Disk Scheduling

Dr. Pankaj Kumar
Associate Professor – CSE
SRMGPC Lucknow

Background

Magnetic disks provide bulk of secondary storage of modern computers

- 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

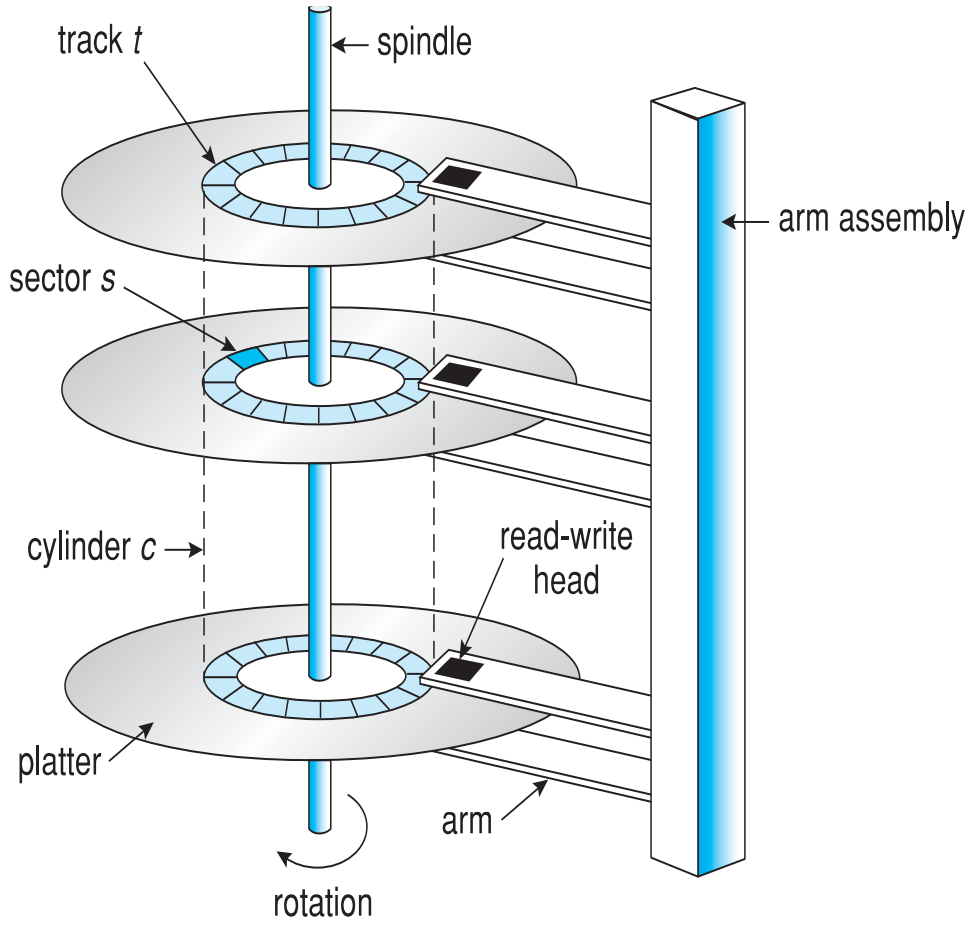
Disks can be removable

Drive attached to computer via **I/O bus**

- Busses vary, including **EIDE, ATA, SATA, USB, Fibre Channel, SCSI, SAS, Firewire**
- **Host controller** in computer uses bus to talk to **disk controller** built into drive or storage array

Hard Disk Mechanism

- 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 / (RPM / 60) = 60 / RPM
  - Average latency = ½ latency



Hard Disk Performance

Access Latency = Average access time = average seek time + average latency

For fastest disk 3ms + 2ms = 5ms

For slow disk 9ms + 5.56ms = 14.56ms

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 =

5ms + 4.17ms + 0.1ms + transfer time =

Transfer time = 4KB / 1Gb/s * 8Gb / GB * 1GB / 1024^2KB = 32 / (1024^2) = 0.031 ms

Average I/O time for 4KB block = 9.27ms + .031ms = 9.301ms

Hard Disk Performance

Disk drives are addressed as large 1-dimensional arrays of **logical blocks**, where the logical block is the smallest unit of transfer

- Low-level formatting creates **logical blocks** on physical media

The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially

- Sector 0 is the first sector of the first track on the outermost cylinder
- Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost
- Logical to physical address should be easy
  - Except for bad sectors
  - Non-constant # of sectors per track via constant angular velocity

Disk 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 ≈ 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

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

Disk Scheduling

- **Seek Time:** Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.
- **Rotational Latency:** Rotational Latency is the time taken by the desired sector of disk to rotate into a position so that it can access the read/write heads. So the disk scheduling algorithm that gives minimum rotational latency is better.
- **Transfer Time:** Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and number of bytes to be transferred.
- **Disk Response Time:** Response Time is the average of time spent by a request waiting to perform its I/O operation. Average Response time is the response time of the all requests. Variance Response Time is measure of how individual request are serviced with respect to average response time. So the disk scheduling algorithm that gives minimum variance response time is better.
- **Disk Access Time:** Disk Access Time is:

  Disk Access Time = Seek Time + Rotational Latency + Transfer Time

First Come First Serve (FCFS)

FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue. Let us understand this with the help of an example.

Example:

Suppose the order of request is- (82,170,43,140,24,16,190)

And current position of Read/Write head is : 50

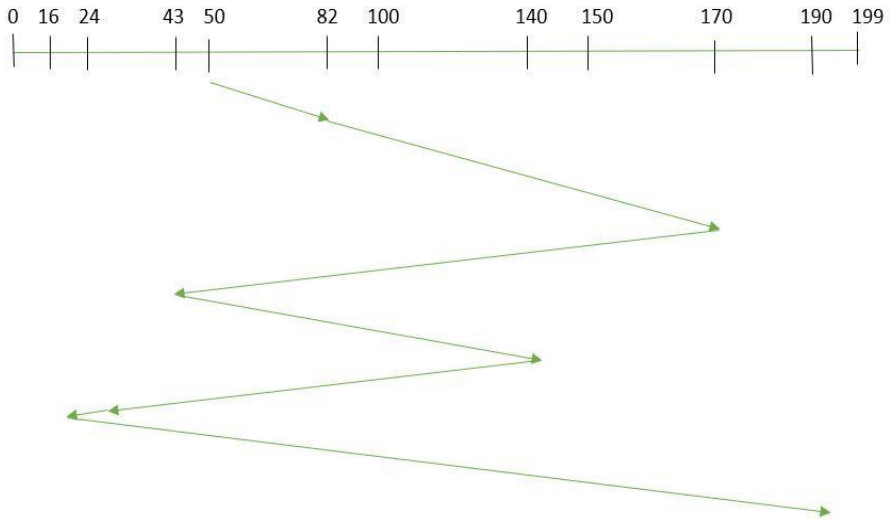
So, total seek time: =(82-50)+(170-82)+(170-43)+(140-43)+(140-24)+(24-16)+(190-16) =642

Advantages:

- Every request gets a fair chance
- No indefinite postponement

Disadvantages:

- Does not try to optimize seek time
- May not provide the best possible service



Shortest Seek Time First (SSTF)

In SSTF (Shortest Seek Time First), requests having shortest seek time are executed first. So, the seek time of every request is calculated in advance in the queue and then they are scheduled according to their calculated seek time. As a result, the request near the disk arm will get executed first.

Example:

Suppose the order of request is- (82,170,43,140,24,16,190) And current position of Read/Write head is : 50

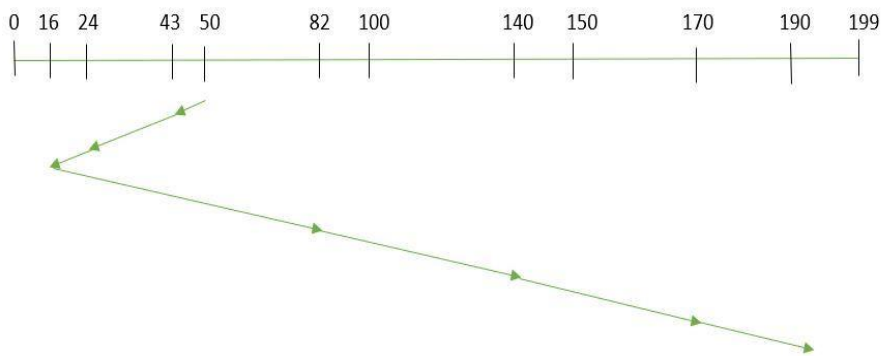
So, total seek time: = (50-43)+(43-24)+(24-16)+(82-16)+(140-82)+(170-40)+(190-170) =208

Advantages:

- Average Response Time decreases
- Throughput increases

Disadvantages:

- Overhead to calculate seek time in advance
- Can cause Starvation for a request if it has higher seek time as compared to incoming requests
- High variance of response time as SSTF favors only some requests



(SCAN)

In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works as an elevator and hence also known as **elevator algorithm.** As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

Example:

Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **"towards the larger value".**

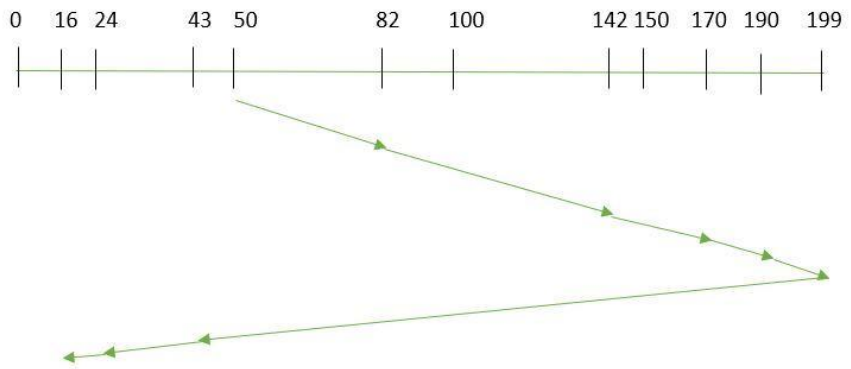
So, total seek time: =(199-50)+(199-16) =332

Advantages:

- High throughput
- Low variance of response time
- Average response time

Disadvantages:

- Long waiting time for requests for locations just visited by disk arm



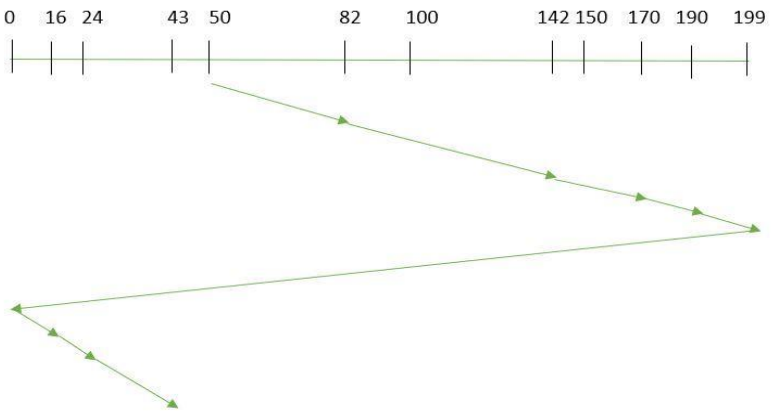
(CSCAN)

In SCAN algorithm, the disk arm again scans the path that has been scanned, after reversing its direction. So, it may be possible that too many requests are waiting at the other end or there may be zero or few requests pending at the scanned area.

These situations are avoided in *CSCAN* algorithm in which the disk arm instead of reversing its direction goes to the other end of the disk and starts servicing the requests from there. So, the disk arm moves in a circular fashion and this algorithm is also similar to SCAN algorithm and hence it is known as C-SCAN (Circular SCAN).

Example:

Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **"towards the larger value".**



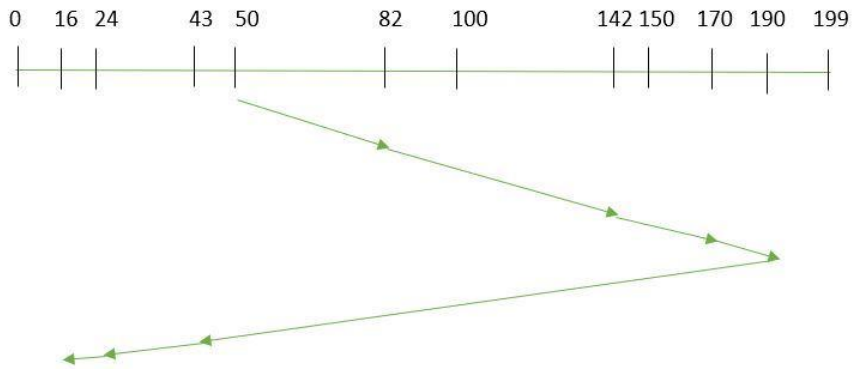
So, total seek time: =(199-50)+(199-0)+(43-0)=391

(LOOK)

It is similar to the SCAN disk scheduling algorithm except for the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only. Thus it prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

Example:

Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **"towards the larger value".**



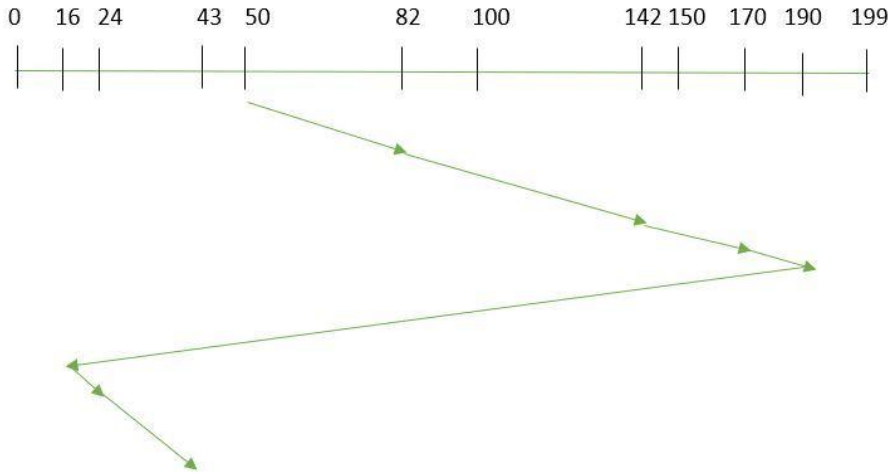
So, total seek time: =(190-50)+(190-16) =314

(CLOOK)

It is similar to the SCAN disk scheduling algorithm except for the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only. Thus it prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

Example:

Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **"towards the larger value".**



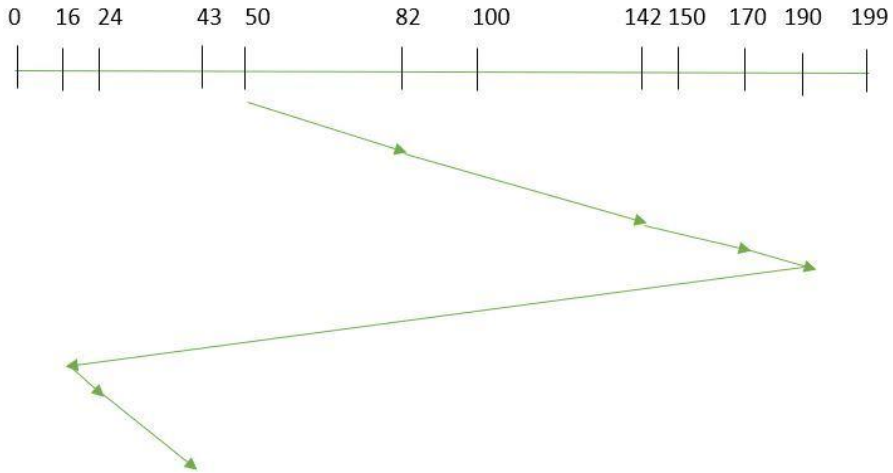
So, total seek time: =(190-50)+(190-16)+(43-16) =341

(CLOOK)

It is similar to the SCAN disk scheduling algorithm except for the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only. Thus it prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

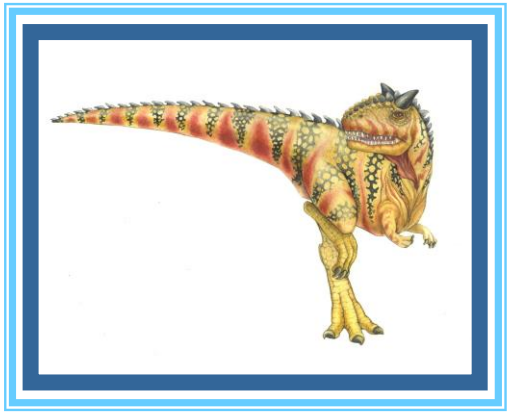
Example:

Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **"towards the larger value".**



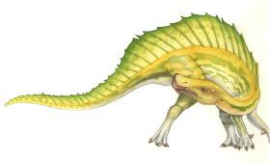
So, total seek time: =(190-50)+(190-16)+(43-16) =341

Chapter 12: File System Implementation



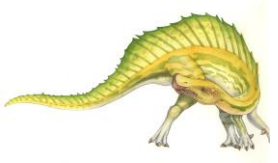
Chapter 12: File System Implementation

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



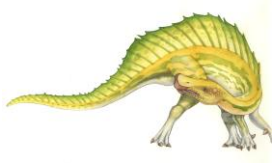
Objectives

- To describe the details of implementing local file systems and directory structures
- To describe the implementation of remote file systems
- To discuss block allocation and free-block algorithms and trade-offs

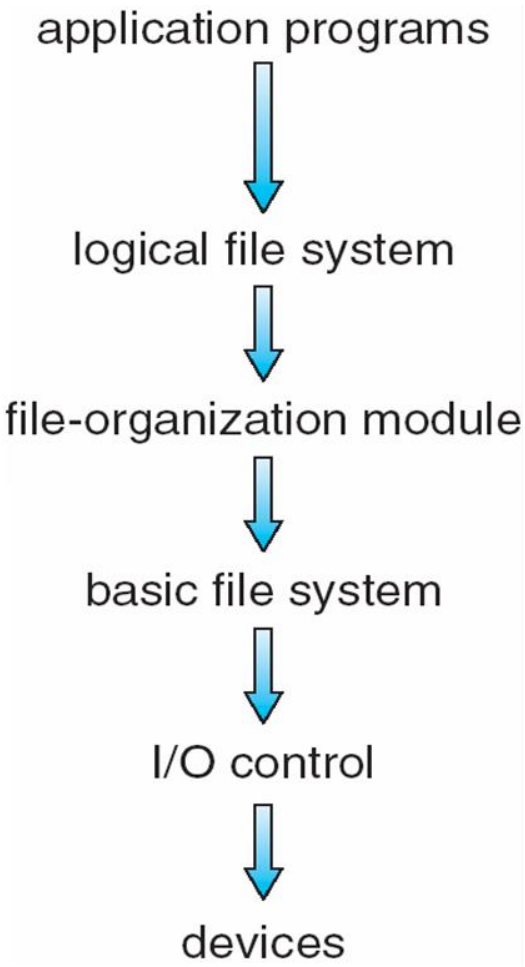


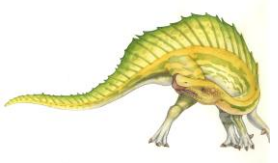
File-System Structure

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



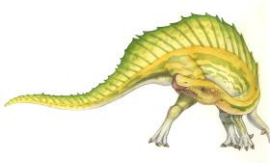
Layered File System





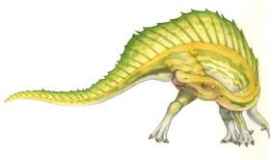
File System Layers

- **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
- **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 performanceTranslates file name into file number, file handle, location by maintaining file control blocks (**inodes** in UNIX)
  - Logical layers can be implemented by any coding method according to OS designer



File System Layers (Cont.)

