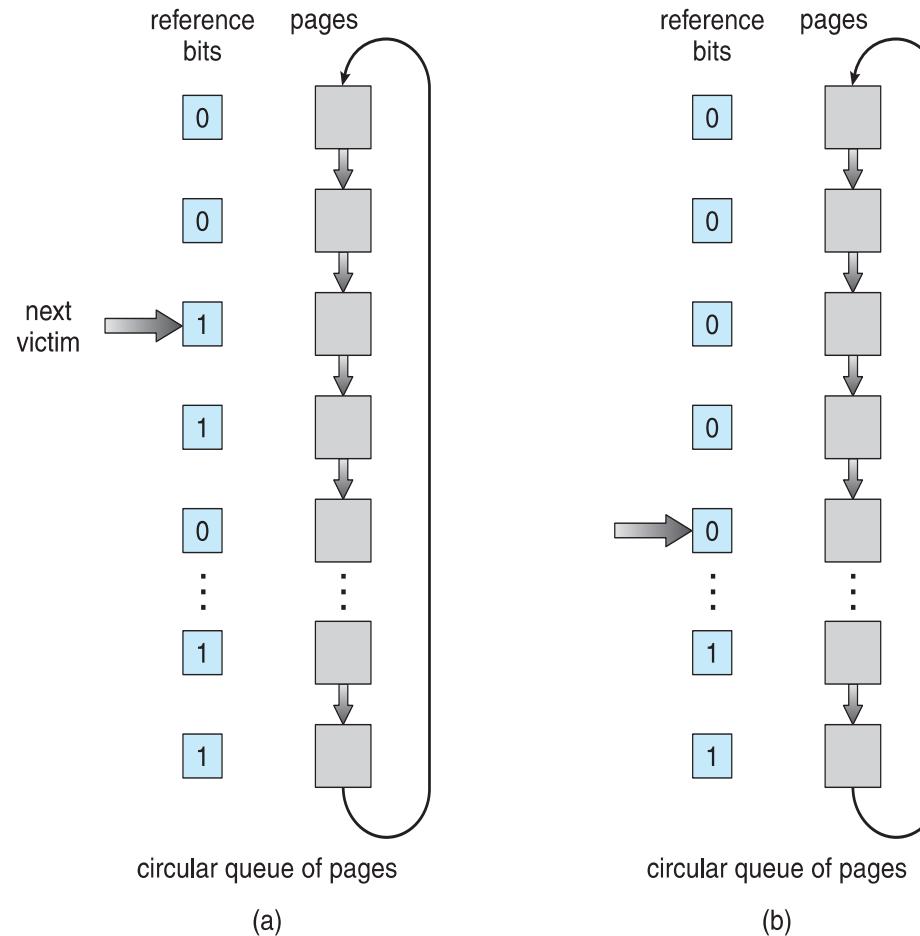
## ■ Second-chance algorithm

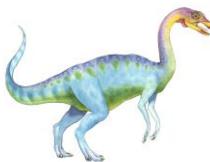
- Generally FIFO, plus hardware-provided reference bit
- **Clock** replacement
- If page to be replaced has
  - ▶ Reference bit = 0 -> replace it
  - ▶ reference bit = 1 then:
    - set reference bit 0, leave page in memory
    - replace next page, subject to same rules





# Second-chance Algorithm

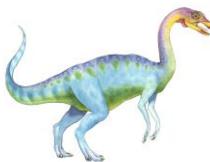




# Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify):
  - (0, 0) neither recently used nor modified – best page to replace
  - (0, 1) not recently used but modified – not quite as good, must write out before replacement
  - (1, 0) recently used but clean – probably will be used again soon
  - (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement
- When page replacement called for, use the clock scheme but use the four classes to replace page in lowest non-empty class
  - Might need to search circular queue several times

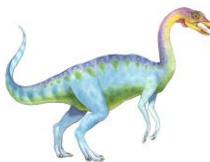




# Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **Least Frequently Used (LFU) Algorithm:**
  - Replaces page with smallest count
- **Most Frequently Used (MFU) Algorithm:**
  - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used
- Not common
  - Implementation is expensive

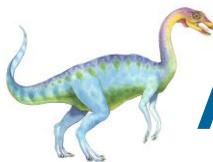




# Page-Buffering Algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected



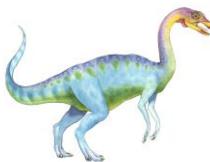


# Applications and Page Replacement

---

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can give direct access to the disk, getting out of the way of the applications
  - **Raw disk** mode
- Bypasses buffering, locking, etc.





# Allocation of Frames

---

- Each process needs ***minimum*** number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- ***Maximum*** of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations





# Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

$s_i$  = size of process  $p_i$

$S = \sum s_i$

$m$  = total number of frames

$a_i$  = allocation for  $p_i = \frac{s_i}{S} \times m$

$$m = 64$$

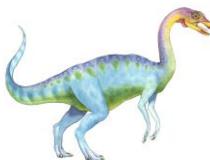
$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

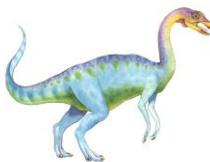




# Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput, so it's more common
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly under-utilized memory

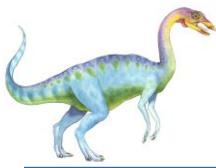




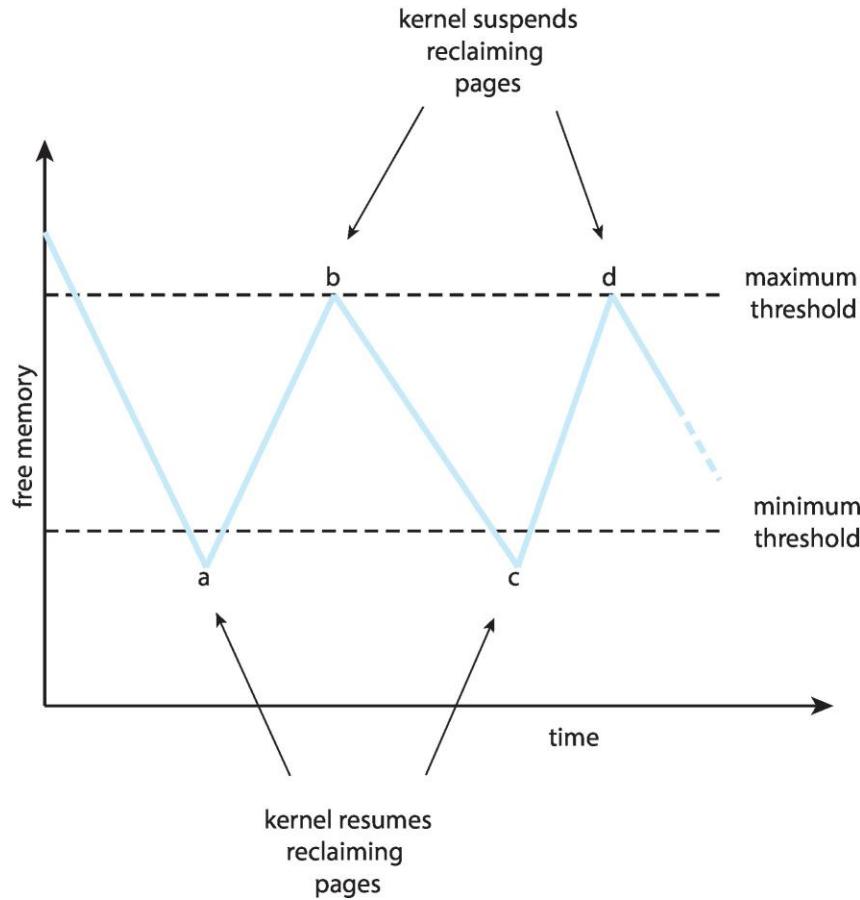
# Reclaiming Pages

- A strategy to implement global page-replacement policy
- All memory requests are satisfied from the free-frame list, rather than waiting for the list to drop to zero before we begin selecting pages for replacement
- Page replacement is triggered when the list falls below a certain threshold
- This strategy attempts to ensure there is always sufficient free memory to satisfy new requests.





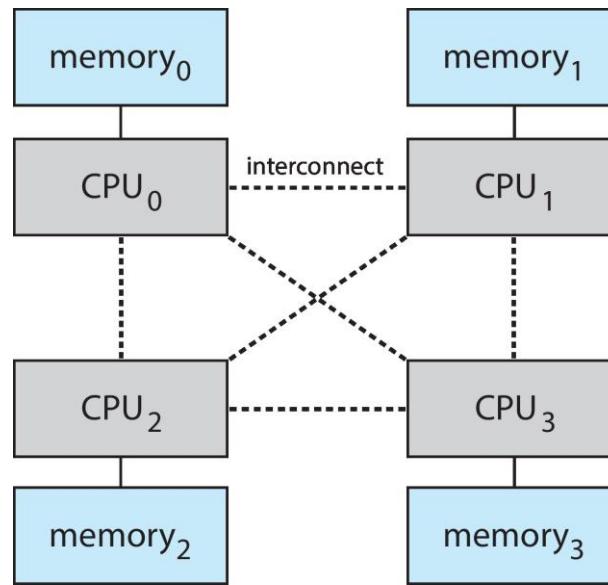
# Reclaiming Pages Example

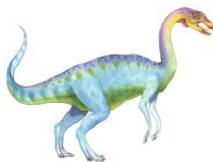




# Non-Uniform Memory Access

- So far, we assumed that all memory accessed equally
- Many systems are **NUMA** – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- NUMA multiprocessing architecture

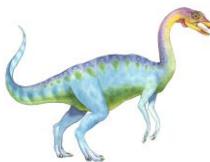




# Non-Uniform Memory Access (Cont.)

- Optimal performance comes from allocating memory “close to” the CPU on which the thread is scheduled
  - And modifying the scheduler to schedule the thread on the same system board when possible
  - Solved by Solaris by creating **Igroups (latency groups)**
    - ▶ Structure to track CPU / Memory low latency groups
    - ▶ When possible schedule all threads of a process and allocate all memory for that process within the Igroup



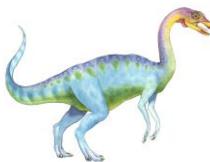


# Thrashing

---

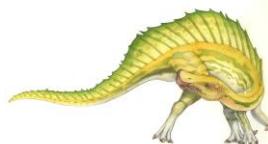
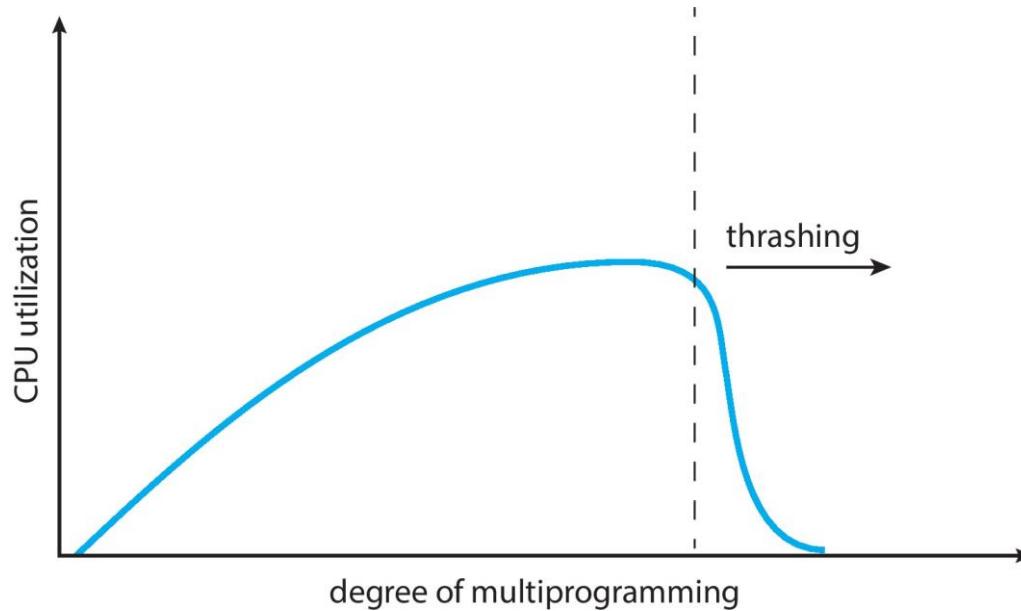
- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - ▶ Low CPU utilization
    - ▶ Operating system thinking that it needs to increase the degree of multiprogramming
    - ▶ Another process added to the system

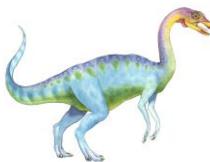




# Thrashing (Cont.)

- **Thrashing.** A process is busy swapping pages in and out





# Demand Paging and Thrashing

---

- Why does demand paging work?

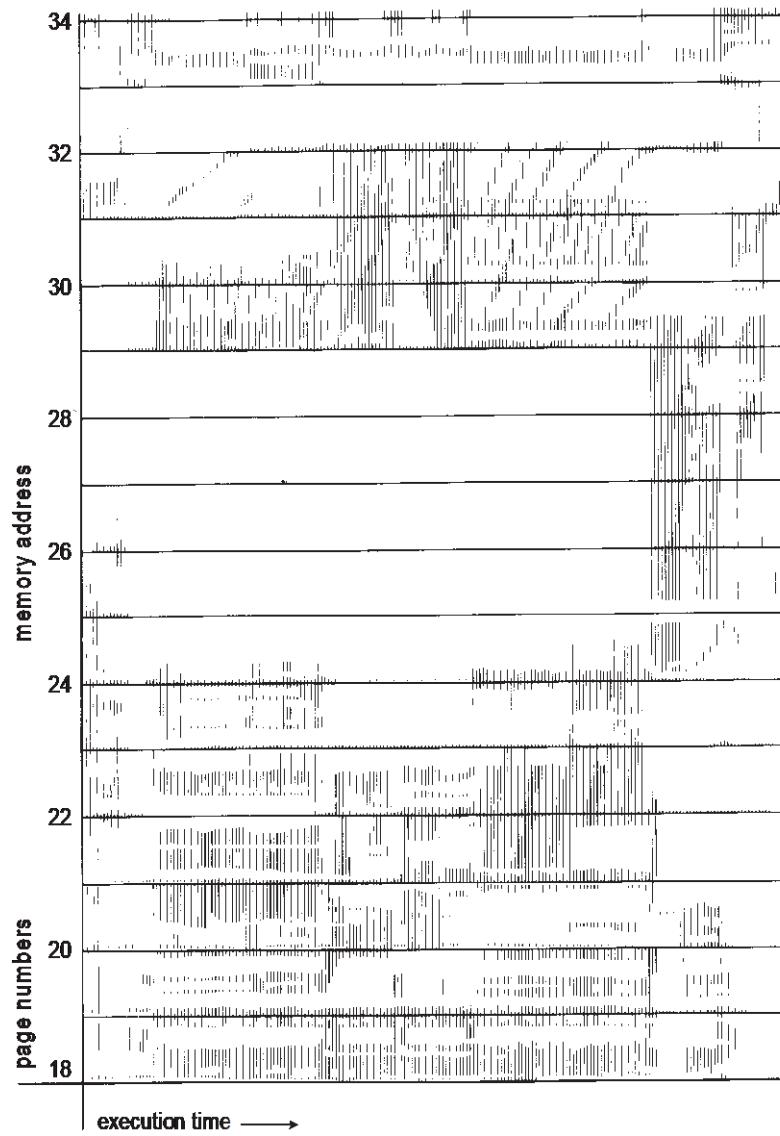
## Locality model

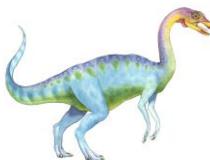
- Process migrates from one locality to another
- Localities may overlap
- Why does thrashing occur?  
 $\Sigma$  size of locality > total memory size
- Limit effects by using local or priority page replacement





# Locality In A Memory-Reference Pattern

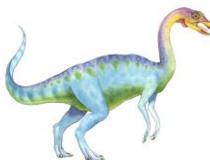




# Working-Set Model

- $\Delta \equiv$  working-set window  $\equiv$  a fixed number of page references  
Example: 10,000 instructions
- $WSS_i$  (working set of Process  $P_i$ ) = total number of pages referenced in the most recent  $\Delta$  (varies in time)
  - if  $\Delta$  too small will not encompass entire locality
  - if  $\Delta$  too large will encompass several localities
  - if  $\Delta = \infty \Rightarrow$  will encompass entire program
- $D = \sum WSS_i \equiv$  total demand frames
  - Approximation of locality



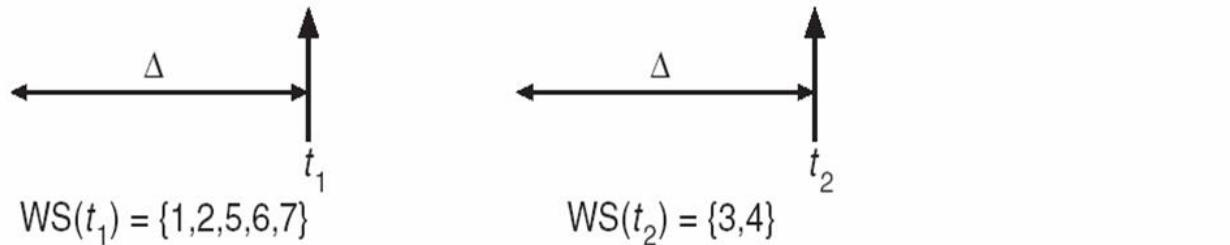


# Working-Set Model (Cont.)

- if  $D > m \Rightarrow$  Thrashing
- Policy if  $D > m$ , then suspend or swap out one of the processes

page reference table

... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...

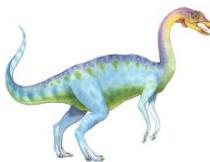




# Keeping Track of the Working Set

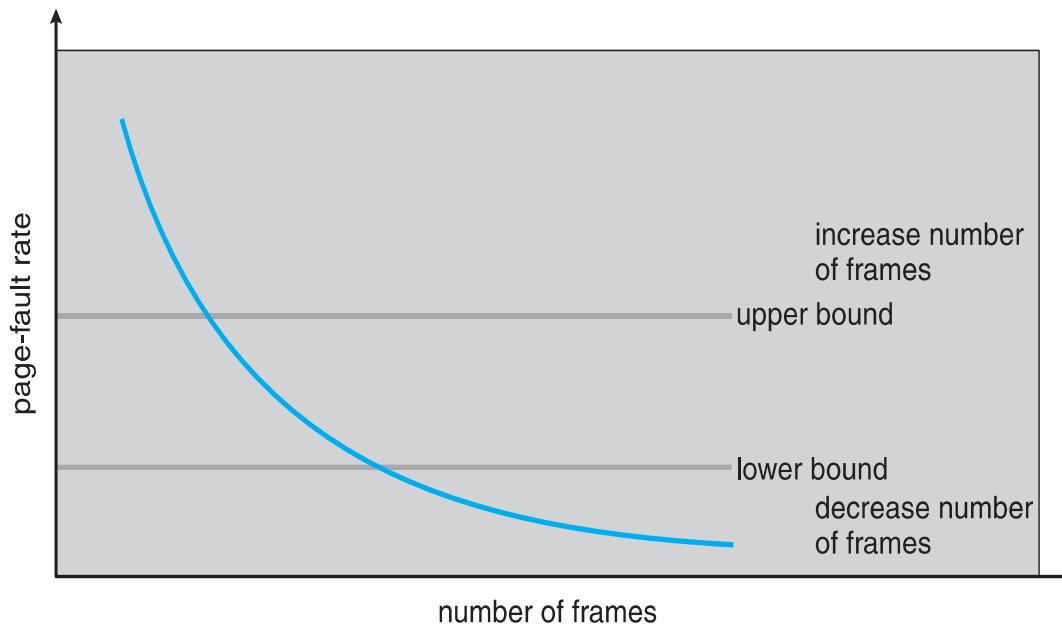
- Approximate with interval timer + a reference bit
- Example:  $\Delta = 10,000$ 
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1  $\Rightarrow$  page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

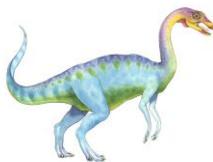




# Page-Fault Frequency

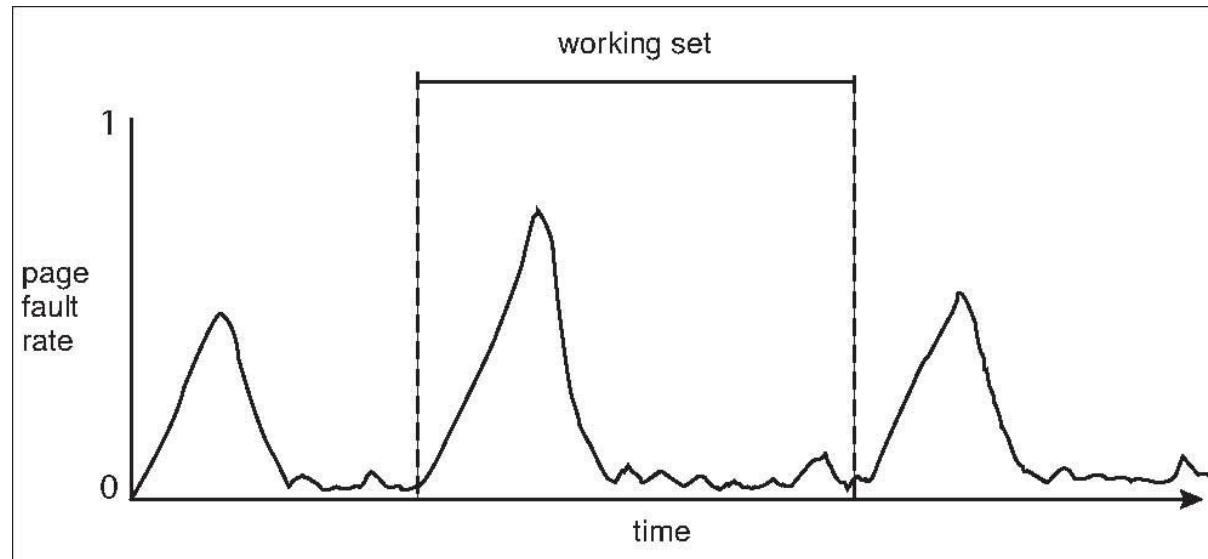
- More direct approach than WSS
- Establish “acceptable” **page-fault frequency (PFF)** rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame

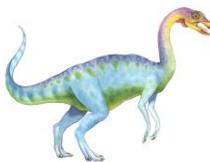




# Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time



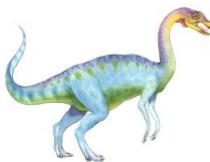


# Allocating Kernel Memory

---

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - ▶ i.e., for device I/O

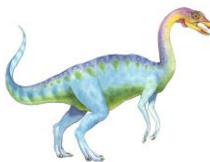




# Buddy System

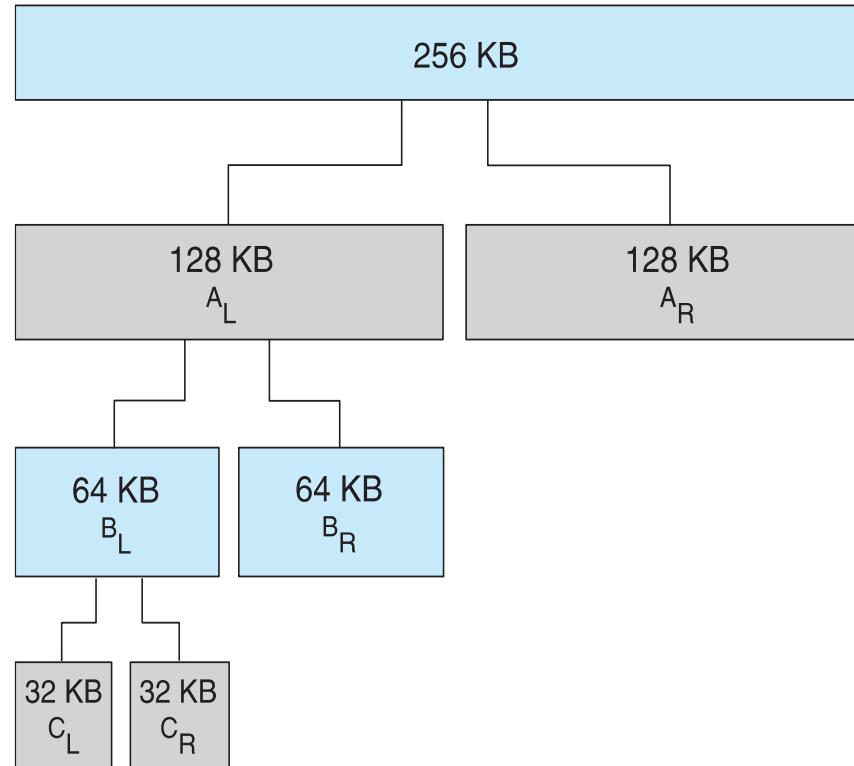
- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - ▶ Continue until appropriate sized chunk available
- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into  $A_L$  and  $A_R$  of 128KB each
    - ▶ One further divided into  $B_L$  and  $B_R$  of 64KB
      - One further into  $C_L$  and  $C_R$  of 32KB each – one used to satisfy request
- Advantage – quickly **coalesce** unused chunks into larger chunk
- Disadvantage - fragmentation





# Buddy System Allocator

physically contiguous pages

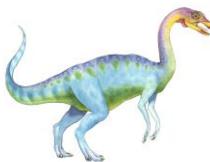




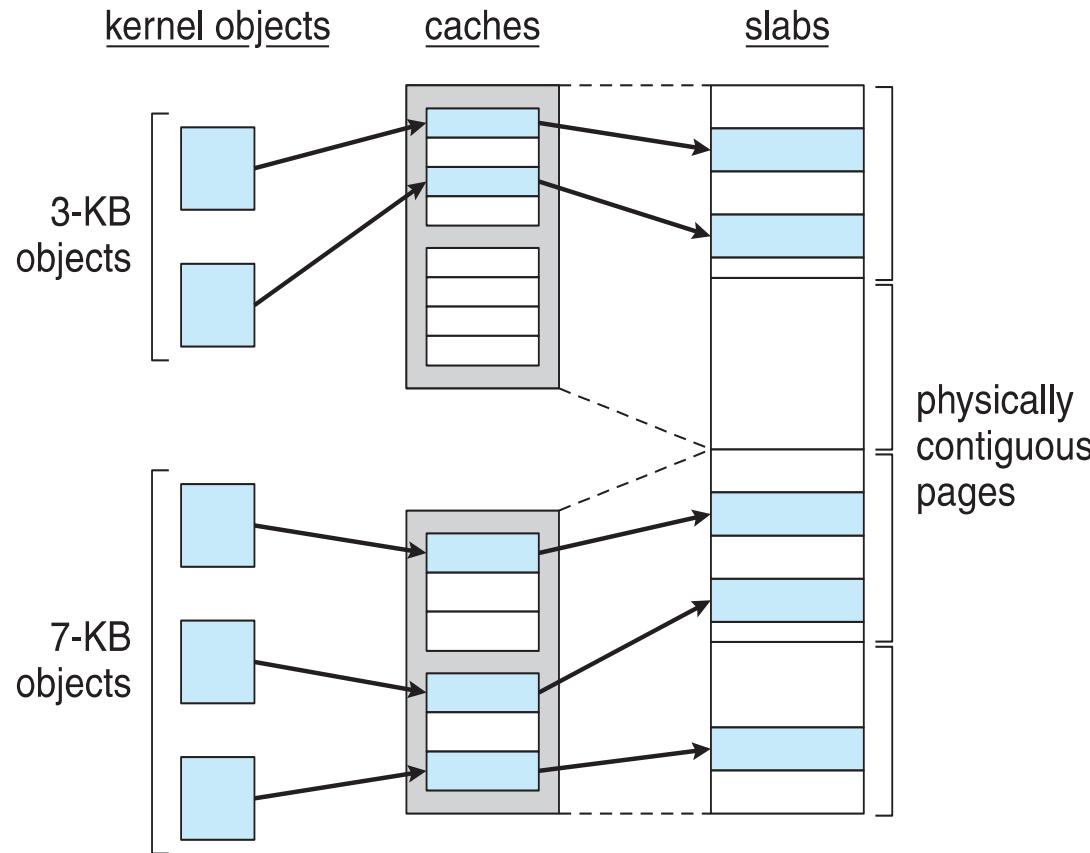
# Slab Allocator

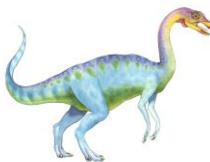
- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction





# Slab Allocation

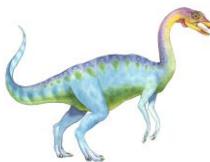




# Slab Allocator in Linux

- For example process descriptor is of type `struct task_struct`
- Approx 1.7KB of memory
- New task -> allocate new struct from cache
  - Will use existing free `struct task_struct`
- Slab can be in three possible states
  1. Full – all used
  2. Empty – all free
  3. Partial – mix of free and used
- Upon request, slab allocator
  1. Uses free struct in partial slab
  2. If none, takes one from empty slab
  3. If no empty slab, create new empty

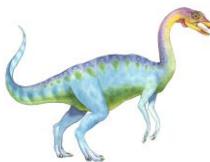




# Slab Allocator in Linux (Cont.)

- Slab started in Solaris, now wide-spread for both kernel mode and user memory in various OSes
- Linux 2.2 had SLAB, now has both SLOB and SLUB allocators
  - SLOB for systems with limited memory
    - ▶ Simple List of Blocks – maintains 3 list objects for small, medium, large objects
  - SLUB is performance-optimized SLAB removes per-CPU queues, metadata stored in page structure

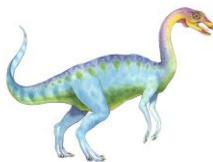




# Other Considerations

- Prepaging
- Page size
- TLB reach
- Inverted page table
- Program structure
- I/O interlock and page locking



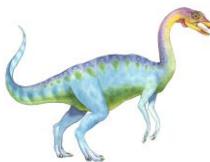


# Prepaging

---

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume  $s$  pages are prepaged and  $\alpha$  of the pages is used
  - Is cost of  $s * \alpha$  save pages faults  $>$  or  $<$  than the cost of prepaginaing  
 $s * (1 - \alpha)$  unnecessary pages?
  - $\alpha$  near zero  $\Rightarrow$  prepaginaing loses

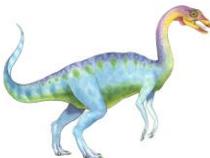




# Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - **Resolution**
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range  $2^{12}$  (4,096 bytes) to  $2^{22}$  (4,194,304 bytes)
- On average, growing over time





# TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- $\text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size})$
- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults
- Increase the Page Size
  - This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation





# Program Structure

- Program structure

- int[128,128] data;
- Each row is stored in one page
- Program 1

```
for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i, j] = 0;
```

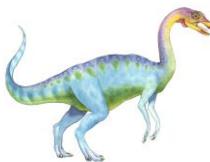
$$128 \times 128 = 16,384 \text{ page faults}$$

- Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i, j] = 0;
```

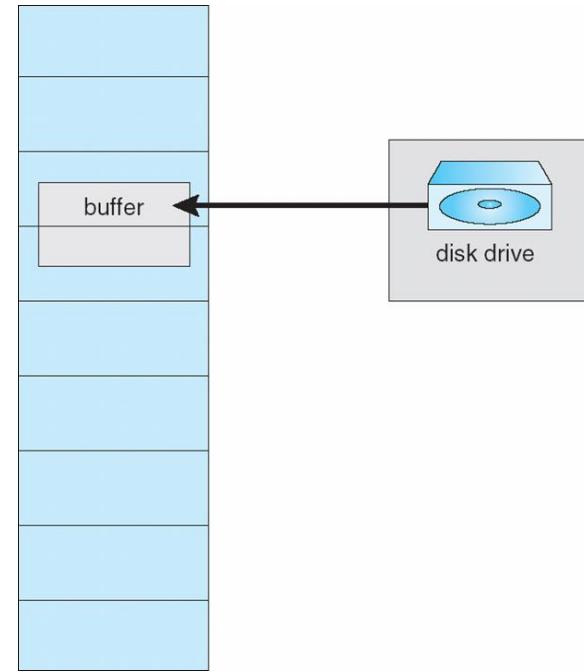
$$128 \text{ page faults}$$

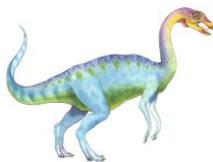




# I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
- **Pinning** of pages to lock into memory



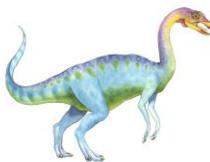


# Operating System Examples

---

- Windows
- Solaris



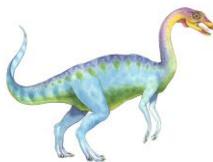


# Windows

---

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page
- Processes are assigned **working set minimum** and **working set maximum**
  - Working set minimum is the minimum number of pages the process is guaranteed to have in memory
  - A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
  - Working set trimming removes pages from processes that have pages in excess of their working set minimum



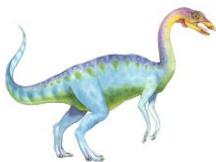


# Solaris

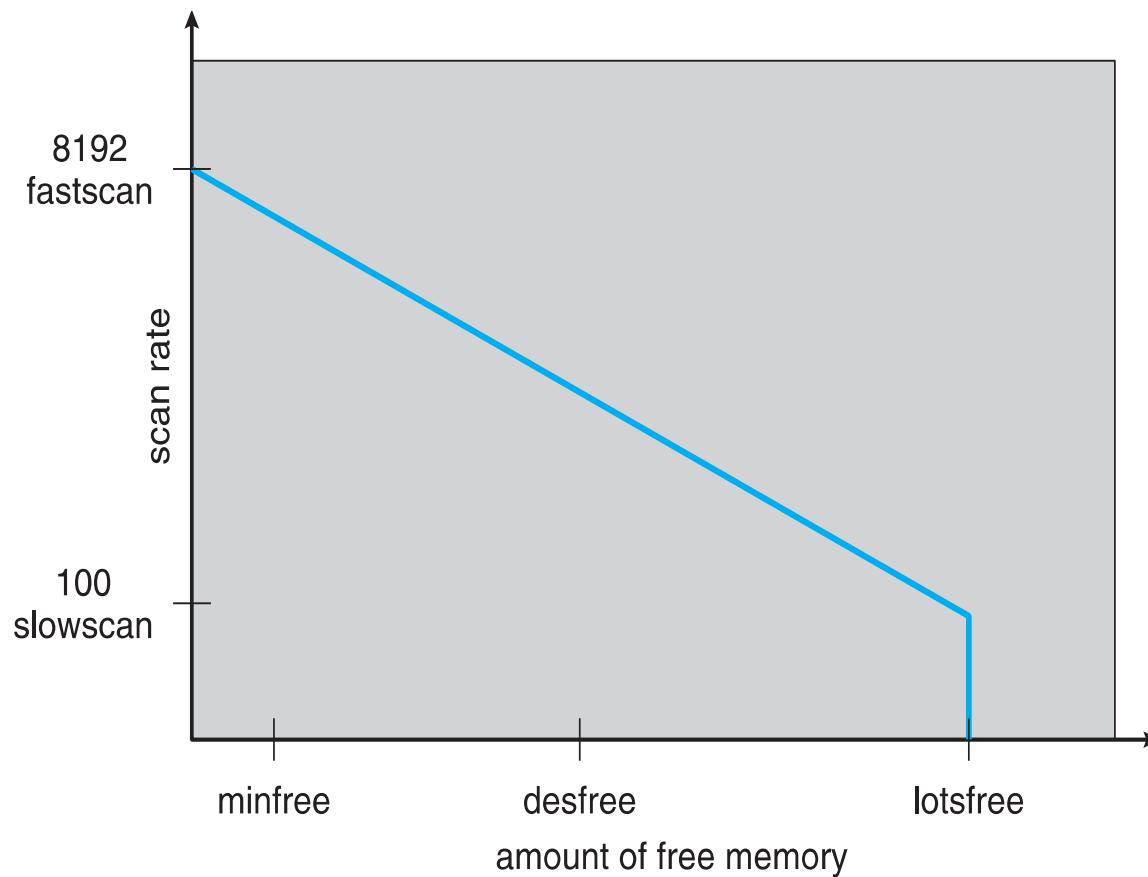
---

- Maintains a list of free pages to assign faulting processes
- **Lotsfree** – threshold parameter (amount of free memory) to begin paging
- **Desfree** – threshold parameter to increasing paging
- **Minfree** – threshold parameter to begin swapping
- Paging is performed by **pageout** process
- **Pageout** scans pages using modified clock algorithm
- **Scanrate** is the rate at which pages are scanned. This ranges from **slowscan** to **fastscan**
- **Pageout** is called more frequently depending upon the amount of free memory available
- **Priority paging** gives priority to process code pages

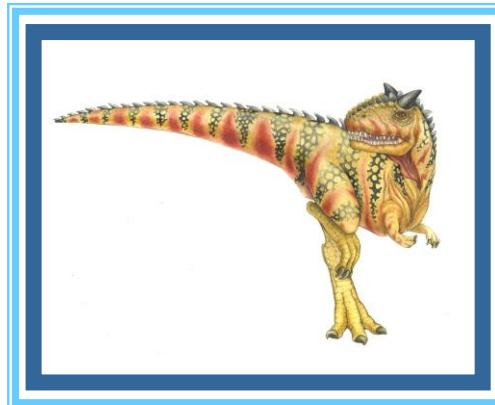


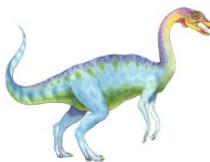


# Solaris 2 Page Scanner



# End of Chapter 10

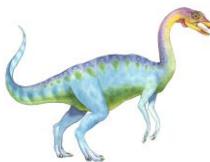




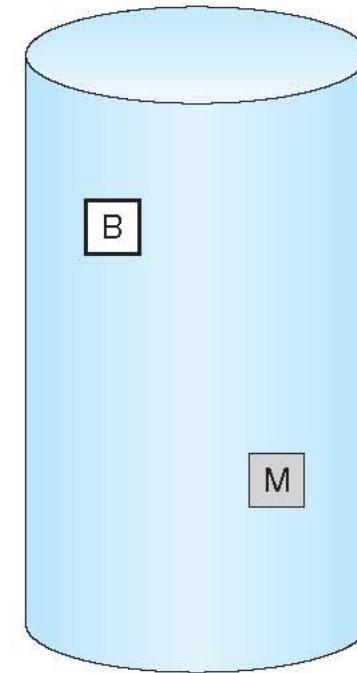
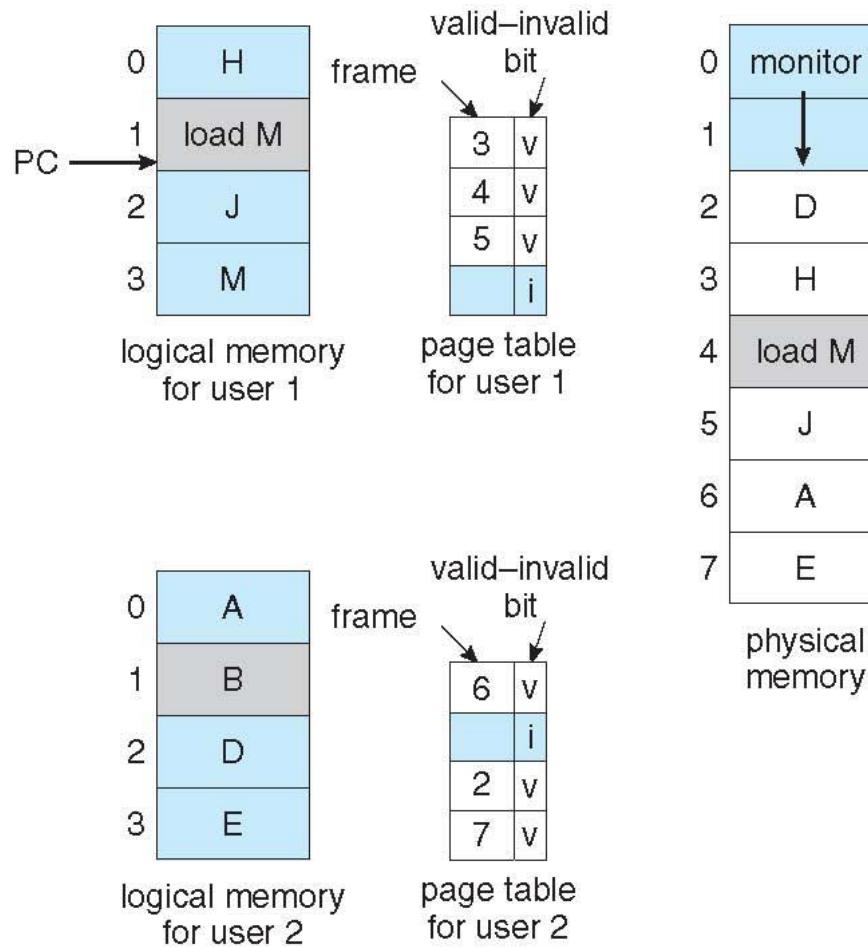
# Performance of Demand Paging

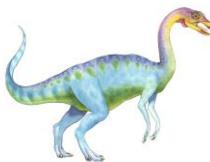
- Stages in Demand Paging (worse case)
  1. Trap to the operating system
  2. Save the user registers and process state
  3. Determine that the interrupt was a page fault
  4. Check that the page reference was legal and determine the location of the page on the disk
  5. Issue a read from the disk to a free frame:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame
  6. While waiting, allocate the CPU to some other user
  7. Receive an interrupt from the disk I/O subsystem (I/O completed)
  8. Save the registers and process state for the other user
  9. Determine that the interrupt was from the disk
  10. Correct the page table and other tables to show page is now in memory
  11. Wait for the CPU to be allocated to this process again
  12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction





# Need For Page Replacement





# Priority Allocation

---

- Use a proportional allocation scheme using priorities rather than size
- If process  $P_i$  generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number





# Memory Compression

- **Memory compression** -- rather than paging out modified frames to swap space, we compress several frames into a single frame, enabling the system to reduce memory usage without resorting to swapping pages.
- Consider the following free-frame-list consisting of 6 frames

free-frame list

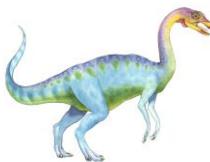


modified frame list



- Assume that this number of free frames falls below a certain threshold that triggers page replacement. The replacement algorithm (say, an LRU approximation algorithm) selects four frames -- 15, 3, 35, and 26 to place on the free-frame list. It first places these frames on a modified-frame list. Typically, the modified-frame list would next be written to swap space, making the frames available to the free-frame list. An alternative strategy is to compress a number of frames—say, three—and store their compressed versions in a single page frame.





# Memory Compression (Cont.)

- An alternative to paging is **memory compression**.
- Rather than paging out modified frames to swap space, we compress several frames into a single frame, enabling the system to reduce memory usage without resorting to swapping pages.

free-frame list



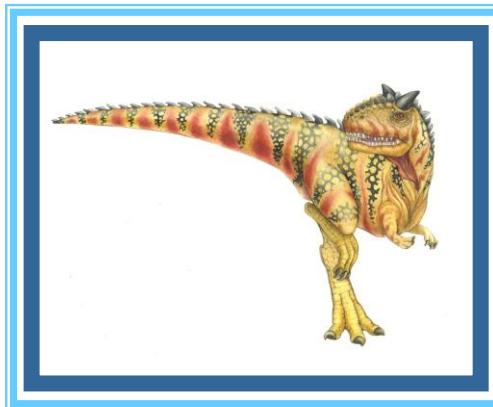
modified frame list

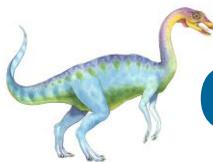


compressed frame list



# Chapter 11: Mass-Storage Systems



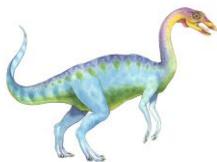


# Chapter 11: Mass-Storage Systems

---

- Overview of Mass Storage Structure
- HDD Scheduling
- NVM Scheduling
- Error Detection and Correction
- Storage Device Management
- Swap-Space Management
- Storage Attachment
- RAID Structure



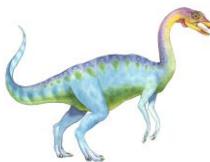


# Objectives

---

- Describe the physical structure of secondary storage devices and the effect of a device's structure on its uses
- Explain the performance characteristics of mass-storage devices
- Evaluate I/O scheduling algorithms
- Discuss operating-system services provided for mass storage, including RAID

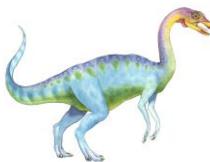




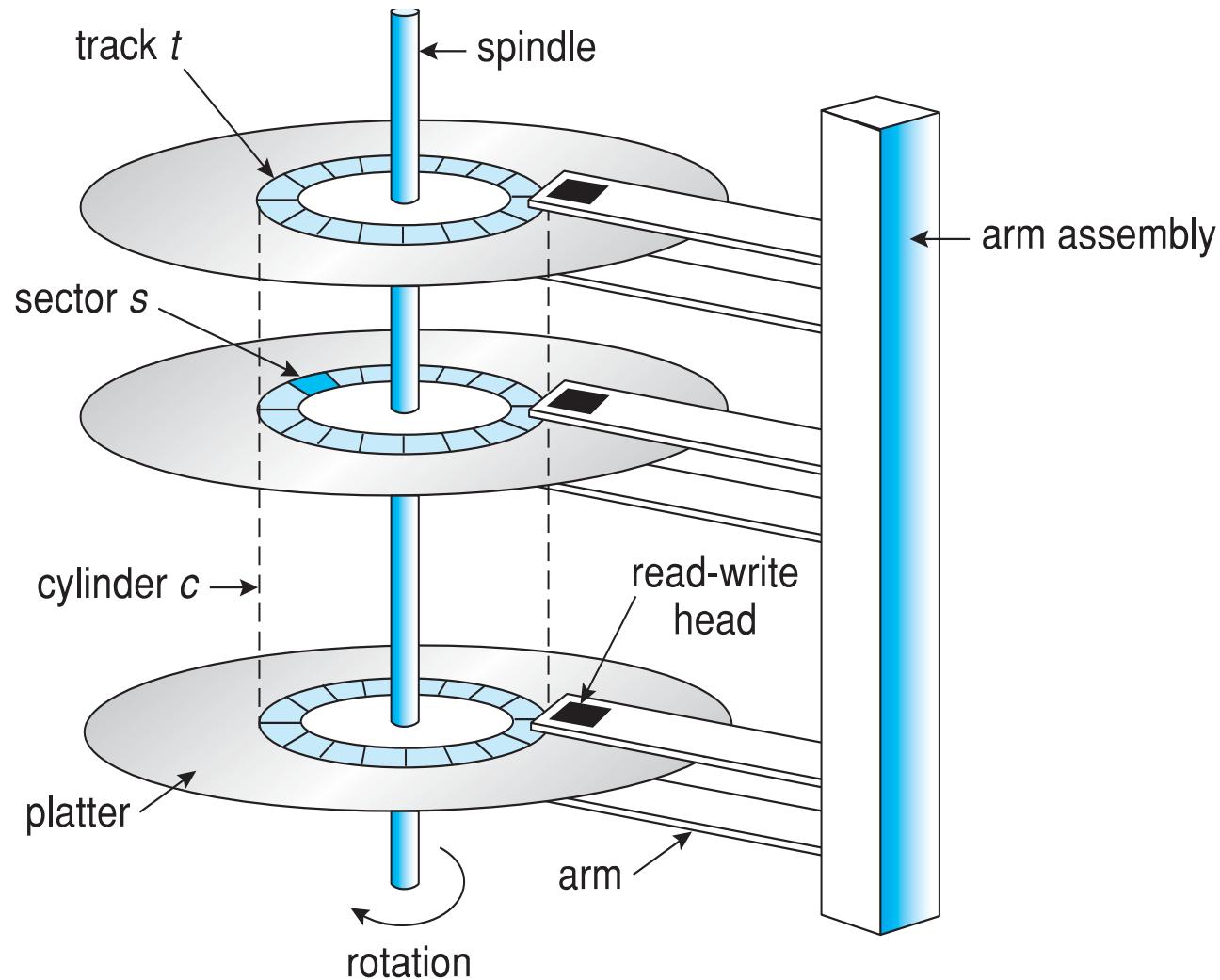
# Overview of Mass Storage Structure

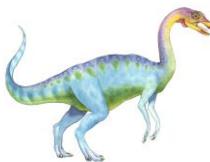
- Bulk of secondary storage for modern computers is **hard disk drives (HDDs)** and **nonvolatile memory (NVM)** devices
- **HDDs** spin platters of magnetically-coated material under moving read-write heads
  - Drives rotate at 60 to 250 times per second
  - **Transfer rate** is rate at which data flow between drive and computer
  - **Positioning time (random-access time)** is time to move disk arm to desired cylinder (**seek time**) and time for desired sector to rotate under the disk head (**rotational latency**)
  - **Head crash** results from disk head making contact with the disk surface -- That's bad
- Disks can be removable





# Moving-head Disk Mechanism

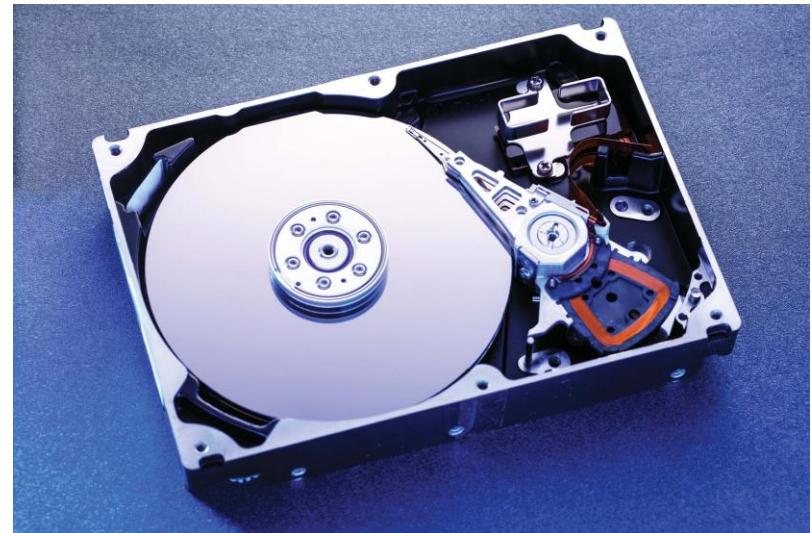


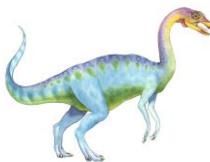


# Hard Disk Drives

---

- Platters range from .85" to 14" (historically)
  - Commonly 3.5", 2.5", and 1.8"
- Range from 30GB to 3TB per drive

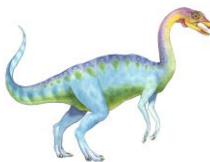




## ■ Performance

- Transfer Rate – theoretical – 6 Gb/sec
  - ▶ Effective Transfer Rate – real – 1Gb/sec
- Seek time from 3ms to 12ms – 9ms common for desktop drives
  - ▶ Average seek time measured or calculated based on 1/3 of tracks
- Latency based on spindle speed
  - ▶  $1 / (\text{RPM} / 60) = 60 / \text{RPM}$
  - ▶ Average latency =  $\frac{1}{2}$  latency

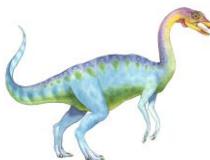




# Hard Disk Performance

- **Access Latency = Average access time** = average seek time + average latency
  - For fastest disk:  $3\text{ms} + 2\text{ms} = 5\text{ms}$
  - For slow disk:  $9\text{ms} + 5.56\text{ms} = 14.56\text{ms}$
- Average I/O time = average access time + (amount to transfer / transfer rate) + controller overhead
  - For example, to transfer a 4KB block on a **7200 RPM** disk with a 5ms average seek time, 1Gb/sec transfer rate with a .1ms controller overhead,  
average I/O time =  $5\text{ms} + 4.17\text{ms} + 0.1\text{ms} + \text{transfer time}$
  - Transfer time =  $4\text{KB} / 1\text{Gb/s} * 8\text{Gb / GB} * 1\text{GB} / 1024^2\text{KB} = 32 / (1024^2) = 0.031 \text{ ms}$
  - Average I/O time for 4KB block =  $9.27\text{ms} + .031\text{ms} = 9.301\text{ms}$





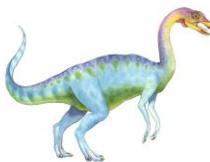
# The First Commercial Disk Drive



1956  
IBM RAMDAC  
computer included  
the IBM Model 350  
disk storage system

5M (7 bit) characters  
50 x 24" platters  
Access time = < 1  
second



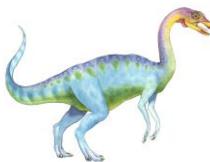


# Nonvolatile Memory Devices

---

- If disk-drive like, then called **solid-state disks (SSDs)**
  - Other forms include **USB drives** (thumb drive, flash drive), DRAM disk replacements, surface-mounted on motherboards, and main storage in devices like smartphones
- Can be more reliable than HDDs
- More expensive per MB
- Maybe have shorter life span – need careful management
- Less capacity
- But much faster
  - Buses can be too slow -> connect directly to PCI for example
- No moving parts, so no seek time or rotational latency

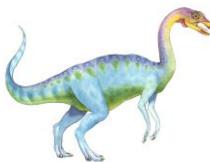




# Nonvolatile Memory Devices

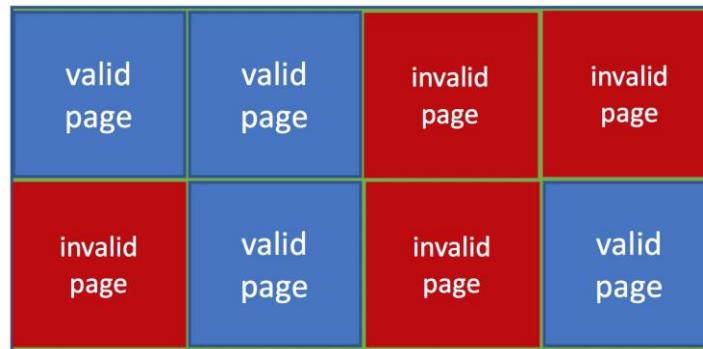
- Have characteristics that present challenges
- Read and written in “page” increments (think sector) but **can’t overwrite** in place
  - Must first be erased, and erases happen in larger “block” increments
  - Can only be erased a limited number of times before worn out – ~ 100,000
  - Life span measured in **drive writes per day (DWPD)**
    - ▶ A 1TB NAND drive with rating of 5DWPD is expected to have 5TB per day written within warranty period without failing





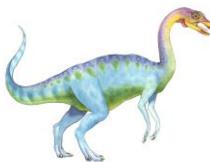
# NAND Flash Controller Algorithms

- With no overwrite, pages end up with mix of valid and invalid data
- To track which logical blocks are valid, controller maintains **flash translation layer (FTL)** table
- Also implements **garbage collection** to free invalid page space
- Allocates **overprovisioning** to provide working space for GC
- Each cell has lifespan, so **wear leveling** needed to write equally to all cells



NAND block with valid and invalid pages

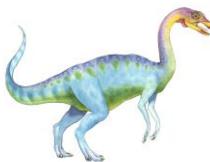




# Volatile Memory

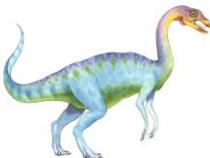
- DRAM frequently used as mass-storage device
  - Not technically secondary storage because volatile, but can have file systems, be used like very fast secondary storage
- **RAM drives** (with many names, including RAM disks) present as raw block devices, commonly file system formatted
- Used as high speed temporary storage
  - Programs could share bulk data, quickly, by reading/writing to RAM drive





- Computers have buffering, caching via RAM, so why RAM drives?
  - Caches / buffers allocated / managed by programmer, operating system, hardware
  - RAM drives under **user** control
  - Found in all major operating systems
    - ▶ Linux `/dev/ram`
    - ▶ macOS `diskutil` to create them
    - ▶ Linux `/tmp` of file system type `tmpfs`





# Magnetic Tape

**Magnetic tape** was used as an early secondary-storage medium. Although it is nonvolatile and can hold large quantities of data, its access time is slow compared with that of main memory and drives. In addition, random access to magnetic tape is about a thousand times slower than random access to HDDs and about a hundred thousand times slower than random access to SSDs so tapes are not very useful for secondary storage. Tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another.

A tape is kept in a spool and is wound or rewound past a read–write head. Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can read and write data at speeds comparable to HDDs. Tape capacities vary greatly, depending on the particular kind of tape drive, with current capacities exceeding several terabytes. Some tapes have built-in compression that can more than double the effective storage. Tapes and their drivers are usually categorized by width, including 4, 8, and 19 millimeters and 1/4 and 1/2 inch. Some are named according to technology, such as LTO-6 (Figure 11.5) and SDLT.



**Figure 11.5** An LTO-6 Tape drive with tape cartridge inserted.





# Disk Attachment

---

- Host-attached storage accessed through I/O ports talking to **I/O buses**
  - Several busses available, including **advanced technology attachment (ATA)**, **serial ATA (SATA)**, **eSATA**, **serial attached SCSI (SAS)**, **universal serial bus (USB)**, and **fibre channel (FC)**
- Most common is SATA
- Because NVM much faster than HDD, new fast interface for NVM called **NVM express (NVMe)**, connecting directly to PCI bus





- Data transfers on a bus carried out by special electronic processors called **controllers** (or **host-bus adapters, HBAs**)
  - Host controller on the computer end of the bus, device controller on device end
  - Computer places command on host controller, using memory-mapped I/O ports
    - ▶ Host controller sends messages to device controller
    - ▶ Data transferred via DMA between device and computer DRAM



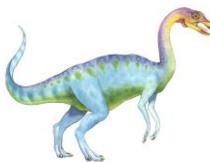


# HDD Scheduling

---

- The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth
- Minimize seek time
  - Seek time  $\approx$  seek distance
- Disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer

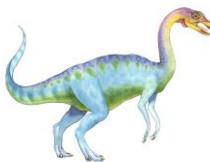




# Disk Scheduling (Cont.)

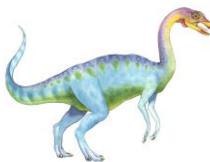
- There are many sources of disk I/O request
  - OS
  - System processes
  - Users processes
- I/O request includes input or output mode, disk address, memory address, number of sectors to transfer
- OS maintains queue of requests, per disk or device





- Idle disk can immediately work on I/O request, busy disk means work must queue
  - Optimization algorithms only make sense when a queue exists
- In the past, operating system responsible for queue management, disk drive head scheduling
  - Now, built into the storage device controllers
  - Just provide LBAs, handle sorting of requests
    - ▶ Some of the algorithms they use described next





# Disk Scheduling (Cont.)

---

- Note that drive controllers have small buffers and can manage a queue of I/O requests (of varying “depth”)
- Several algorithms exist to schedule the servicing of disk I/O requests
- The analysis is true for one or many platters
- We illustrate scheduling algorithms with a request queue (0-199)

98, 183, 37, 122, 14, 124, 65, 67

Head pointer 53



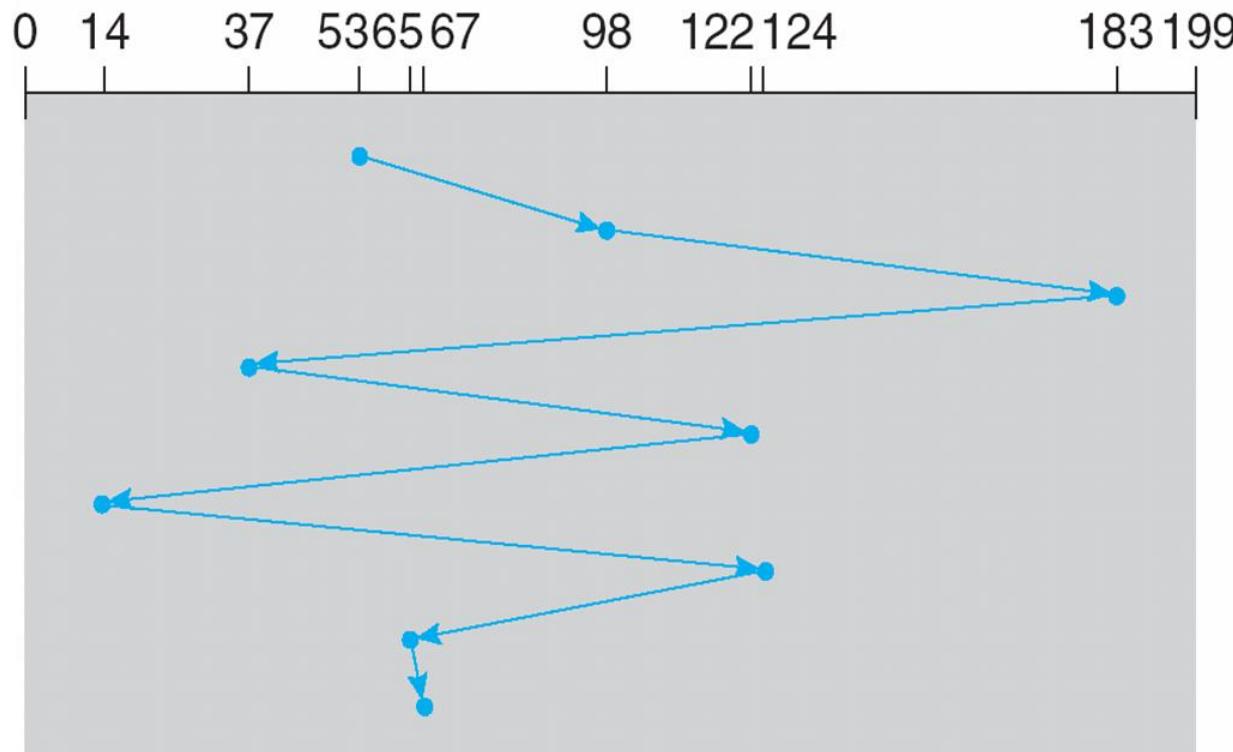


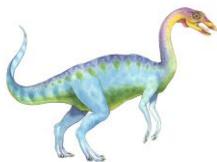
# FCFS

Illustration shows total head movement of 640 cylinders

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

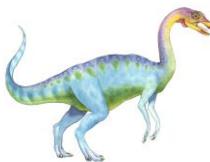




# SCAN

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues
- **SCAN algorithm** Sometimes called the **elevator algorithm**
- Illustration shows total head movement of 208 cylinders
- But note that if requests are uniformly dense, largest density at other end of disk and those wait the longest

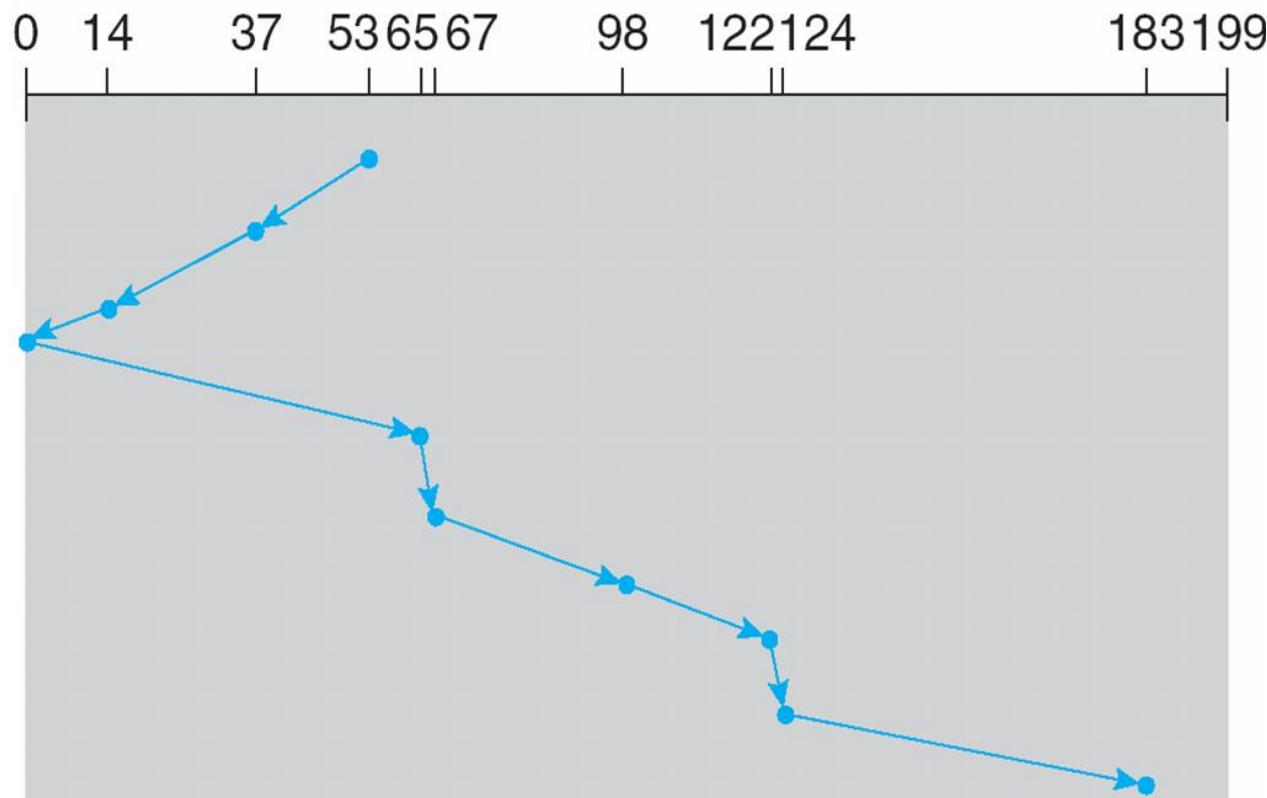


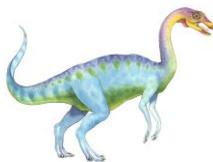


# SCAN (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



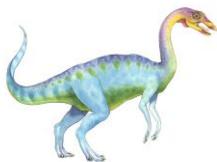


## C-SCAN

---

- Provides a more uniform wait time than SCAN
- The head moves from one end of the disk to the other, servicing requests as it goes
  - When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one
- Total number of cylinders?

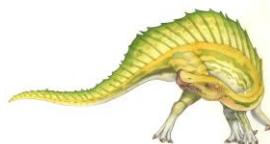
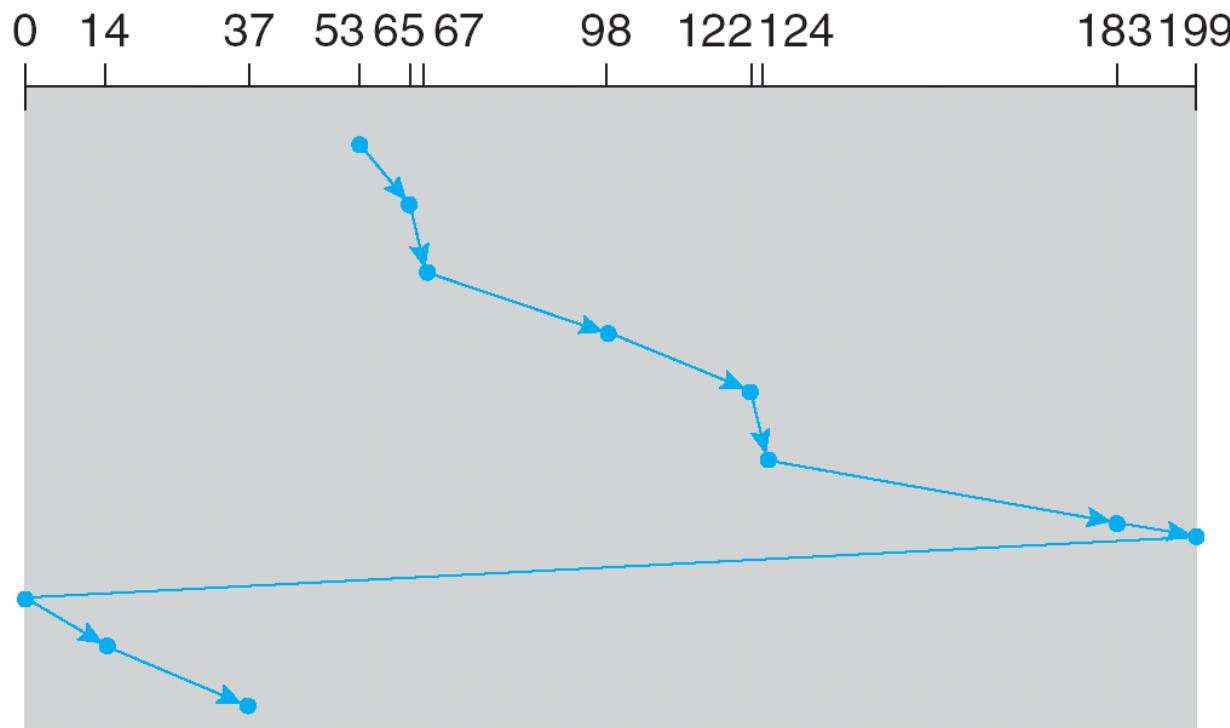


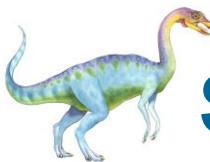


# C-SCAN (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53





# Selecting a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk
  - Less starvation, but still possible
- To avoid starvation Linux implements **deadline** scheduler
  - Maintains separate read and write queues, gives read priority
    - ▶ Because processes more likely to block on read than write





- Implements four queues: 2 x read and 2 x write
  - ▶ 1 read and 1 write queue sorted in LBA order, essentially implementing C-SCAN
  - ▶ 1 read and 1 write queue sorted in FCFS order
  - ▶ All I/O requests sent in batch sorted in that queue's order
  - ▶ After each batch, checks if any requests in FCFS older than configured age (default 500ms)
    - If so, LBA queue containing that request is selected for next batch of I/O
- In RHEL 7 also **NOOP** and **completely fair queueing** scheduler (**CFQ**) also available, defaults vary by storage device



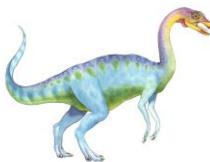


# NVM Scheduling

---

- No disk heads or rotational latency but still room for optimization
- In RHEL 7 **NOOP** (no scheduling) is used but adjacent LBA requests are combined
  - NVM best at random I/O, HDD at sequential
  - Throughput can be similar
  - **Input/Output operations per second (IOPS)** much higher with NVM (hundreds of thousands vs hundreds)
  - But **write amplification** (one write, causing garbage collection and many read/writes) can decrease the performance advantage

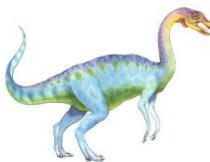




# Error Detection and Correction

- Fundamental aspect of many parts of computing (memory, networking, storage)
- **Error detection** determines if there a problem has occurred (for example a bit flipping)
  - If detected, can halt the operation
  - Detection frequently done via parity bit
- Parity one form of **checksum** – uses modular arithmetic to compute, store, compare values of fixed-length words
  - Another error-detection method common in networking is **cyclic redundancy check (CRC)** which uses hash function to detect multiple-bit errors
- **Error-correction code (ECC)** not only detects, but can correct some errors
  - Soft errors correctable, hard errors detected but not corrected

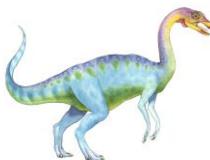




# Storage Device Management

- **Low-level formatting**, or **physical formatting** — Dividing a disk into sectors that the disk controller can read and write
  - Each sector can hold header information, plus data, plus error correction code (**ECC**)
  - Usually 512 bytes of data but can be selectable
- To use a disk to hold files, the operating system still needs to record its own data structures on the disk
  - **Partition** the disk into one or more groups of cylinders, each treated as a logical disk
  - **Logical formatting** or “making a file system”
  - To increase efficiency most file systems group blocks into **clusters**
    - ▶ Disk I/O done in blocks
    - ▶ File I/O done in clusters

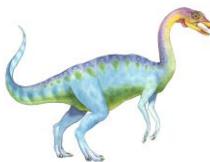




# Storage Device Management (cont.)

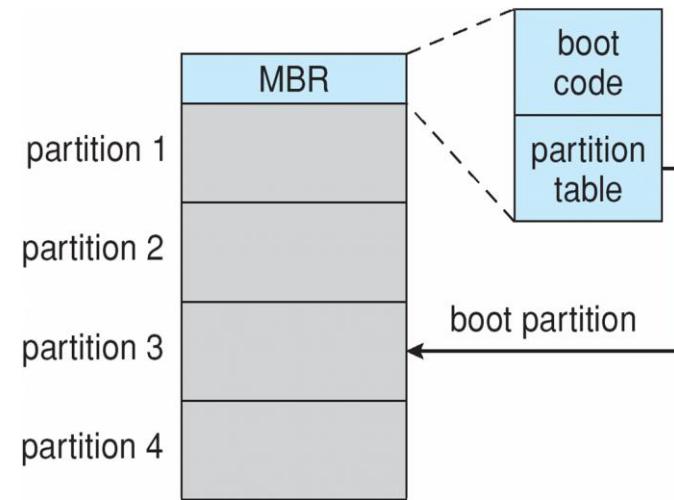
- **Root partition** contains the OS, other partitions can hold other Oses, other file systems, or be raw
  - **Mounted** at boot time
  - Other partitions can mount automatically or manually
- At mount time, file system consistency checked
  - Is all metadata correct?
    - ▶ If not, fix it, try again
    - ▶ If yes, add to mount table, allow access
- Boot block can point to boot volume or boot loader set of blocks that contain enough code to know how to load the kernel from the file system
  - Or a boot management program for multi-OS booting





# Storage Device Management (Cont.)

- Raw disk access for apps that want to do their own block management, keep OS out of the way (databases for example)
- Boot block initializes system
  - The bootstrap is stored in ROM, firmware
  - **Bootstrap loader** program stored in boot blocks of boot partition
- Methods such as **sector sparing** used to handle bad blocks



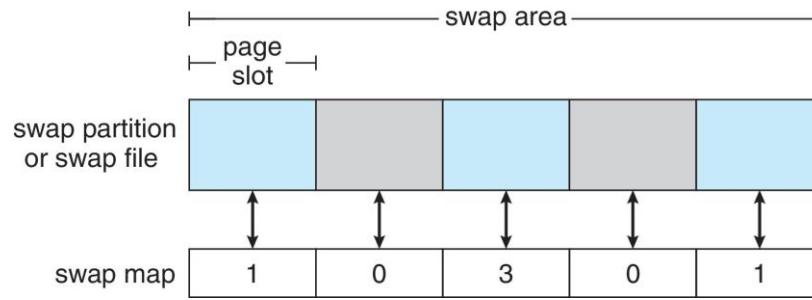
Booting from secondary storage in Windows





# Swap-Space Management

- Used for moving entire processes (swapping), or pages (paging), from DRAM to secondary storage when DRAM not large enough for all processes
- Operating system provides **swap space management**
  - Secondary storage slower than DRAM, so important to optimize performance
  - Usually multiple swap spaces possible – decreasing I/O load on any given device
  - Best to have dedicated devices
  - Can be in raw partition or a file within a file system (for convenience of adding)
  - Data structures for swapping on Linux systems:

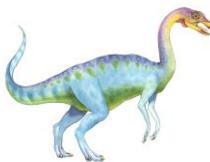




# Storage Attachment

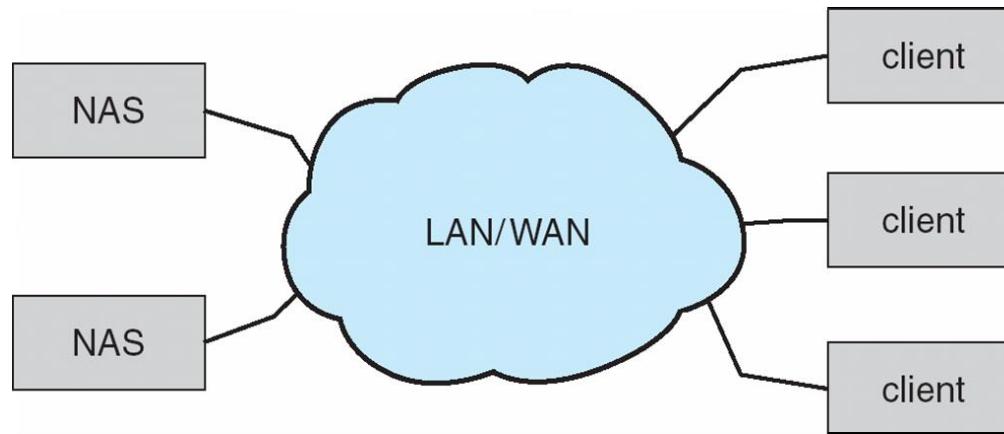
- Computers access storage in three ways
  - host-attached
  - network-attached
  - cloud
- Host attached access through local I/O ports, using one of several technologies
  - To attach many devices, use storage busses such as USB, firewire, thunderbolt
  - High-end systems use **fibre channel (FC)**
    - ▶ High-speed serial architecture using fibre or copper cables
    - ▶ Multiple hosts and storage devices can connect to the FC fabric

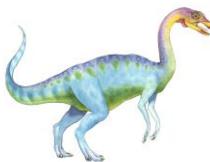




# Network-Attached Storage

- Network-attached storage (**NAS**) is storage made available over a network rather than over a local connection (such as a bus)
  - Remotely attaching to file systems
- NFS and CIFS are common protocols
- Implemented via remote procedure calls (RPCs) between host and storage over typically TCP or UDP on IP network
- **iSCSI** protocol uses IP network to carry the SCSI protocol
  - Remotely attaching to devices (blocks)





# Cloud Storage

- Similar to NAS, provides access to storage across a network
  - Unlike NAS, accessed over the Internet or a WAN to remote data center
- NAS presented as just another file system, while cloud storage is API based, with programs using the APIs to provide access
  - Examples include Dropbox, Amazon S3, Microsoft OneDrive, Apple iCloud
  - Use APIs because of latency and failure scenarios (NAS protocols wouldn't work well)





# Storage Array

---

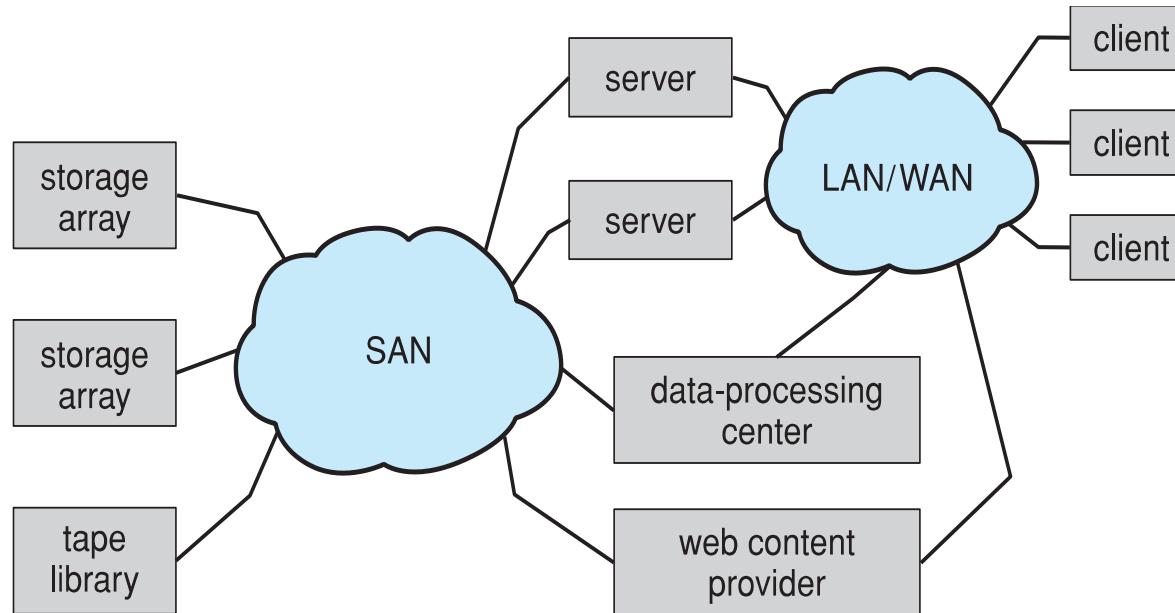
- Can just attach disks, or arrays of disks
- Avoids the NAS drawback of using network bandwidth
- Storage Array has controller(s), provides features to attached host(s)
  - Ports to connect hosts to array
  - Memory, controlling software (sometimes NVRAM, etc)
  - A few to thousands of disks
  - RAID, hot spares, hot swap (discussed later)
  - Shared storage -> more efficiency
  - Features found in some file systems
    - ▶ Snapshots, clones, thin provisioning, replication, deduplication, etc

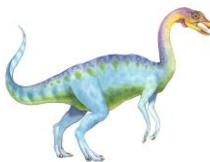




# Storage Area Network

- Common in large storage environments
- Multiple hosts attached to multiple storage arrays – flexible





# Storage Area Network (Cont.)

- SAN is one or more storage arrays
  - Connected to one or more Fibre Channel switches or **InfiniBand (IB)** network
- Hosts also attach to the switches
- Storage made available via **LUN Masking** from specific arrays to specific servers
- Easy to add or remove storage, add new host and allocate it storage
- Why have separate storage networks and communications networks?
  - Consider iSCSI, FCOE



A Storage Array





# RAID Structure

---

- RAID – redundant array of inexpensive disks
  - multiple disk drives provides reliability via redundancy
- Increases the mean time to failure
- Mean time to repair – exposure time when another failure could cause data loss
- Mean time to data loss based on above factors
- If mirrored disks fail independently, consider disk with 100,000 hour mean time to failure and 10 hour mean time to repair
  - Mean time to data loss is  $100,000^2 / (2 * 10) = 500 * 10^6$  hours, or 57,000 years!
- Frequently combined with NVRAM to improve write performance
- Several improvements in disk-use techniques involve the use of multiple disks working cooperatively



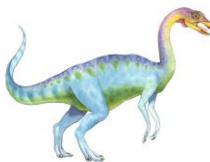


# RAID (Cont.)

---

- RAID is arranged into six different levels
- RAID schemes improve performance and improve the reliability of the storage system by storing redundant data
  - Disk **striping** uses a group of disks as one storage unit
  - **Mirroring** or **shadowing (RAID 1)** keeps duplicate of each disk
  - **Striped mirrors (RAID 1+0)** or **mirrored stripes (RAID 0+1)** provides high performance and high reliability
  - **Block interleaved parity (RAID 4, 5, 6)** uses much less redundancy
- RAID within a storage array can still fail if the array fails, so automatic **replication** of the data between arrays is common
- Frequently, a small number of **hot-spare** disks are left unallocated, automatically replacing a failed disk and having data rebuilt onto them

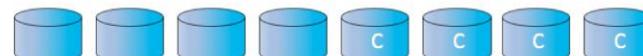




# RAID Levels



(a) RAID 0: non-redundant striping.



(b) RAID 1: mirrored disks.



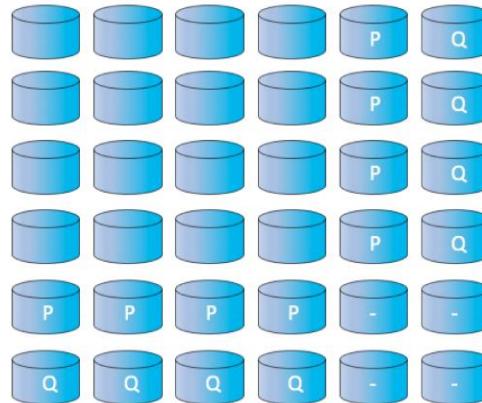
(c) RAID 4: block-interleaved parity.



(d) RAID 5: block-interleaved distributed parity.

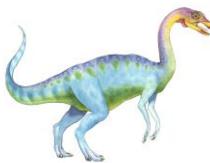


(e) RAID 6: P + Q redundancy.

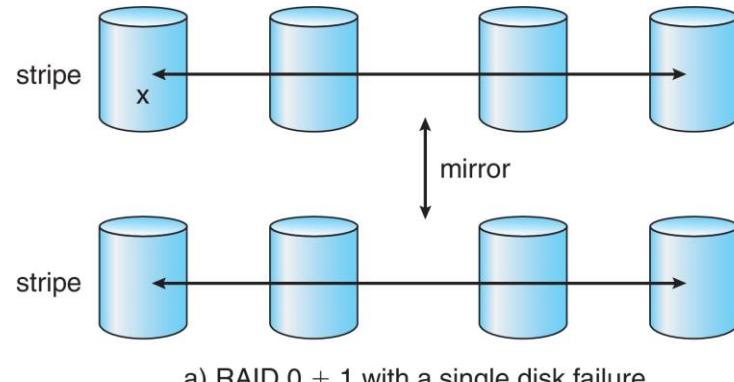


(f) Multidimensional RAID 6.

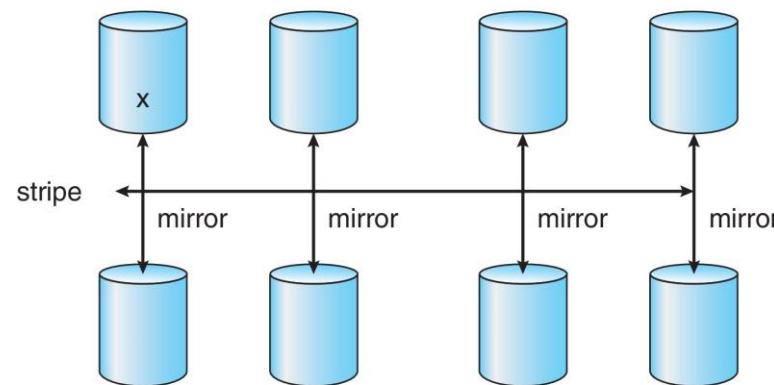




# RAID (0 + 1) and (1 + 0)



a) RAID 0 + 1 with a single disk failure.



b) RAID 1 + 0 with a single disk failure.

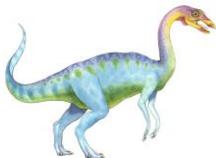




# Other Features

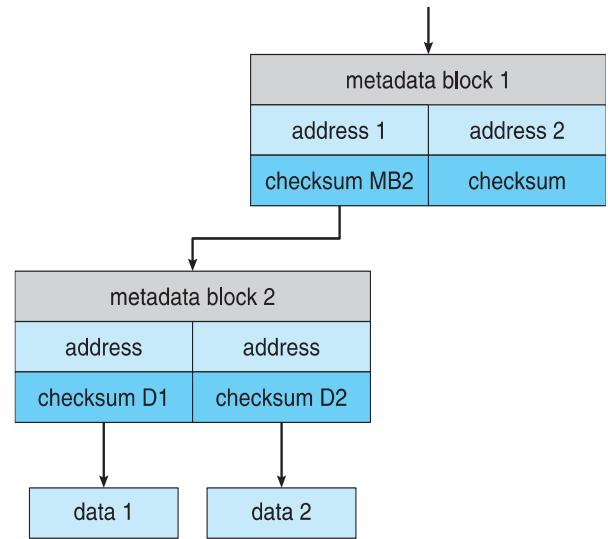
- Regardless of where RAID implemented, other useful features can be added
- **Snapshot** is a view of file system before a set of changes take place (i.e. at a point in time)
  - More in Ch. 12
- **Replication** is automatic duplication of writes between separate sites
  - For redundancy and disaster recovery
  - Can be synchronous or asynchronous
- Hot spare disk is unused, automatically used by RAID production if a disk fails to replace the failed disk and rebuild the RAID set if possible
  - Decreases mean time to repair



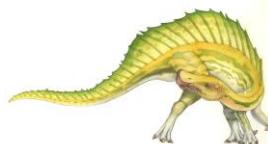


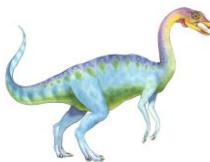
# Extensions

- RAID alone does not prevent or detect data corruption or other errors, just disk failures
- Solaris ZFS adds **checksums** of all data and metadata
- Checksums kept with pointer to object, to detect if object is the right one and whether it changed
- Can detect and correct data and metadata corruption
- ZFS also removes volumes, partitions
  - Disks allocated in **pools**
  - Filesystems with a pool share that pool, use and release space like `malloc()` and `free()` memory allocate / release calls

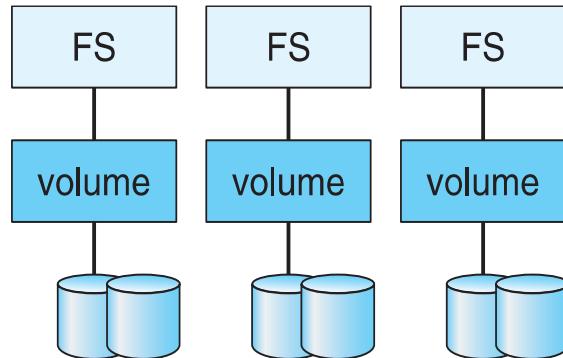


ZFS checksums all metadata and data

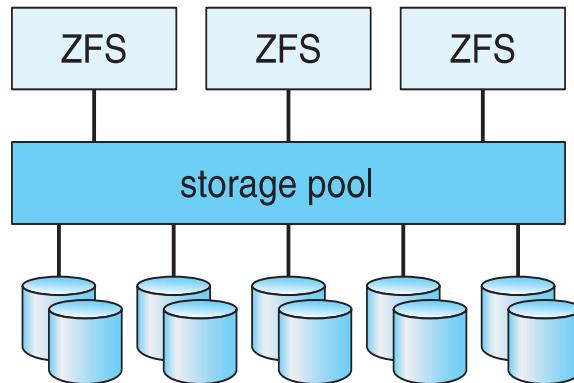




# Traditional and Pooled Storage



(a) Traditional volumes and file systems.



(b) ZFS and pooled storage.

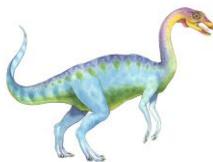




# Object Storage

- For general-purpose computing, file systems not sufficient for very large scale
- Another approach – start with a storage pool and place objects in it
  - Object just a container of **data**
  - No way to navigate the pool to find objects (no directory structures, few services)
  - Computer-oriented, not user-oriented
- Typical sequence
  - Create an object within the pool, receive an object ID
  - Access object via that ID
  - Delete object via that ID



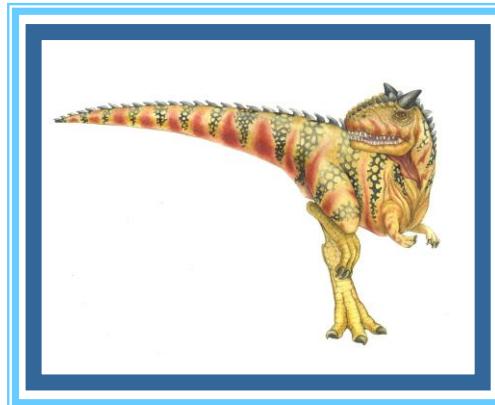


# Object Storage (Cont.)

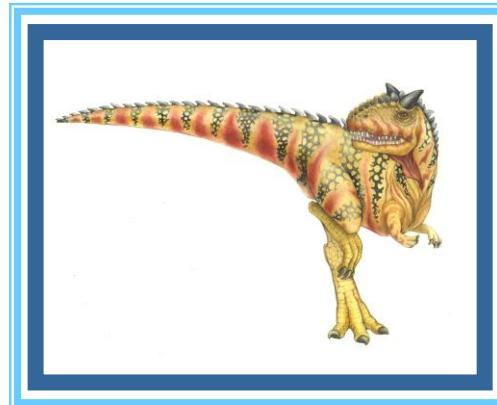
- Object storage management software like **Hadoop file system (HDFS)** and **Ceph** determine where to store objects, manages protection
  - Typically by storing N copies, across N systems, in the object storage cluster
  - **Horizontally scalable**
  - **Content addressable, unstructured**

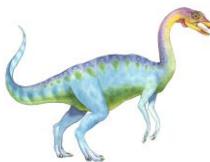


# End of Chapter 11



# Chapter 12: I/O Systems



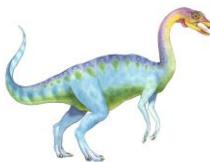


# Chapter 12: I/O Systems

---

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

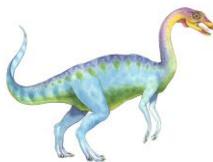




# Overview

- I/O management is a major component of OS 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, buses, device controllers connect to various devices
- **Device drivers** encapsulate device details
  - Present uniform device-access interface to I/O subsystem

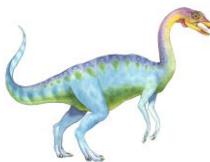




# I/O Hardware

- 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 (**PCle**)
    - ▶ **expansion bus** connects relatively slow devices
    - ▶ **Serial-attached SCSI (SAS)** common disk interface





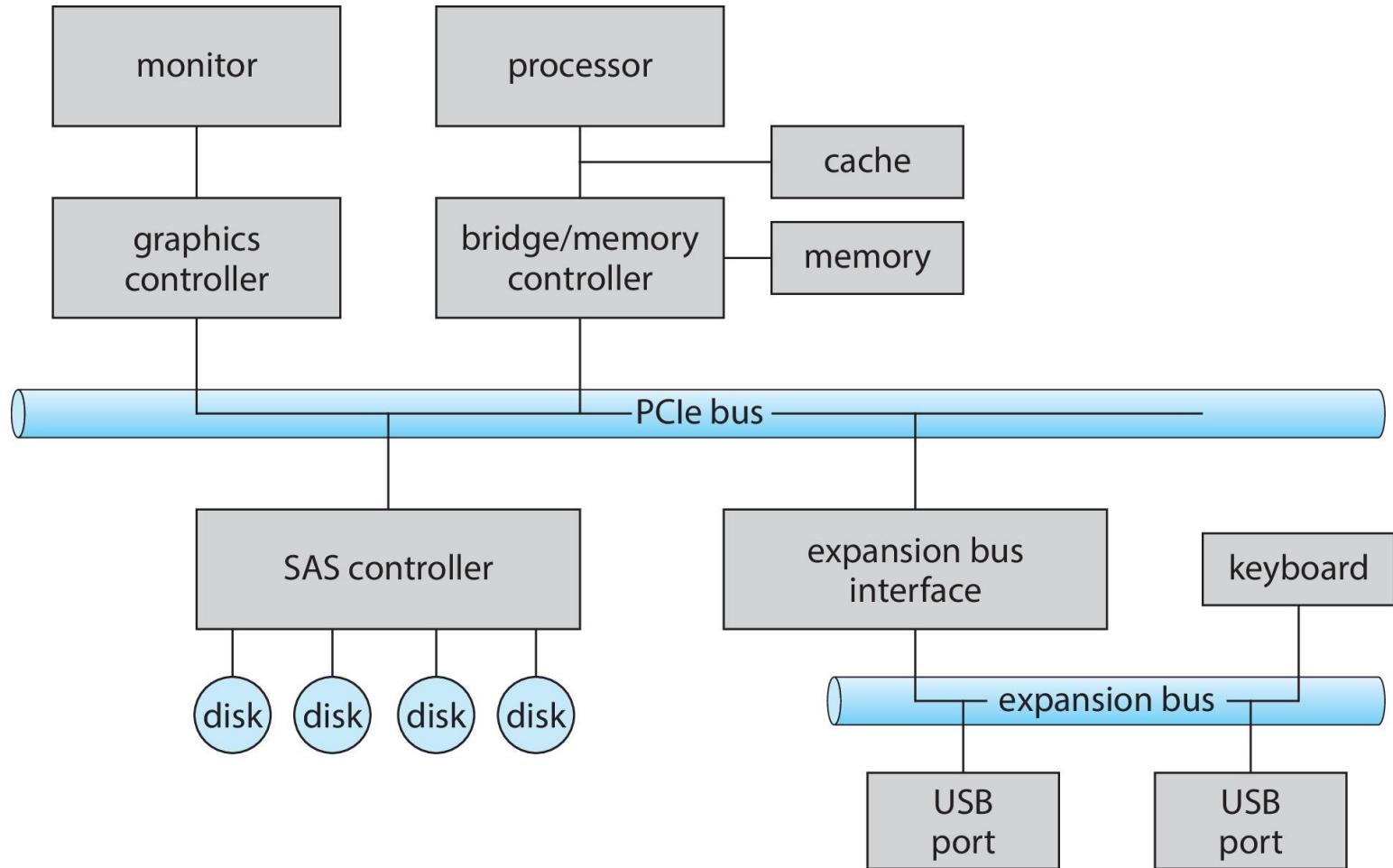
# I/O Hardware (Cont.)

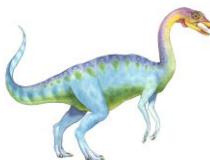
- **Controller (host adapter)** – electronics that operate port, bus, device
  - ▶ Sometimes integrated
  - ▶ Sometimes separate circuit board (host adapter)
  - ▶ Contains processor, microcode, private memory, bus controller, etc.
    - Some talk to per-device controller with bus controller, microcode, memory, etc.





# A Typical PC Bus Structure

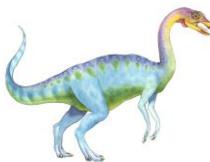




# I/O Hardware (Cont.)

- **Fibre channel (FC)** is complex controller, usually separate circuit board (**host-bus adapter, HBA**) plugging into bus
- I/O instructions control devices
- Devices usually have registers where device driver places commands, addresses, and data to write, or read data from registers after command execution
  - Data-in register, data-out register, status register, control register
  - Typically 1-4 bytes, or FIFO buffer





# 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)





# 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)





# 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



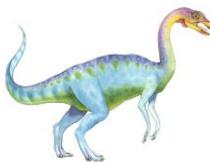


# Interrupts

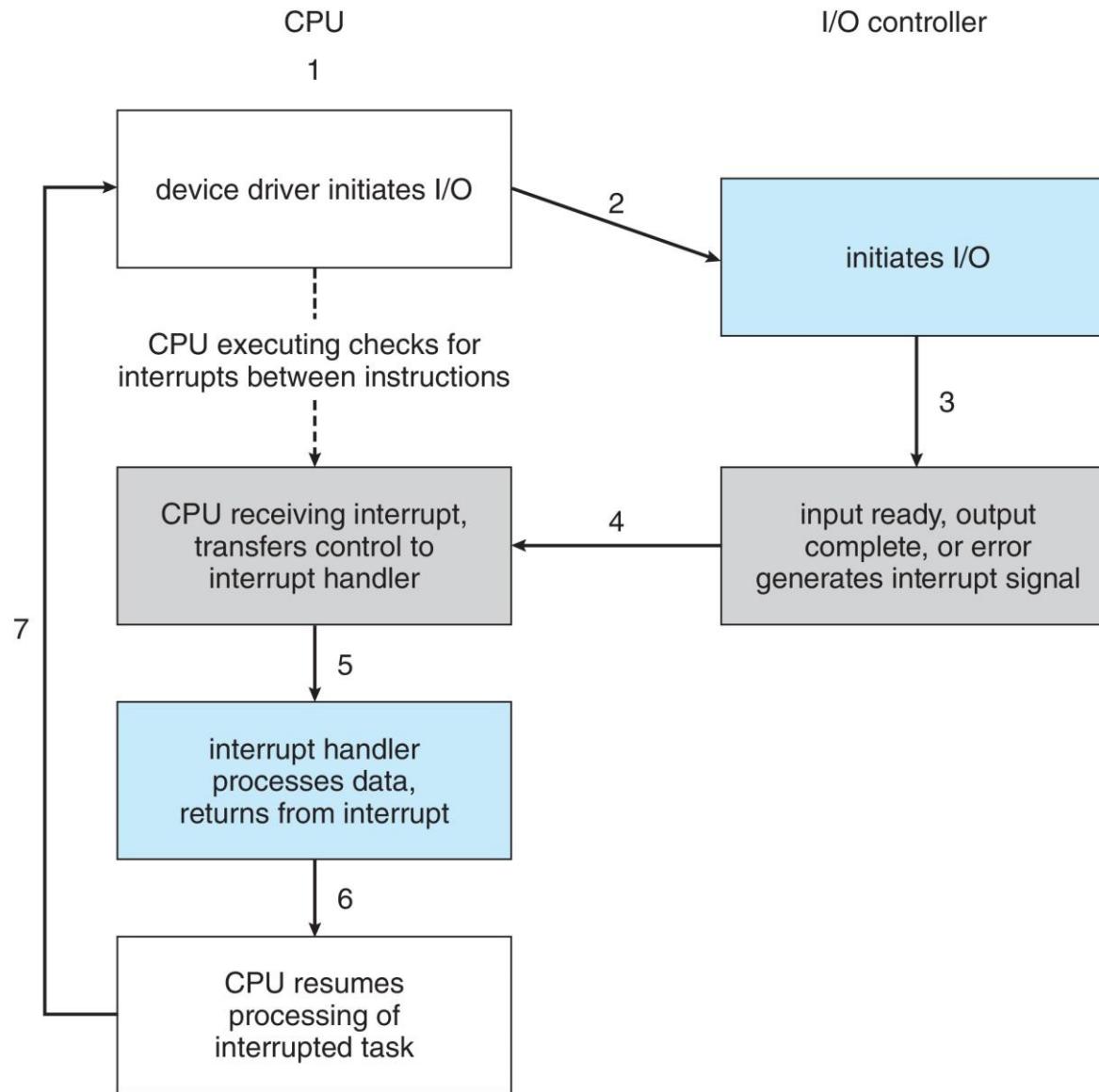
---

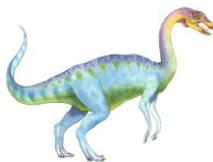
- Polling can happen in 3 instruction cycles
  - Read status, logical-and to extract status bit, branch if not zero
  - How to be more efficient if non-zero infrequently?
- CPU **Interrupt-request line** triggered by I/O device
  - Checked by processor after each instruction
- **Interrupt handler** receives interrupts
  - **Maskable** 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**
  - Interrupt chaining if more than one device at same interrupt number





# Interrupt-Driven I/O Cycle





# Interrupts (Cont.)

- 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 OS designed to handle it
- Used for time-sensitive processing, frequent, must be fast



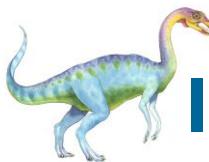


# Latency

- 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

	SCHEDULER	INTERRUPTS	0:00:10
-----			
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	

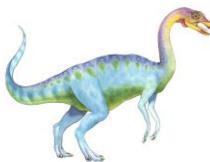




# Intel Pentium Processor Event-Vector Table

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



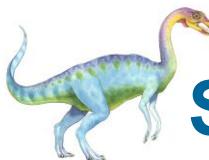


# Direct Memory Access

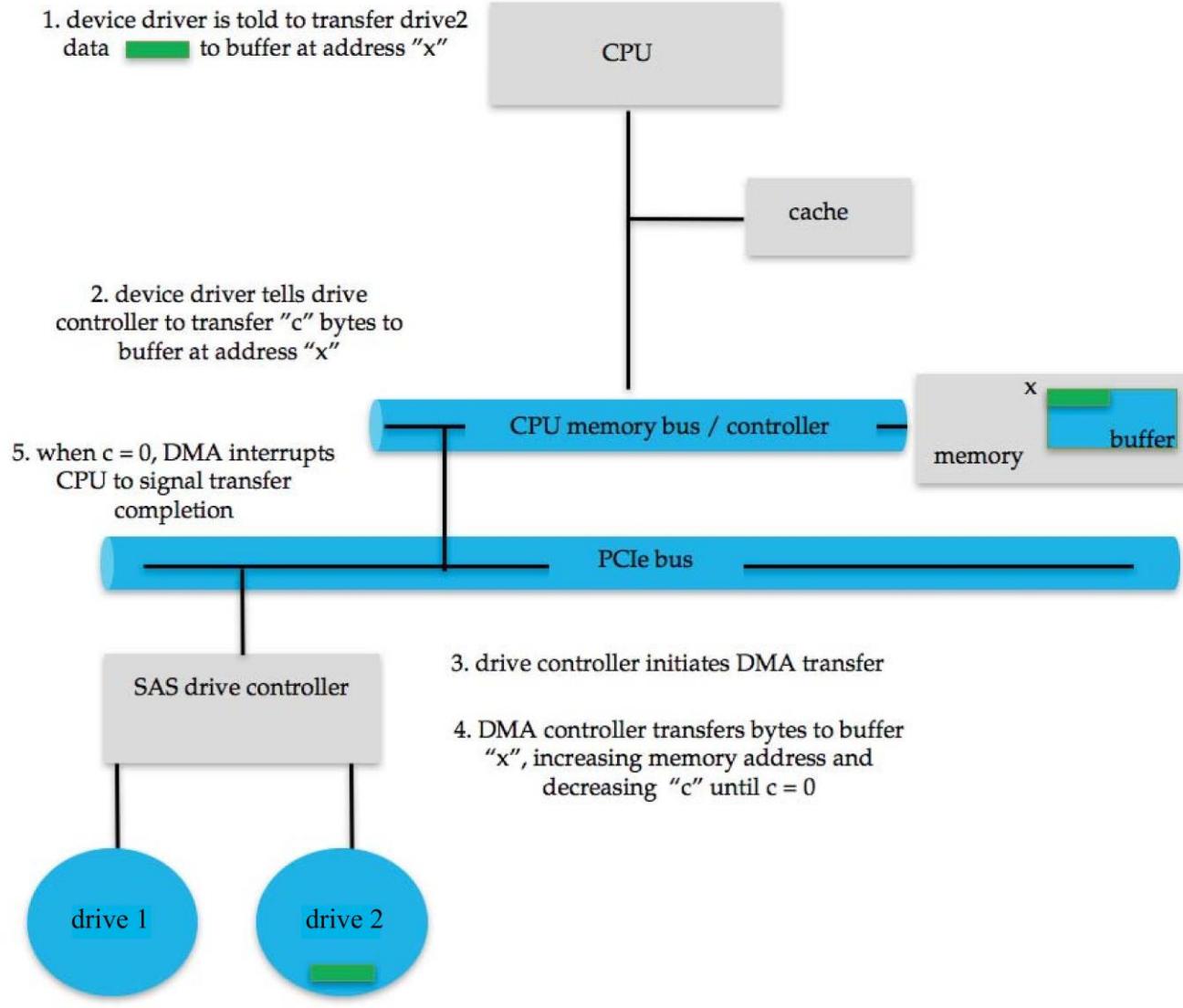
---

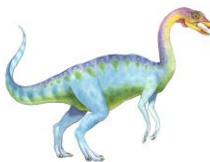
- Used to avoid **programmed I/O** (one byte at a time) for large data movement
- Requires **DMA** controller
- Bypasses CPU to transfer data directly between I/O device and memory
- OS writes DMA command block into memory
  - Source and destination addresses
  - Read or write mode
  - Count of bytes
  - 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**





# Six Step Process to Perform DMA Transfer

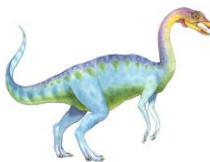




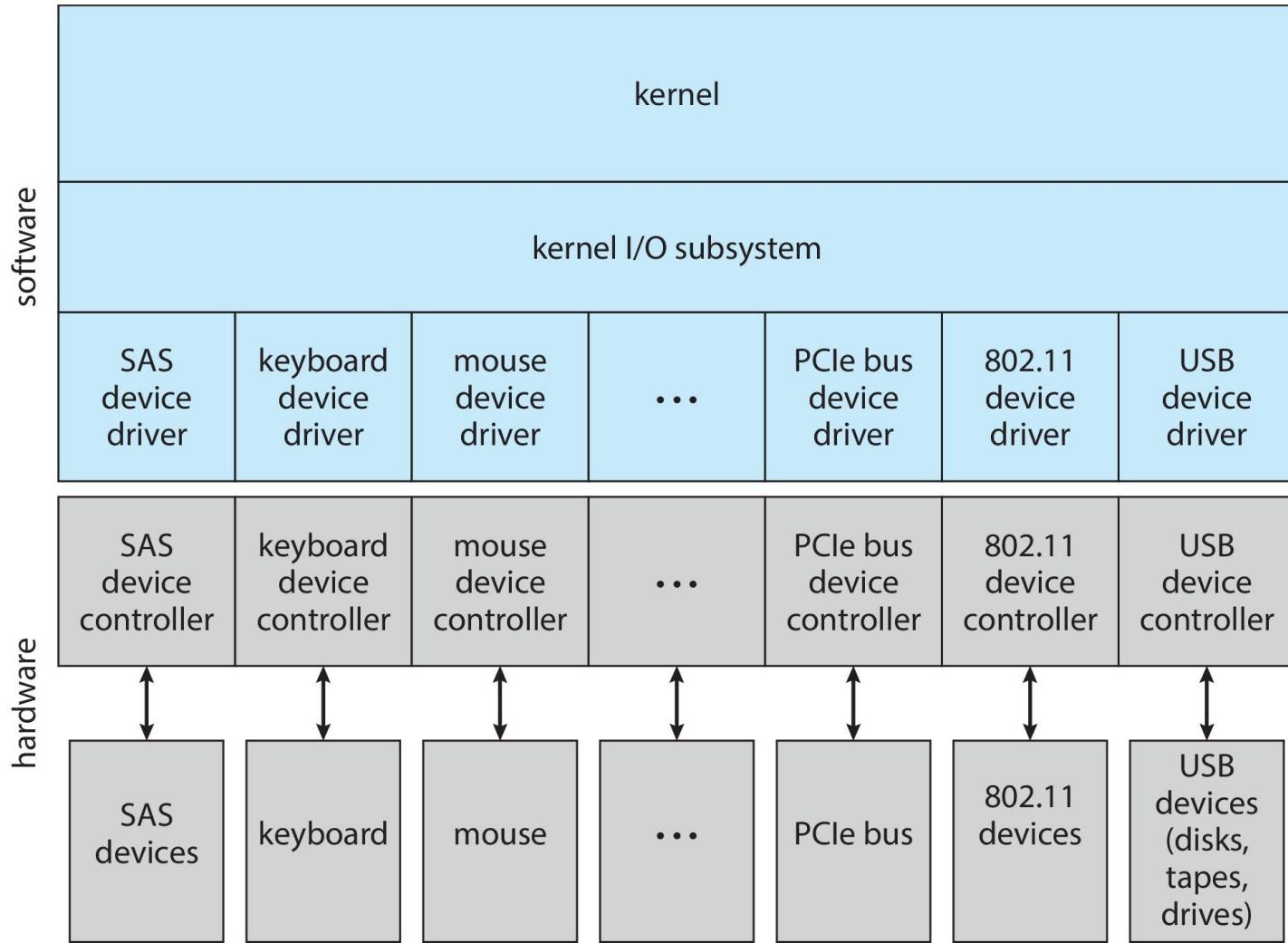
# Application I/O Interface

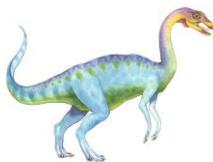
- I/O system calls encapsulate device behaviors in generic classes
- Device-driver layer hides differences among I/O controllers from kernel
- New devices talking already-implemented protocols need no extra work
- Each OS has its own I/O subsystem structures and device driver frameworks
- Devices vary 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

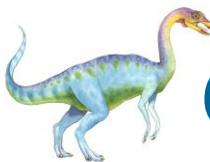




# Characteristics of I/O Devices

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





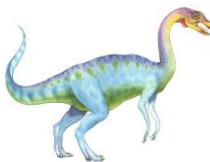
# Characteristics of I/O Devices (Cont.)

- 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
```