- Many file systems, sometimes many within an operating system
  - 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 40 types, with **extended file system** ext2 and ext3 leading; plus distributed file systems, etc.)
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE

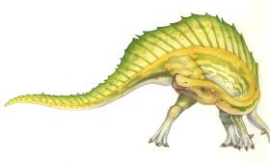
File-System Implementation

- We have system calls at the API level, but how do we implement their functions?
  - On-disk and in-memory structures
- **Boot control block** contains info needed by system to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
- **Volume control block (superblock, master file table)** contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- Directory structure organizes the files
  - Names and inode numbers, master file table

File-System Implementation (Cont.)

- Per-file **File Control Block (FCB)** contains many details about the file
  - inode number, permissions, size, dates
  - NFTS stores into in master file table using relational DB structures

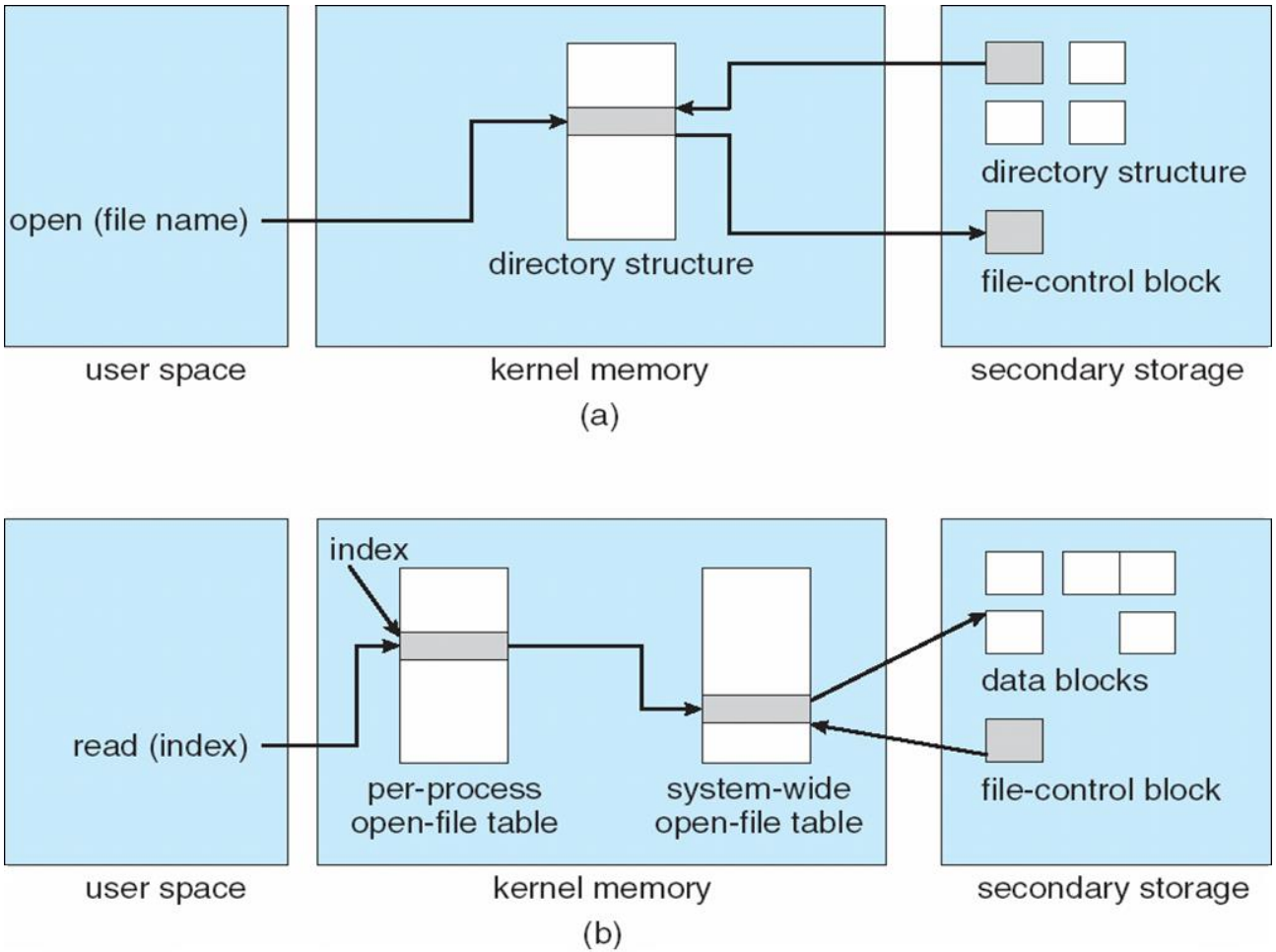
file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

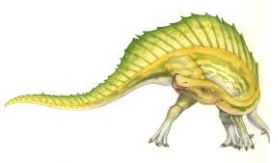


In-Memory File System Structures

- Mount table storing file system mounts, mount points, file system types
- The following figure illustrates the necessary file system structures provided by the operating systems
- Figure 12-3(a) refers to opening a file
- Figure 12-3(b) refers to reading a file
- Plus buffers hold data blocks from secondary storage
- Open returns a file handle for subsequent use
- Data from read eventually copied to specified user process memory address

In-Memory File System Structures



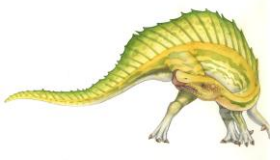


Partitions and Mounting

- 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
- 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

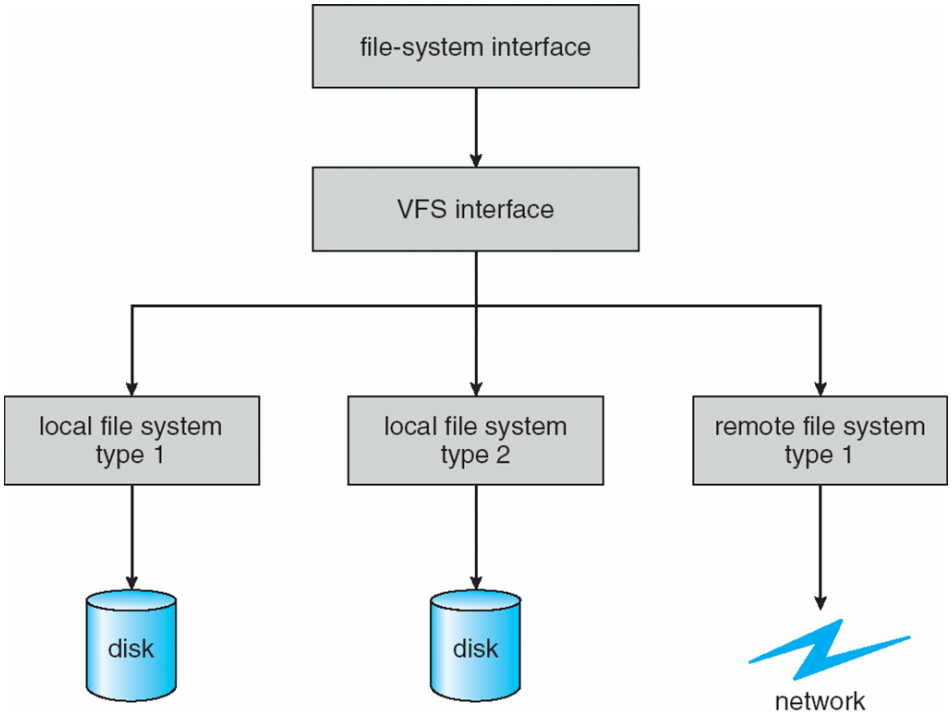
Virtual File Systems

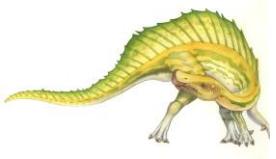
- **Virtual File Systems** (**VFS**) on Unix provide an object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - Separates file-system generic operations from implementation details
  - Implementation can 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



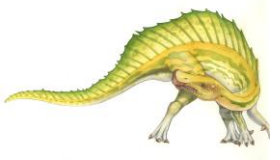


Virtual File System Implementation

- 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

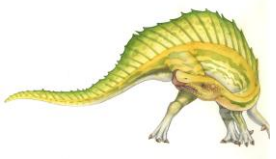
Directory Implementation

- **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



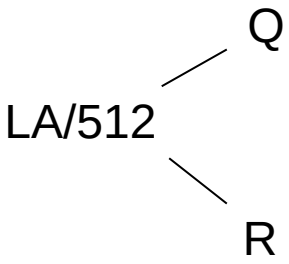
Allocation Methods - Contiguous

- 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**

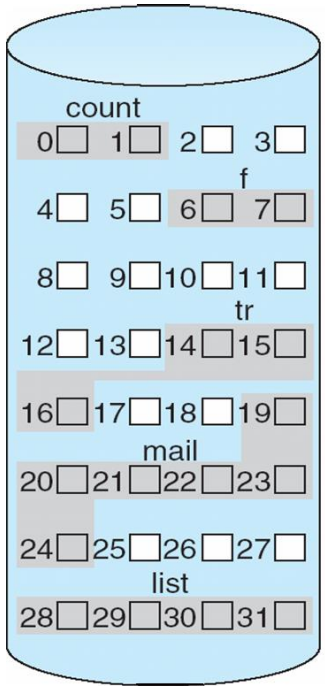


Contiguous Allocation

- Mapping from logical to physical



Block to be accessed = Q + starting address
Displacement into block = R



directory

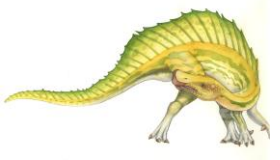
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

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 a contiguous block of disks
  - Extents are allocated for file allocation
  - A file consists of one or more extents

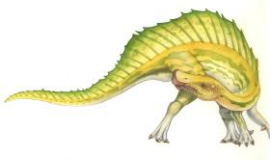
Allocation Methods - Linked

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



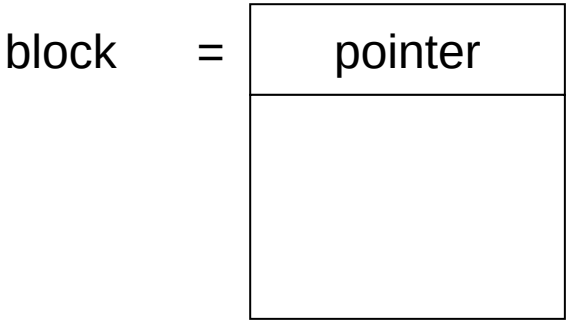
Allocation Methods – Linked (Cont.)

- FAT (File Allocation Table) variation
  - Beginning of volume has table, indexed by block number
  - Much like a linked list, but faster on disk and cacheable
  - New block allocation simple

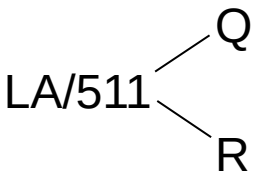


Linked Allocation

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

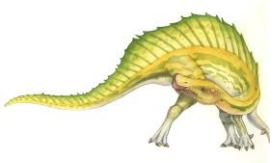


- Mapping

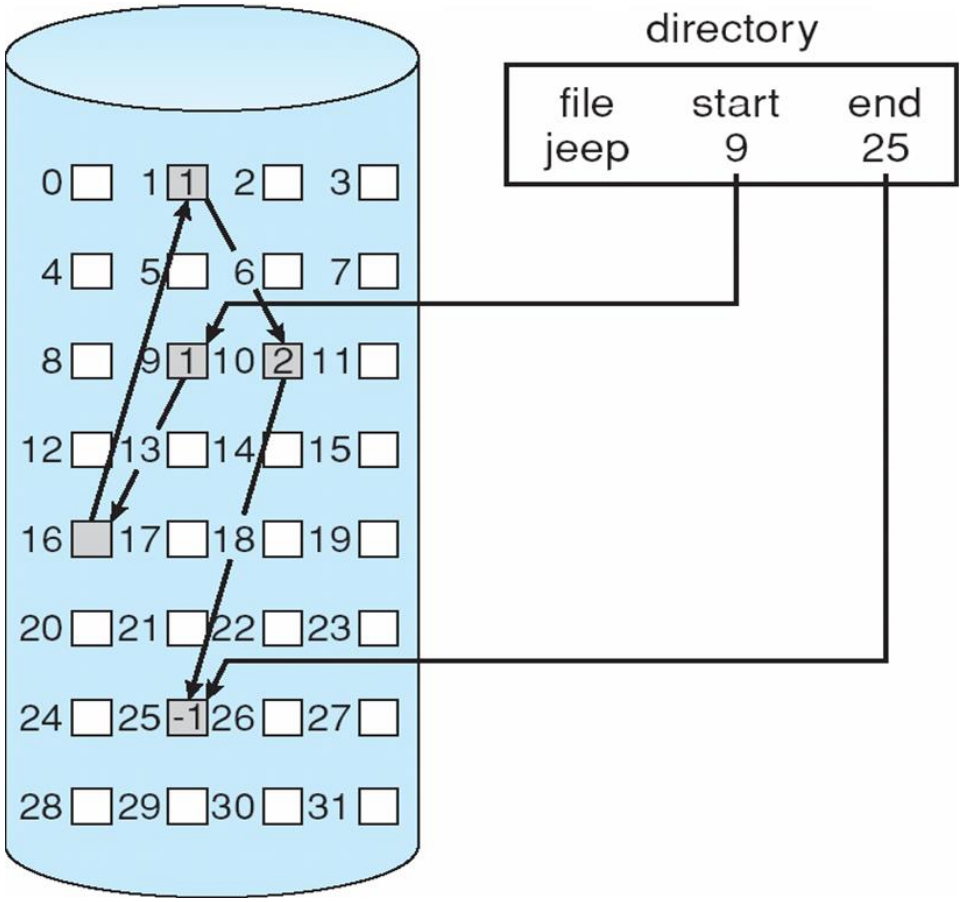


Block to be accessed is the Qth block in the linked chain of blocks representing the file.

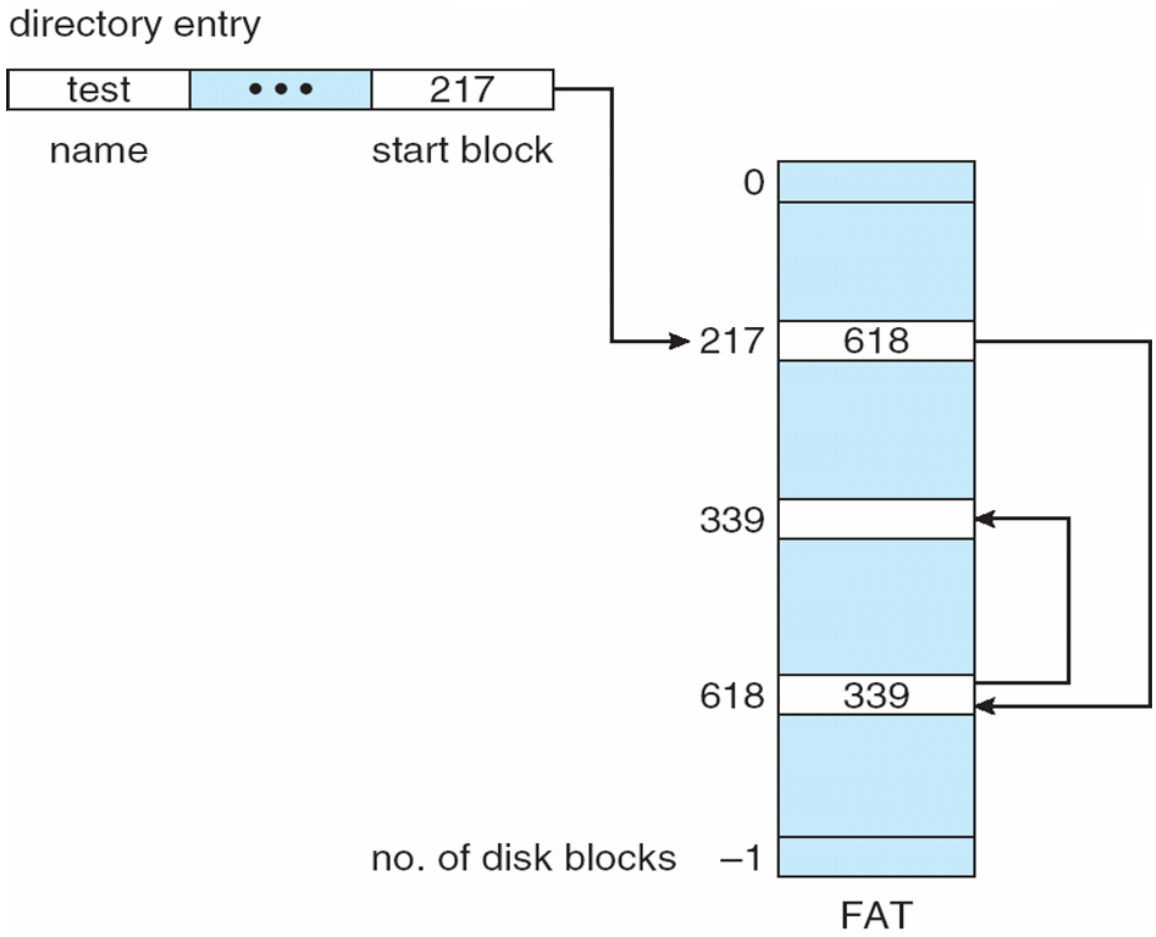
Displacement into block = R + 1



Linked Allocation



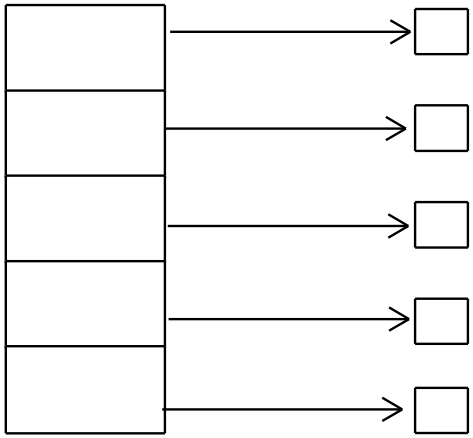
File-Allocation Table

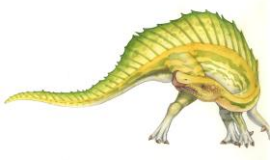


Allocation Methods - Indexed

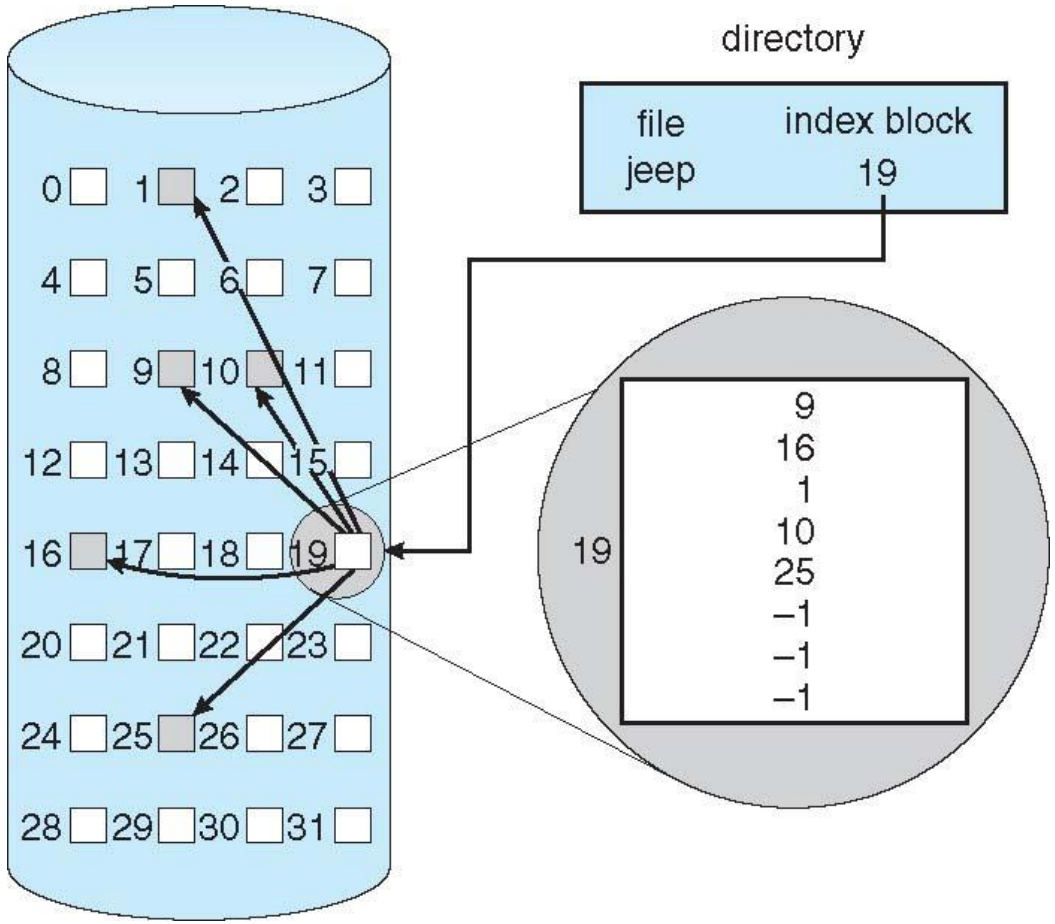
- **Indexed allocation**
  - Each file has its own **index block**(s) of pointers to its data blocks

- Logical view



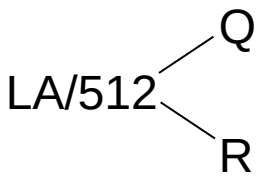


Example of Indexed Allocation

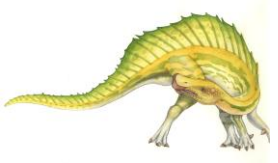


Indexed Allocation (Cont.)

- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K bytes and block size of 512 bytes. We need only 1 block for index table



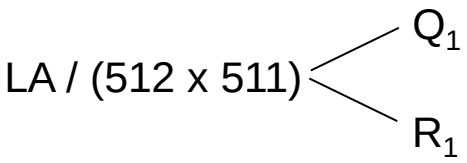
Q = displacement into index table
R = displacement into block



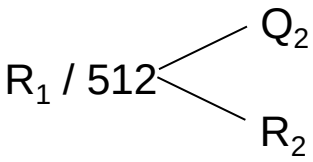
Indexed Allocation – Mapping (Cont.)

- Mapping from logical to physical in a file of unbounded length (block size of 512 words)

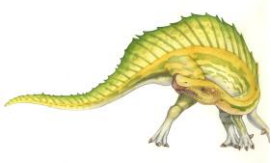
- Linked scheme – Link blocks of index table (no limit on size)



Q_1 = block of index table
R_1 is used as follows:

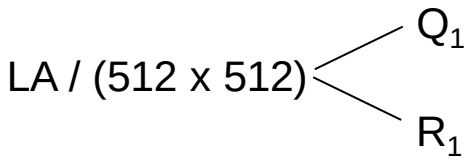


Q_2 = displacement into block of index table
R_2 displacement into block of file:

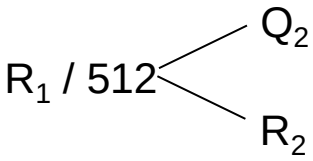


Indexed Allocation – Mapping (Cont.)

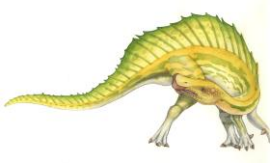
- Two-level index (4K blocks could store 1,024 four-byte pointers in outer index -> 1,048,567 data blocks and file size of up to 4GB)



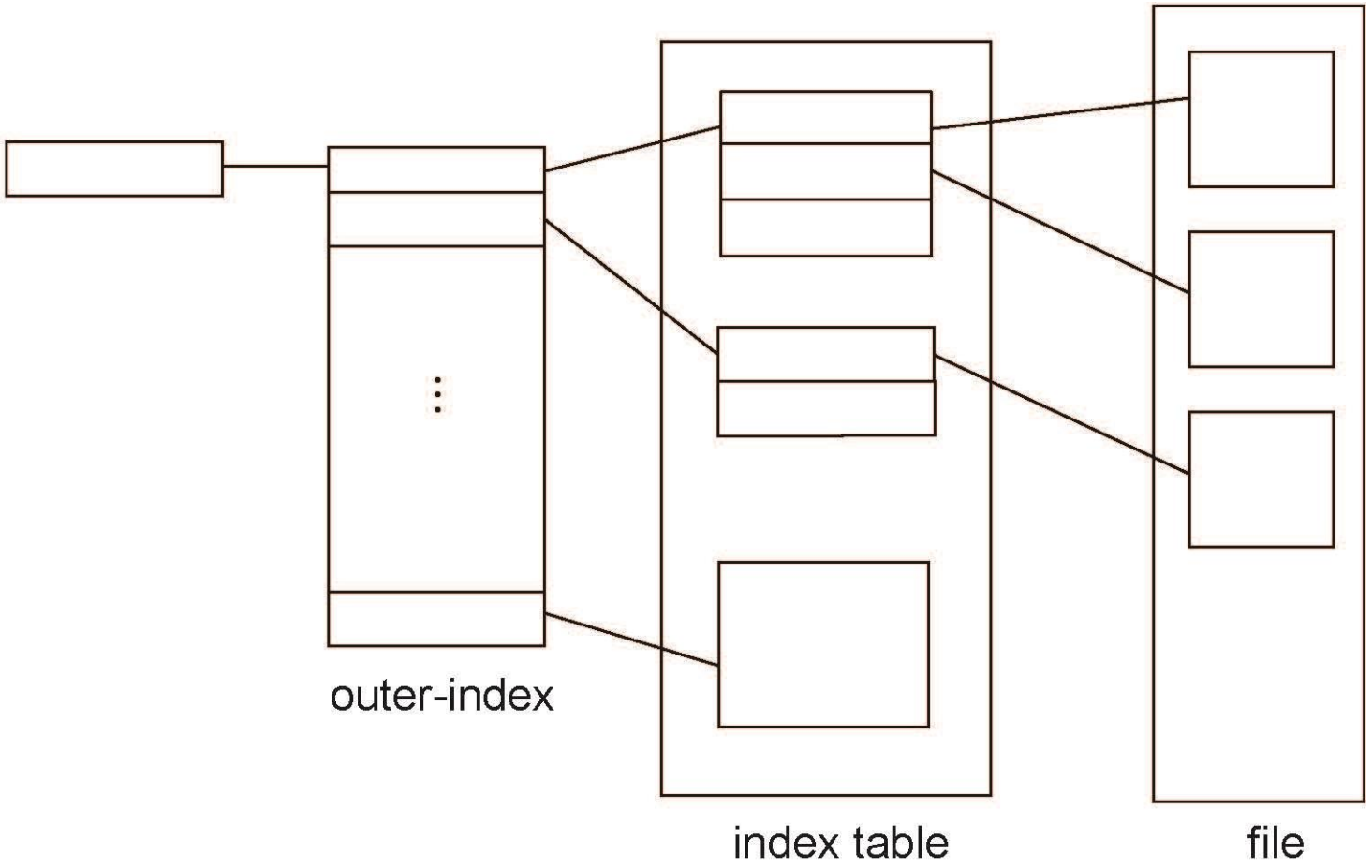
Q_1 = displacement into outer-index
R_1 is used as follows:



Q_2 = displacement into block of index table
R_2 displacement into block of file:

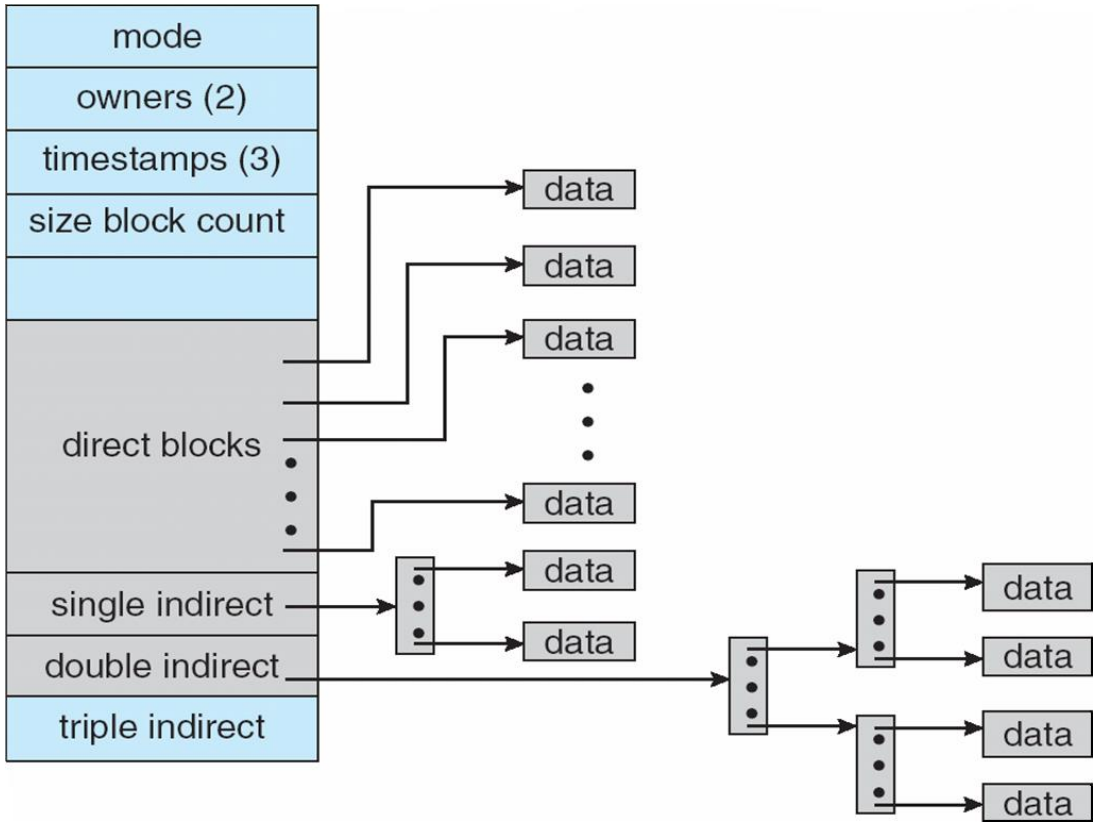


Indexed Allocation – Mapping (Cont.)

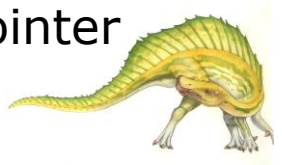


Combined Scheme: UNIX UFS

4K bytes per block, 32-bit addresses



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



Performance

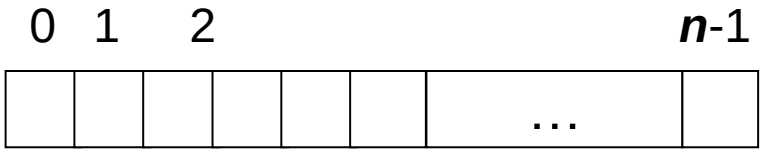
- Best method depends on file access type
  - Contiguous great for sequential and random
- Linked good for sequential, not random
- Declare access type at creation -> select either contiguous or linked
- Indexed more complex
  - Single block access could require 2 index block reads then data block read
  - Clustering can help improve throughput, reduce CPU overhead

Performance (Cont.)

- Adding instructions to the execution path to save one disk I/O is reasonable
  - Intel Core i7 Extreme Edition 990x (2011) at 3.46Ghz = 159,000 MIPS
    - http://en.wikipedia.org/wiki/Instructions_per_second
  - Typical disk drive at 250 I/Os per second
    - 159,000 MIPS / 250 = 630 million instructions during one disk I/O
  - Fast SSD drives provide 60,000 IOPS
    - 159,000 MIPS / 60,000 = 2.65 millions instructions during one disk 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)

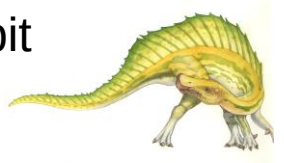


$$\text{bit}[i] = \begin{cases} 1 \Rightarrow \text{block}[i] \text{ free} \\ 0 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

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)

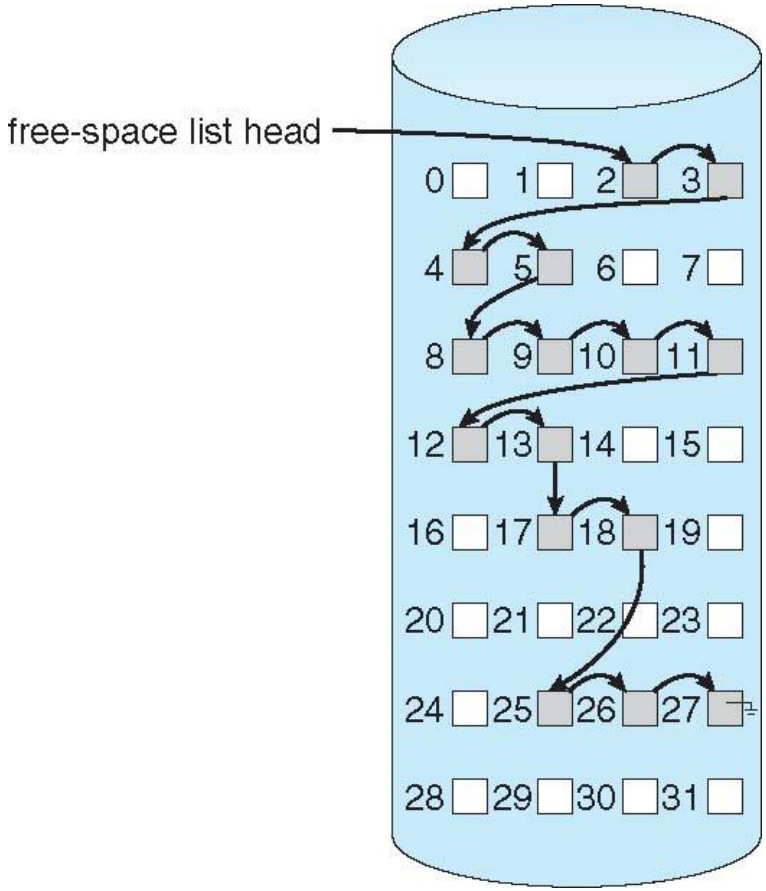
    $n = 2^{40}/2^{12} = 2^{28}$ bits (or 32MB)

    if clusters of 4 blocks -> 8MB of memory

- Easy to get contiguous files

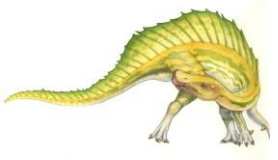
Linked Free Space List on Disk

- 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

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

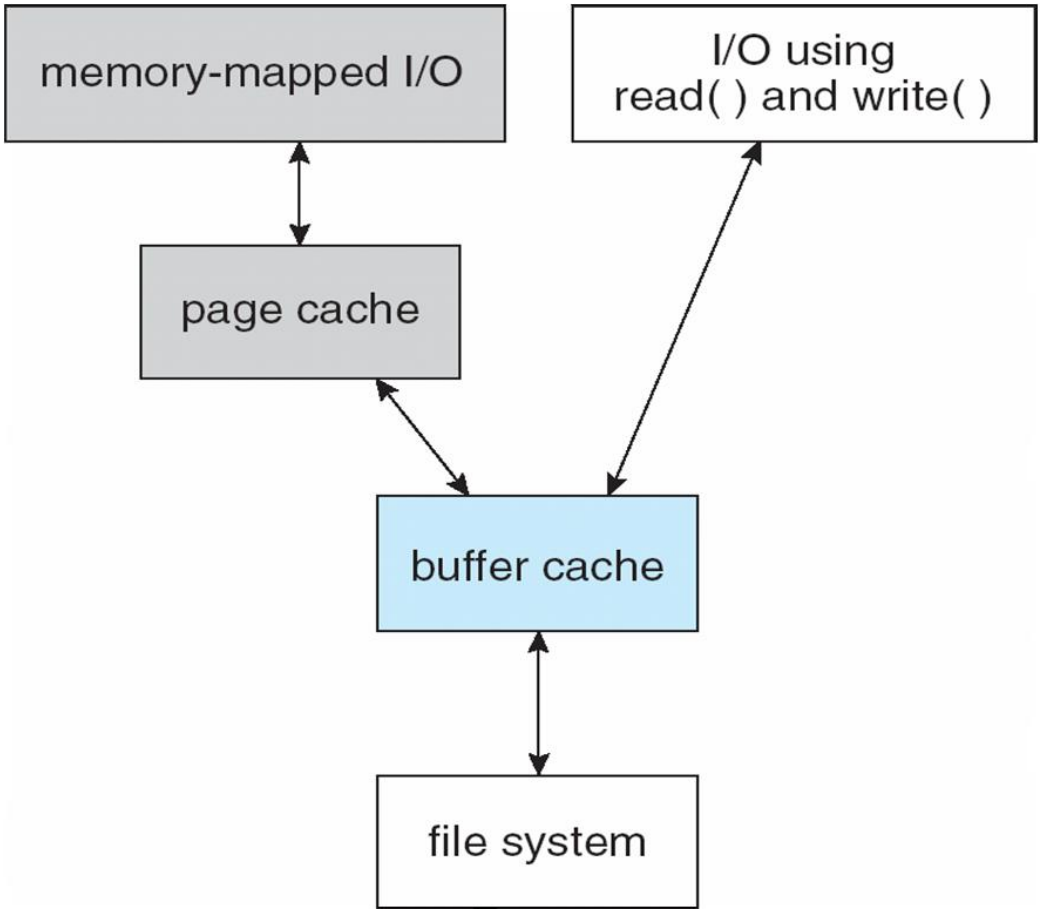
Efficiency and Performance (Cont.)