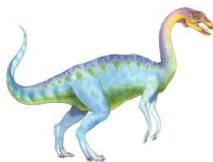




# Block and Character Devices

- 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



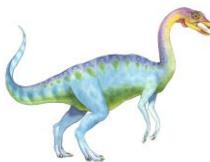


# Network Devices

---

- 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)



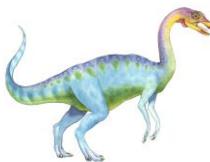


# Clocks and Timers

---

- 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

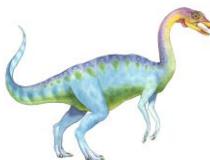




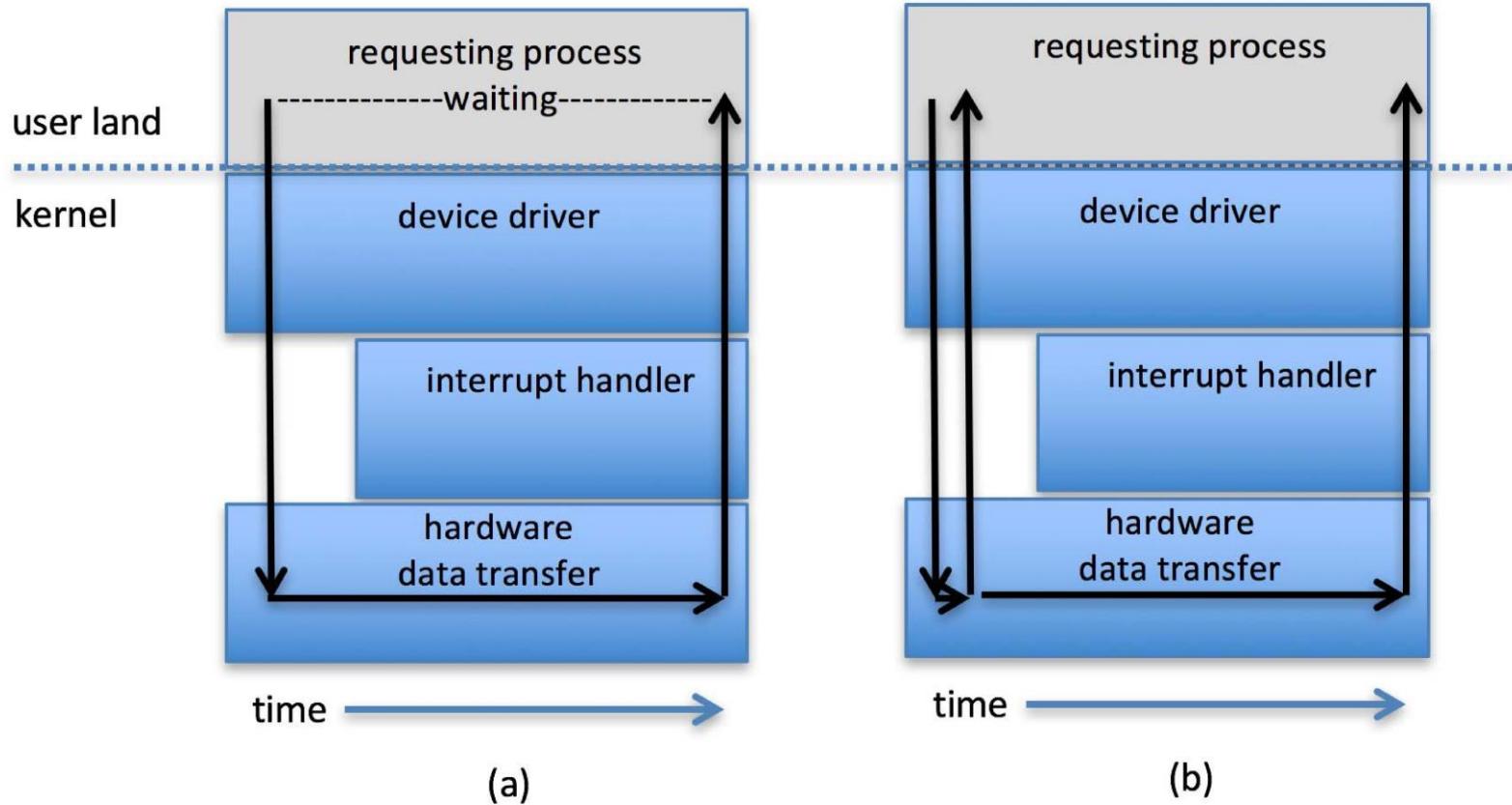
# Nonblocking and Asynchronous I/O

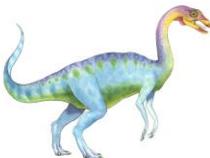
- **Blocking** - process suspended until I/O completed
  - Easy to use and understand
  - Insufficient for some needs
- **Nonblocking** - I/O call returns as much as available
  - User interface, data copy (buffered I/O)
  - Implemented via multi-threading
  - Returns quickly with count of bytes read 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



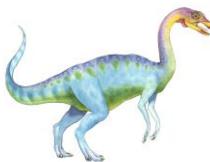


# Vectored I/O

---

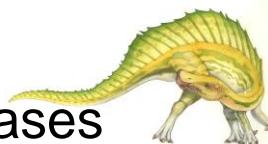
- **Vectored I/O** allows one system call to perform multiple I/O operations
- For example, Unix **readve()** accepts a vector of multiple buffers to read into or write from
- This scatter-gather method better than multiple individual I/O calls
  - Decreases context switching and system call overhead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

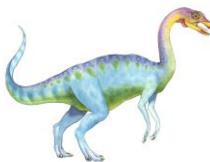




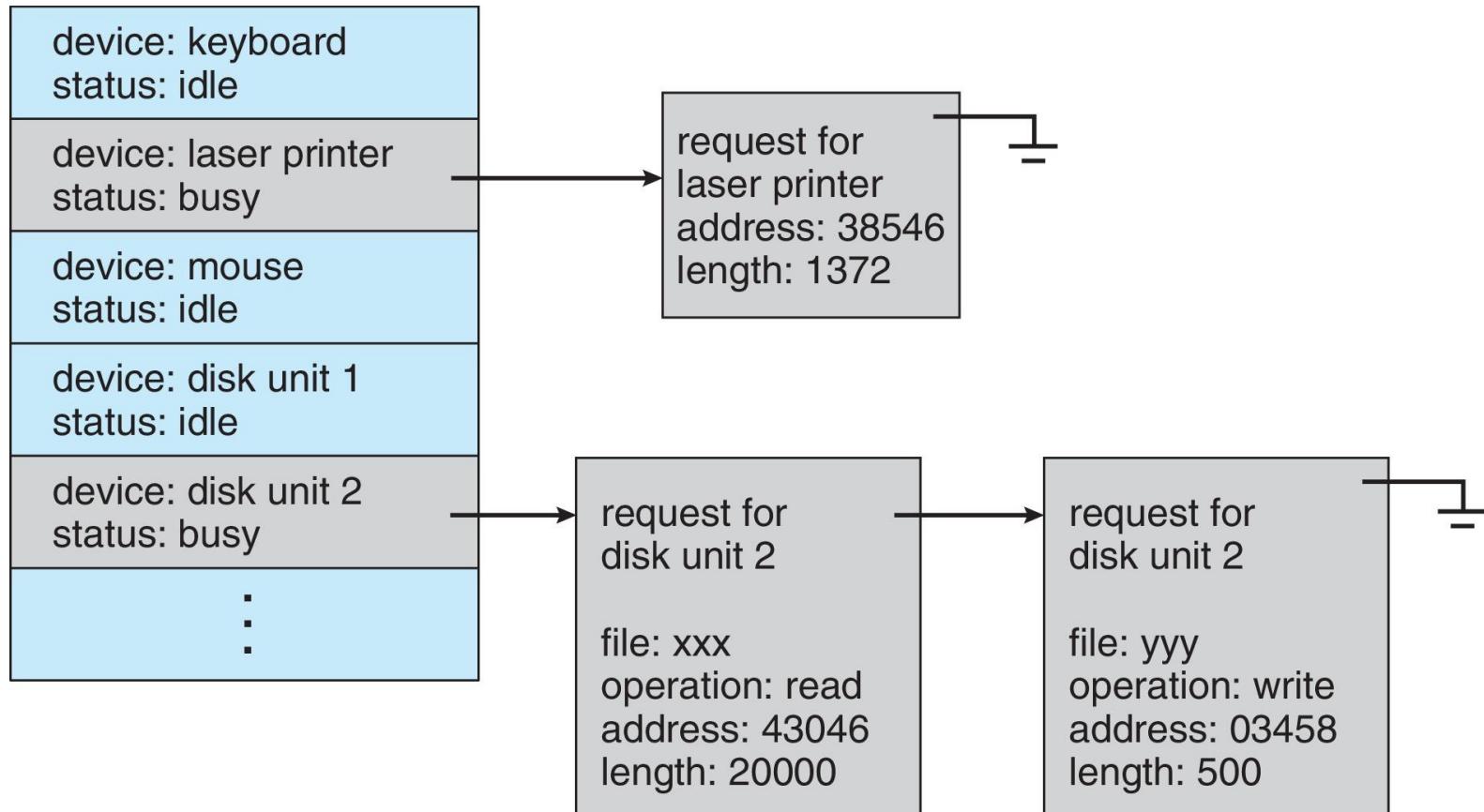
# Kernel I/O Subsystem

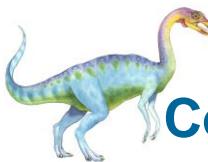
- Scheduling
  - Some I/O request ordering via per-device queue
  - Some OSs try fairness
  - Some implement Quality Of Service (i.e. IPQOS)
- **Buffering** - store data in memory while transferring between devices
  - To cope with device speed mismatch
  - To cope with device transfer size mismatch
  - 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



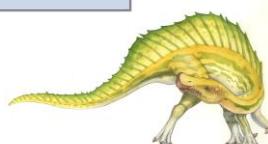
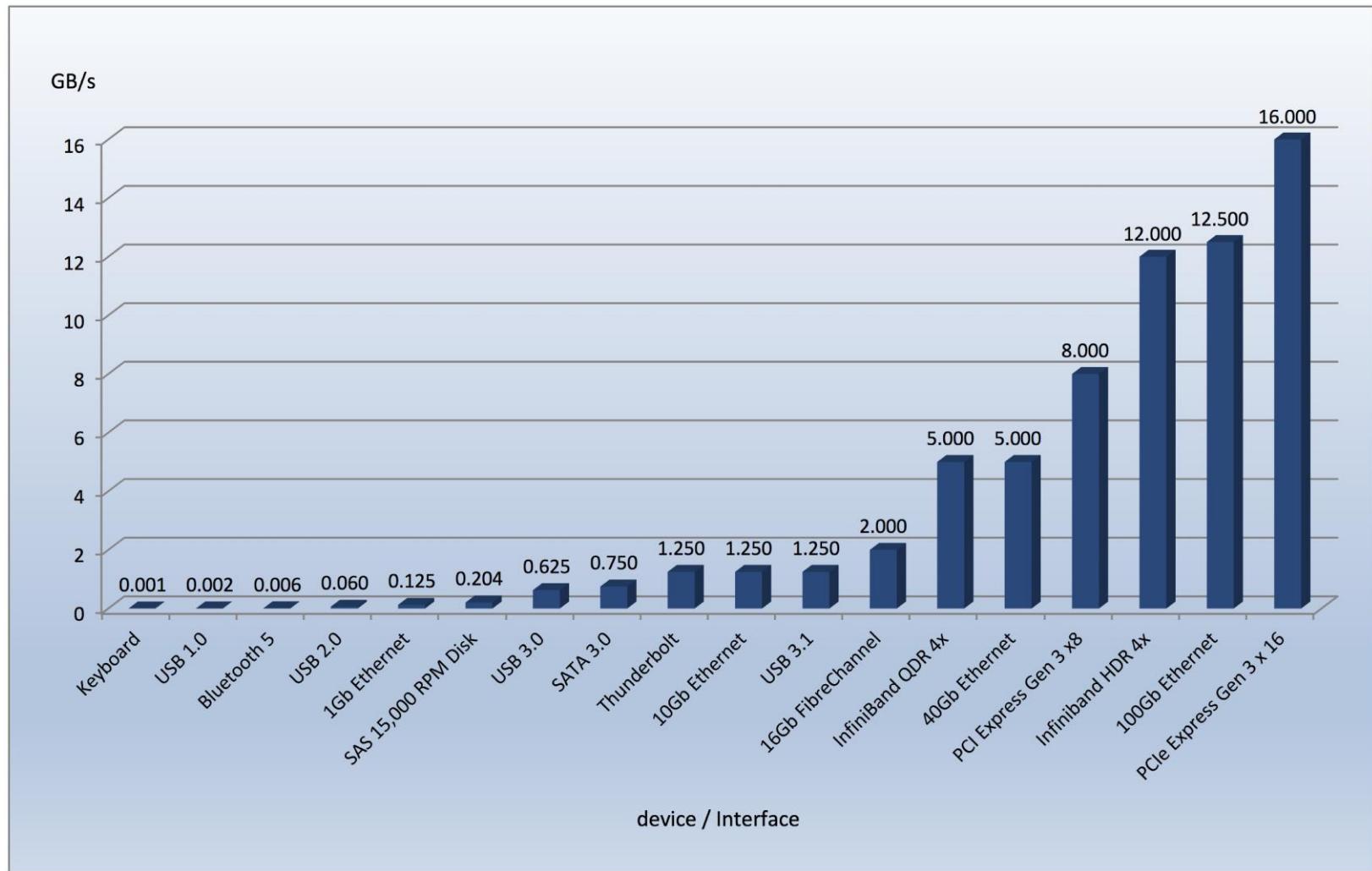


# Device-status Table





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

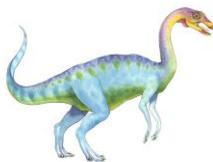




# Kernel I/O Subsystem

- **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

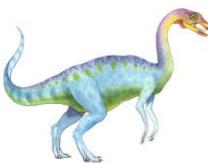




# Error Handling

- 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





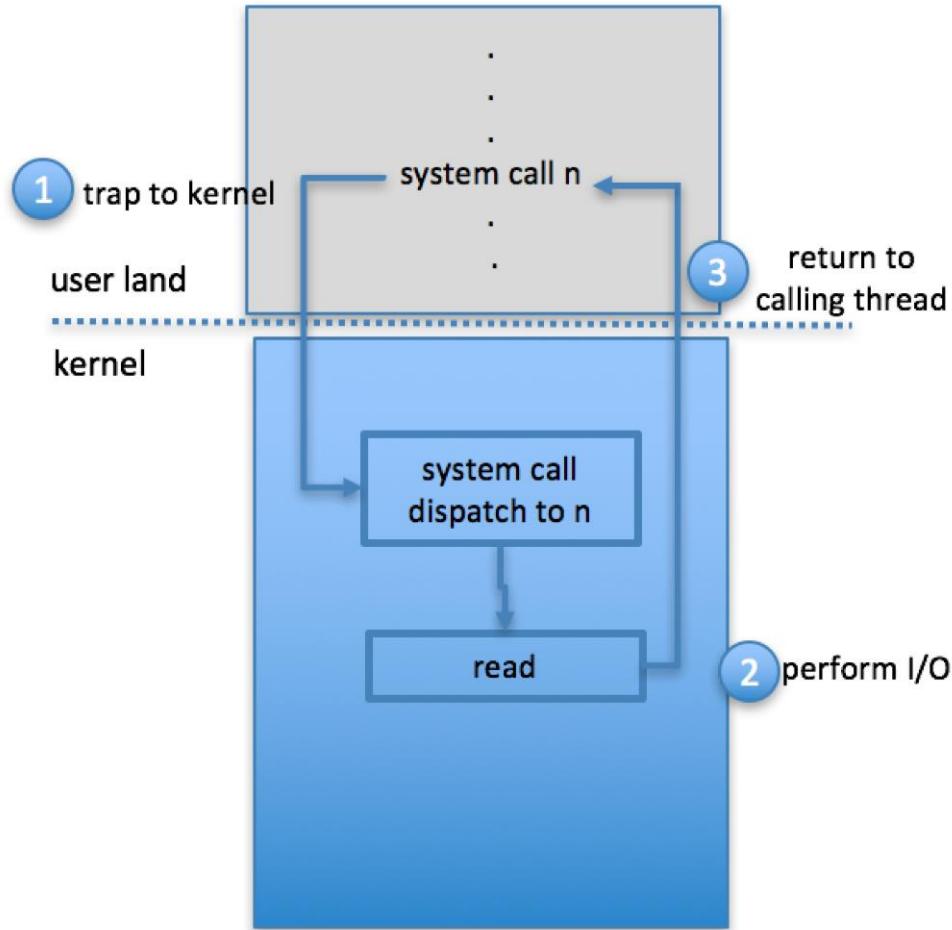
# I/O Protection

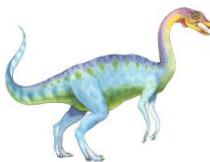
- User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions
  - All I/O instructions defined to be privileged
  - I/O must be performed via system calls
    - ▶ Memory-mapped and I/O port memory locations must be protected too





# Use of a System Call to Perform I/O

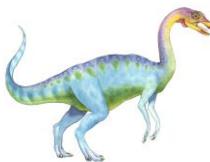




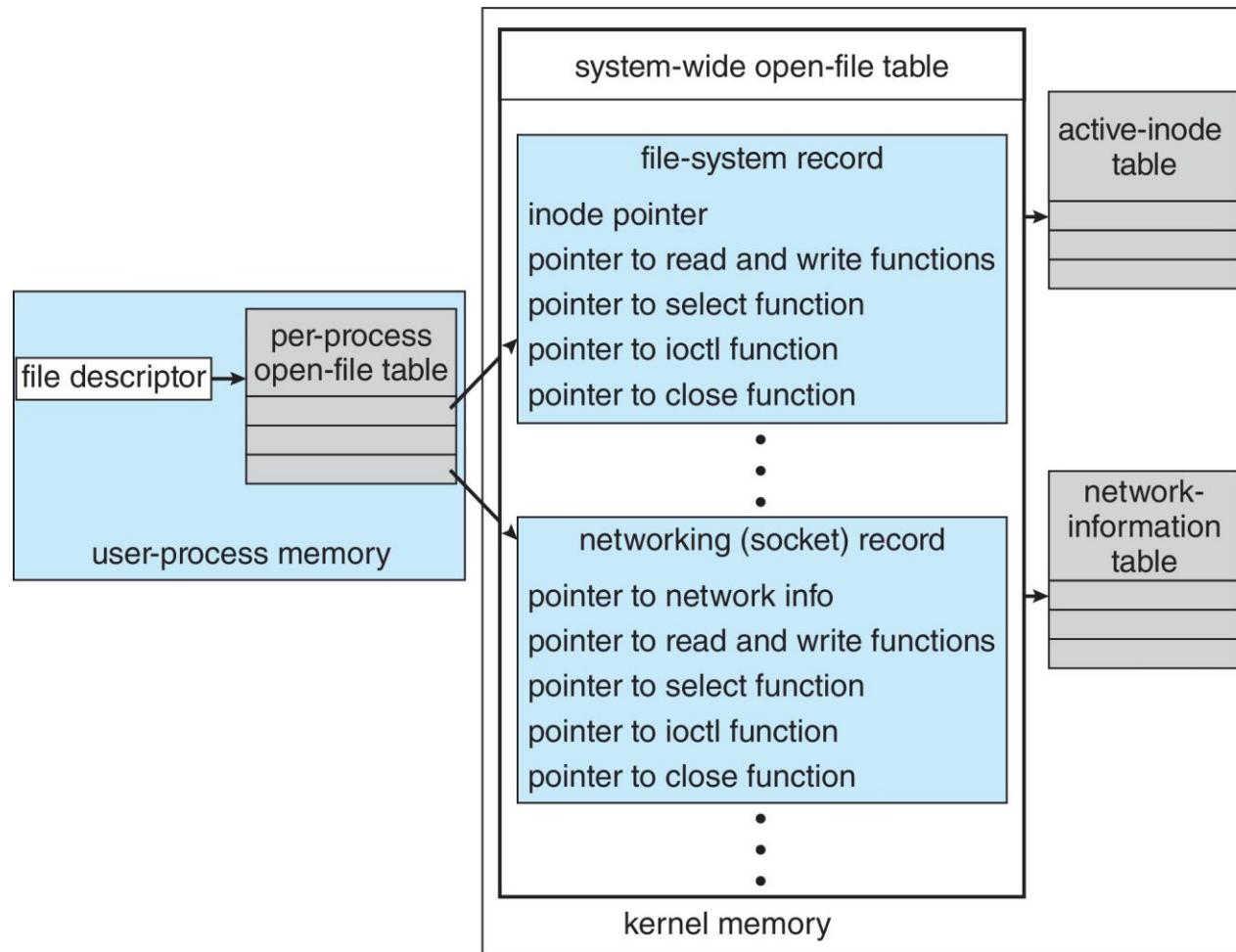
# Kernel Data Structures

- 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



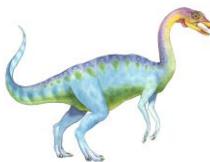


# Power Management

---

- 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

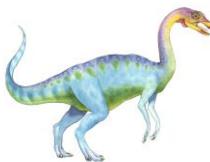




# Power Management (Cont.)

- 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

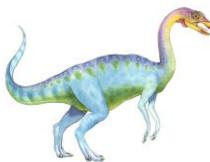




# Power Management (Cont.)

- For example, Android implements (Cont.)
  - 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)
- 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



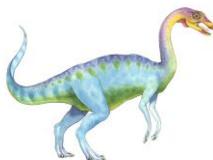


# Kernel I/O Subsystem Summary

---

- 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



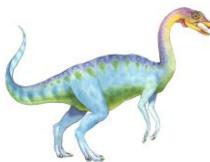


# Transforming I/O Requests to Hardware Operations

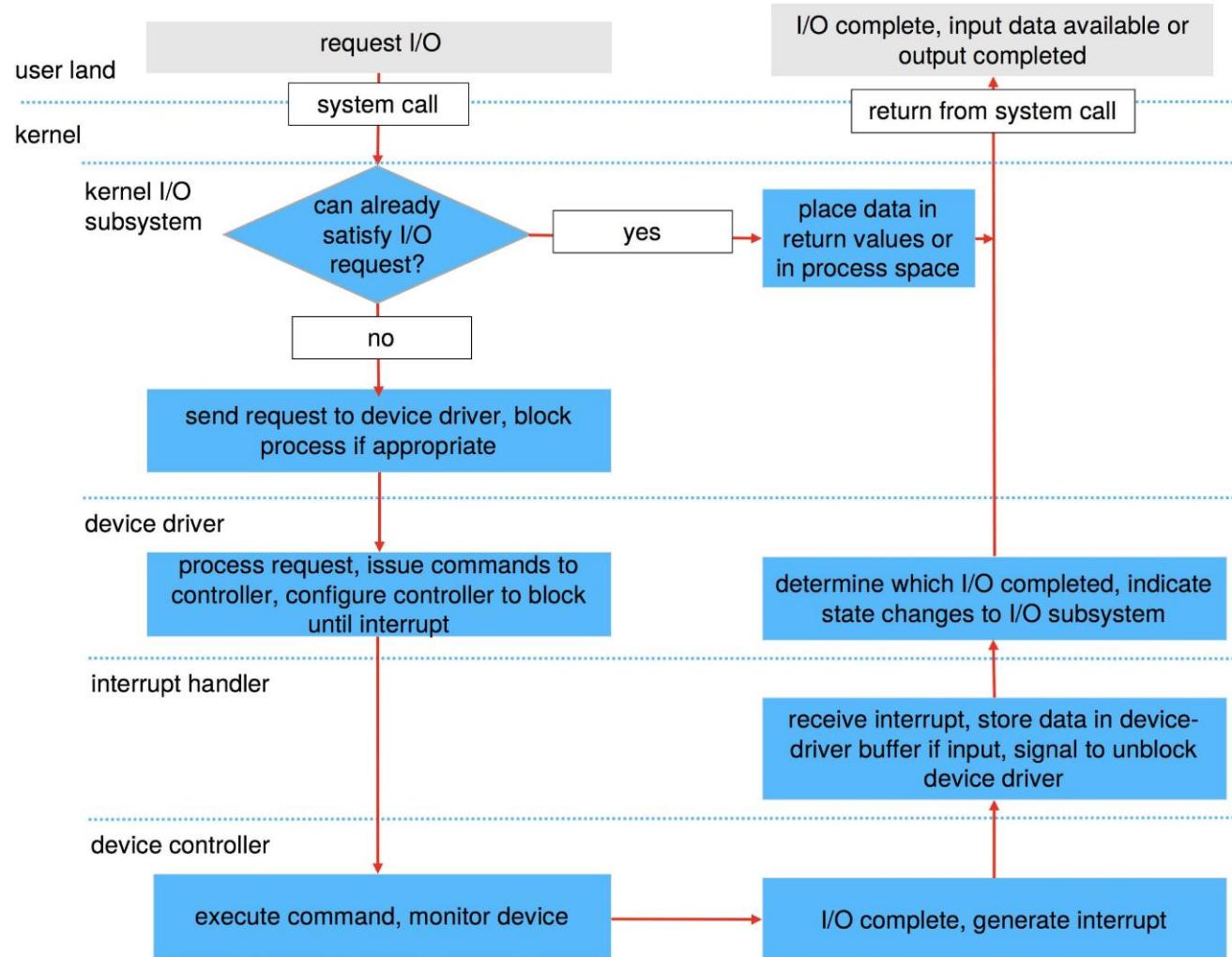
---

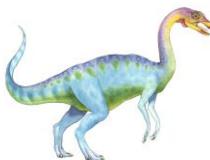
- Consider reading a file from disk for a process:
  - Determine device holding file
  - Translate name to device representation
  - Physically read data from disk into buffer
  - Make data available to requesting process
  - Return control to process





# Life Cycle of An I/O Request





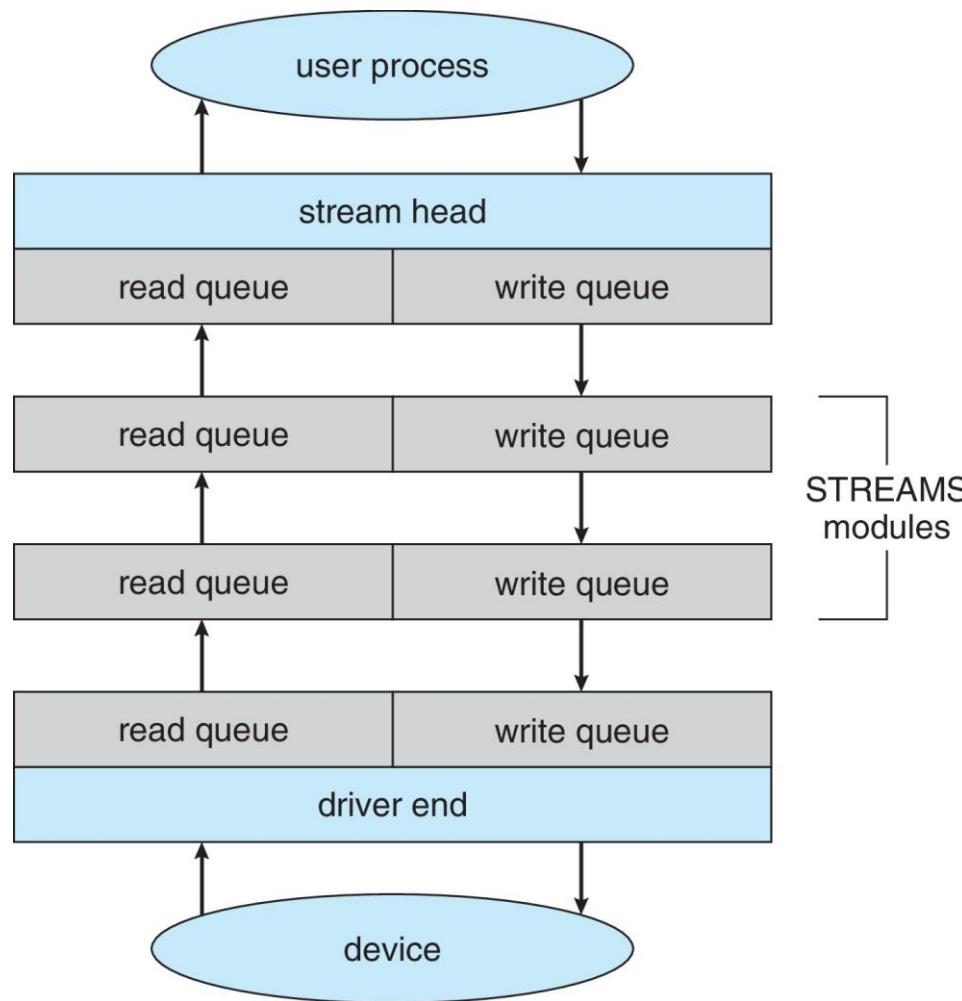
# STREAMS

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

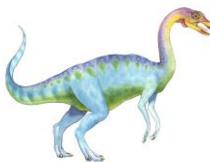




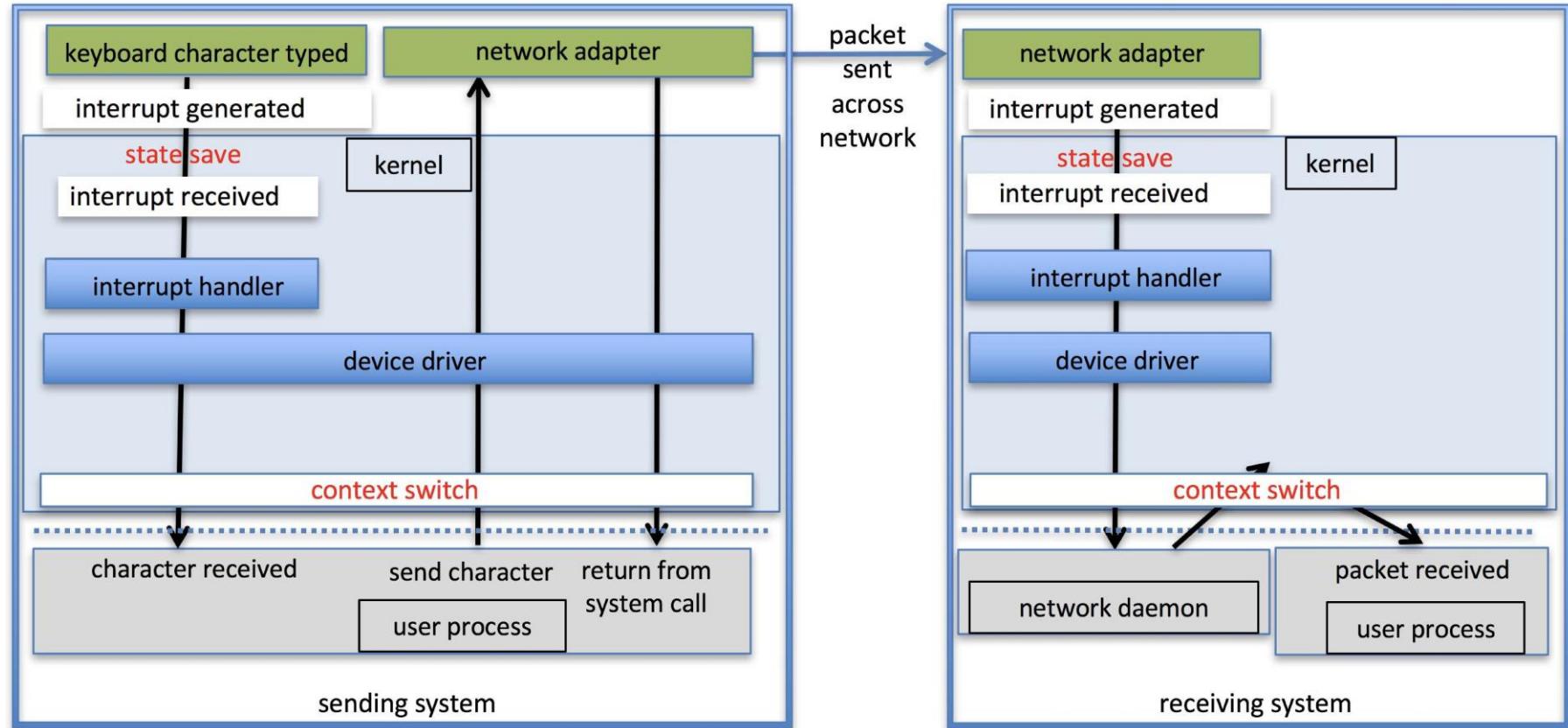
# Performance

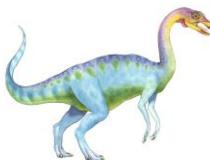
- I/O a major factor in system performance:
  - Demands CPU to execute device driver, kernel I/O code
  - Context switches due to interrupts
  - Data copying
  - Network traffic especially stressful





# Intercomputer Communications



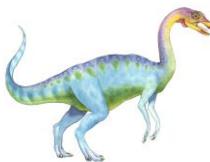


# Improving Performance

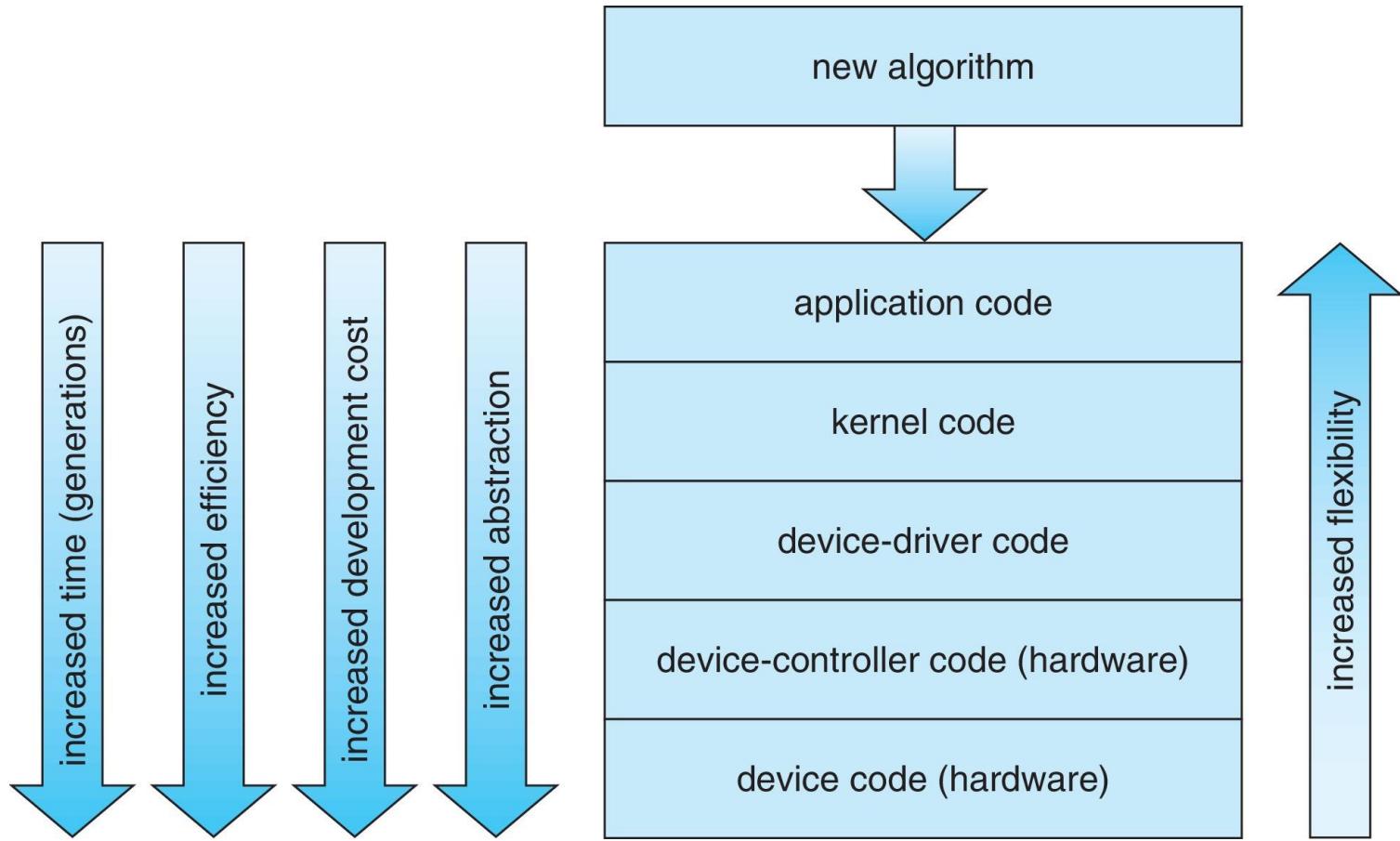
---

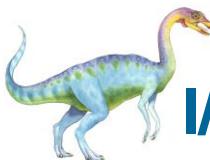
- Reduce number of context switches
- Reduce data copying
- Reduce interrupts by using large transfers, smart controllers, polling
- Use DMA
- Use smarter hardware devices
- Balance CPU, memory, bus, and I/O performance for highest throughput
- Move user-mode processes / daemons to kernel threads



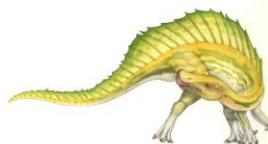
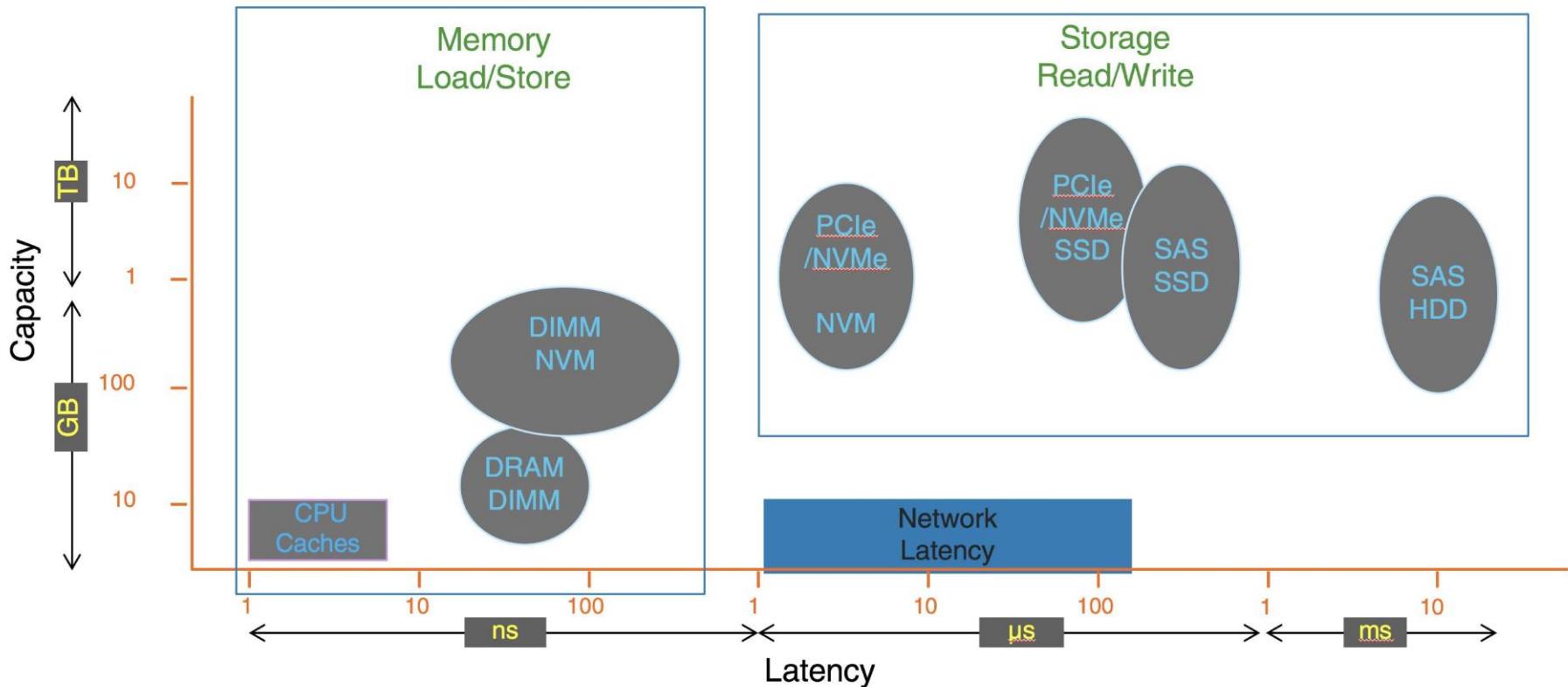


# Device-Functionality Progression

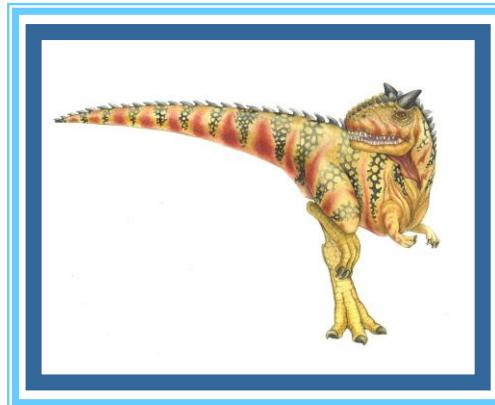




# I/O Performance of Storage (and Network Latency)

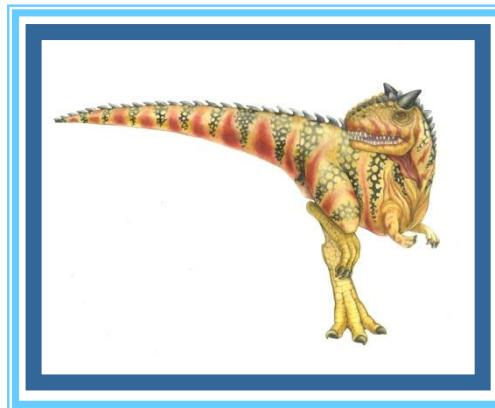


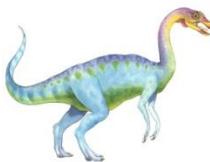
# End of Chapter 12



# Chapter 13:

# File-System Interface



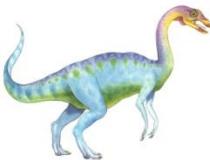


# Outline

---

- File Concept
- Access Methods
- Disk and Directory Structure
- File-System Mounting
- File Sharing
- Protection





# Objectives

---

- To explain the functions of file systems
- To describe the interfaces to file systems
- To discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures
- To explore file-system protection





# File Concept

---

- Contiguous logical address space
- Types:
  - Data
    - ▶ numeric
    - ▶ character
    - ▶ binary
  - Program
- Contents defined by file's creator
  - Many types
    - ▶ Consider **text file, source file, executable file**





# File Attributes

---

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk
- Many variations, including extended file attributes such as file checksum





# File info Window on Mac OS X





# File Operations

- File is an **abstract data type**
- 6 basic operations
  - **Create**
  - **Write** – at **write pointer** location
  - **Read** – at **read pointer** location
  - **Reposition within file - seek**
  - **Delete**
  - **Truncate**
- To avoid constant searching of the file entry in the directory,
  - **Open( $F_i$ )** – search the directory structure on disk for entry  $F_i$ , and move the content of entry to **memory**
  - **Close ( $F_i$ )** – move the content of entry  $F_i$  in memory to directory structure on disk





# Open Files

- Several pieces of data are needed to manage open files:
  - **Open-file table**: tracks open files
  - File pointer: pointer to last read/write location, per process that has the file open
  - **File-open count**: counter of number of times a file is open – to allow removal of data from open-file table when the last process closes it
  - Disk location of the file: cache of data access information
  - Access rights: per-process access mode information





# Open File Locking

- Provided by some OS and file systems
  - Similar to reader-writer locks
  - **Shared lock** similar to reader lock – several processes can acquire concurrently
  - **Exclusive lock** similar to writer lock
- Mediates access to a file
- Mandatory or advisory:
  - **Mandatory** – access is denied depending on locks held and requested
  - **Advisory** – processes can find status of locks and decide what to do





# File Locking Example – Java API

```
import java.io.*;
import java.nio.channels.*;
public class LockingExample {
    public static final boolean EXCLUSIVE = false;
    public static final boolean SHARED = true;
    public static void main(String args[]) throws IOException {
        FileLock sharedLock = null;
        FileLock exclusiveLock = null;
        try {
            RandomAccessFile raf = new RandomAccessFile("file.txt",
                "rw");
            // get the channel for the file
            FileChannel ch = raf.getChannel();
            // this locks the first half of the file - exclusive
            exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);
            /** Now modify the data . . . */
            // release the lock
            exclusiveLock.release();
        }
    }
}
```





# File Locking Example – Java API (Cont.)

```
// this locks the second half of the file - shared  
sharedLock = ch.lock(raf.length()/2+1, raf.length(),  
                     SHARED);  
/** Now read the data . . . */  
// release the lock  
sharedLock.release();  
} catch (java.io.IOException ioe) {  
    System.err.println(ioe);  
}finally {  
    if (exclusiveLock != null)  
        exclusiveLock.release();  
    if (sharedLock != null)  
        sharedLock.release();  
}  
}  
}
```





# File Types – Filename Extension

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information



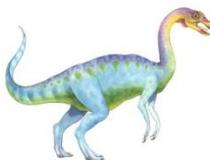


# File Structure

---

- None - sequence of words, bytes
- Simple record structure
  - Lines
  - Fixed length
  - Variable length
- Complex Structures
  - Formatted document
  - Relocatable load file
- Who decides:
  - OS
  - Program





# Access Methods

- Sequential Access

```
read next  
write next  
reset  
no read after last write  
(rewrite)
```

- Direct Access – file is fixed length **logical records**

```
read n  
write n  
position to n  
read next  
write next  
rewrite n
```

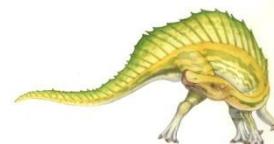
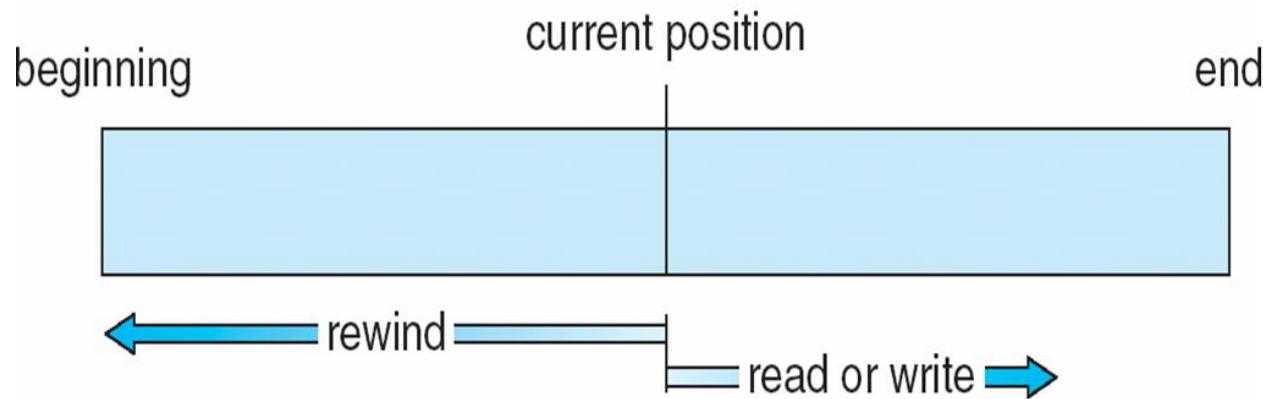
$n$  = **relative block number**

- Relative block numbers allow OS to decide where file should be placed
  - See **allocation problem** in Ch.14





# Sequential-access File





# Simulation of Sequential Access on Direct-access File

sequential access	implementation for direct access
<i>reset</i>	$cp = 0;$
<i>read next</i>	<i>read cp;</i> $cp = cp + 1;$
<i>write next</i>	<i>write cp;</i> $cp = cp + 1;$





# Other Access Methods

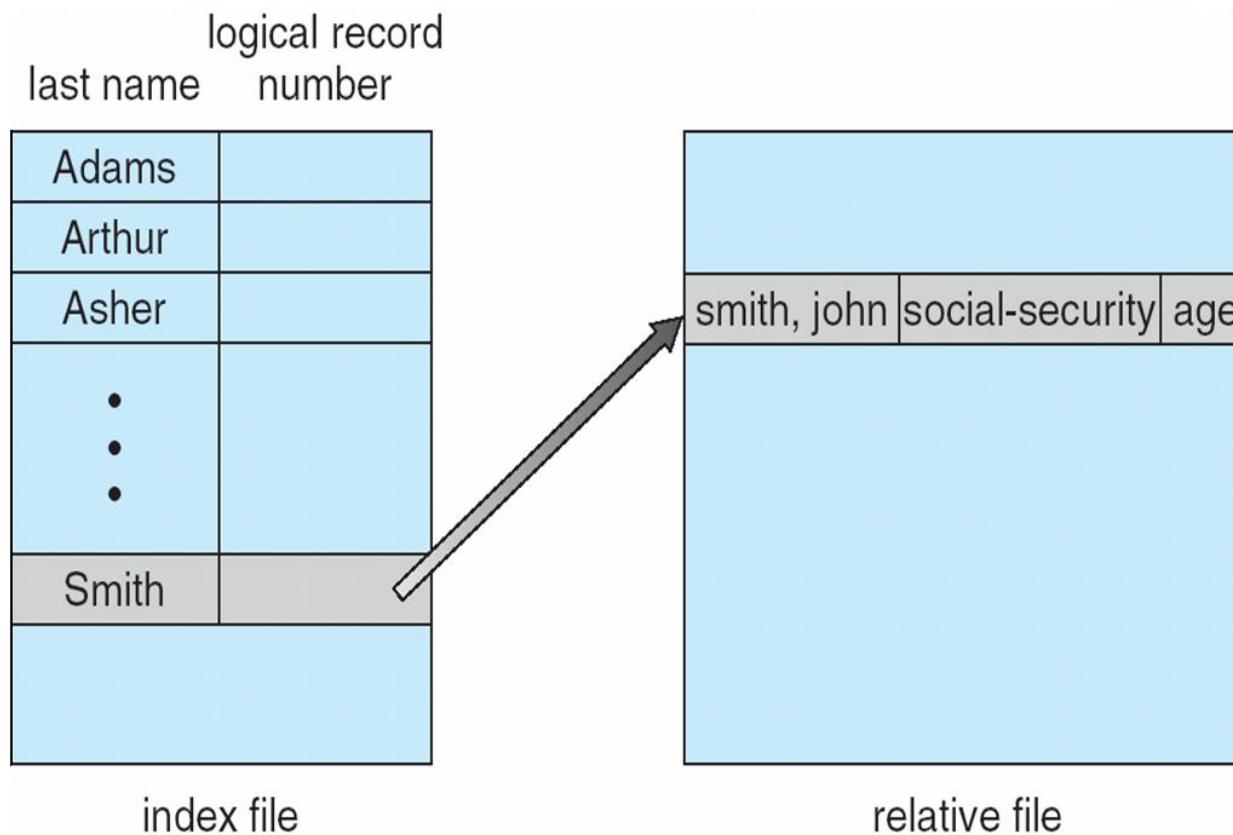
---

- Can be built on top of base methods
- Generally involve creation of an **index** for the file
  - Keep index in memory for fast determination of location of data to be operated on (consider UPC code plus record of data about that item)
- If too large, index (in memory) of the index (on disk)
  - E.g. IBM indexed sequential-access method (**ISAM**)
    - ▶ Small master index, points to disk blocks of secondary index
    - ▶ File kept sorted on a defined key
    - ▶ All done by the OS
- VMS operating system provides index and relative files as another example (see next slide)





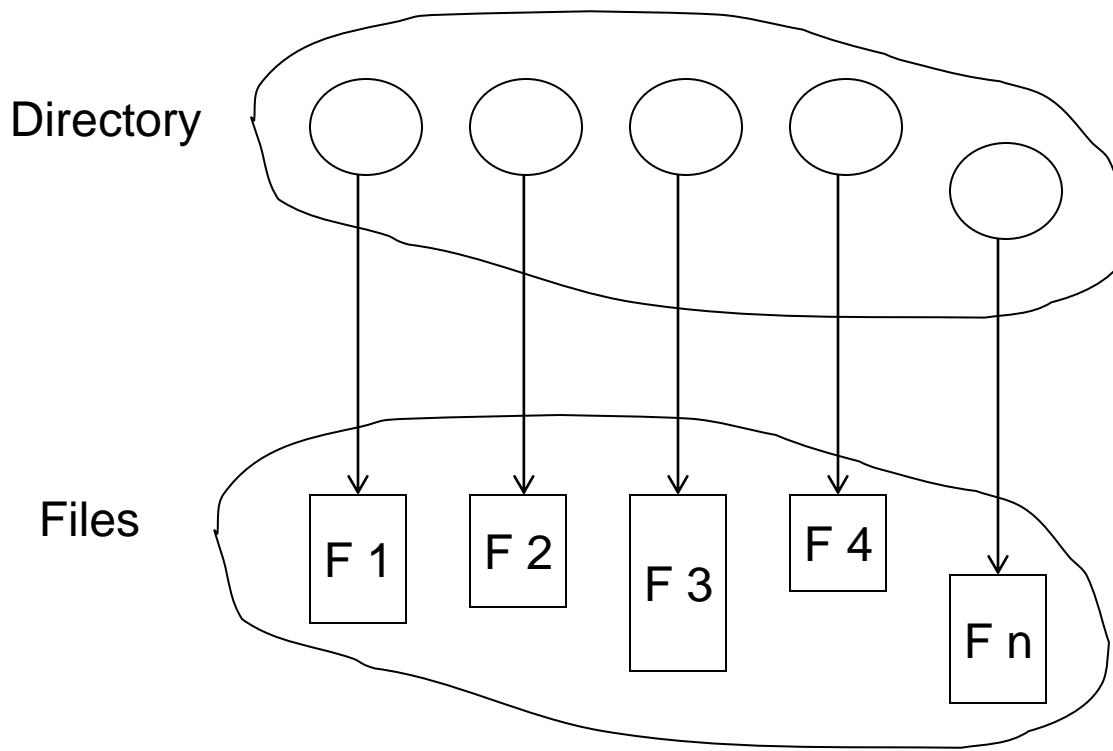
# Example of Index and Relative Files





# Directory Structure

- A collection of nodes containing information about all files



Both the directory structure and the files reside on disk





# Disk Structure

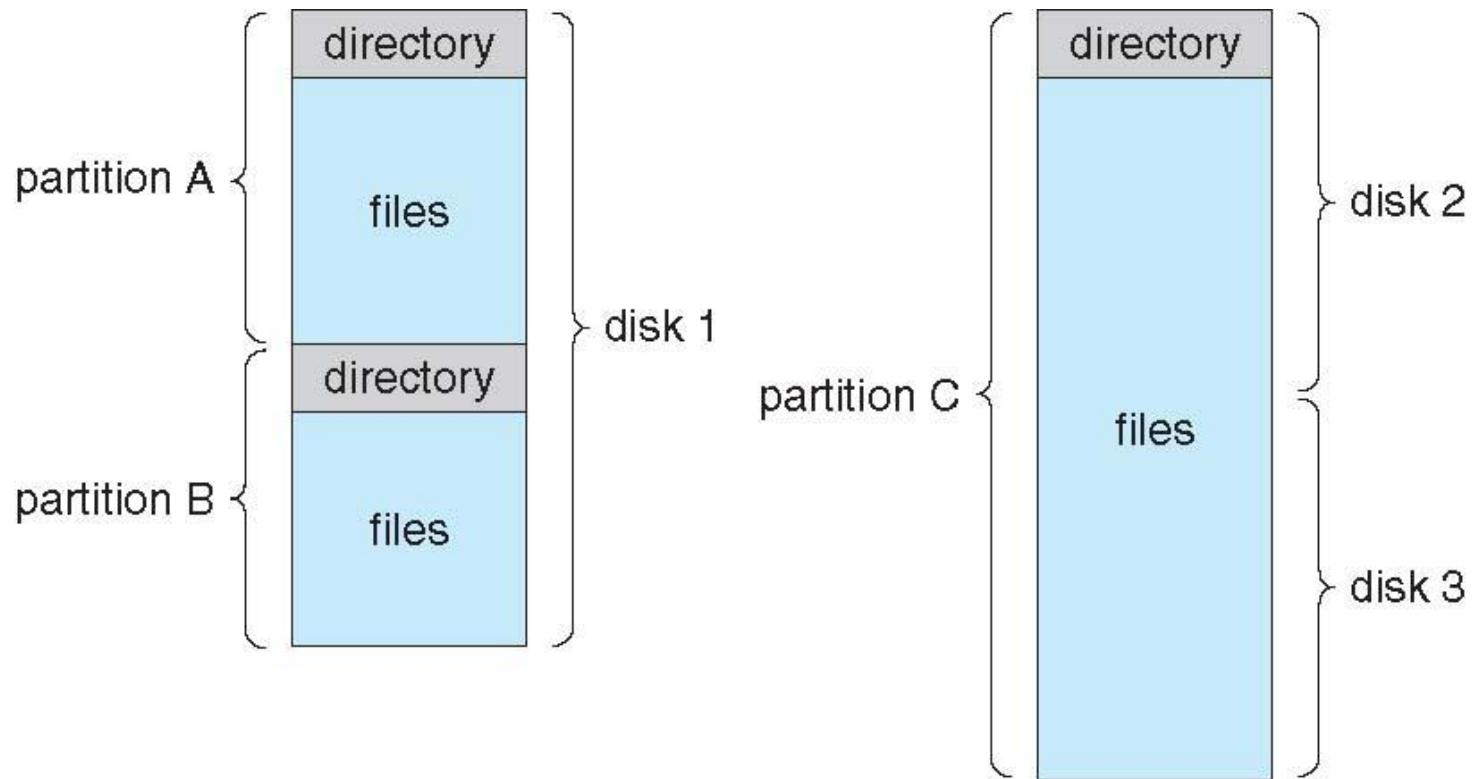
---

- Disk can be subdivided into **partitions**
  - Disks or partitions can be **RAID** protected against failure
  - Disk or partition can be used **raw** – without a file system, or **formatted** with a file system
  - Partitions also known as minidisks, slices
- Entity containing file system known as a **volume**
  - Each volume containing file system also tracks that file system's info in **device directory** or **volume table of contents**
- As well as **general-purpose file systems** there are many **special-purpose file systems**, frequently all within the same OS or computer





# A Typical File-system Organization





# Types of File Systems

- We mostly talk of general-purpose file systems
  - But systems frequently have may file systems, some general- and some special- purpose
- Consider Solaris has
  - tmpfs – **memory**-based volatile FS for fast, temporary I/O
  - objfs – interface into **kernel memory** to get kernel symbols for debugging
  - ctfs – contract file system for managing **daemons**
  - lofs – loopback file system allows one FS to be accessed in place of another
  - procfs – kernel interface to **process** structures
  - ufs, zfs – general purpose file systems





# Operations Performed on Directory

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system





# Directory Organization

The directory is organized logically to obtain

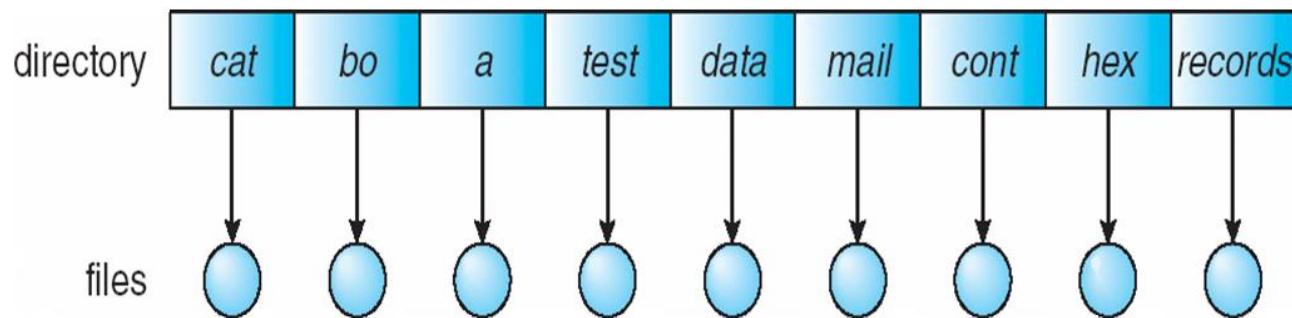
- Efficiency – locating a file quickly
- Naming – convenient to users
  - Two users can have the same name for different files
  - The same file can have several different names
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, ...)