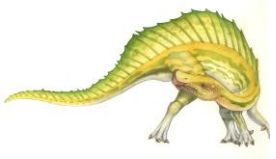
- Performance
  - Keeping data and metadata close together
  - **Buffer cache** – separate section of main memory for frequently used blocks
  - **Synchronous** writes sometimes requested by apps or needed by OS
    - No buffering / caching – writes must hit disk before acknowledgement
    - **Asynchronous** 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



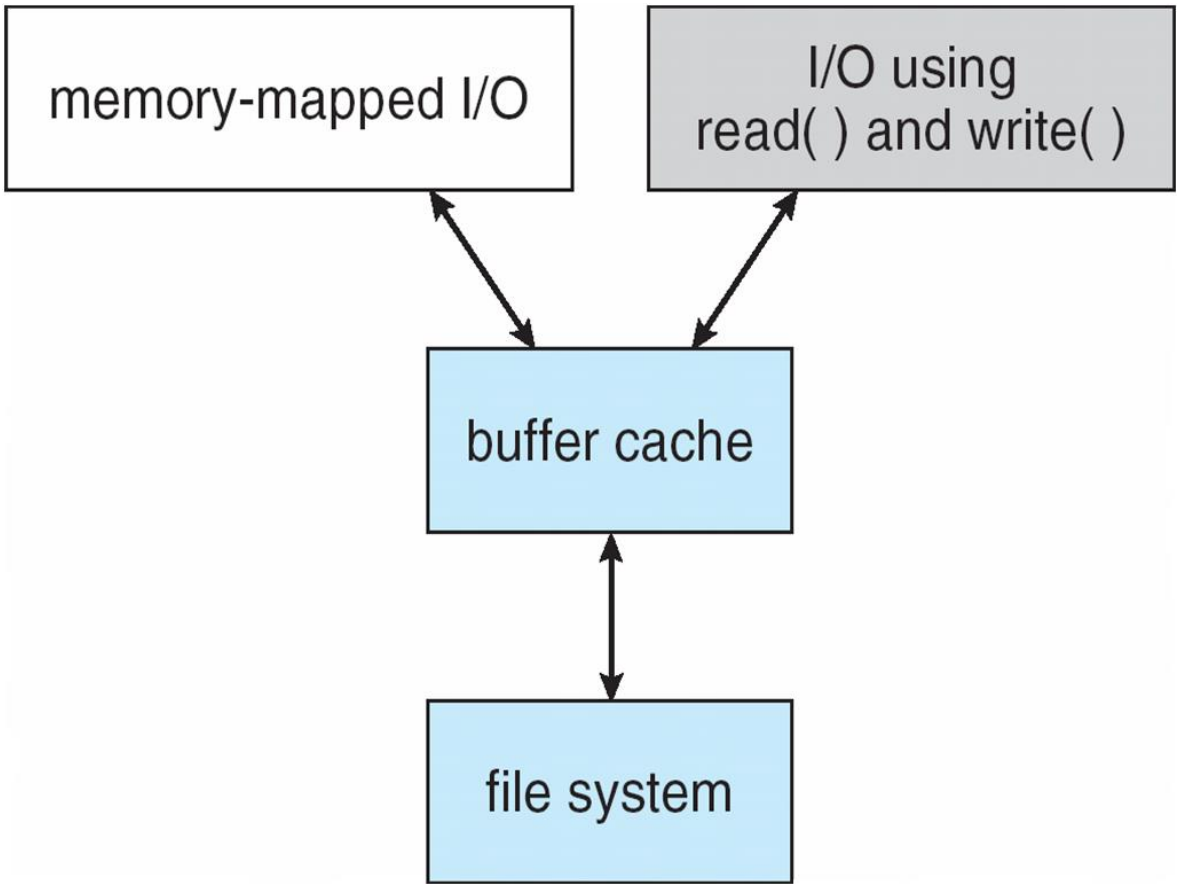


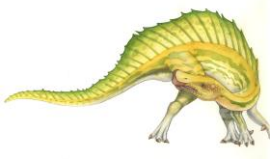
Unified Buffer Cache

- 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



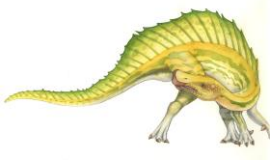


Recovery

- **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

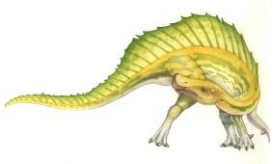
Log Structured File Systems

- **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



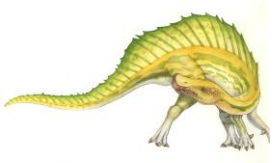
The Sun Network File System (NFS)

- An implementation and a specification of a software system for accessing remote files across LANs (or WANs)
- The implementation is part of the Solaris and SunOS operating systems running on Sun workstations using an unreliable datagram protocol (UDP/IP protocol and Ethernet



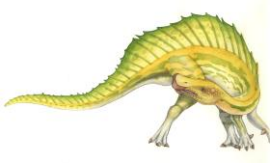
NFS (Cont.)

- 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

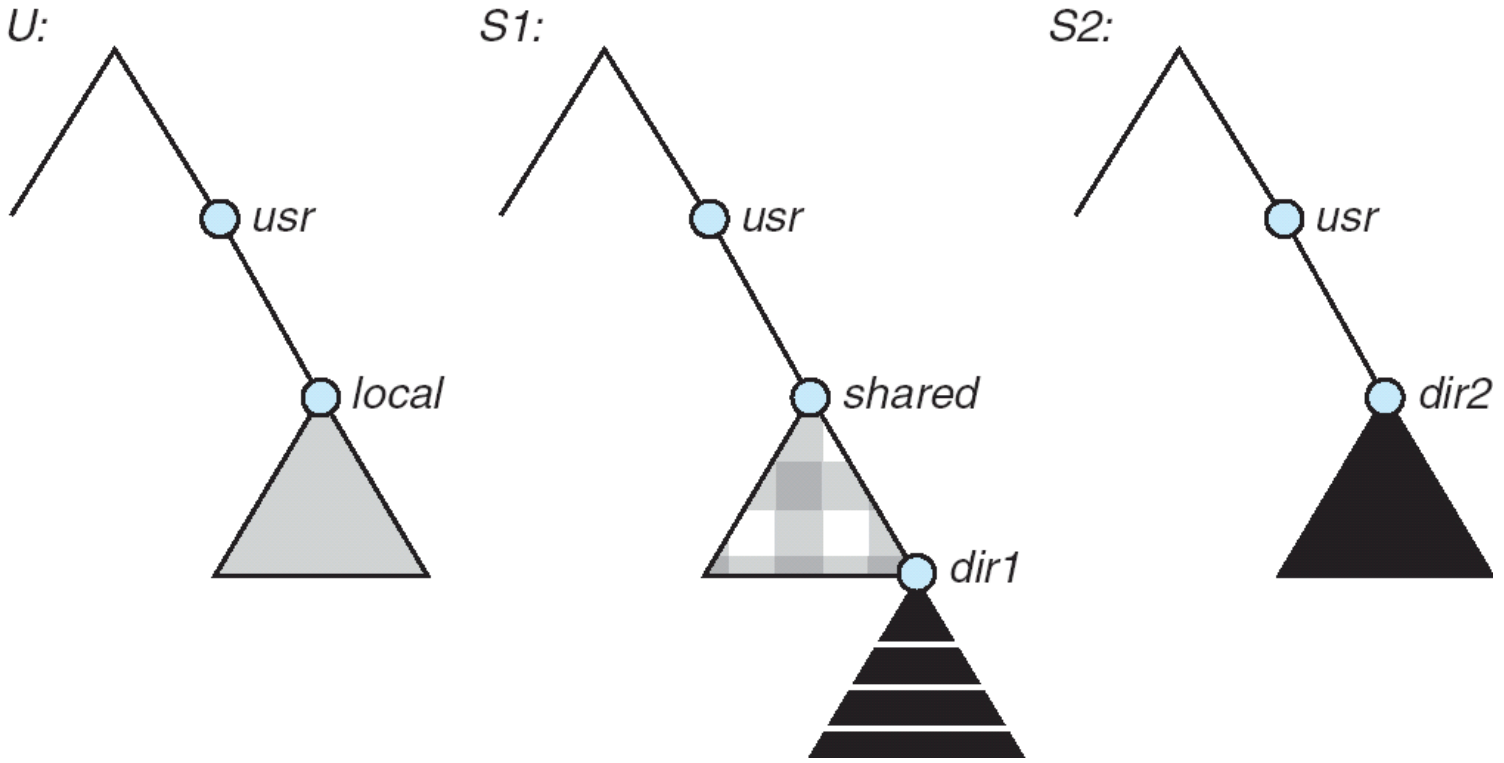


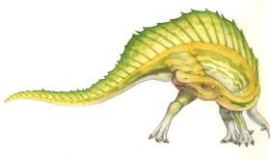
NFS (Cont.)

- 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

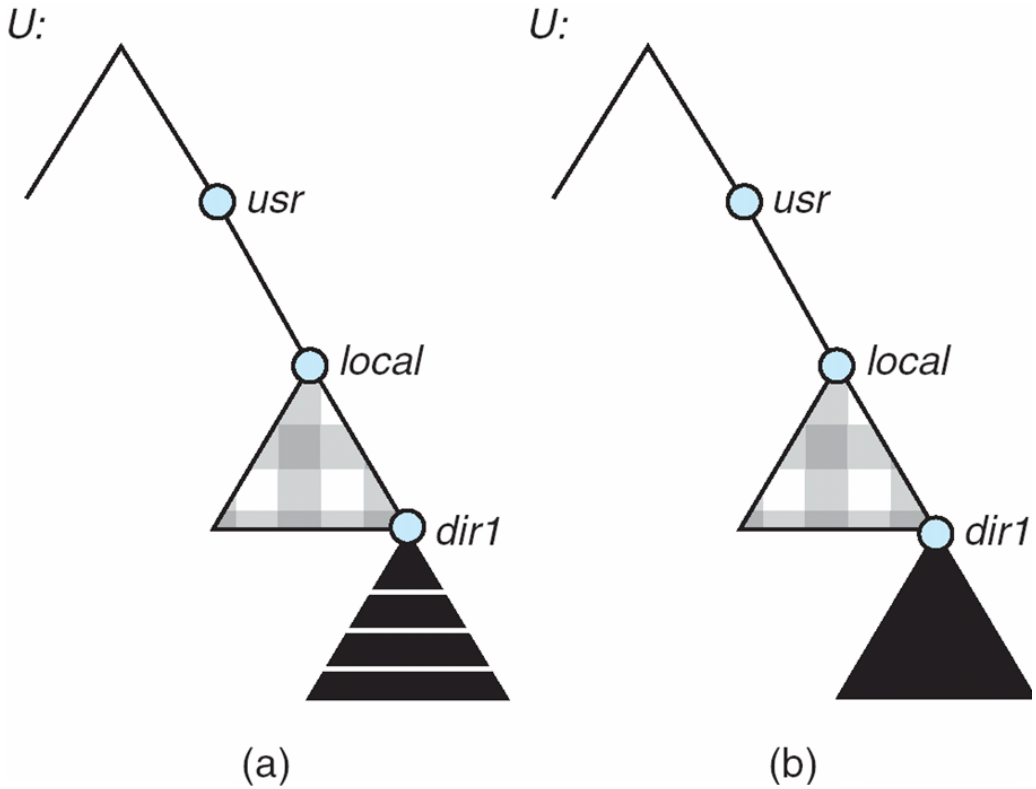


Three Independent File Systems



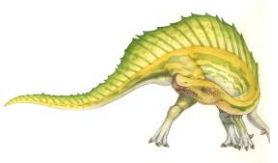


Mounting in NFS



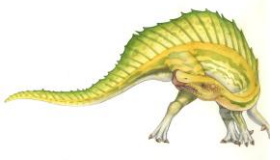
Mounts

Cascading mounts



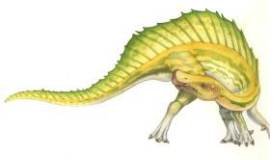
NFS Mount Protocol

- 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



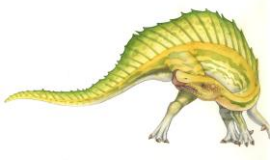
NFS Protocol

- Provides a set of remote procedure calls for remote file operations. The procedures support the following 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 just coming available – 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

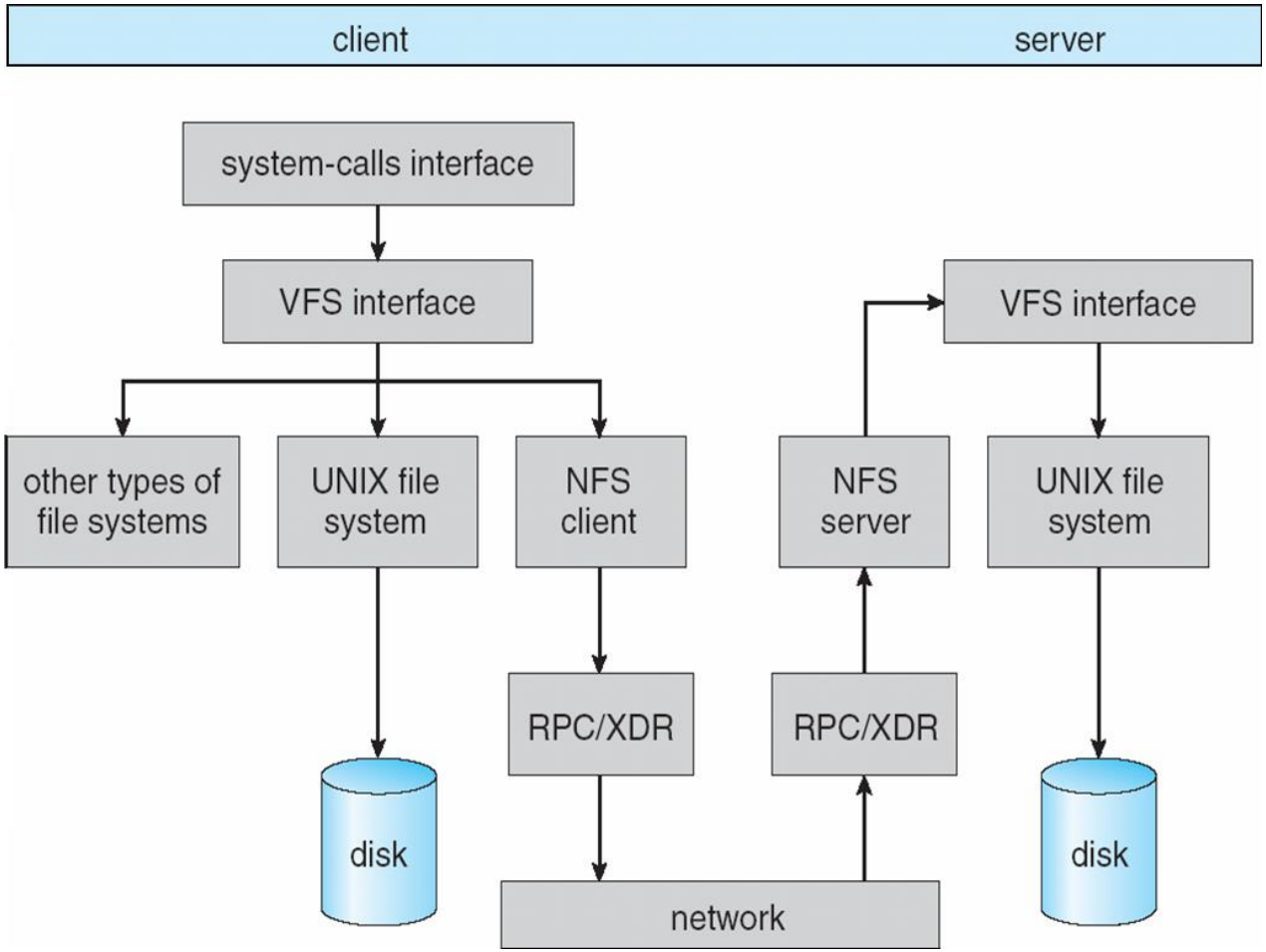


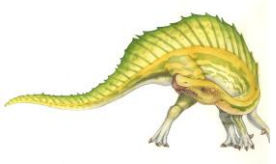
Three Major Layers of NFS Architecture

- 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



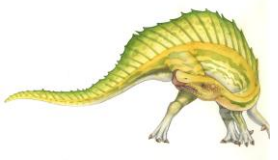
Schematic View of NFS Architecture





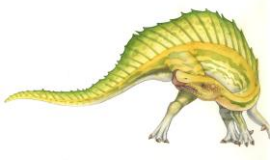
NFS Path-Name Translation

- 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's side holds the vnodes for remote directory names



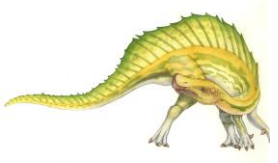
NFS Remote Operations

- 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

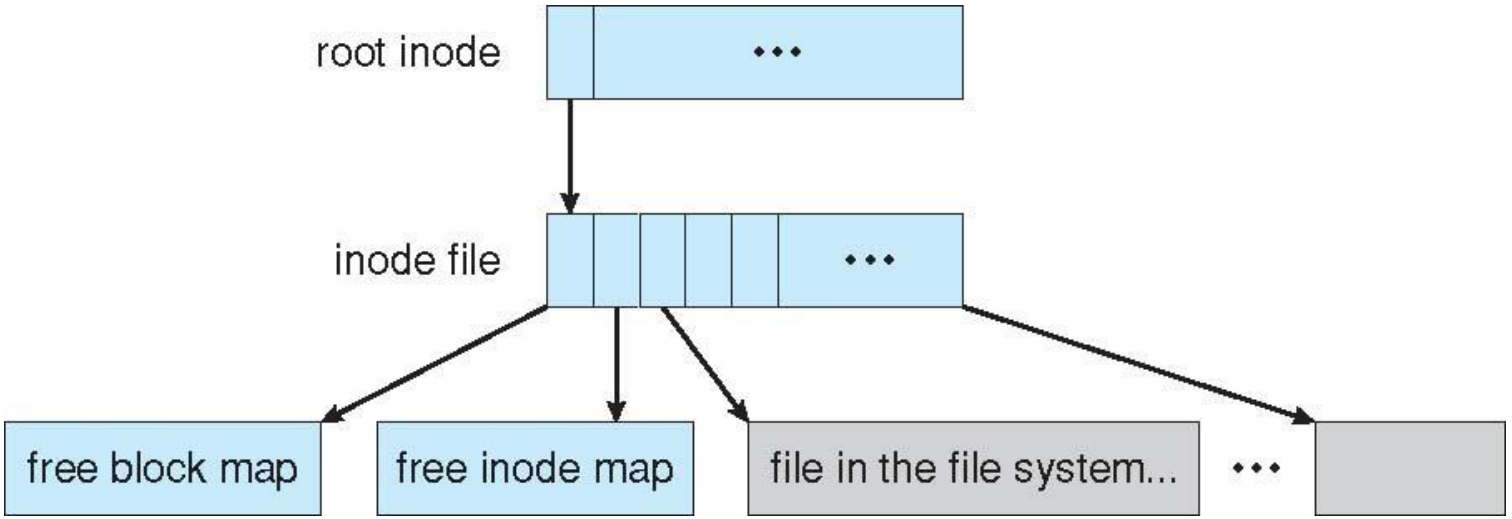


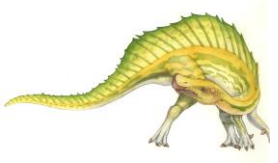
Example: WAFL File System

- Used on Network Appliance “Filers” – distributed file system appliances
- “Write-anywhere file layout”
- 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

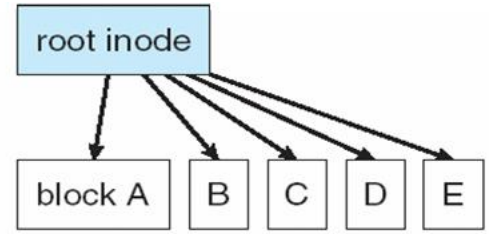


The WAFL File Layout

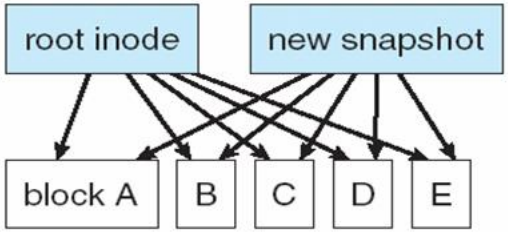




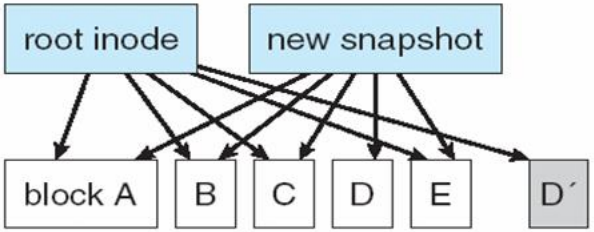
Snapshots in WAFL



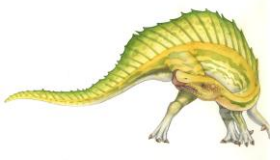
(a) Before a snapshot.



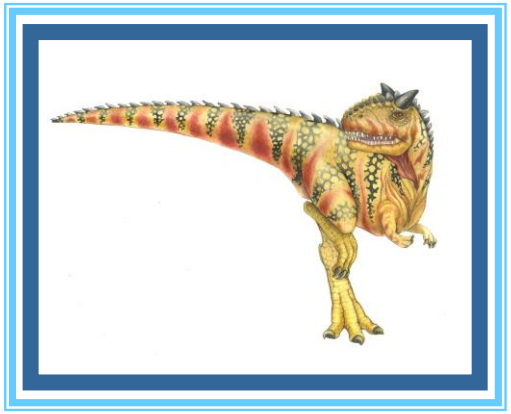
(b) After a snapshot, before any blocks change.



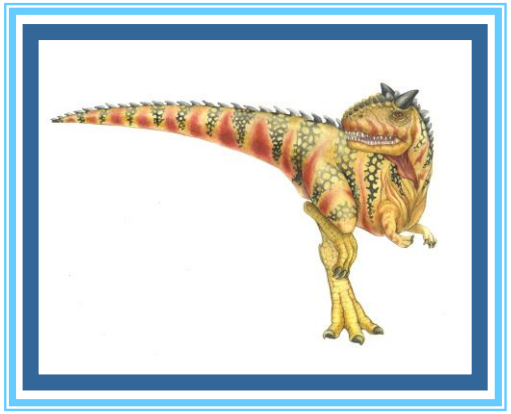
(c) After block D has changed to D´.



End of Chapter 12

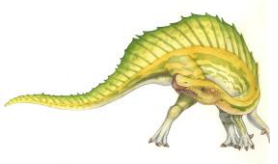


Chapter 12: File System Implementation



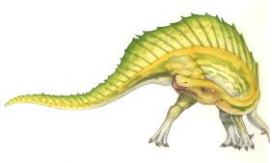
Chapter 12: File System Implementation

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



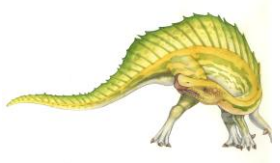
Objectives

- To describe the details of implementing local file systems and directory structures
- To describe the implementation of remote file systems
- To discuss block allocation and free-block algorithms and trade-offs

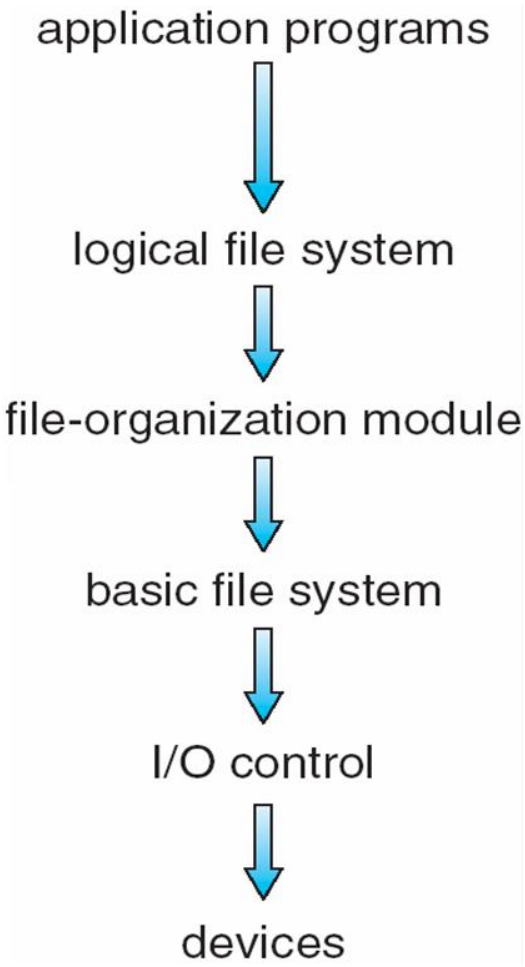


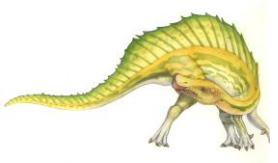
File-System Structure

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



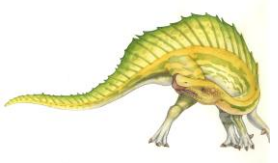
Layered File System





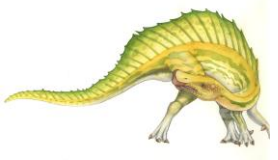
File System Layers

- **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
- **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 performanceTranslates file name into file number, file handle, location by maintaining file control blocks (**inodes** in UNIX)
  - Logical layers can be implemented by any coding method according to OS designer



File System Layers (Cont.)

- Many file systems, sometimes many within an operating system
  - 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 40 types, with **extended file system** ext2 and ext3 leading; plus distributed file systems, etc.)
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE

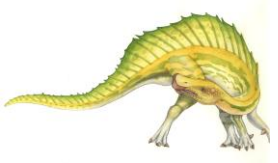
File-System Implementation

- We have system calls at the API level, but how do we implement their functions?
  - On-disk and in-memory structures
- **Boot control block** contains info needed by system to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
- **Volume control block (superblock, master file table)** contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- Directory structure organizes the files
  - Names and inode numbers, master file table

File-System Implementation (Cont.)

- Per-file **File Control Block (FCB)** contains many details about the file
  - inode number, permissions, size, dates
  - NFTS stores into in master file table using relational DB structures

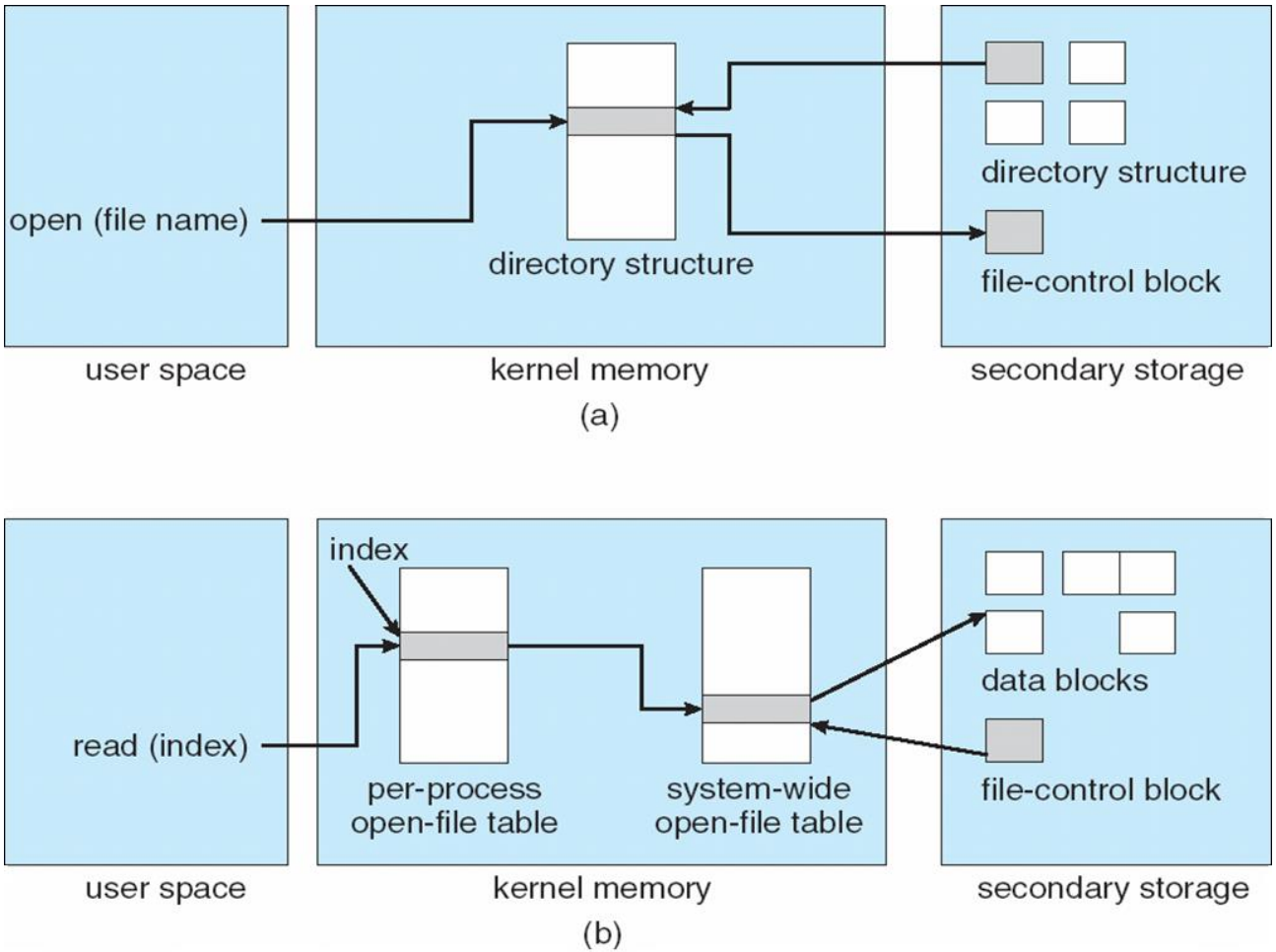
file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

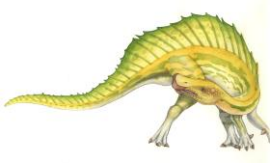


In-Memory File System Structures

- Mount table storing file system mounts, mount points, file system types
- The following figure illustrates the necessary file system structures provided by the operating systems
- Figure 12-3(a) refers to opening a file
- Figure 12-3(b) refers to reading a file
- Plus buffers hold data blocks from secondary storage
- Open returns a file handle for subsequent use
- Data from read eventually copied to specified user process memory address

In-Memory File System Structures



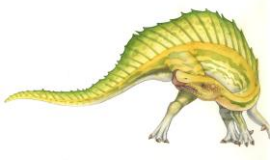


Partitions and Mounting

- 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
- 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

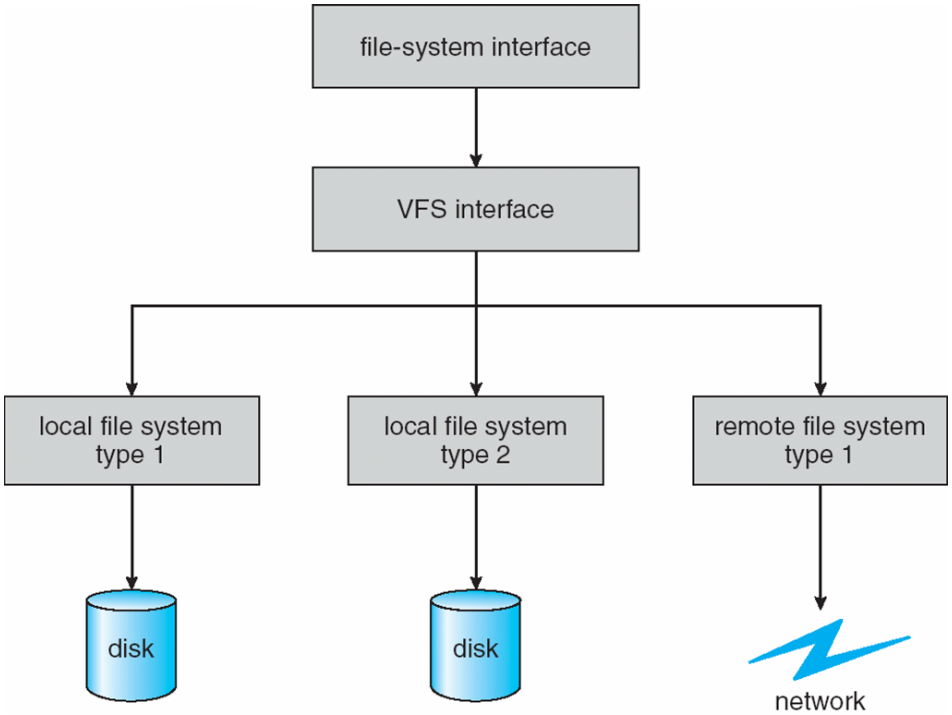
Virtual File Systems

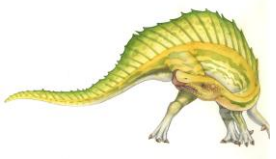
- **Virtual File Systems** (**VFS**) on Unix provide an object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - Separates file-system generic operations from implementation details
  - Implementation can 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



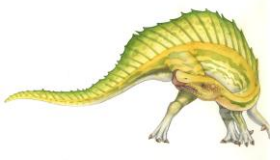


Virtual File System Implementation

- 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

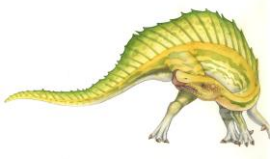
Directory Implementation

- **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



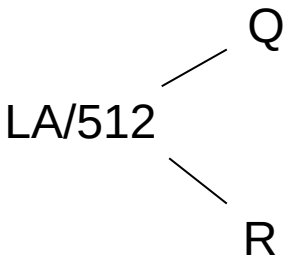
Allocation Methods - Contiguous

- 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**

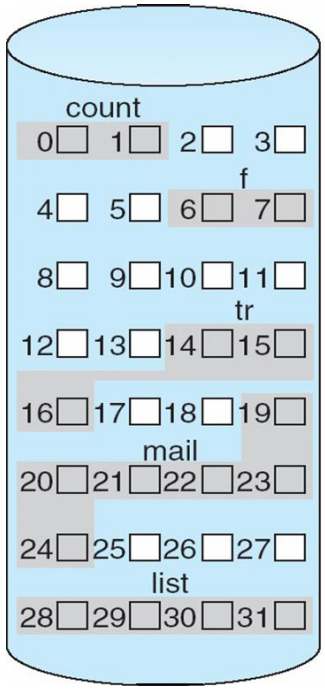


Contiguous Allocation

- Mapping from logical to physical



Block to be accessed = Q + starting address
Displacement into block = R



directory

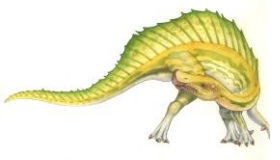
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

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 a contiguous block of disks
  - Extents are allocated for file allocation
  - A file consists of one or more extents

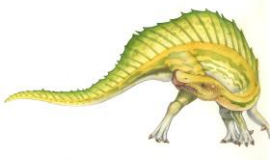
Allocation Methods - Linked

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



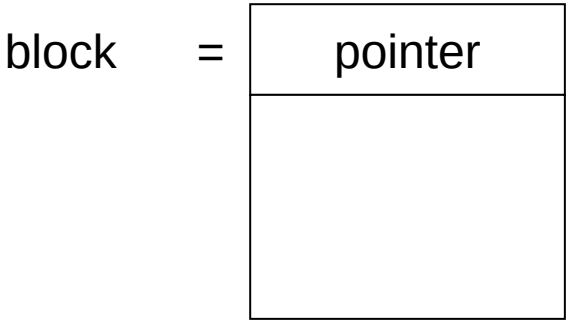
Allocation Methods – Linked (Cont.)

- FAT (File Allocation Table) variation
  - Beginning of volume has table, indexed by block number
  - Much like a linked list, but faster on disk and cacheable
  - New block allocation simple

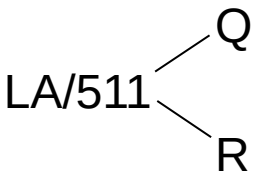


Linked Allocation

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

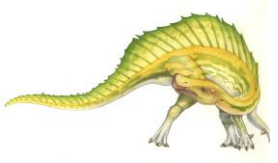


- Mapping

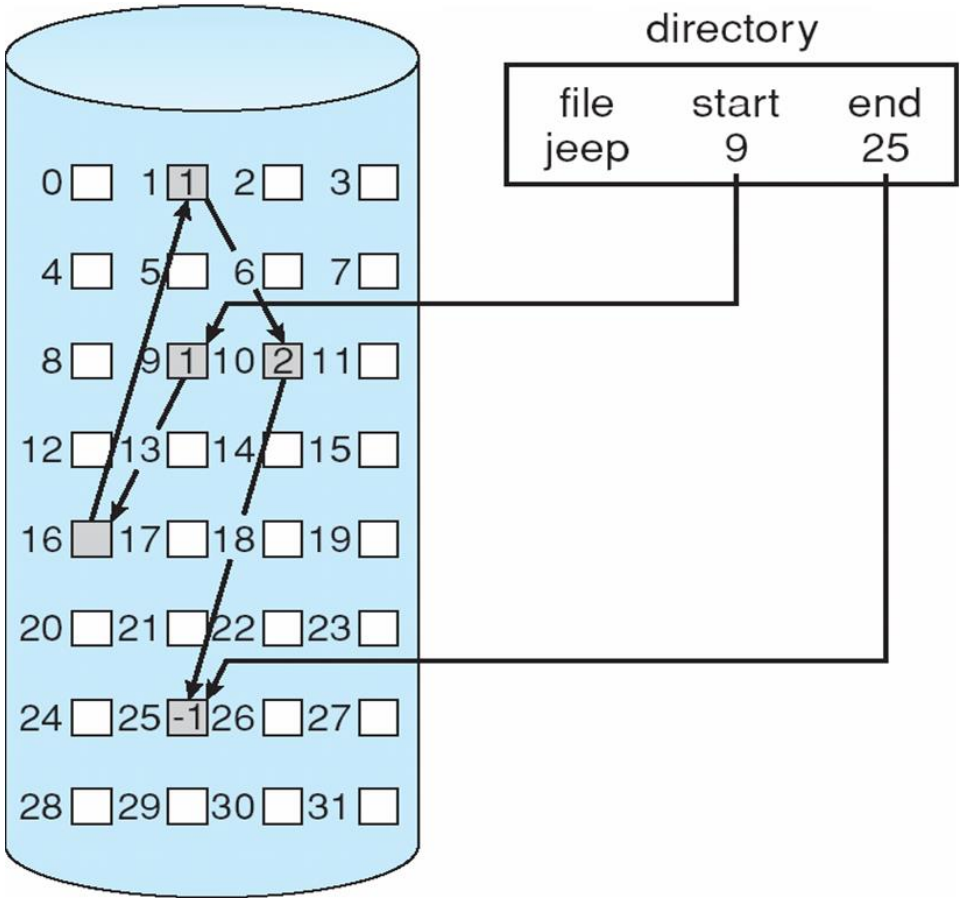


Block to be accessed is the Qth block in the linked chain of blocks representing the file.