# Single-Level Directory

- A single directory for all users



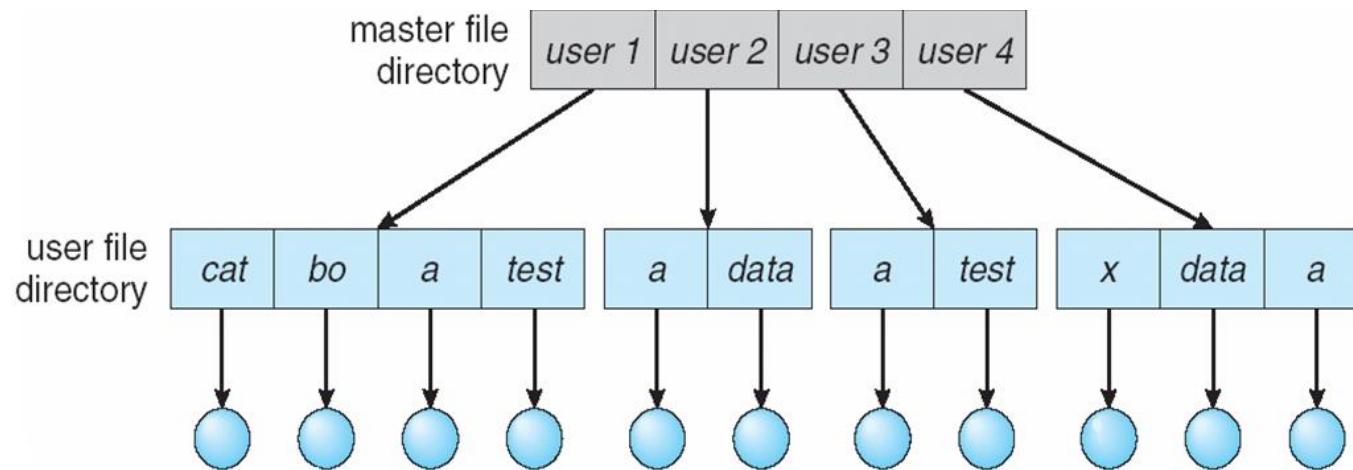
- Naming problem
- Grouping problem





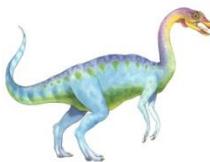
# Two-Level Directory

- Separate directory for each user

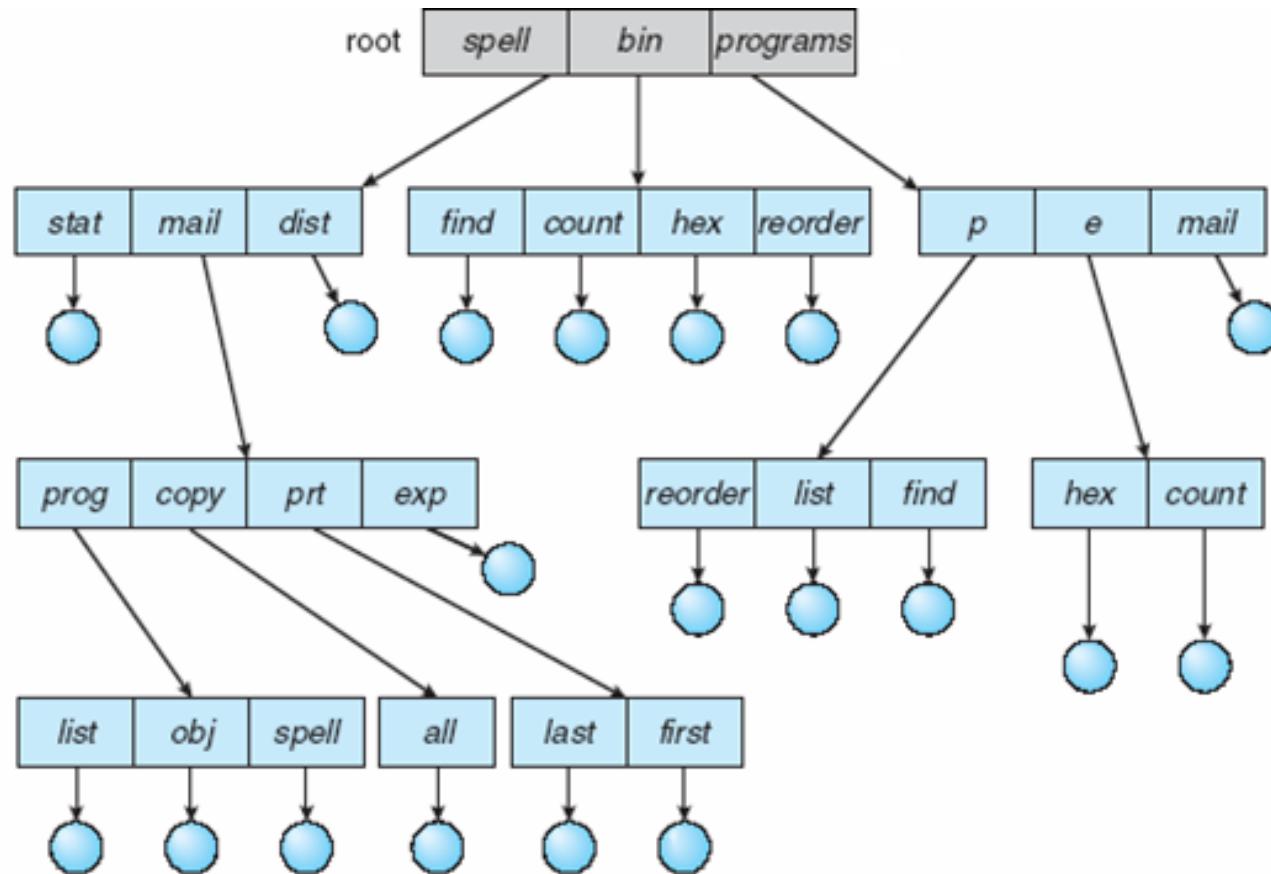


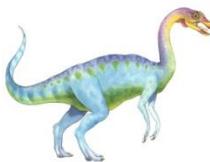
- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability





# Tree-Structured Directories





# Tree-Structured Directories (Cont.)

- Efficient searching
- Grouping Capability
- Current directory (working directory)
  - `cd /spell/mail/prog`
  - `type list`





# Tree-Structured Directories (Cont.)

- **Absolute** or **relative** path name
- Creating a new file is done in current directory
- Delete a file

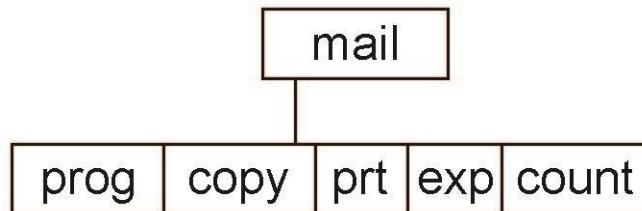
`rm <file-name>`

- Creating a new subdirectory is done in current directory

`mkdir <dir-name>`

Example: if in current directory `/mail`

`mkdir count`



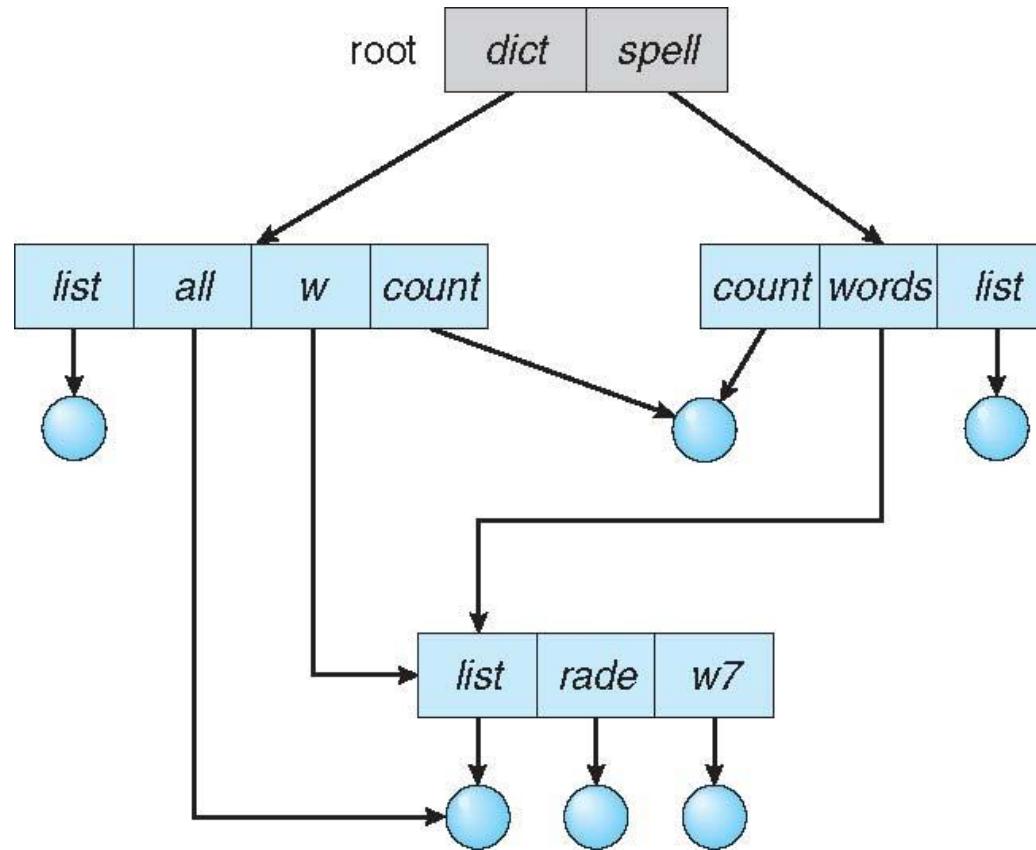
Deleting “mail” ⇒ deleting the entire subtree rooted by “mail”

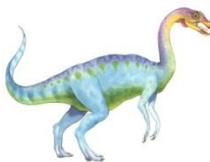




# Acyclic-Graph Directories

- Have shared subdirectories and files





# Acyclic-Graph Directories (Cont.)

- Two different names (aliasing)
- If **dict** deletes **list** ⇒ dangling pointer

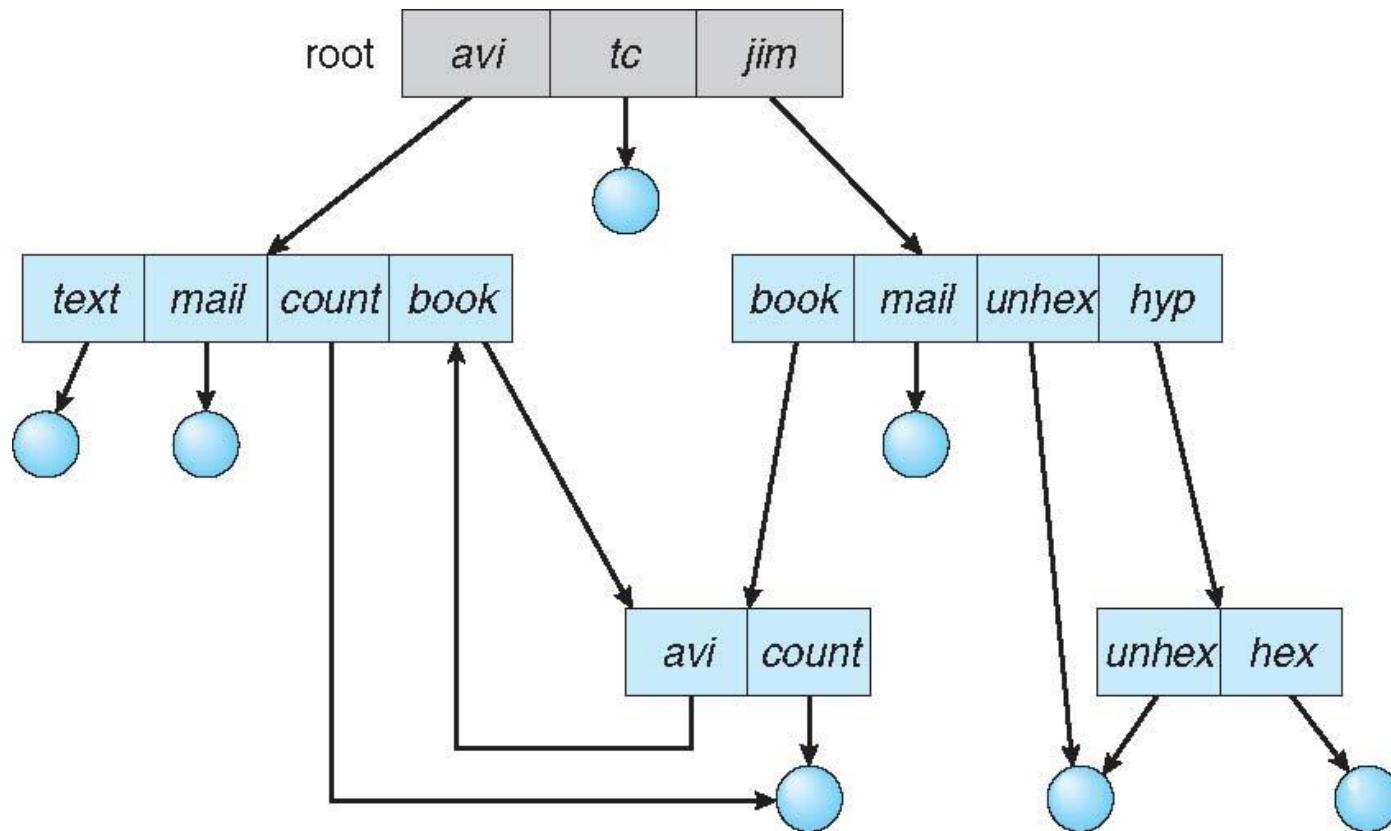
Solutions:

- Backpointers, so we can delete all pointers
  - Variable size records problem
- Backpointers using a daisy chain organization
  - Entry-hold-count solution
- New directory entry type
  - **Link** – another name (pointer) to an existing file
  - **Resolve the link** – follow pointer to locate the file





# General Graph Directory





# General Graph Directory (Cont.)

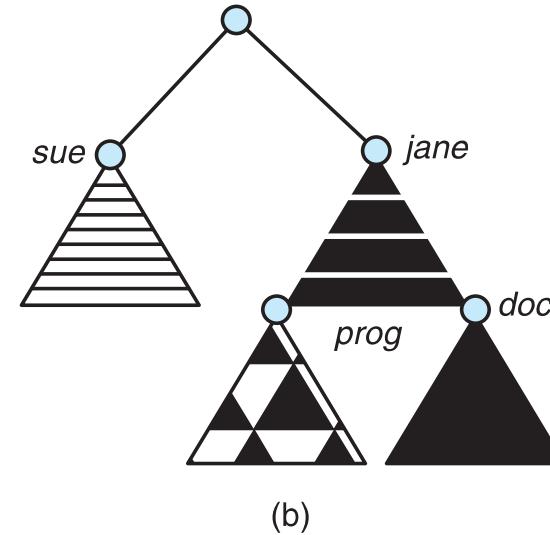
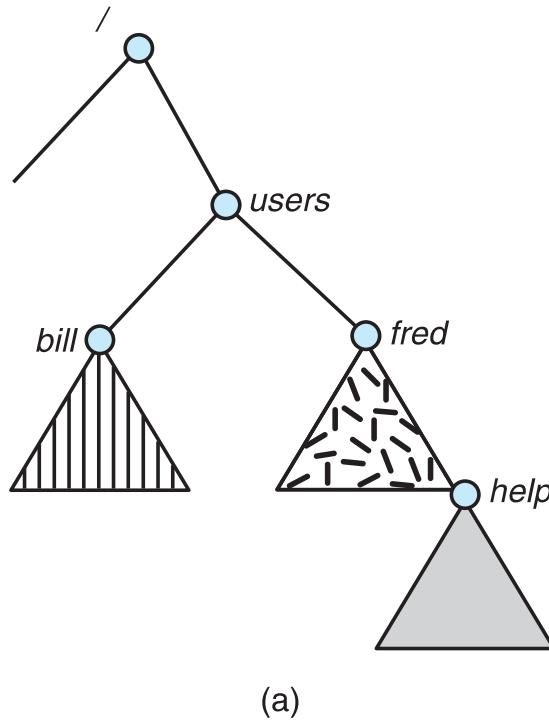
- How do we guarantee no cycles?
  - Allow only links to file not subdirectories
  - **Garbage collection**
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK

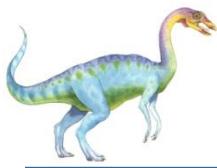




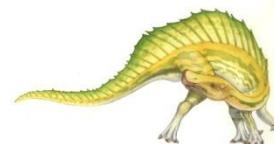
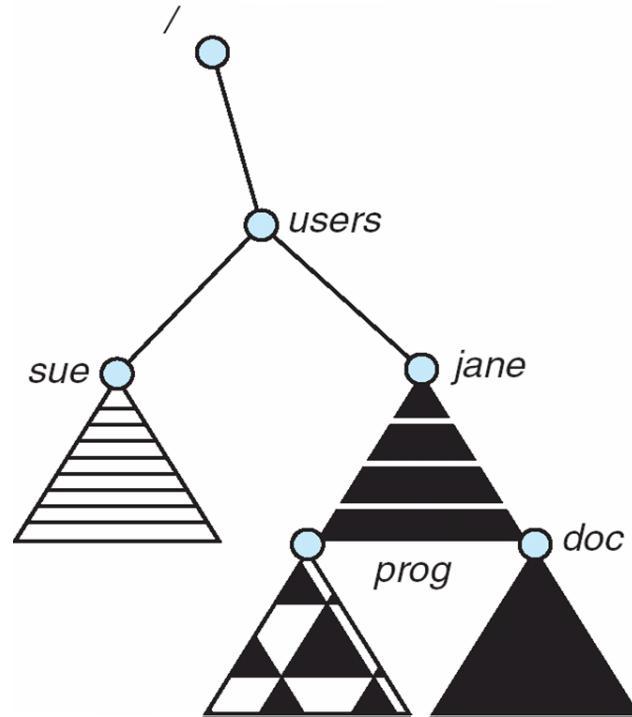
# File System Mounting

- A file system must be **mounted** before it can be accessed
- An unmounted file system (i.e., Fig. 11-11(b)) is mounted at a **mount point**





# Mount Point





# File Sharing

- Sharing of files on multi-user systems is desirable
- Sharing may be done through a **protection** scheme
- On distributed systems, files may be shared across a network
- Network File System (NFS) is a common distributed file-sharing method
- If multi-user system
  - **User IDs** identify users, allowing permissions and protections to be per-user
  - **Group IDs** allow users to be in groups, permitting group access rights
  - Owner of a file / directory
  - Group of a file / directory





# File Sharing – Remote File Systems

- Uses networking to allow file system access between systems
  - Manually via programs like FTP
  - Automatically, seamlessly using **distributed file systems**
  - Semi automatically via the **world wide web**
- **Client-server** model allows clients to mount remote file systems from servers
  - Server can serve multiple clients
  - Client and user-on-client identification is insecure or complicated
  - **NFS** is standard UNIX client-server file sharing protocol
  - **CIFS** is standard Windows protocol
  - Standard OS file system calls are translated into remote calls
- Distributed Information Systems (**distributed naming services**) such as LDAP, DNS, NIS, Active Directory implement unified access to information needed for remote computing

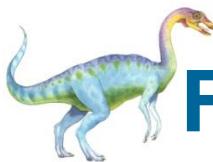




# File Sharing – Failure Modes

- All file systems have failure modes
  - For example corruption of directory structures or other non-user data, called **metadata**
- Remote file systems add new failure modes, due to network failure, server failure
- Recovery from failure can involve **state information** about status of each remote request
- **Stateless** protocols such as NFS v3 include all information in each request, allowing easy recovery but less security





# File Sharing – Consistency Semantics

- Specify how multiple users are to access a shared file simultaneously
  - Similar to Ch.6 process synchronization algorithms
    - ▶ Tend to be less complex due to disk I/O and network latency (for remote file systems)
  - Unix file system (UFS) implements **UNIX semantics**:
    - ▶ Writes to an open file visible immediately to other users of the same open file
    - ▶ Sharing file pointer to allow multiple users to read and write concurrently
  - Andrew File System (AFS) implemented complex remote file sharing semantics - **session semantics**
    - ▶ Writes only visible to sessions starting after the file is closed





# Protection

---

- File owner/creator should be able to control:
  - what can be done
  - by whom
- Types of access
  - **Read**
  - **Write**
  - **Execute**
  - **Append**
  - **Delete**
  - **List**



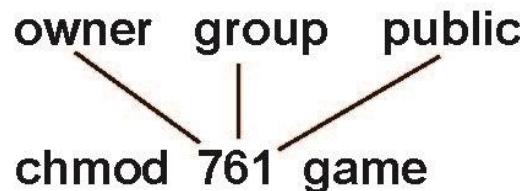


# Access Lists and Groups

- Mode of access: read, write, execute
- Three classes of users on Unix / Linux

		RWX
a) <b>owner access</b>	7	$\Rightarrow$ 1 1 1
b) <b>group access</b>	6	$\Rightarrow$ 1 1 0
c) <b>public access</b>	1	$\Rightarrow$ 0 0 1

- Ask manager to create a group (unique name), say G, and add some users to the group
- For a particular file (say *game*) or subdirectory, define an appropriate access



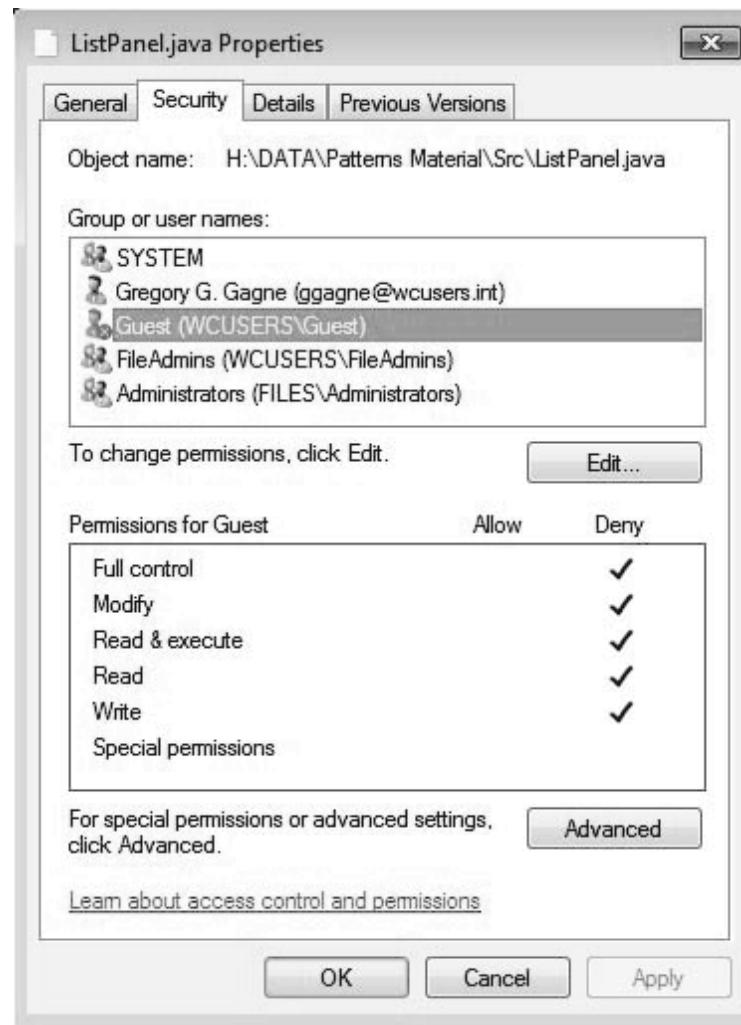
Attach a group to a file

`chgrp G game`





# Windows Access-Control List Management



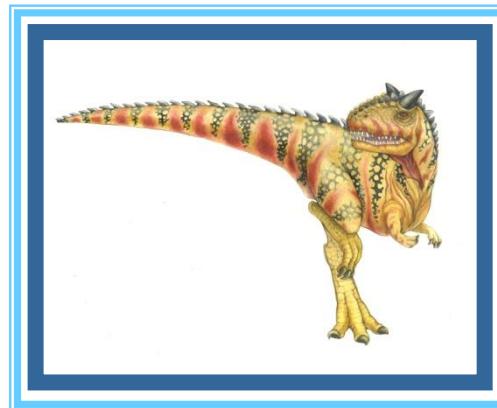


# A Sample UNIX Directory Listing

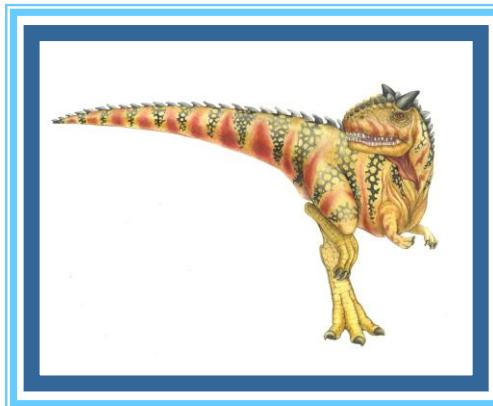
-rw-rw-r--	1	pbg	staff	31200	Sep 3 08:30	intro.ps
drwx-----	5	pbg	staff	512	Jul 8 09:33	private/
drwxrwxr-x	2	pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2	pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1	pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1	pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4	pbg	faculty	512	Jul 31 10:31	lib/
drwx-----	3	pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3	pbg	staff	512	Jul 8 09:35	test/

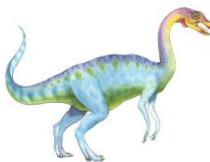


# End of Chapter 13



# Chapter 14: File System Implementation



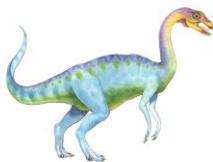


# Outline

---

- File-System Structure
- File-System Operations
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery
- Example: WAFL File System



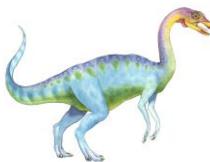


# Objectives

---

- Describe the details of implementing local file systems and directory structures
- Discuss **block allocation** and free-block algorithms and trade-offs
- Explore file system efficiency and performance issues
- Look at recovery from file system failures
- Describe the WAFL file system as a concrete example

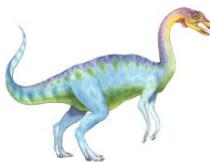




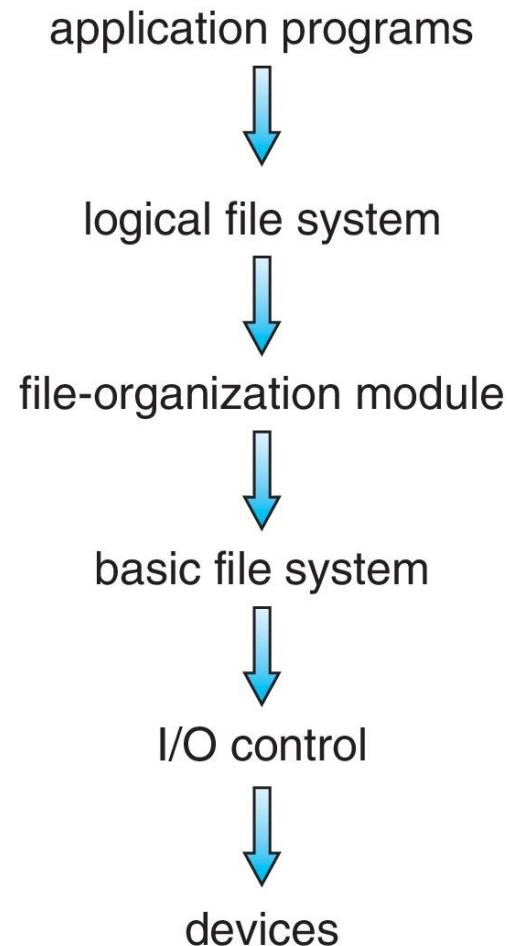
# File-System Structure

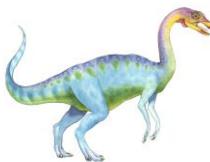
- File structure
  - Logical storage unit
  - Collection of related information
- **File system** resides on secondary storage (disks)
  - Provides user interface to storage, mapping logical to physical
  - Provides efficient and convenient access to disk by allowing data to be stored, located, and retrieved easily
- Disk provides in-place rewrite and random access
  - I/O transfers performed in **blocks** of **sectors** (usually 512 bytes)
- **File control block (FCB)** – storage structure consisting of information about a file
- **Device driver** controls the physical device
- File system organized into layers





# Layered File System

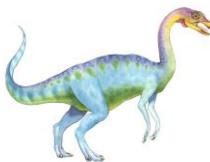




# File System Layers

- **Device drivers** manage I/O devices at the I/O control layer
  - Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” outputs low-level hardware specific commands to hardware controller (Ch.12)
- **Basic file system** given command like “retrieve block 123” translates to device driver
  - Also manages memory buffers and caches (allocation, freeing, replacement)
    - ▶ Buffers hold data in transit
    - ▶ Caches hold frequently used data
- **File organization module** understands files, logical address, and physical blocks
  - Translates logical block # to physical block #
  - Manages free space, disk allocation

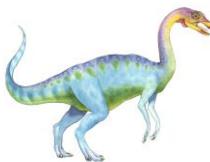




# File System Layers (Cont.)

- **Logical file system** manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in UNIX)
  - Directory management
  - Protection
- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
- Logical layers can be implemented by any coding method according to OS designer

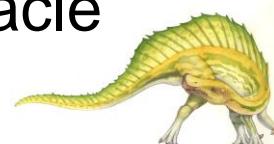


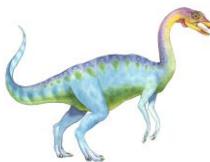


# File System Layers (Cont.)

---

- Many file systems, sometimes many are supported within an OS
  - Each with its own format
    - ▶ CD-ROM is ISO 9660
    - ▶ Unix has **UFS**, FFS
    - ▶ Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray
    - ▶ Linux has more than 130 types, with **extended file system** ext3 and ext4 leading; plus distributed file systems, etc.
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE

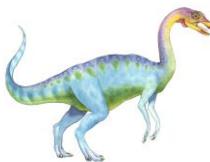




# File-System Operations

- We have system calls at the API level, but how do we implement their functions?
  - On-disk and in-memory structures
- On-disk structures
  - Boot control block (per volume)
  - Volume control block (per volume)
  - Directory structure (per file system)
  - Per-file File-control block (FCB)





# On-disk Structures

---

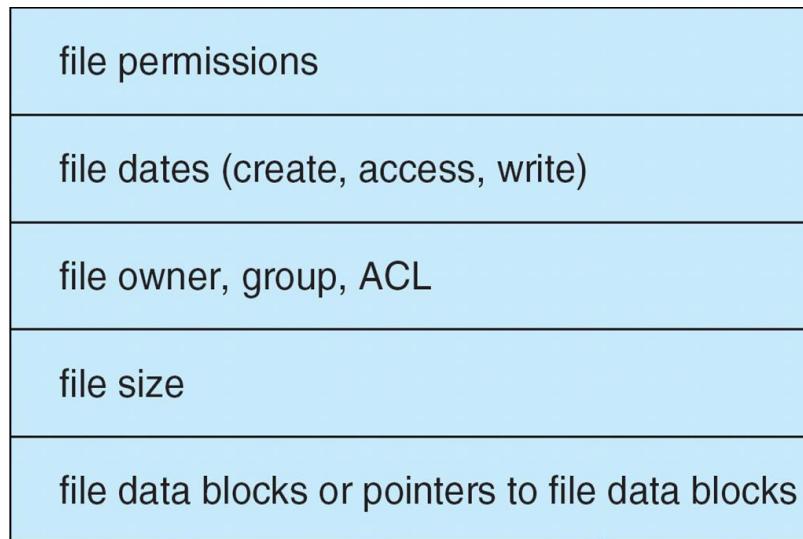
- **Boot control block:** info needed to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
  - UNIX boot block, NTFS partition boot sector
- **Volume control block (superblock, master file table)** contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
  - UNIX superblock, NTFS master file table
- Directory structure organizes the files
  - UNIX filenames and inode numbers, NTFS master file table
- Per-file **File Control Block (FCB)** contains many details about the file
  - typically inode number, permissions, size, dates
  - NTFS stores info in master file table using relational DB structures

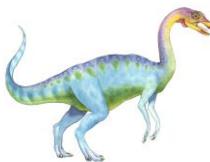




# File-System Implementation (Cont.)

- A typical file-control block

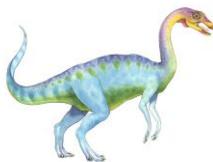




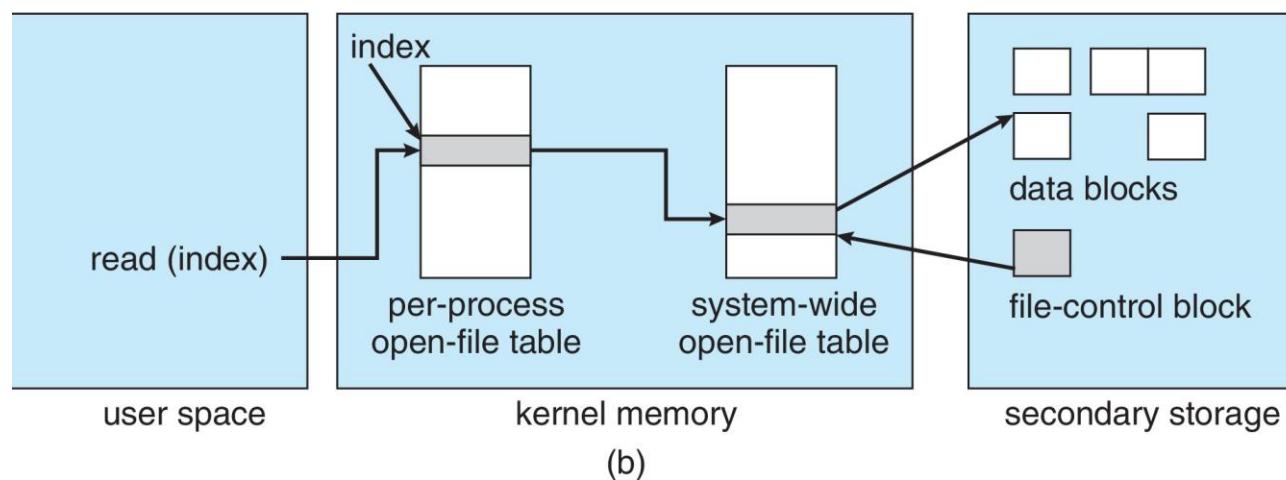
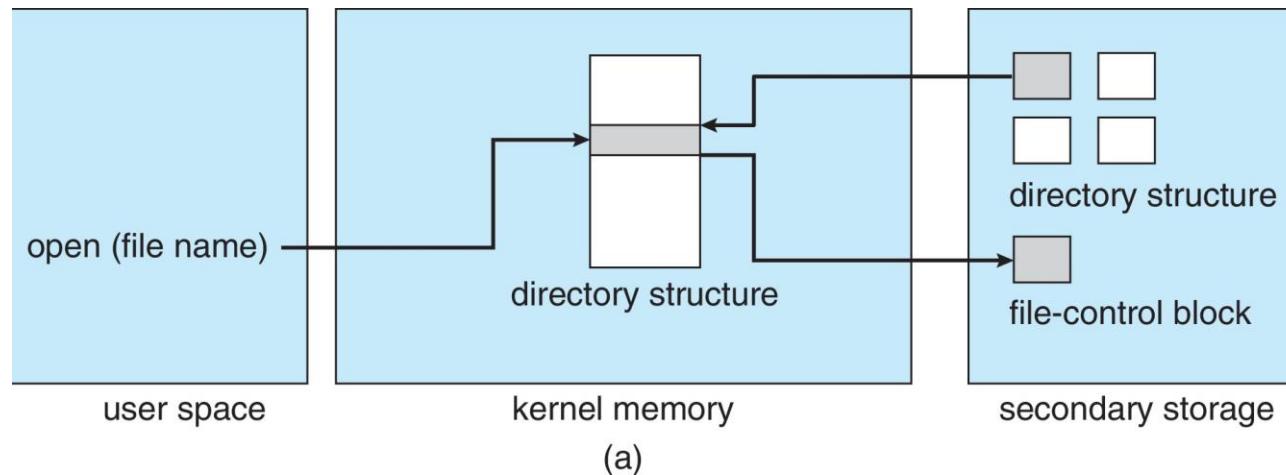
# In-Memory File System Structures

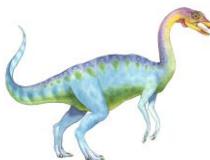
- **Mount table** storing file system mounts, mount points, file system types
- **system-wide open-file table** contains a copy of the FCB of each file and other info
- **per-process open-file table** contains pointers to appropriate entries in system-wide open-file table as well as other info
- Directory-structure cache holds directory information of recently accessed directories
- Plus buffers hold data blocks from secondary storage





# In-Memory File System Structures

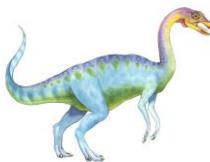




# Directory Implementation

- **Linear list** of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - ▶ Linear search time
    - ▶ Could keep ordered alphabetically via linked list or use B+ tree
- **Hash Table** – linear list with hash data structure
  - Decreases directory search time
  - **Collisions** – situations where two file names hash to the same location
  - Only good if entries are fixed size, or use chained-overflow method

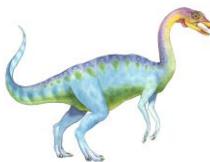




# Allocation Methods - Contiguous

- An allocation method refers to how disk blocks are allocated for files:
- **Contiguous allocation** – each file occupies set of contiguous blocks
  - Best performance in most cases
  - Simple – only starting location (block #) and length (number of blocks) are required
  - Problems include finding space for file, knowing file size, external fragmentation, need for **compaction off-line (downtime)** or **on-line**





# Contiguous Allocation

- Mapping from logical to physical

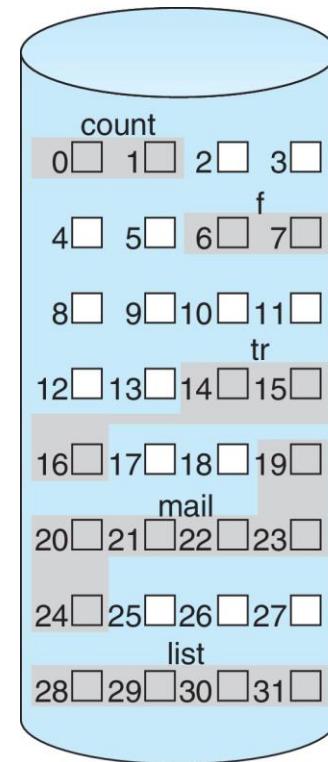
LA/512

Q

R

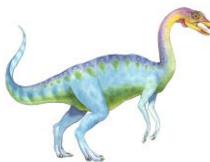
Block to be accessed = Q +  
starting address

Displacement into block = R



directory		
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2



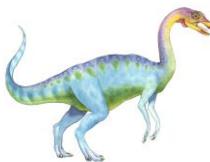


# Extent-Based Systems

---

- Many newer file systems (i.e., Veritas File System) use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in extents
- An **extent** is contiguous blocks on disk
  - Extents are allocated for file allocation
  - A file consists of one or more extents





# Allocation Methods - Linked

---

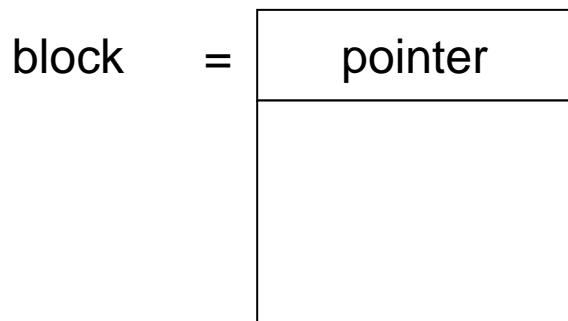
- **Linked allocation** – each file a linked list of blocks
  - File ends at nil pointer
  - Each block contains pointer to next block
  - Free space management system called when new block needed
- Pros and cons
  - No compaction, external fragmentation
  - **Reliability** can be a problem
  - Locating a block can take many I/Os and disk seeks
    - ▶ Improve efficiency by clustering blocks into groups but increases internal fragmentation



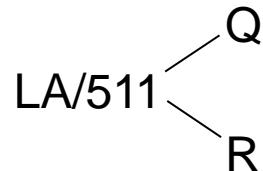


# Linked Allocation

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk



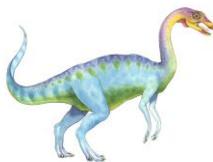
- Mapping



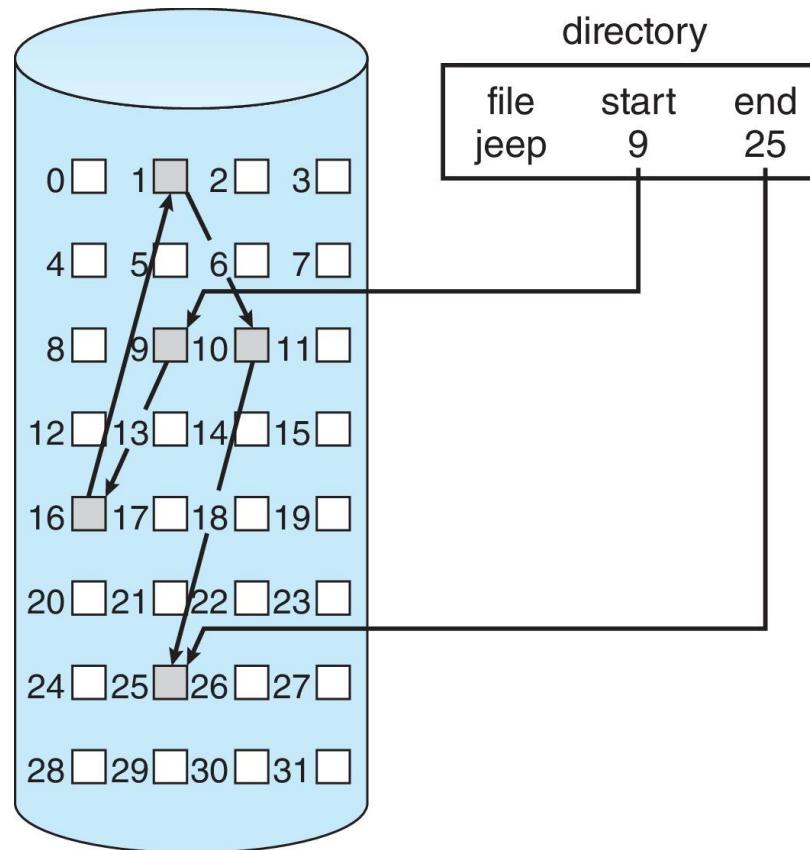
Block to be accessed is the  $Q$ th block in the linked chain of blocks representing the file

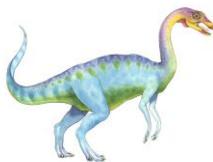
Displacement into block =  $R + 1$



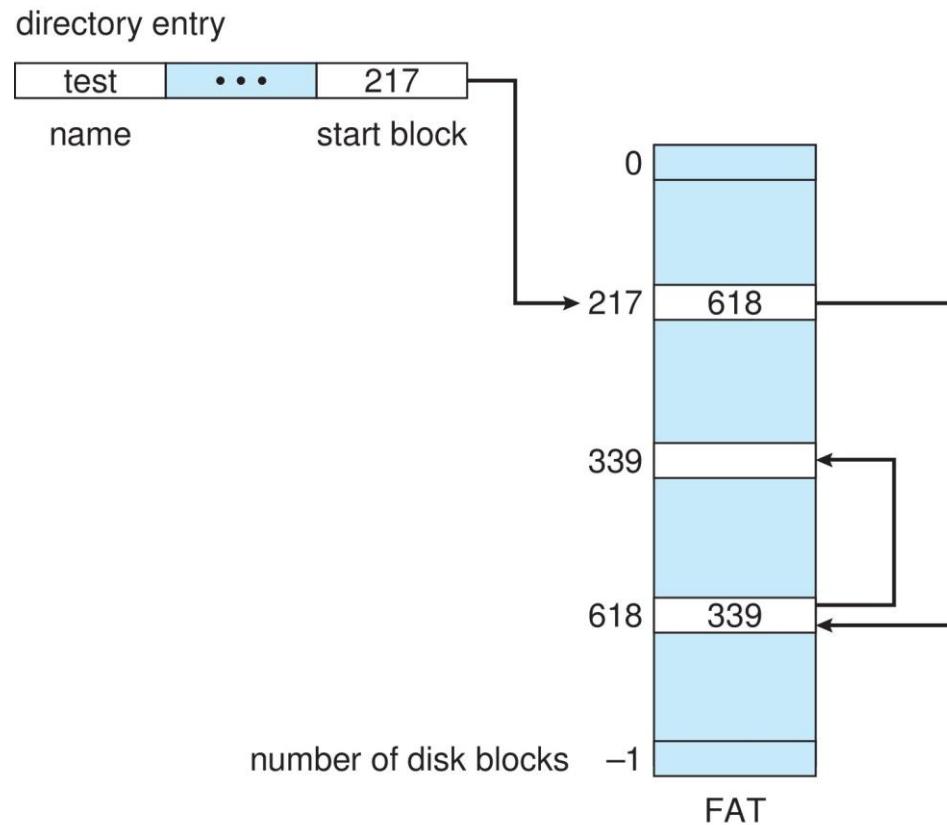


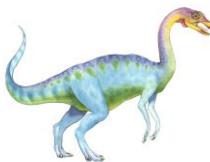
# Linked Allocation





# File-Allocation Table



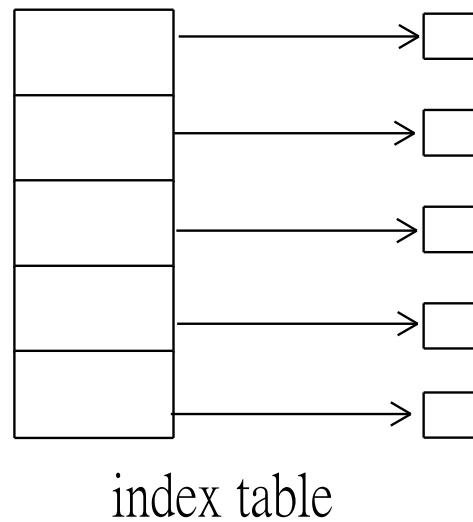


# Allocation Methods - Indexed

- **Indexed allocation**

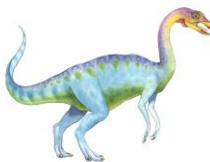
- Each file has its own **index block(s)** of pointers to its data blocks

- Logical view

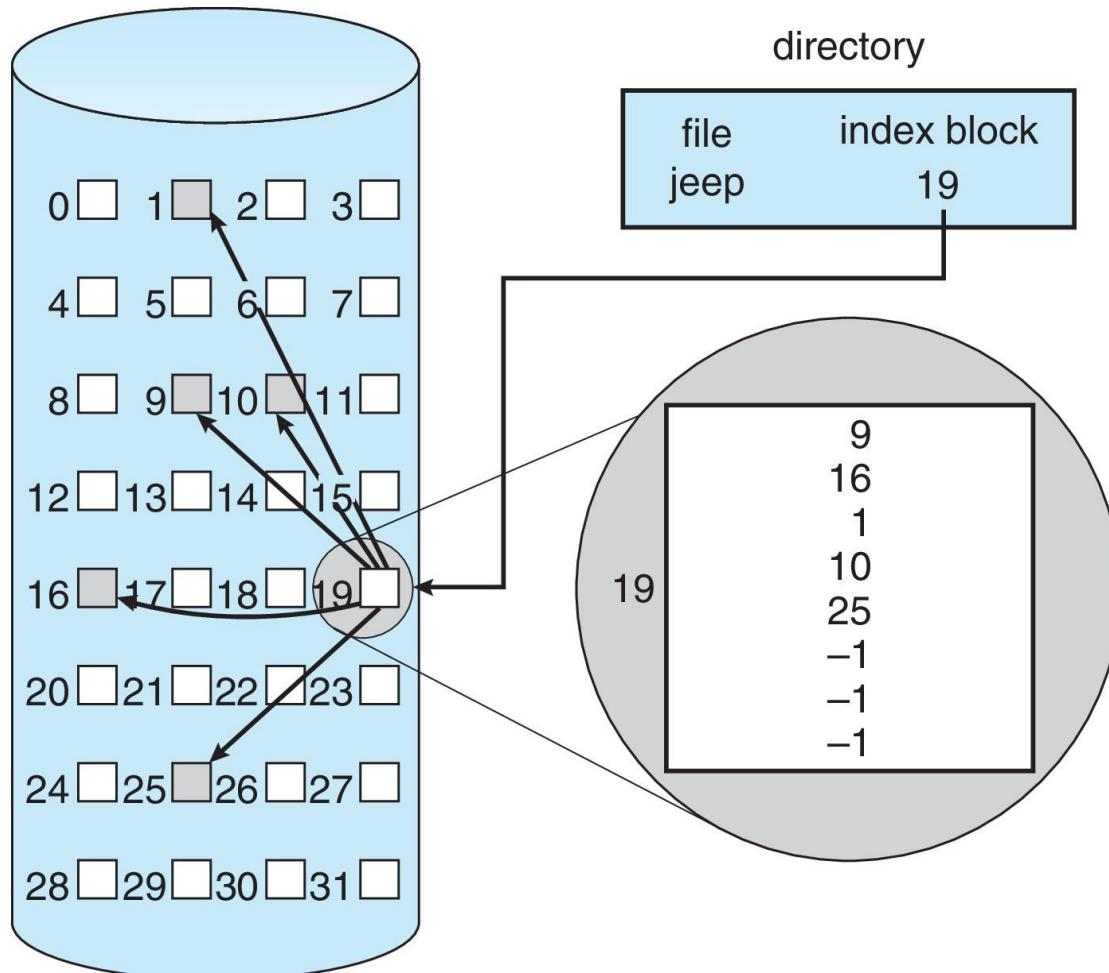


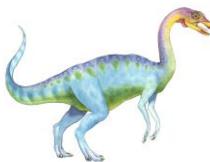
index table





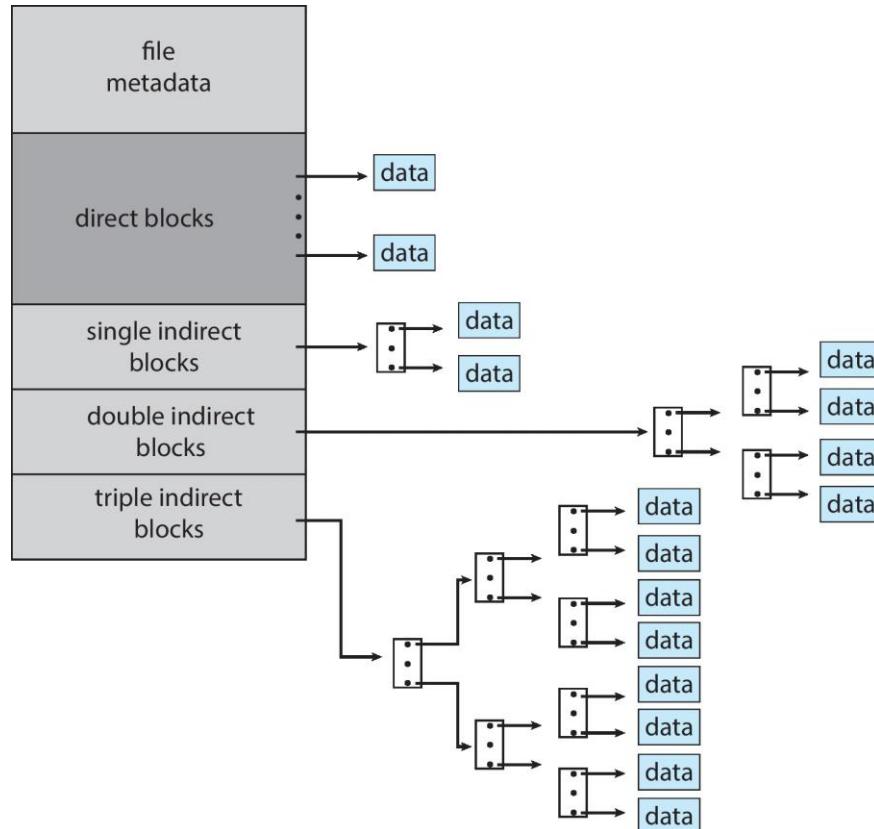
# Example of Indexed Allocation





# Combined Scheme: UNIX UFS

4K bytes per block, 32-bit addresses



More index blocks than can be addressed with 32-bit file pointer



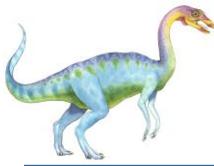


# Performance

---

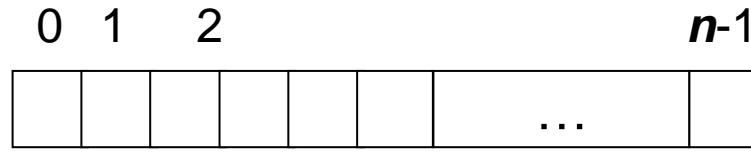
- Best method depends on file access type
  - Contiguous great for sequential and random
- Linked good for sequential, not random
  - Declare access type at creation -> select either contiguous or linked
- Indexed more complex
  - Single block access could require 2 index block reads then data block read
  - Clustering can help improve throughput, reduce CPU overhead
- For NVM, no disk head so different algorithms and optimizations needed
  - Using old algorithm takes many CPU cycles trying to avoid non-existent head movement
  - With NVM goal is to reduce CPU cycles and overall path needed for I/O





# Free-Space Management

- File system maintains **free-space list** to track available blocks/clusters
    - (Using term “block” for simplicity)
  - **Bit vector** or **bit map** ( $n$  blocks)



$$\text{bit}[i] = \begin{cases} 1 & \Rightarrow \text{block}[i] \text{ free} \\ 0 & \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

## Block number calculation

(number of bits per word) \*  
(number of 0-value words) +  
offset of first 1 bit

CPUs have instructions to return offset within word of first “1” bit





# Free-Space Management (Cont.)

- Bit map requires extra space

- Example:

block size = 4KB =  $2^{12}$  bytes

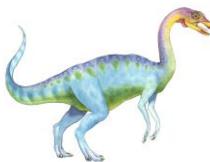
disk size =  $2^{40}$  bytes (1 terabyte)

$n = 2^{40}/2^{12} = 2^{28}$  bits (or 32MB)

if clusters of 4 blocks -> 8MB of memory

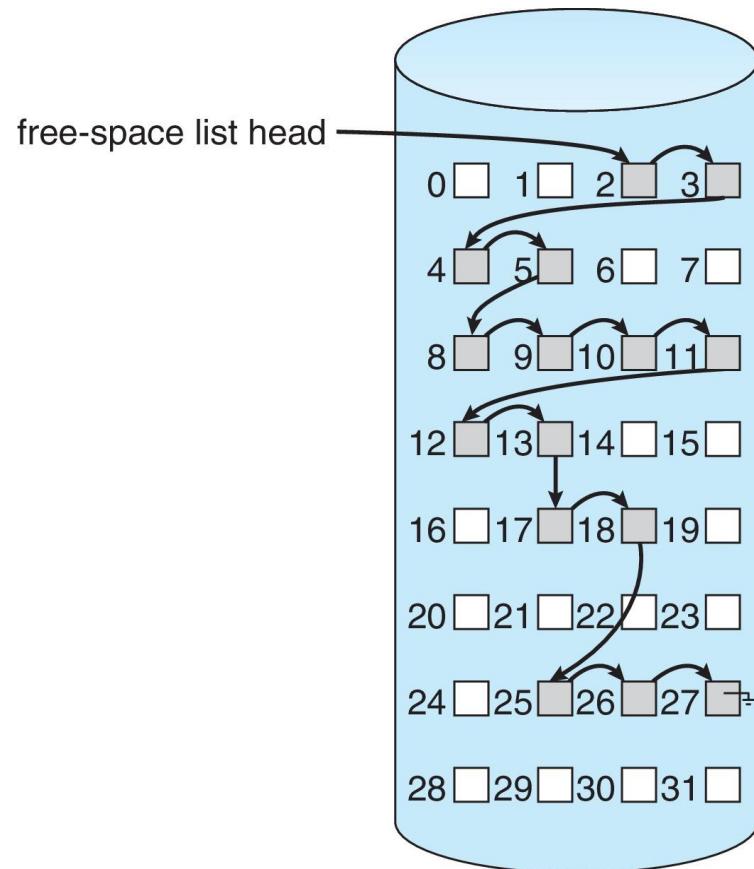
- Easy to get contiguous files

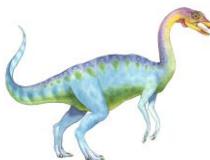




# Linked Free Space List on Disk

- Linked list (free list)
  - Cannot get contiguous space easily
  - No waste of space
  - No need to traverse the entire list (if # free blocks recorded)

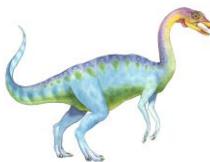




# Free-Space Management (Cont.)

- Grouping
  - Modify linked list to store address of next  $n-1$  free blocks in first free block, plus a pointer to next block that contains free-block-pointers (like this one)
- Counting
  - Because space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering
    - ▶ Keep address of first free block and count of following free blocks
    - ▶ Free space list then has entries containing addresses and counts





# Free-Space Management (Cont.)

- Space Maps
  - Used in **ZFS**
  - Consider meta-data I/O on very large file systems
    - ▶ Full data structures like bit maps couldn't fit in memory -> thousands of I/Os
  - Divides device space into **metaslab** units and manages metaslabs
    - ▶ Given volume can contain hundreds of metaslabs
  - Each metaslab has associated space map
    - ▶ Uses counting algorithm
  - But records to log file rather than file system
    - ▶ Log of all block activity, in time order, in counting format
  - Metaslab activity -> load space map into memory in balanced-tree structure, indexed by offset
    - ▶ Replay log into that structure
    - ▶ Combine contiguous free blocks into single entry



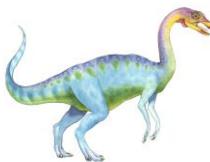


# TRIMing Unused Blocks

---

- HDDS overwrite in place so need only free list
- Blocks not treated specially when freed
  - Keeps its data but without any file pointers to it, until overwritten
- Storage devices not allowing overwrite (like NVM) suffer badly with same algorithm
  - Must be erased before written, erases made in large chunks (blocks, composed of pages) and are slow
  - TRIM is a newer mechanism for the file system to inform the NVM storage device that a page is free
    - ▶ Can be garbage collected or if block is free, now block can be erased



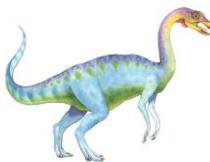


# Efficiency and Performance

---

- Efficiency dependent on:
  - Disk allocation and directory algorithms
  - Types of data kept in file's directory entry
  - Pre-allocation or as-needed allocation of metadata structures
  - Fixed-size or varying-size data structures

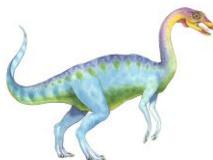




# Efficiency and Performance (Cont.)

- Performance
  - Keeping data and metadata close together
  - **Buffer cache** – separate section of main memory for frequently used blocks
  - **Synchronous** writes sometimes requested by apps or needed by OS
    - ▶ No buffering / caching – writes must hit disk before acknowledgement
    - ▶ **Asynchronous** writes more common, buffer-able, faster
  - **Free-behind** and **read-ahead** – techniques to optimize sequential access
  - Reads frequently slower than writes



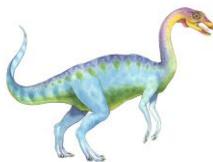


# Page Cache

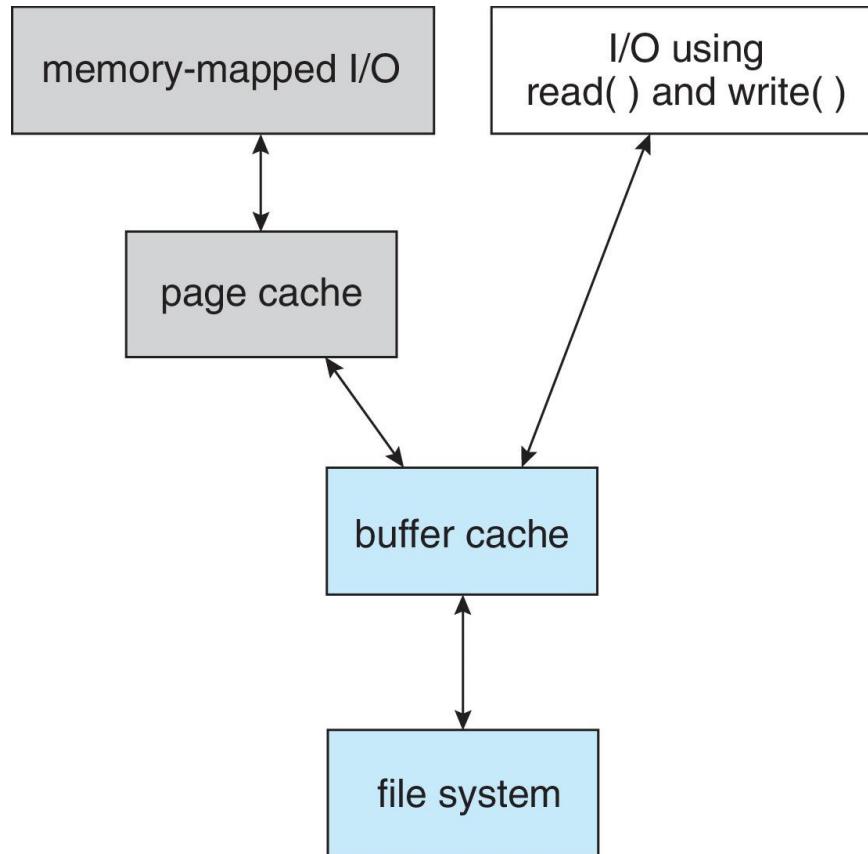
---

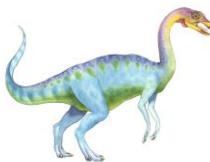
- A **page cache** caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure





# I/O Without a Unified Buffer Cache

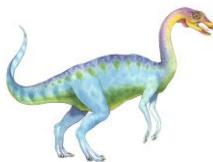




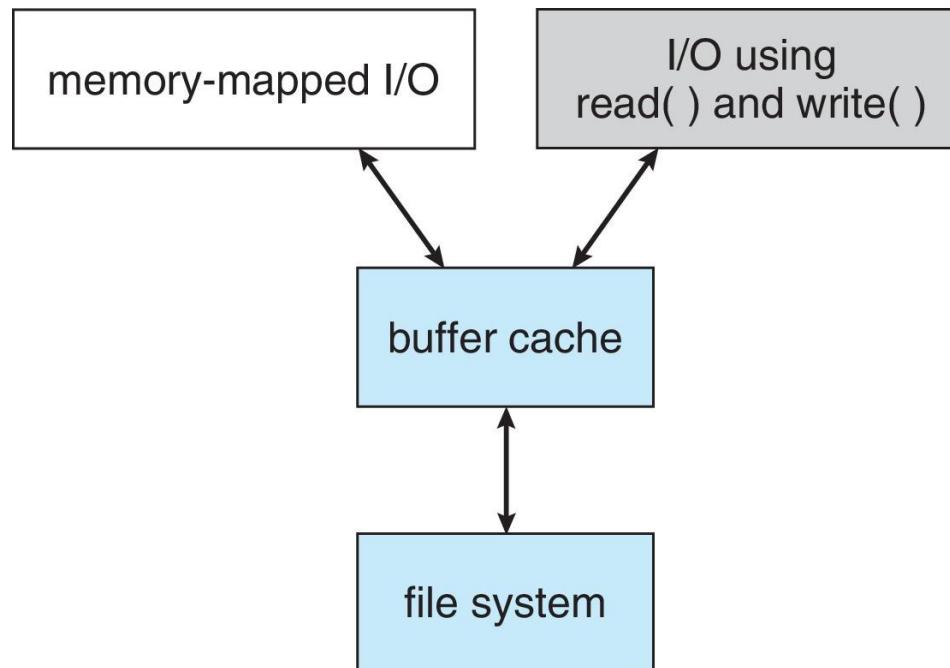
# Unified Buffer Cache

- A **unified buffer cache** uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid **double caching**
- But which caches get priority, and what replacement algorithms to use?