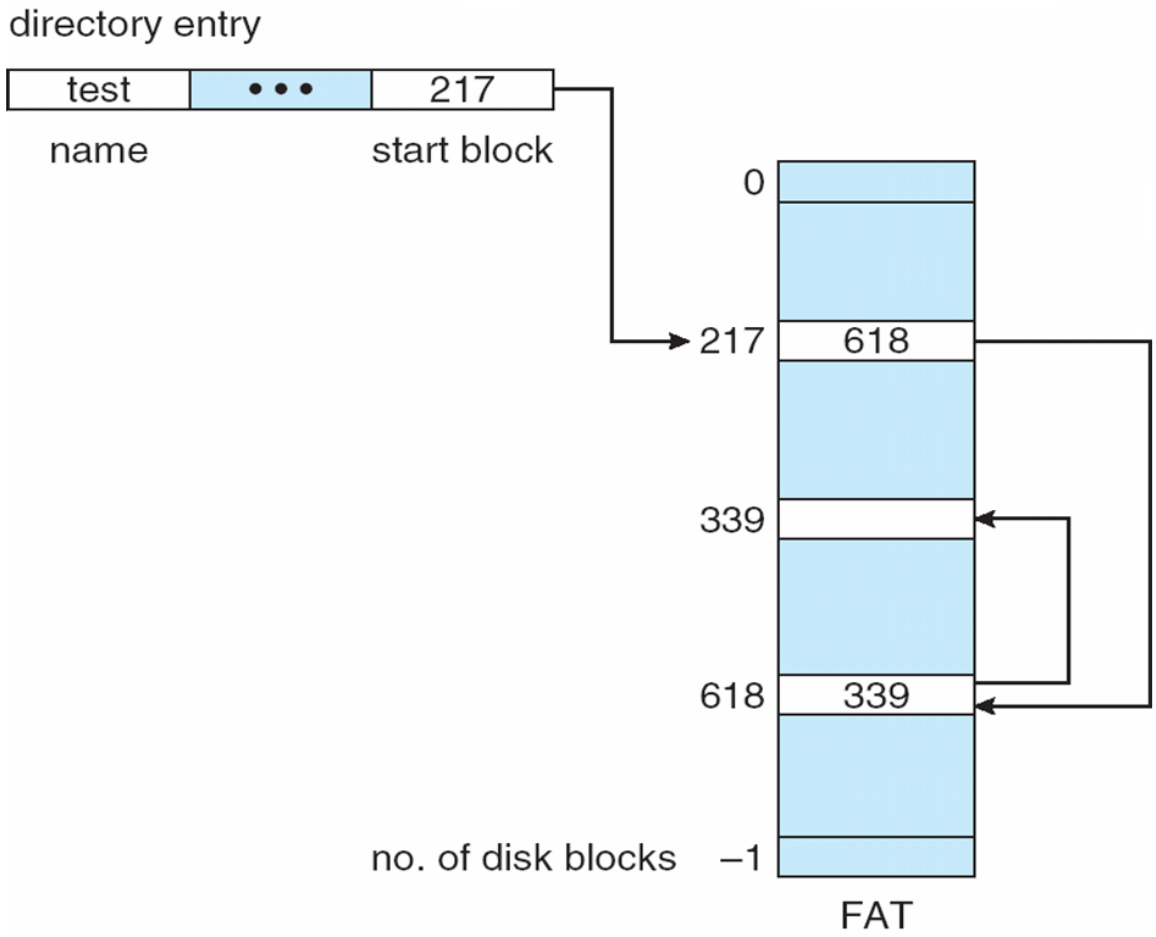
Displacement into block = R + 1



Linked Allocation



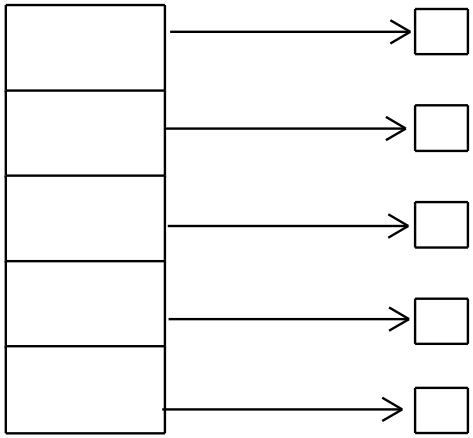
File-Allocation Table

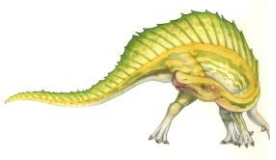


Allocation Methods - Indexed

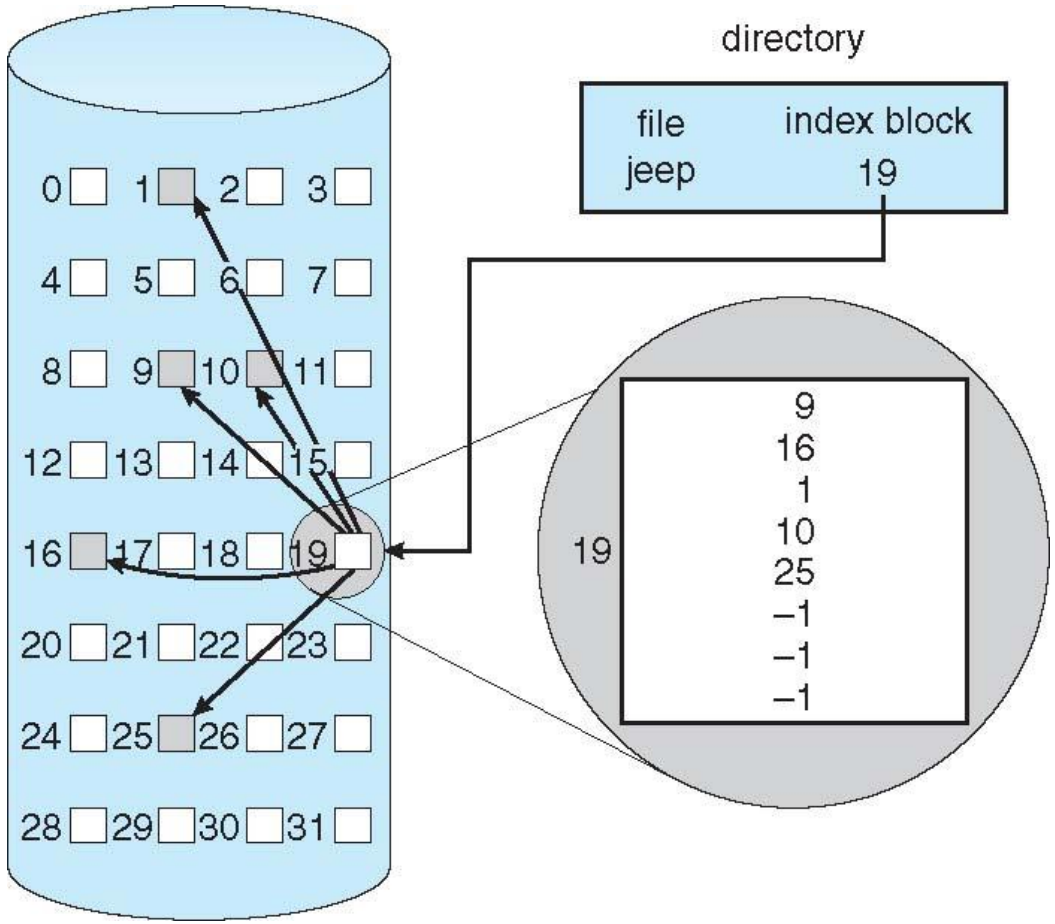
- **Indexed allocation**
  - Each file has its own **index block**(s) of pointers to its data blocks

- Logical view



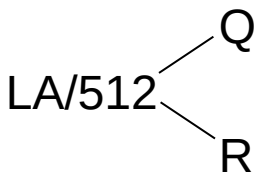


Example of Indexed Allocation

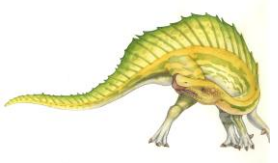


Indexed Allocation (Cont.)

- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K bytes and block size of 512 bytes. We need only 1 block for index table



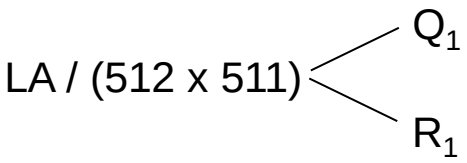
Q = displacement into index table
R = displacement into block



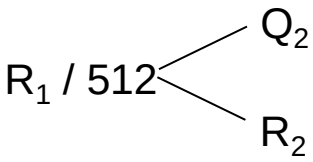
Indexed Allocation – Mapping (Cont.)

- Mapping from logical to physical in a file of unbounded length (block size of 512 words)

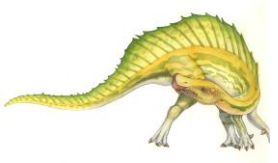
- Linked scheme – Link blocks of index table (no limit on size)



Q_1 = block of index table
R_1 is used as follows:

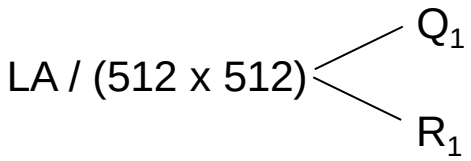


Q_2 = displacement into block of index table
R_2 displacement into block of file:

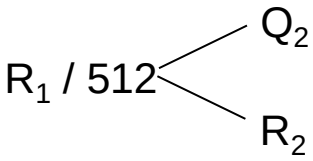


Indexed Allocation – Mapping (Cont.)

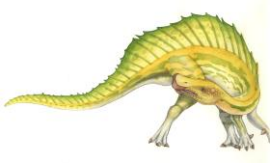
- Two-level index (4K blocks could store 1,024 four-byte pointers in outer index -> 1,048,567 data blocks and file size of up to 4GB)



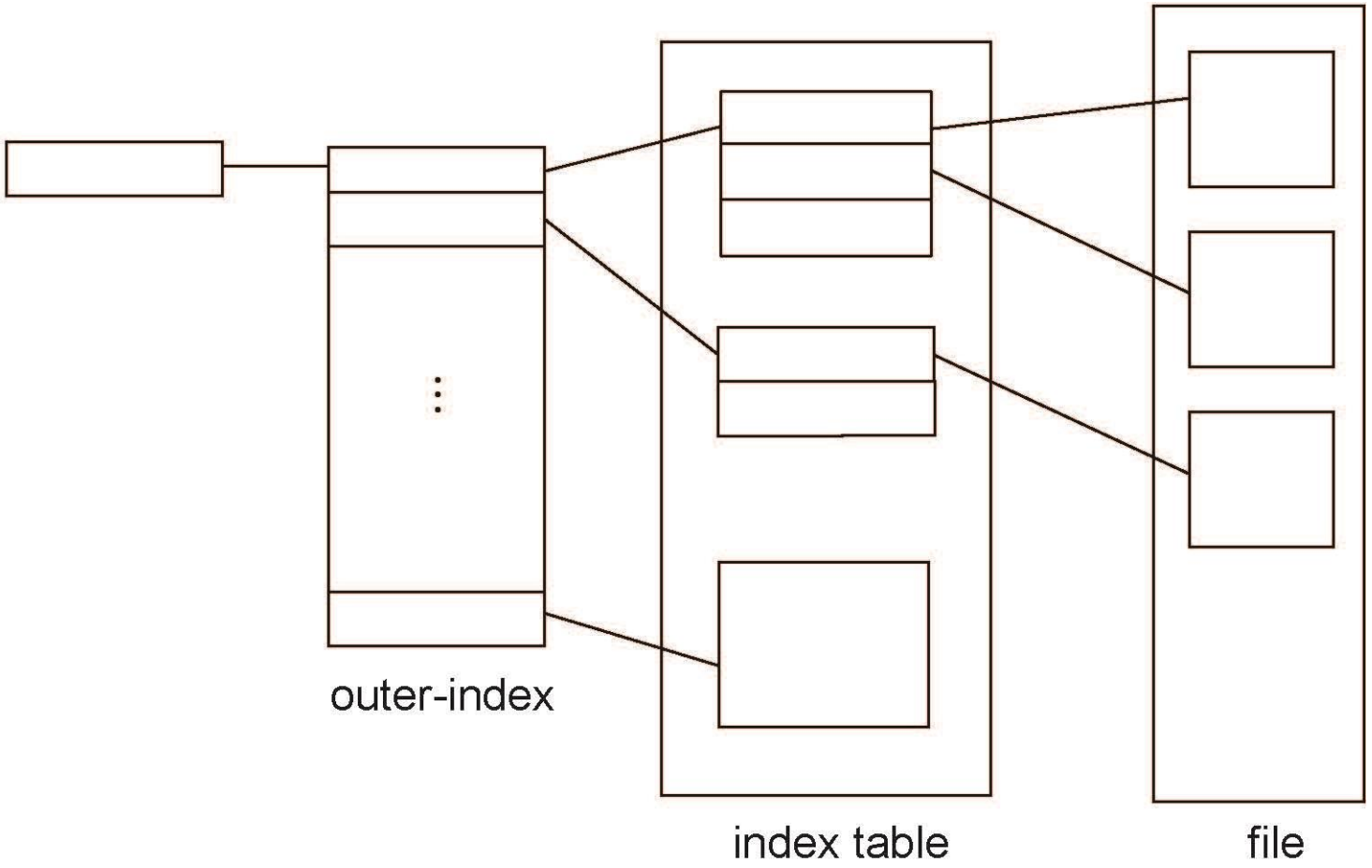
Q_1 = displacement into outer-index
R_1 is used as follows:



Q_2 = displacement into block of index table
R_2 displacement into block of file:

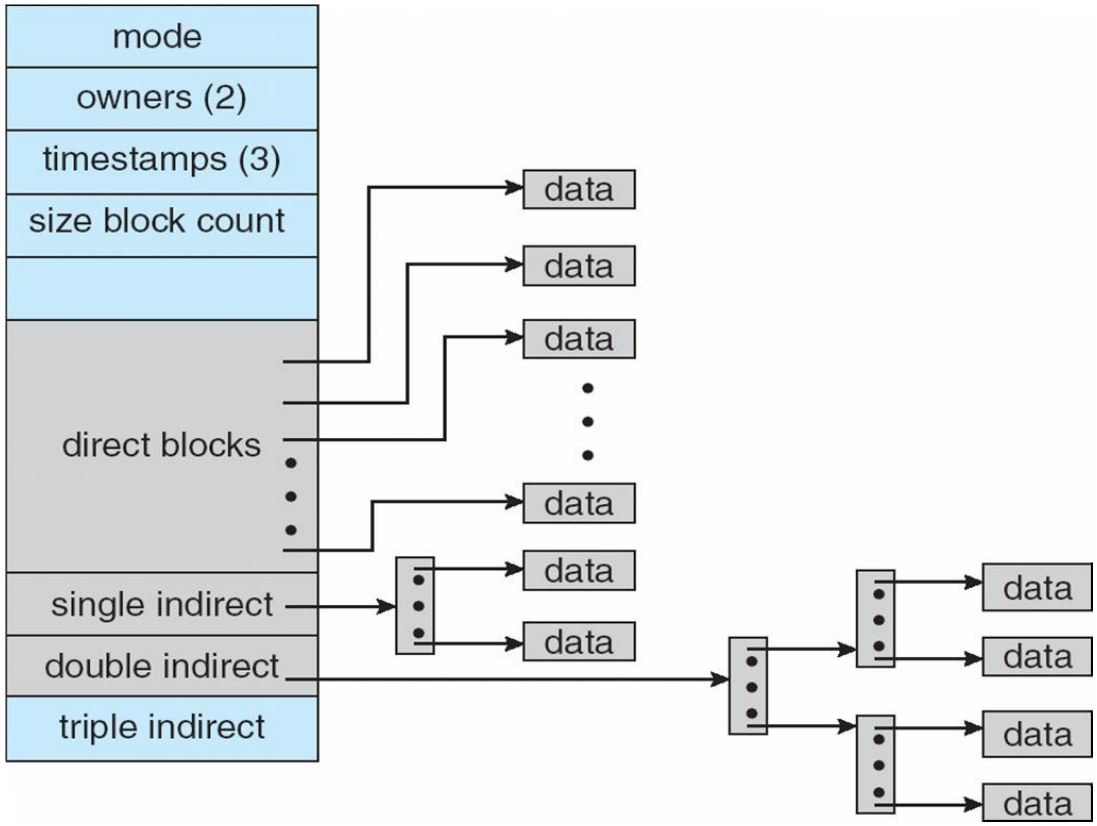


Indexed Allocation – Mapping (Cont.)

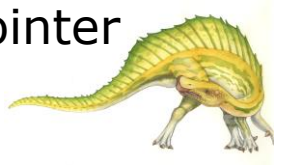


Combined Scheme: UNIX UFS

4K bytes per block, 32-bit addresses



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



Performance

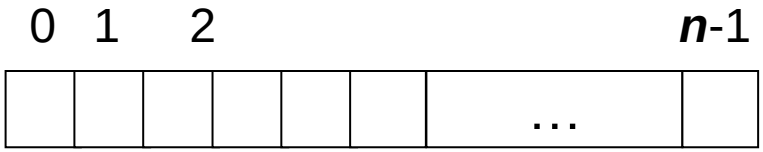
- Best method depends on file access type
  - Contiguous great for sequential and random
- Linked good for sequential, not random
- Declare access type at creation -> select either contiguous or linked
- Indexed more complex
  - Single block access could require 2 index block reads then data block read
  - Clustering can help improve throughput, reduce CPU overhead

Performance (Cont.)

- Adding instructions to the execution path to save one disk I/O is reasonable
  - Intel Core i7 Extreme Edition 990x (2011) at 3.46Ghz = 159,000 MIPS
    - http://en.wikipedia.org/wiki/Instructions_per_second
  - Typical disk drive at 250 I/Os per second
    - 159,000 MIPS / 250 = 630 million instructions during one disk I/O
  - Fast SSD drives provide 60,000 IOPS
    - 159,000 MIPS / 60,000 = 2.65 millions instructions during one disk 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)

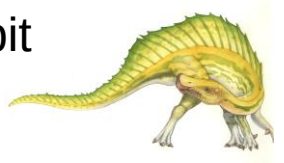


$$\text{bit}[i] = \begin{cases} 1 \Rightarrow \text{block}[i] \text{ free} \\ 0 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

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)

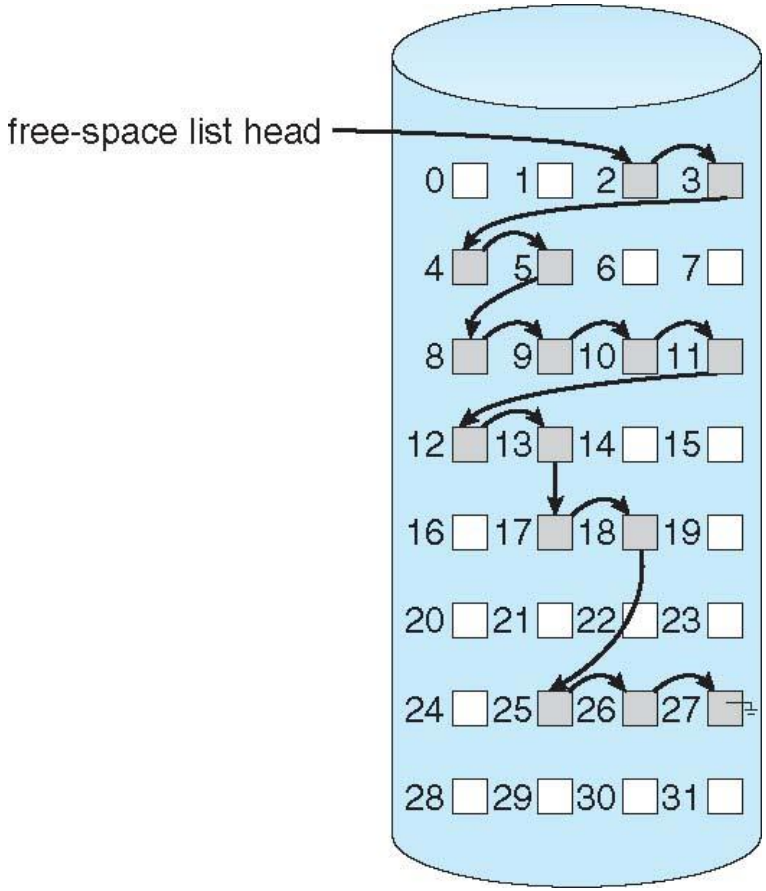
    $n = 2^{40}/2^{12} = 2^{28}$ bits (or 32MB)

    if clusters of 4 blocks -> 8MB of memory

- Easy to get contiguous files

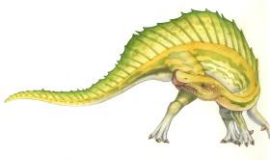
Linked Free Space List on Disk

- 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

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

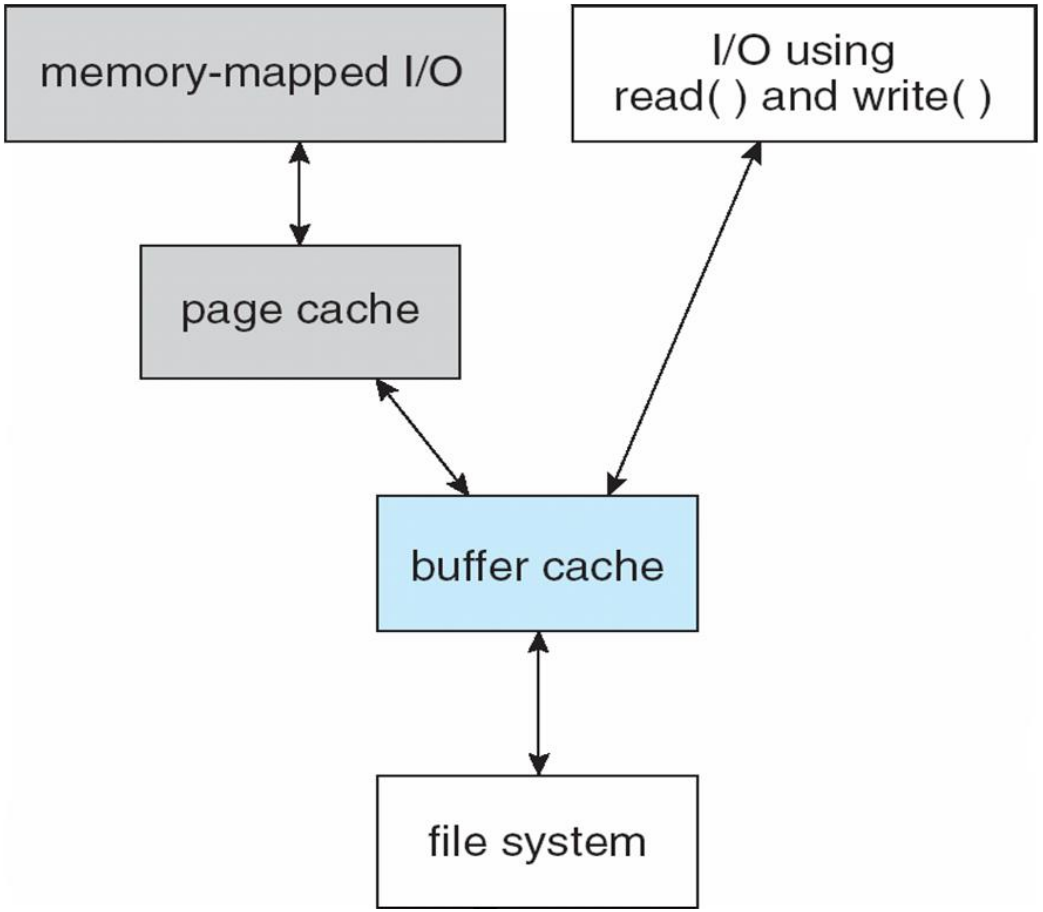
Efficiency and Performance (Cont.)

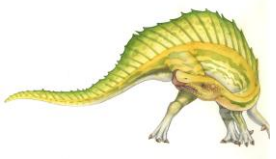
- Performance
  - Keeping data and metadata close together
  - **Buffer cache** – separate section of main memory for frequently used blocks
  - **Synchronous** writes sometimes requested by apps or needed by OS
    - No buffering / caching – writes must hit disk before acknowledgement
    - **Asynchronous** 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



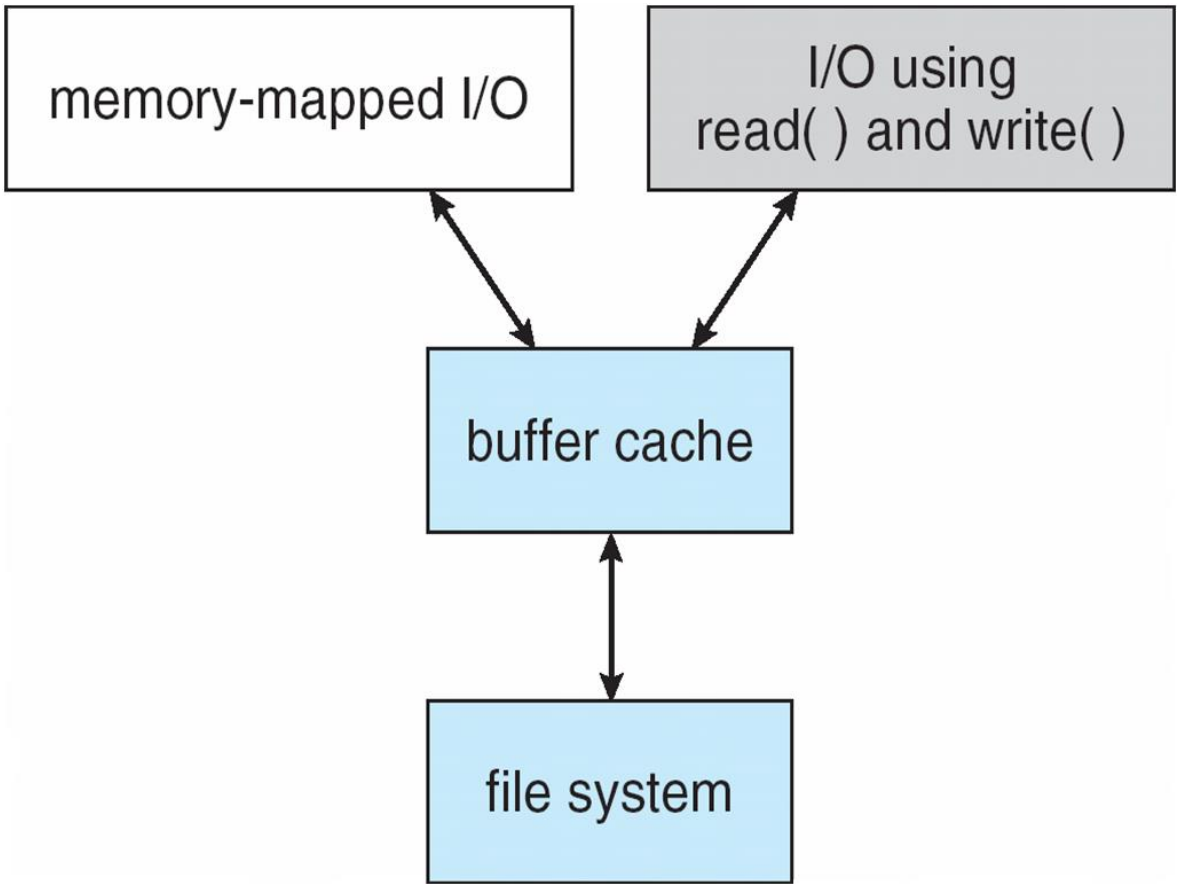


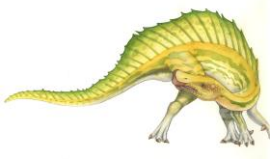
Unified Buffer Cache

- 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



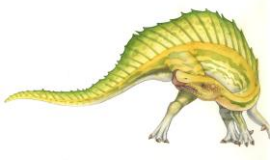


Recovery

- **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

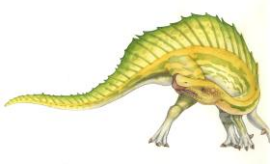
Log Structured File Systems

- **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



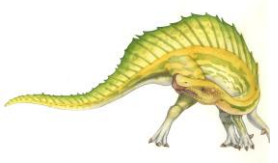
The Sun Network File System (NFS)

- An implementation and a specification of a software system for accessing remote files across LANs (or WANs)
- The implementation is part of the Solaris and SunOS operating systems running on Sun workstations using an unreliable datagram protocol (UDP/IP protocol and Ethernet



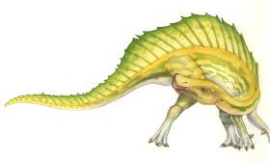
NFS (Cont.)

- 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

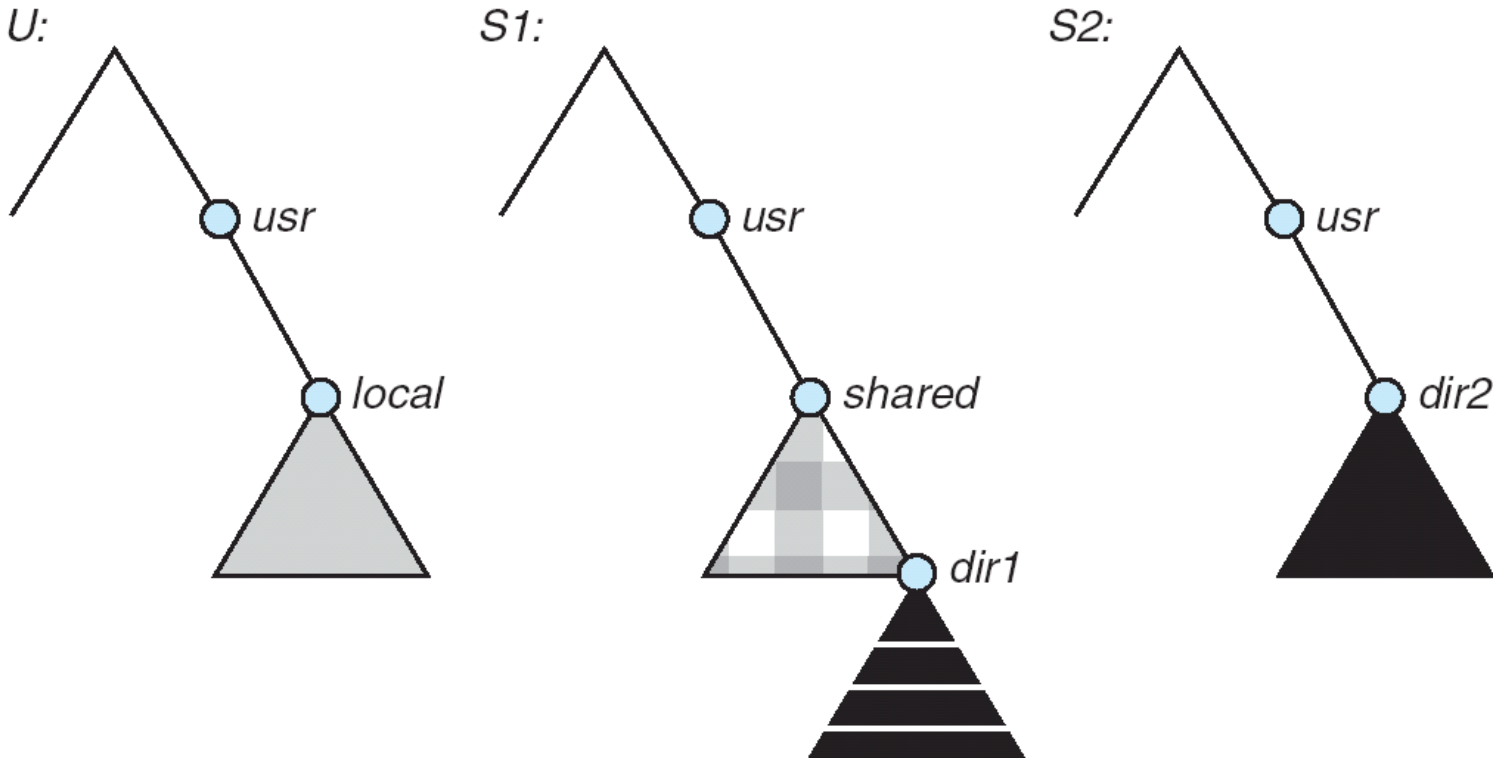


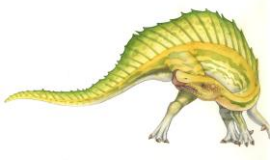
NFS (Cont.)

- 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

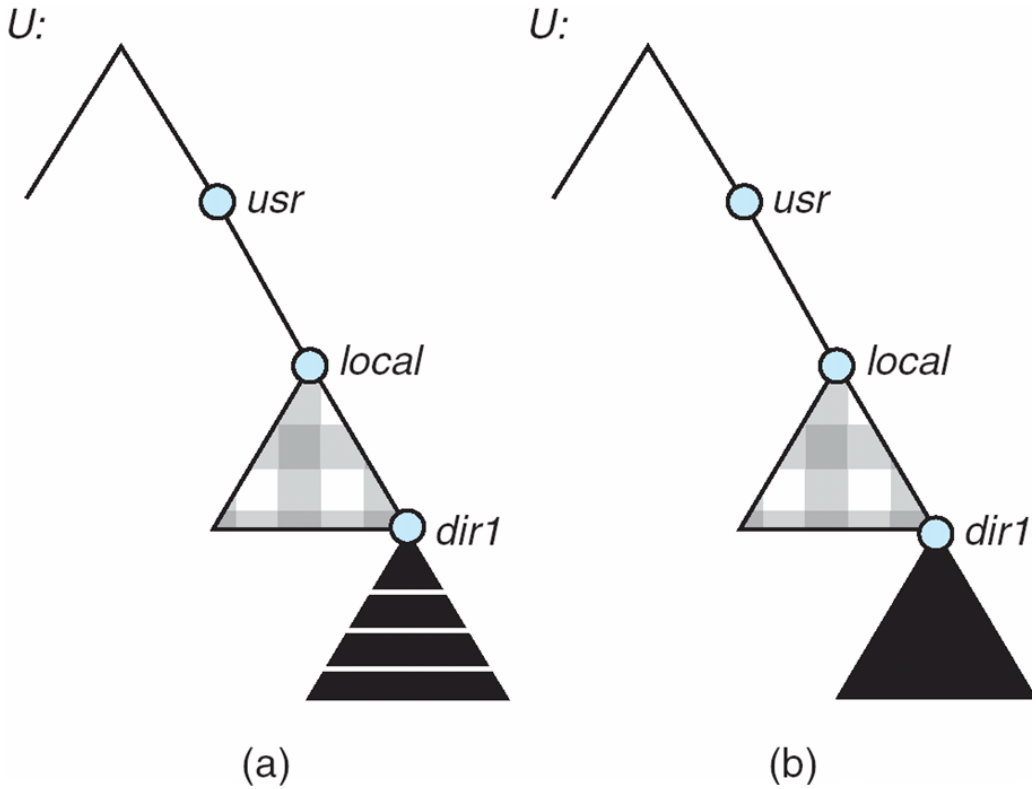


Three Independent File Systems



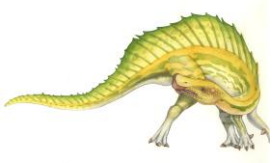


Mounting in NFS



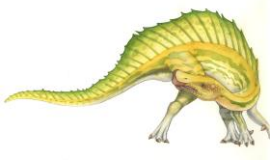
Mounts

Cascading mounts



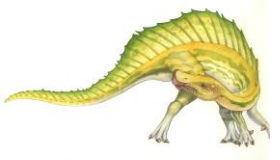
NFS Mount Protocol

- 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



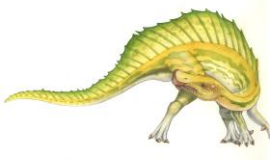
NFS Protocol

- Provides a set of remote procedure calls for remote file operations. The procedures support the following 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 just coming available – 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

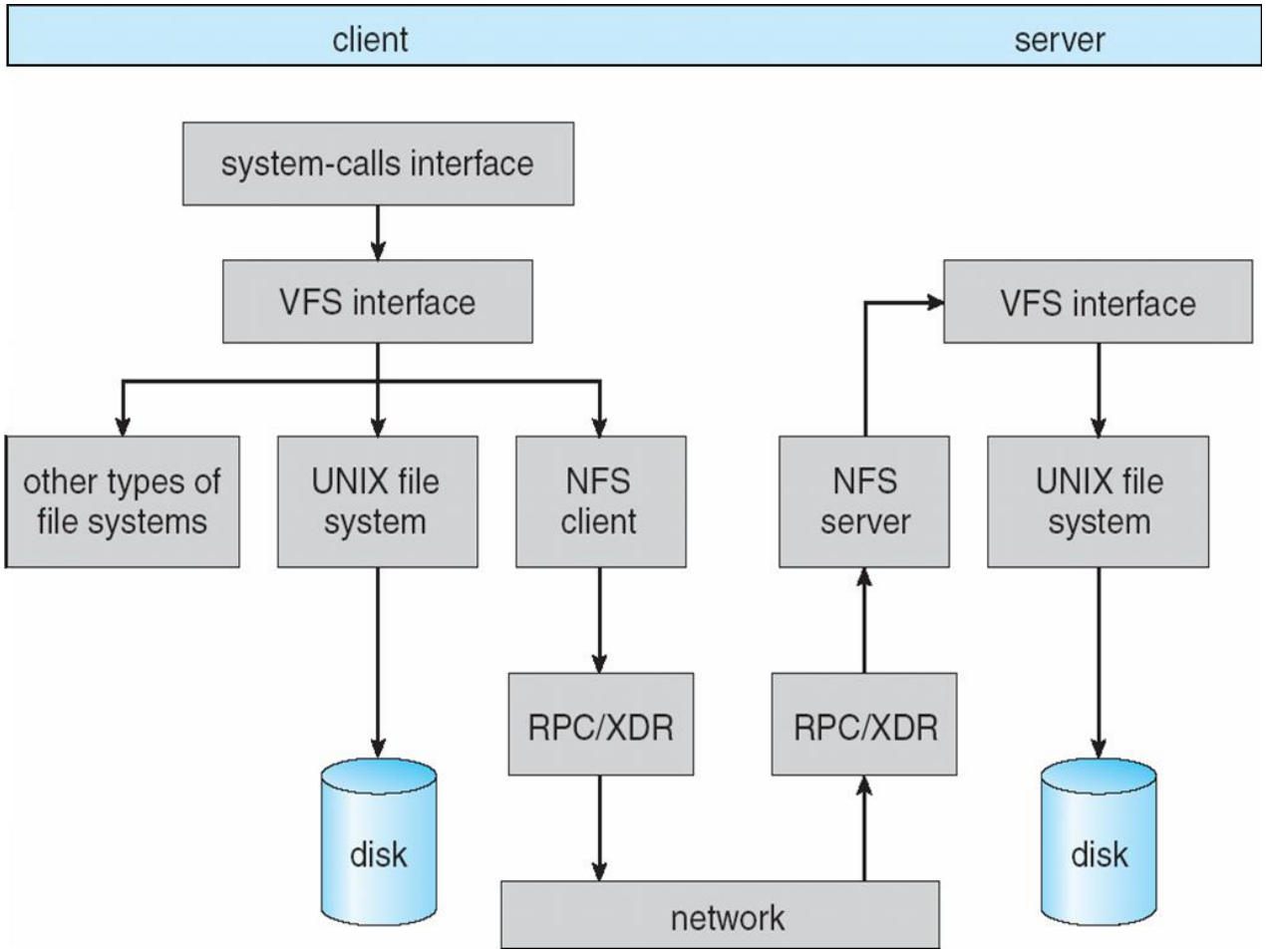


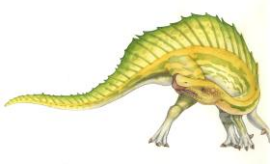
Three Major Layers of NFS Architecture

- 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



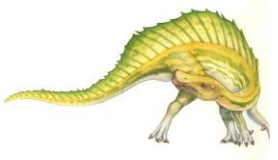
Schematic View of NFS Architecture





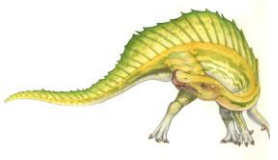
NFS Path-Name Translation

- 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's side holds the vnodes for remote directory names