# I/O Using a Unified Buffer Cache



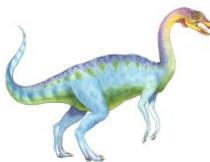


# Recovery

---

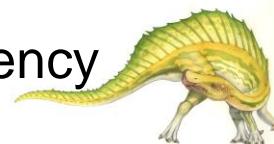
- **Consistency checking** – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - Can be slow and sometimes fails
- Use system programs to **back up** data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by **restoring** data from backup

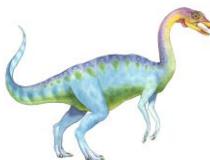




# Log Structured File Systems

- **Log structured** (or **journaling**) file systems record each metadata update to the file system as a **transaction**
- All transactions are written to a log
  - A transaction is considered **committed** once it is written to the log (sequentially)
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated
- The transactions in the log are **asynchronously** written to the file system structures
  - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata

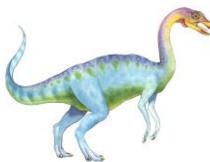




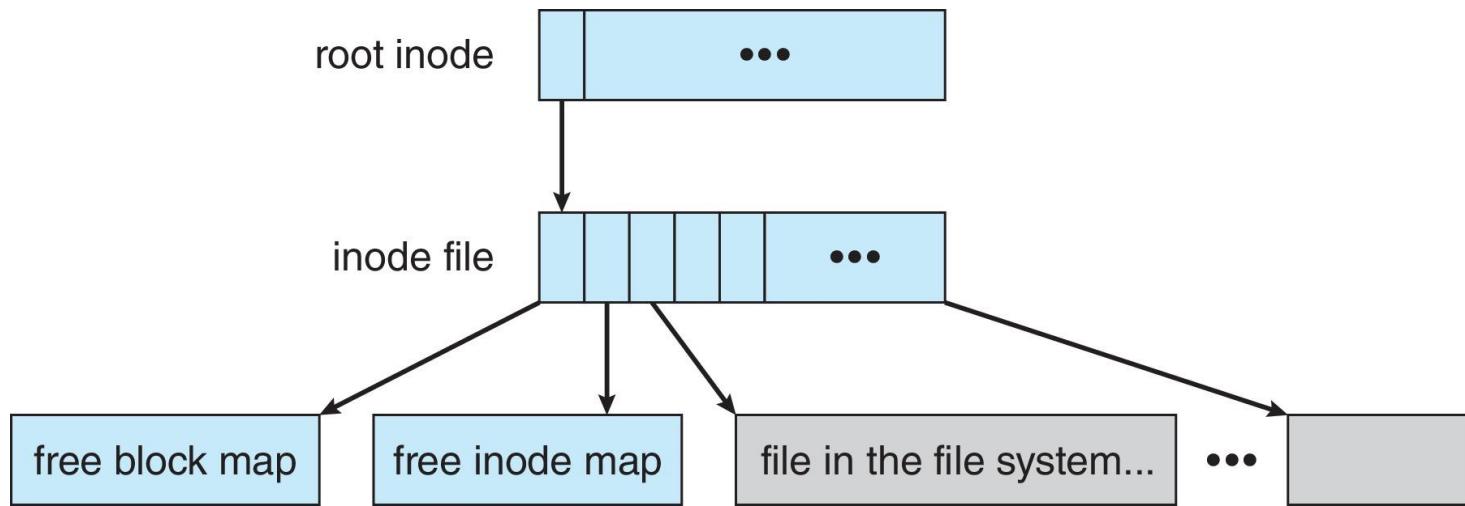
# Example: WAFL File System

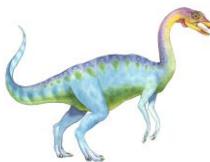
- Used on Network Appliance (NetApp) “Filers” – distributed file system appliances
- “**W**rite-Anywhere **F**ile **L**ayout”
- Serves up NFS, CIFS, HTTP, FTP
- Random I/O optimized, write optimized
  - NVRAM for write caching
- Similar to Berkeley Fast File System, with extensive modifications



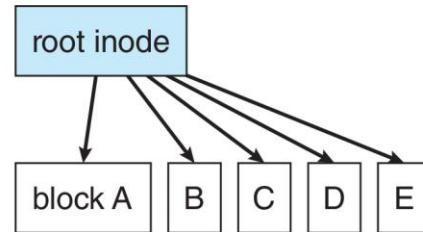


# The WAFL File Layout

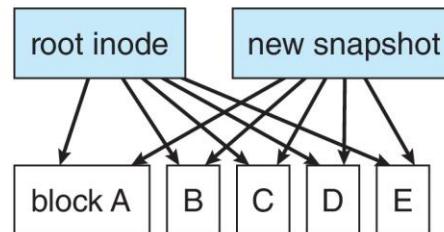




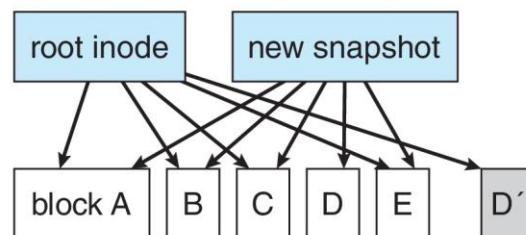
# Snapshots in WAFL



(a) Before a snapshot.

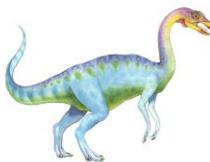


(b) After a snapshot, before any blocks change.



(c) After block D has changed to D'.





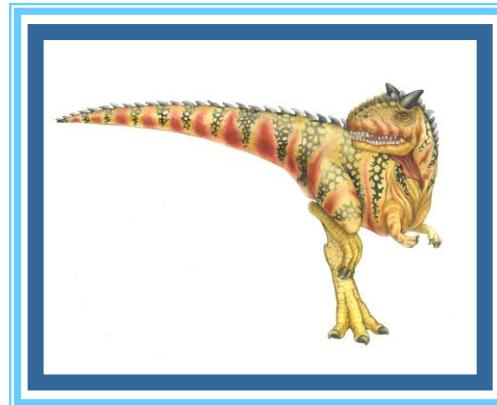
# The Apple File System

---

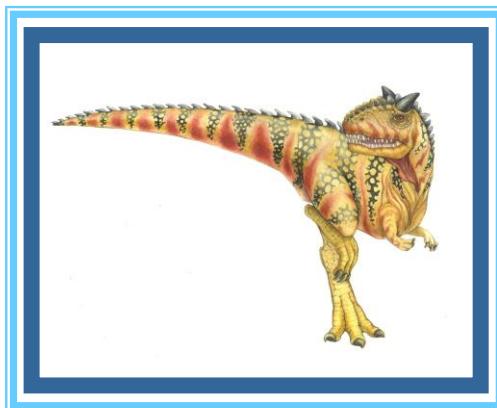
- Apple released a new file system in 2017 called APFS to replace its 30-year-old HFS+
- The goal is to run on all current Apple devices
  - From Apple Watch through the iPhone to the Mac computers
  - watchOS, iOS, tvOS, macOS
- Features include
  - 64-bit pointers, clones for files and directories, snapshots, copy-on-write design, encryption
  - **Space sharing:** storage is available as one or more large free spaces (containers) from which file systems can draw allocations
  - **Fast directory sizing:** provides quick used space calculation and updating
  - **Atomic safe-save primitives:** perform renames of files, bundles of files, and directories as single atomic operations

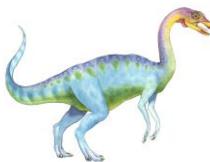


# End of Chapter 14



# Chapter 15: File System Internals



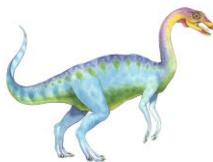


# Outline

---

- File Systems
- File-System Mounting
- Partitions and Mounting
- File Sharing
- Virtual File Systems
- Remote File Systems
- Consistency Semantics
- NFS



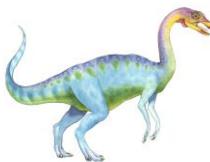


# Objectives

---

- Delve into the details of file systems and their implementation
- Explore booting and file sharing
- Describe remote file systems, using **NFS** as an example

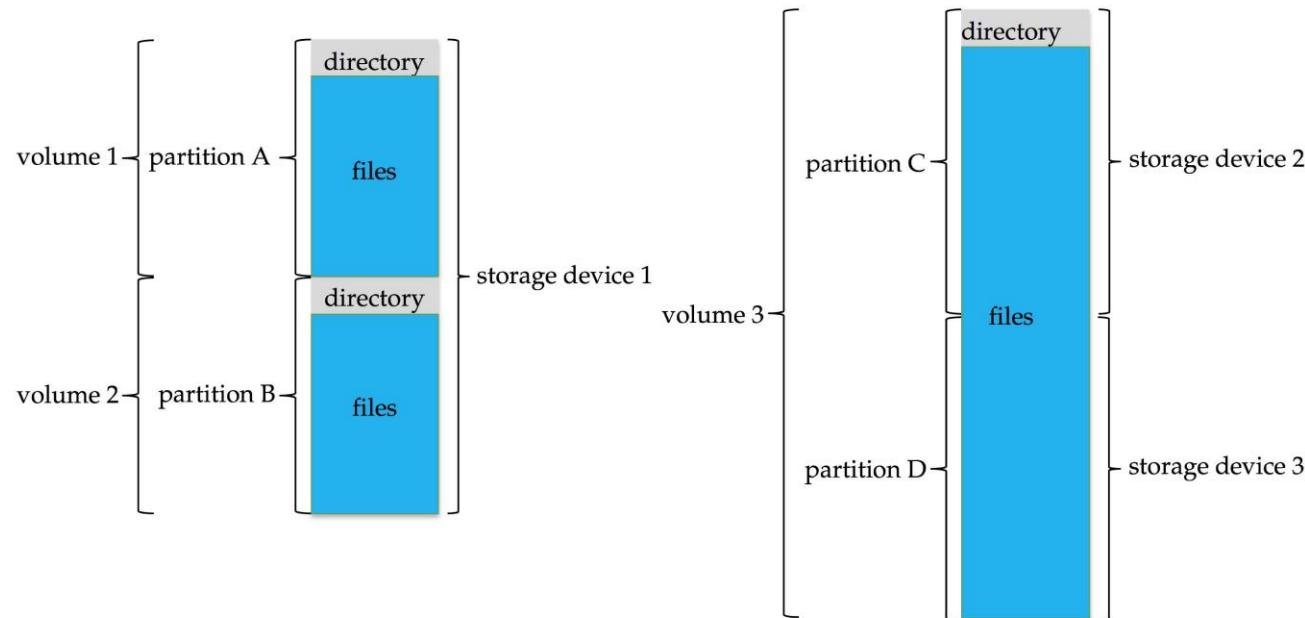




# File System

- General-purpose computers can have multiple storage devices
  - Devices can be sliced into partitions, which hold volumes
  - Volumes can span multiple partitions
  - Each volume usually formatted into a file system
  - # of file systems varies, typically dozens available to choose from

Typical storage device organization:

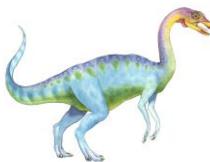




# Example Mount Points and File Systems - Solaris

/	ufs
/devices	devfs
/dev	dev
/system/contract	ctfs
/proc	proc
/etc/mnttab	mntfs
/etc/svc/volatile	tmpfs
/system/object	objfs
/lib/libc.so.1	lofs
/dev/fd	fd
/var	ufs
/tmp	tmpfs
/var/run	tmpfs
/opt	ufs
/zpbge	zfs
/zpbge/backup	zfs
/export/home	zfs
/var/mail	zfs
/var/spool/mqueue	zfs
/zpbg	zfs
/zpbg/zones	zfs



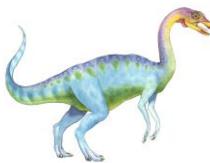


# Partitions and Mounting

---

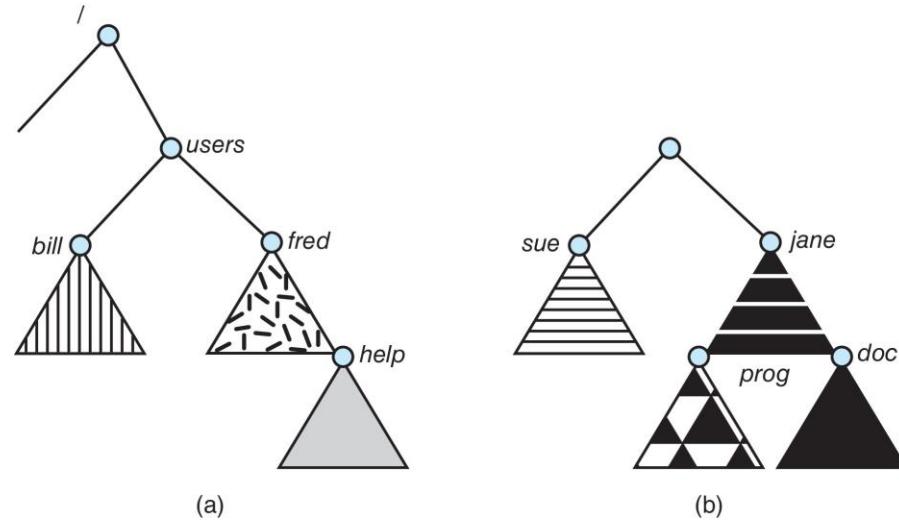
- Partition can be a volume containing a file system (“cooked”) or **raw** – just a sequence of blocks with no file system
- Boot block can point to boot volume or boot loader set of blocks that contain enough code to know how to load the kernel from the file system
  - Or a boot management program for multi-OS booting
- **Root partition** contains the OS, other partitions can hold other OSes, other file systems, or be raw
  - Mounted at boot time
  - Other partitions can mount automatically or manually on **mount points** – location at which they can be accessed
- At mount time, file system consistency checked
  - Is all metadata correct?
    - ▶ If not, fix it, try again
    - ▶ If yes, add to mount table, allow access



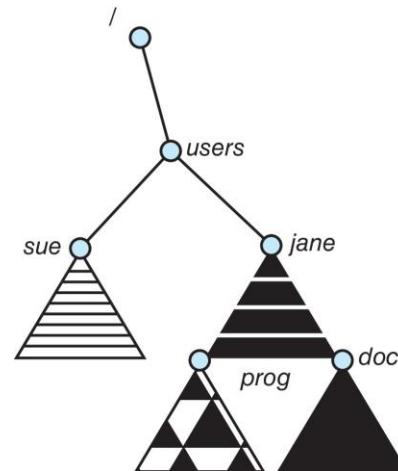


# File Systems and Mounting

- (a) Unix-like file system directory tree  
(b) Unmounted file system



After mounting  
(b) into the  
existing directory  
tree

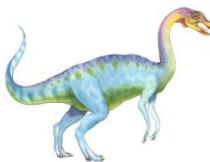




# File Sharing

- Allows multiple users / systems access to the same files
- Permissions / protection must be implemented and accurate
  - Most systems provide concepts of owner, group member
  - Must have a way to apply these between systems

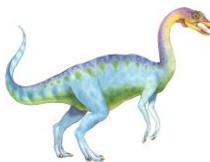




# Virtual File Systems

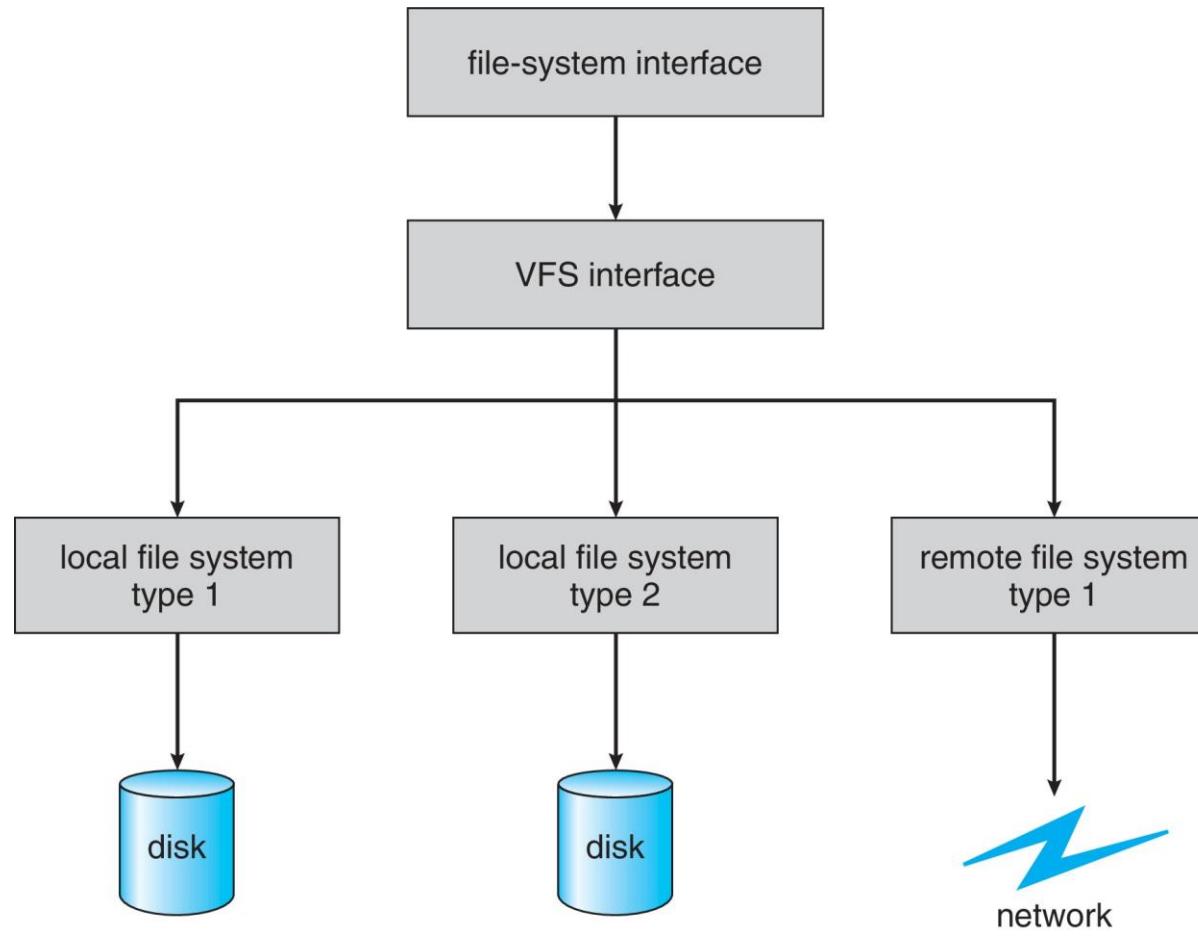
- **Virtual File Systems (VFS)** on Unix provide an object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - Separates file-system generic operations from implementation details
  - Implementation can be one of many file systems types, or network file system
    - ▶ Implements **vnodes** which hold inodes or network file details
  - Then dispatches operation to appropriate file system implementation routines

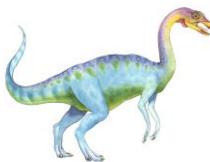




# Virtual File Systems (Cont.)

- The API is to the VFS interface, rather than any specific type of file system

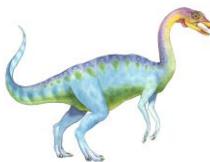




# Virtual File System Implementation

- For example, Linux has four object types:
  - **inode, file, superblock, dentry**
- VFS defines set of operations on the objects that must be implemented
  - Every object has a pointer to a function table
    - ▶ Function table has addresses of routines to implement that function on that object
    - ▶ For example:
      - `int open(. . .)`—Open a file
      - `int close(. . .)`—Close an already-open file
      - `ssize_t read(. . .)`—Read from a file
      - `ssize_t write(. . .)`—Write to a file
      - `int mmap(. . .)`—Memory-map a file



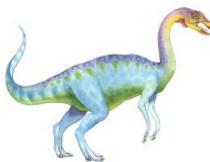


# Remote File Systems

---

- Sharing of files across a network
- First method involved manually sharing each file – programs like ftp
- Second method uses a **distributed file system (DFS)**
  - Remote directories visible from local machine
- Third method – **World Wide Web**
  - A bit of a revision to the first method
  - Use browser to locate file/files and download /upload
  - **Anonymous** access doesn't require authentication



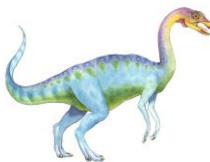


# Client-Server Model

---

- Sharing between a server (providing access to a file system via a network protocol) and a client (using the protocol to access the remote file system)
- Identifying each other via network ID can be spoofed, encryption can be expensive
- NFS an example
  - User auth info on clients and servers must match (UserIDs for example)
  - Remote file system mounted, file operations sent on behalf of user across network to server
  - Server checks permissions, file handle returned
  - Handle used for reads and writes until file closed

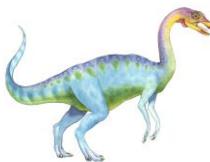




# Distributed Information Systems

- Aka **distributed naming services**, provide unified access to info needed for remote computing
- **Domain name system (DNS)** provides host-name-to-network-address translations for the Internet
- Others like **network information service (NIS)** provide user-name, password, userID, group information
- Microsoft's **common Internet file system (CIFS)** network info used with user auth to create network logins that server uses to allow or deny access
  - **Active directory** distributed naming service
  - **Kerberos-derived** network authentication protocol
- Industry moving toward **lightweight directory-access protocol (LDAP)** as secure distributed naming mechanism



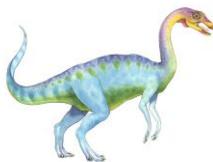


# Consistency Semantics

---

- Important criteria for evaluating file sharing-file systems
- Specify how multiple users are to access shared file simultaneously
  - When modifications of data will be observed by other users
  - Directly related to process synchronization algorithms, but atomicity across a network has high overhead (see Andrew File System)
- The series of accesses between file open and closed called **file session**
- **UNIX semantics**
  - Writes to open file **immediately** visible to others with file open
  - One mode of sharing allows users to share pointer to current I/O location in file
  - Single physical image, accessed exclusively, contention causes process delays
- **Session semantics** (Andrew file system (OpenAFS))
  - Writes to open file not visible during session, only at close
  - Can be several copies, each changed independently

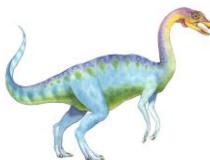




# The Sun Network File System (NFS)

- An implementation and a specification of a software system for accessing remote files across LANs (or WANs)
- The implementation originally part of SunOS operating system, now industry standard / very common
- Can use unreliable datagram protocol (UDP/IP) or TCP/IP, over Ethernet or other network

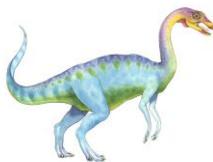




# NFS (Cont.)

- Interconnected workstations viewed as a set of independent machines with independent file systems, which allows sharing among these file systems in a transparent manner
  - A remote directory is **mounted** over a local file system directory
    - ▶ The mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory
  - Specification of the remote directory for the mount operation is **nontransparent**; the host name of the remote directory has to be provided
    - ▶ Files in the remote directory can then be accessed in a transparent manner
  - Subject to access-rights accreditation, potentially any file system (or directory within a file system), can be mounted remotely on top of any local directory

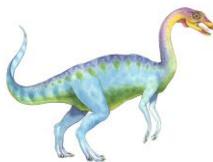




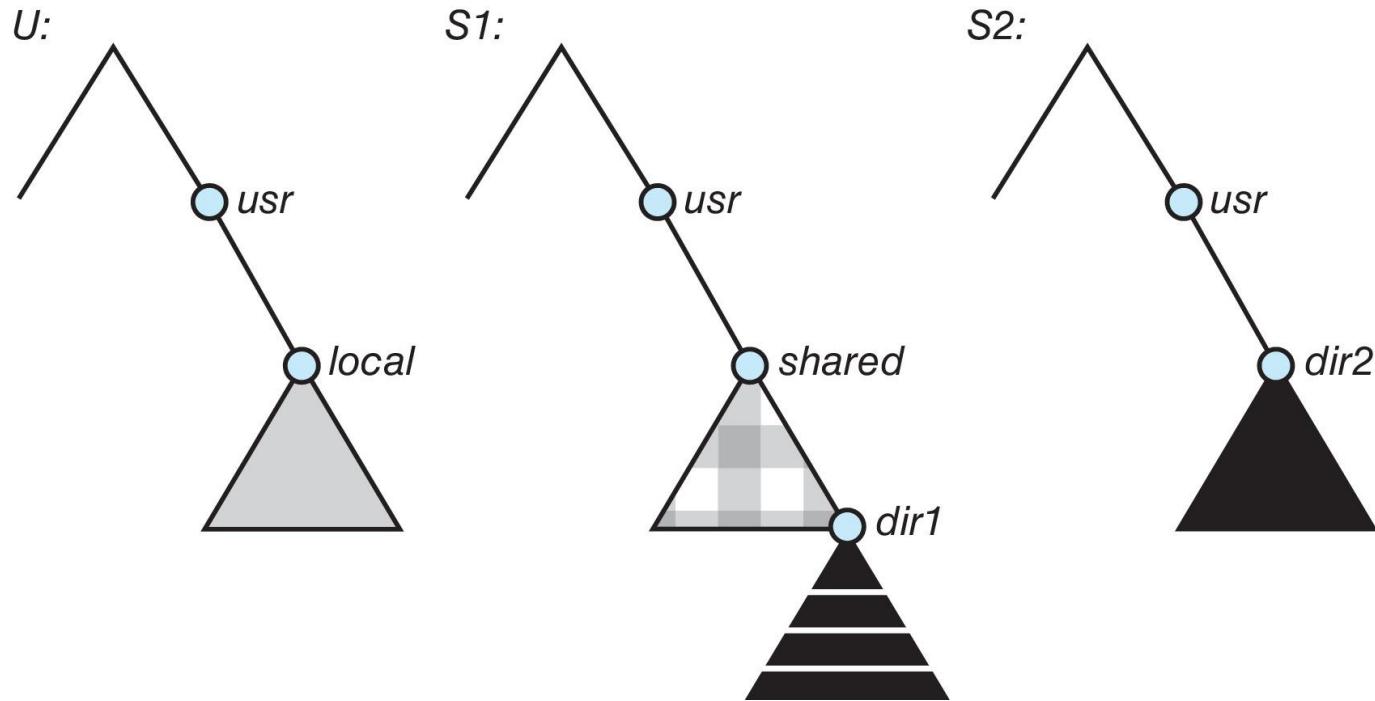
## NFS (Cont.)

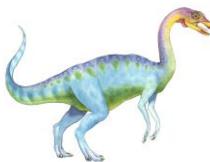
- NFS is designed to operate in a **heterogeneous** environment of different machines, operating systems, and network architectures; the NFS specifications independent of these media
- This independence is achieved through the use of **RPC** primitives built on top of an External Data Representation (XDR) protocol used between two implementation-independent interfaces
- The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote-file-access services





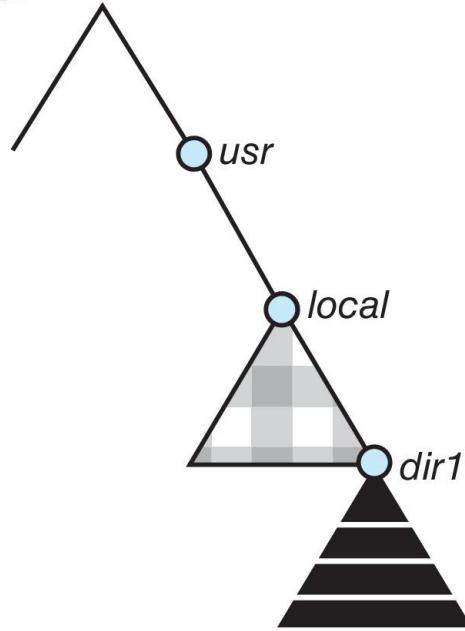
# Three Independent File Systems





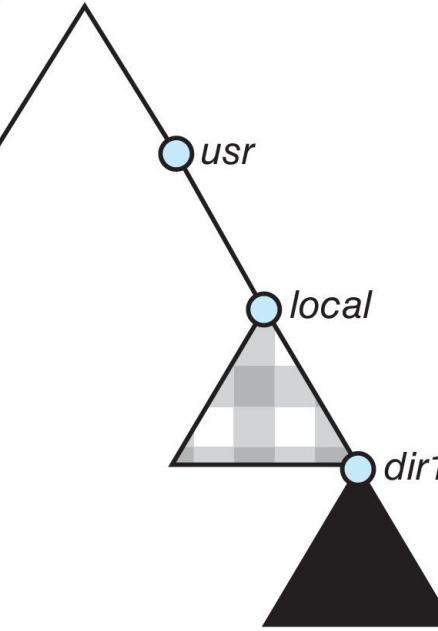
# Mounting in NFS

*U:*



(a)

*U:*

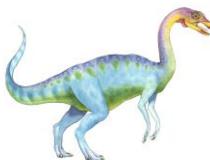


(b)

Mounts

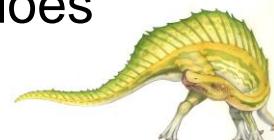
Cascading mounts





# NFS Mount Protocol

- Establishes initial logical connection between server and client
- Mount operation includes name of remote directory to be mounted and name of server machine storing it
  - Mount request is mapped to corresponding RPC and forwarded to mount server running on server machine
  - Export list – specifies local file systems that server exports for mounting, along with names of machines that are permitted to mount them
- Following a mount request that conforms to its export list, the server returns a file handle—a key for further accesses
  - File handle – a file-system identifier, and an inode number to identify the mounted directory within the exported file system
- The mount operation changes only the user's view and does not affect the server side



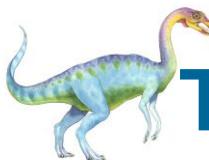


# NFS Protocol

---

- Provides a set of remote procedure calls for remote file operations:
  - searching for a file within a directory
  - reading a set of directory entries
  - manipulating links and directories
  - accessing file attributes
  - reading and writing files
- NFS servers are **stateless**; each request has to provide a full set of arguments (NFS V4 is newer, less used – very different, stateful)
- Modified data must be committed to the server's disk before results are returned to the client (lose advantages of caching)
- The NFS protocol does **not** provide concurrency-control mechanisms





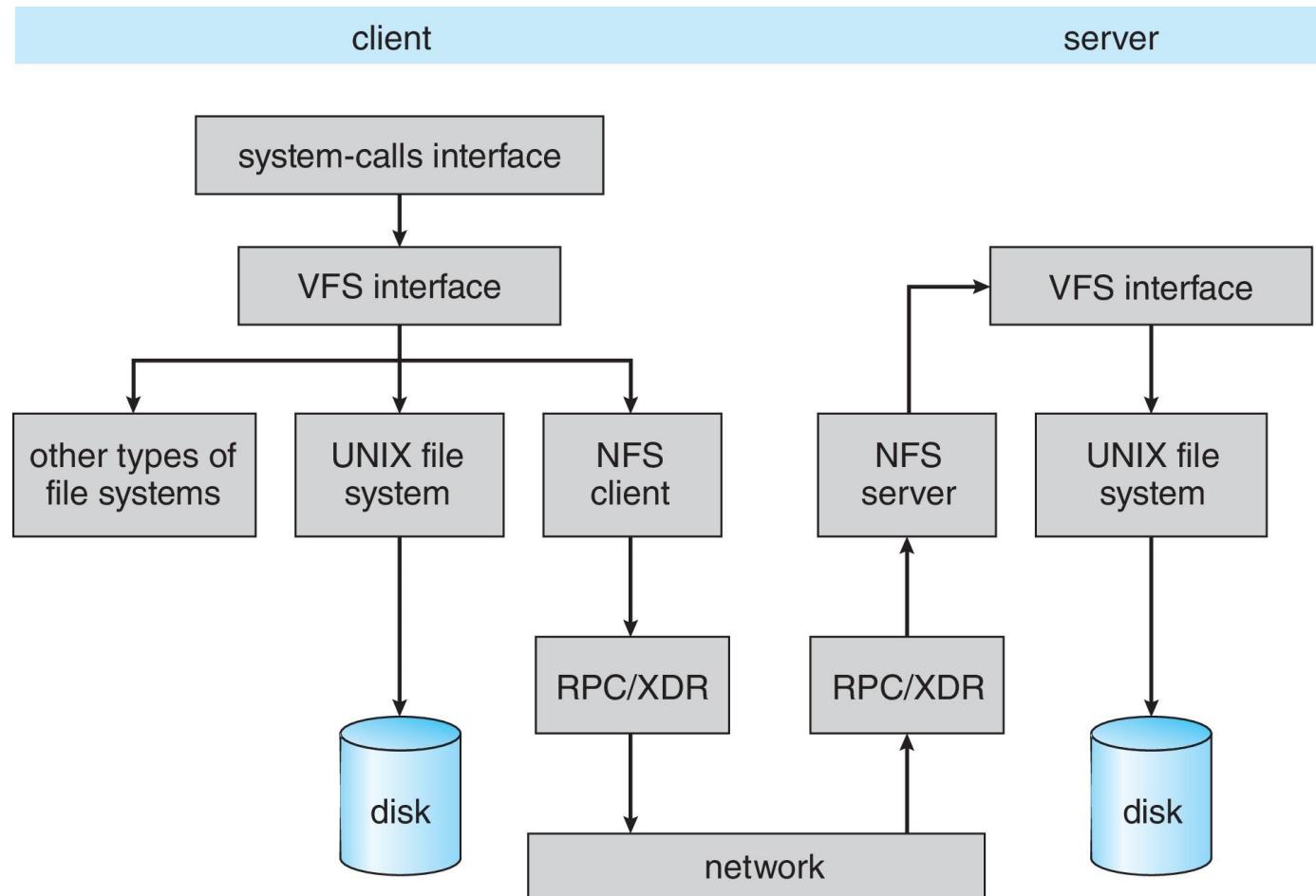
# Three Major Layers of NFS Architecture

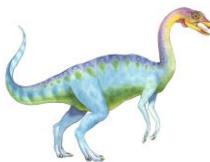
- UNIX file-system interface (based on the **open**, **read**, **write**, and **close** calls, and **file descriptors**)
- Virtual File System (VFS) layer – distinguishes local files from remote ones, and local files are further distinguished according to their file-system types
  - The VFS activates file-system-specific operations to handle local requests according to their file-system types
  - Calls the NFS protocol procedures for remote requests
- NFS service layer – bottom layer of the architecture
  - Implements the NFS protocol





# Schematic View of NFS Architecture





# NFS Path-Name Translation

- Performed by breaking the path into component names and performing a separate NFS lookup call for every pair of component name and directory vnode
- To make lookup faster, a directory name lookup cache on the client side holds the vnodes for remote directory names





# NFS Remote Operations

- Nearly one-to-one correspondence between regular UNIX system calls and the NFS protocol RPCs (except opening and closing files)
- NFS adheres to the remote-service paradigm, but employs buffering and caching techniques for the sake of performance
- **File-blocks cache** – when a file is opened, the kernel checks with the remote server whether to fetch or revalidate the cached attributes
  - Cached file blocks are used only if the corresponding cached attributes are up to date
- **File-attribute cache** – the attribute cache is updated whenever new attributes arrive from the server
- Clients do not free delayed-write blocks until the server confirms that the data have been written to disk



# End of Chapter 15