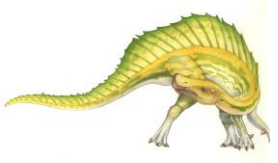
NFS Remote Operations

- 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

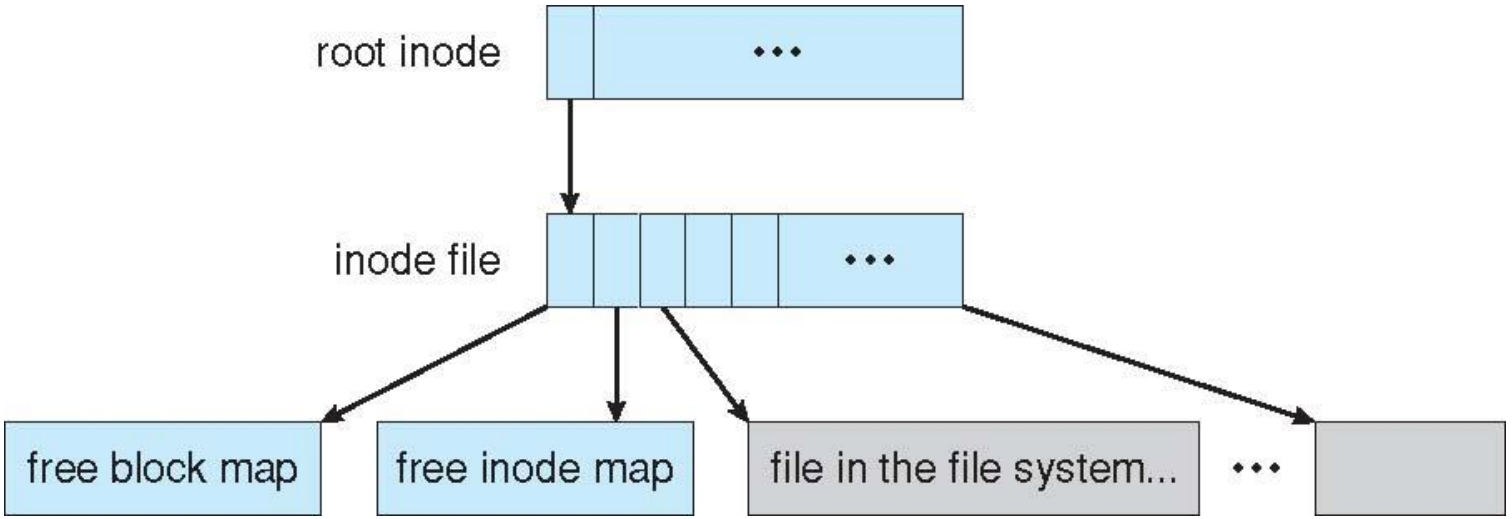


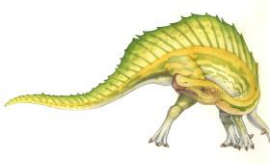
Example: WAFL File System

- Used on Network Appliance “Filers” – distributed file system appliances
- “Write-anywhere file layout”
- 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

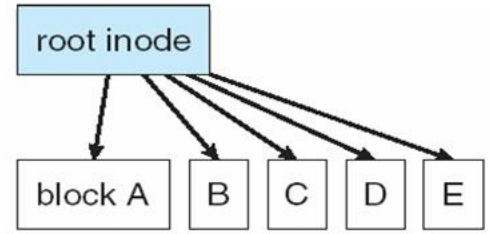


The WAFL File Layout

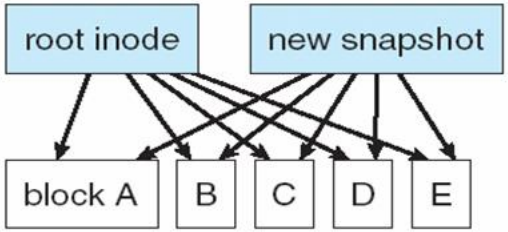




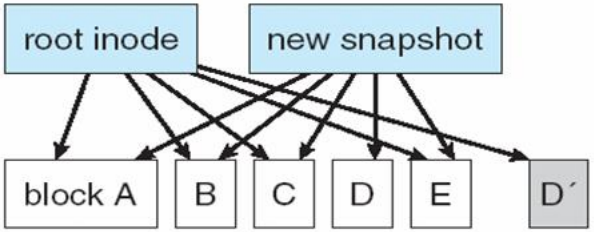
Snapshots in WAFL



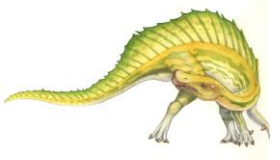
(a) Before a snapshot.



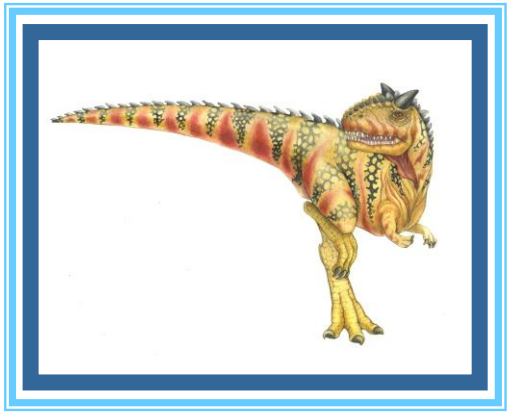
(b) After a snapshot, before any blocks change.



(c) After block D has changed to D´.



End of Chapter 12



Operating System
KCS – 401

I/O System

Dr. Pankaj Kumar
Associate Professor – CSE
SRMGPC Lucknow

Background

I/O management is a major component of operating system design and operation

- Important aspect of computer operation
- I/O devices vary greatly
- Various methods to control them
- Performance management
- New types of devices frequent

Ports, busses, device controllers connect to various devices

Device drivers encapsulate device details

- Present uniform device-access interface to I/O subsystem

I/O System

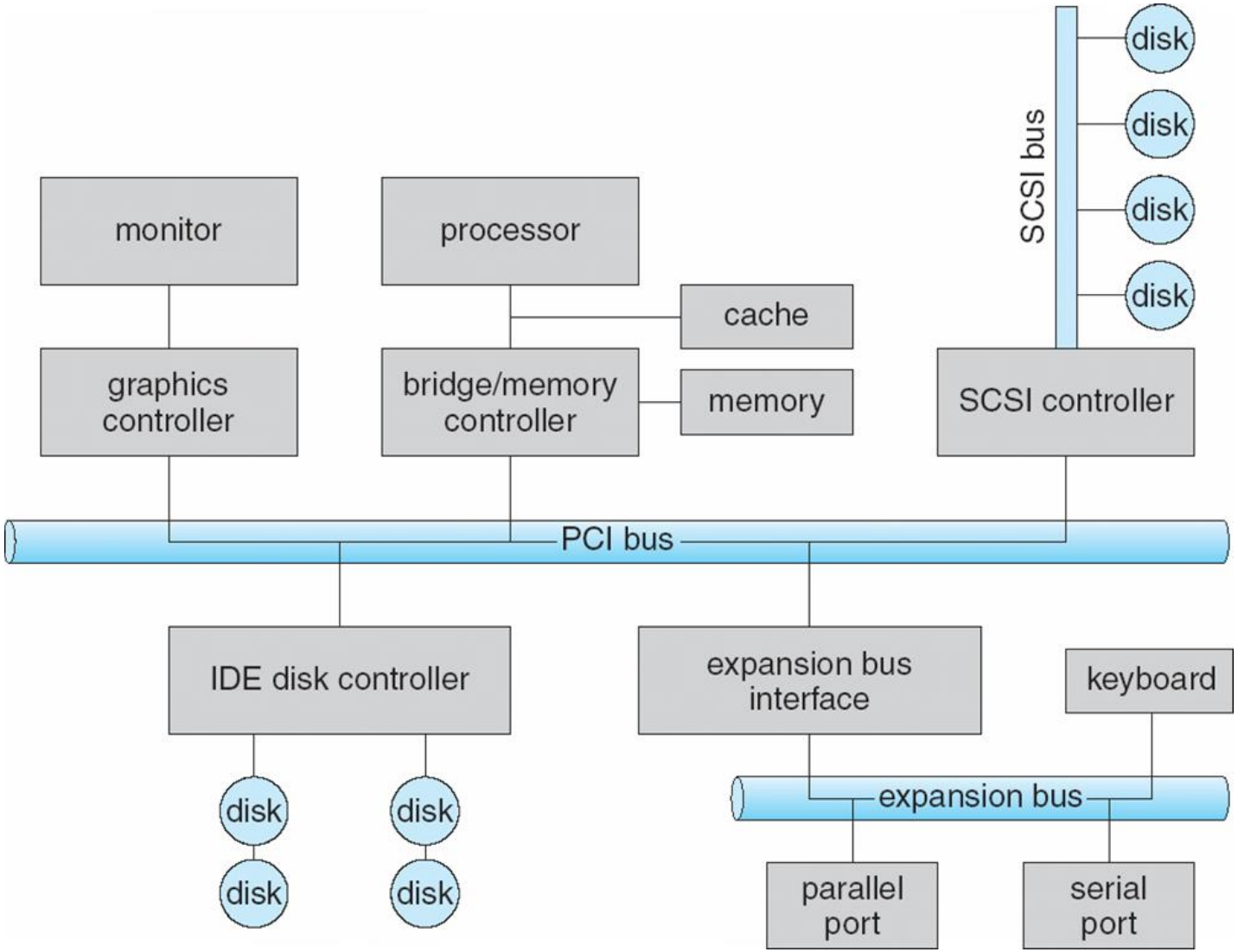
Incredible variety of I/O devices

- Storage
- Transmission
- Human-interface

Common concepts – signals from I/O devices interface with computer

- **Port** – connection point for device
- **Bus - daisy chain** or shared direct access
  - **PCI** bus common in PCs and servers, PCI Express (**PCIe**)
  - **expansion bus** connects relatively slow devices
- **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

I/O System Structure



I/O System Structure

Synchronous vs asynchronous I/O

- **Synchronous I/O** − In this scheme CPU execution waits while I/O proceeds
- **Asynchronous I/O** − I/O proceeds concurrently with CPU execution

Communication to I/O Devices

The CPU must have a way to pass information to and from an I/O device. There are three approaches available to communicate with the CPU and Device.

- Special Instruction I/O
- Memory-mapped I/O
- Direct memory access (DMA)

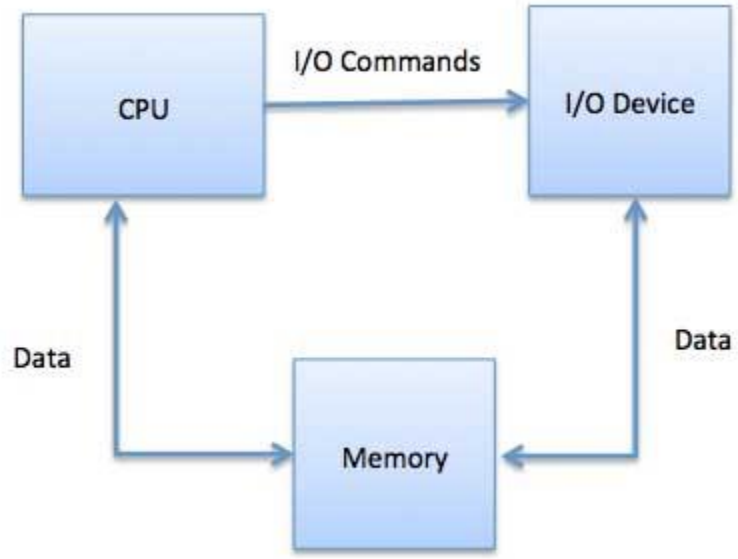
I/O System Structure

Special Instruction I/O

This uses CPU instructions that are specifically made for controlling I/O devices. These instructions typically allow data to be sent to an I/O device or read from an I/O device.

Memory-mapped I/O

When using memory-mapped I/O, the same address space is shared by memory and I/O devices. The device is connected directly to certain main memory locations so that I/O device can transfer block of data to/from memory without going through CPU. While using memory mapped IO, OS allocates buffer in memory and informs I/O device to use that buffer to send data to the CPU. I/O device operates asynchronously with CPU, interrupts CPU when finished.

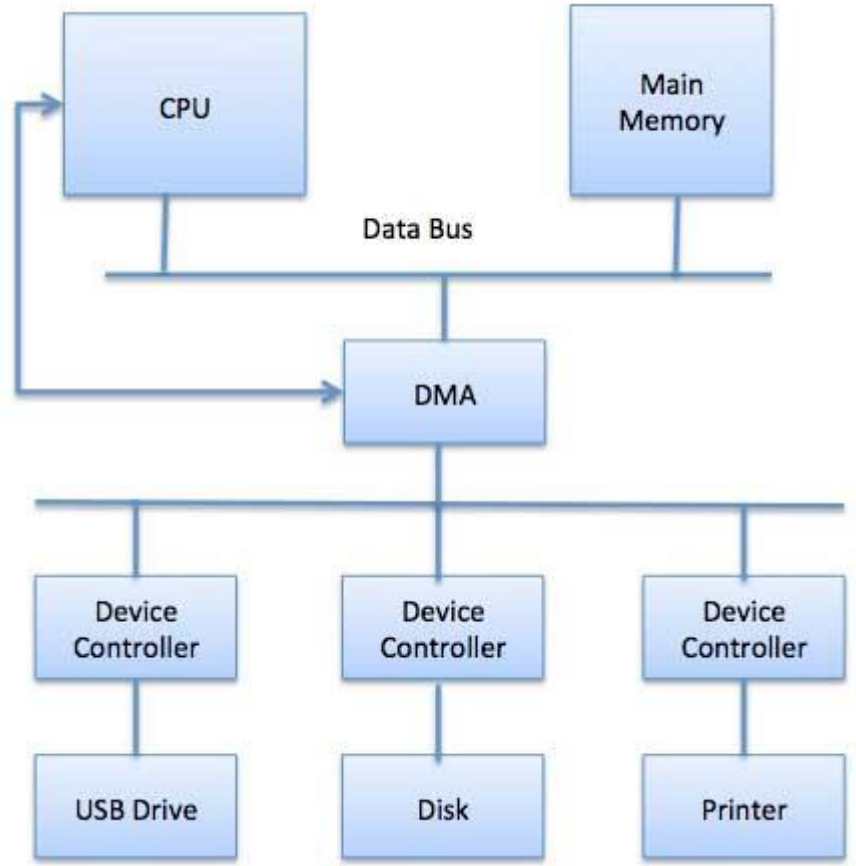


I/O System Structure

Direct Memory Access (DMA)

Slow devices like keyboards will generate an interrupt to the main CPU after each byte is transferred. If a fast device such as a disk generated an interrupt for each byte, the operating system would spend most of its time handling these interrupts. So a typical computer uses direct memory access (DMA) hardware to reduce this overhead.

Direct Memory Access (DMA) means CPU grants I/O module authority to read from or write to memory without involvement. DMA module itself controls exchange of data between main memory and the I/O device. CPU is only involved at the beginning and end of the transfer and interrupted only after entire block has been transferred.



I/O Subsystem

Each I/O device driver can provide a driver specific set of I/O application programming interfaces to the applications However, each application must be aware of the nature of the underlying I/O device

Thus, embedded systems often include an I/O subsystem to reduce this implementation dependence.

I/O subsystem defines a standard set of functions for I/O operations

To hide device peculiarities from applications

All I/O device drivers conform to and support this function set

To provide uniform I/O to applications across a wide spectrum of I/O devices of varying types

I/O Buffering

A buffer is a memory area that stores data being transferred between two devices or between a device and an application.

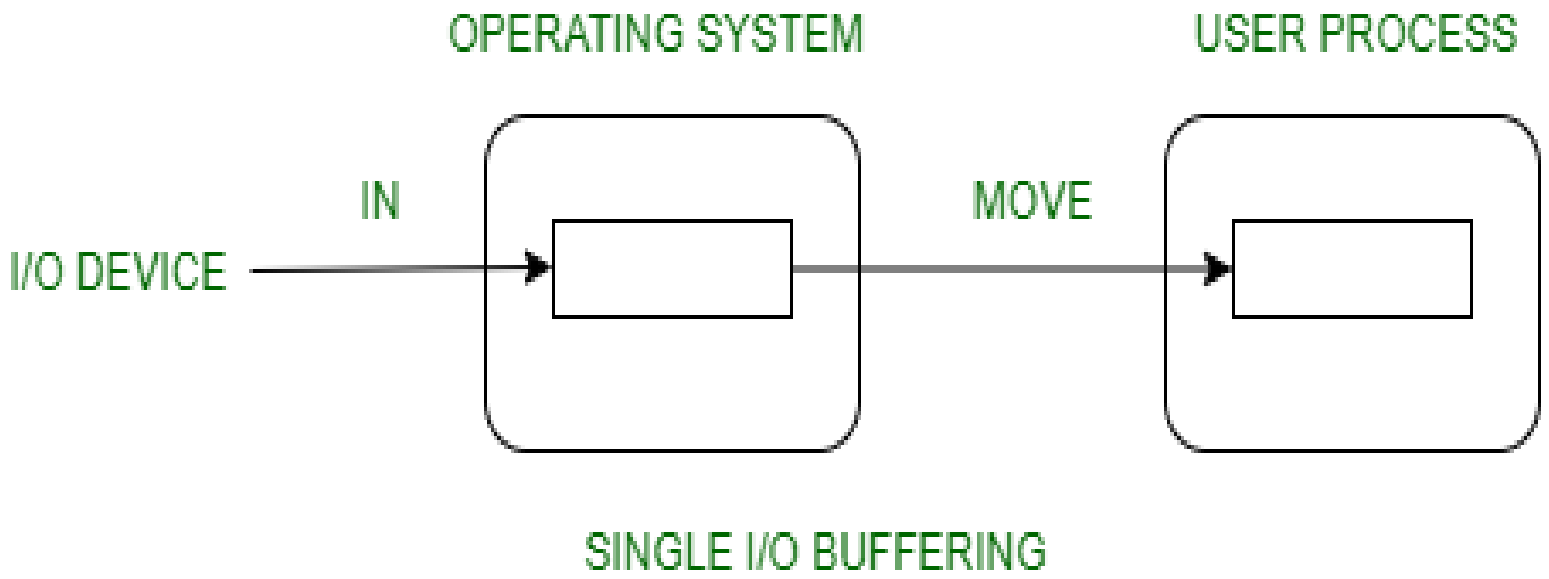
Uses of I/O Buffering :

- Buffering is done to deal effectively with a speed mismatch between the producer and consumer of the data stream.
- After receiving the data in the buffer, the data get transferred to disk from buffer in a single operation.
- This process of data transfer is not instantaneous, therefore the modem needs another buffer in order to store additional incoming data.
- When the first buffer got filled, then it is requested to transfer the data to disk.
- The modem then starts filling the additional incoming data in the second buffer while the data in the first buffer getting transferred to disk.
- When both the buffers completed their tasks, then the modem switches back to the first buffer while the data from the second buffer get transferred to the disk.
- The use of two buffers disintegrates the producer and the consumer of the data, thus minimizes the time requirements between them.
- Buffering also provides variations for devices that have different data transfer sizes.

I/O Buffering - Types

Single buffer :

A buffer is provided by the operating system to the system portion of the main memory.



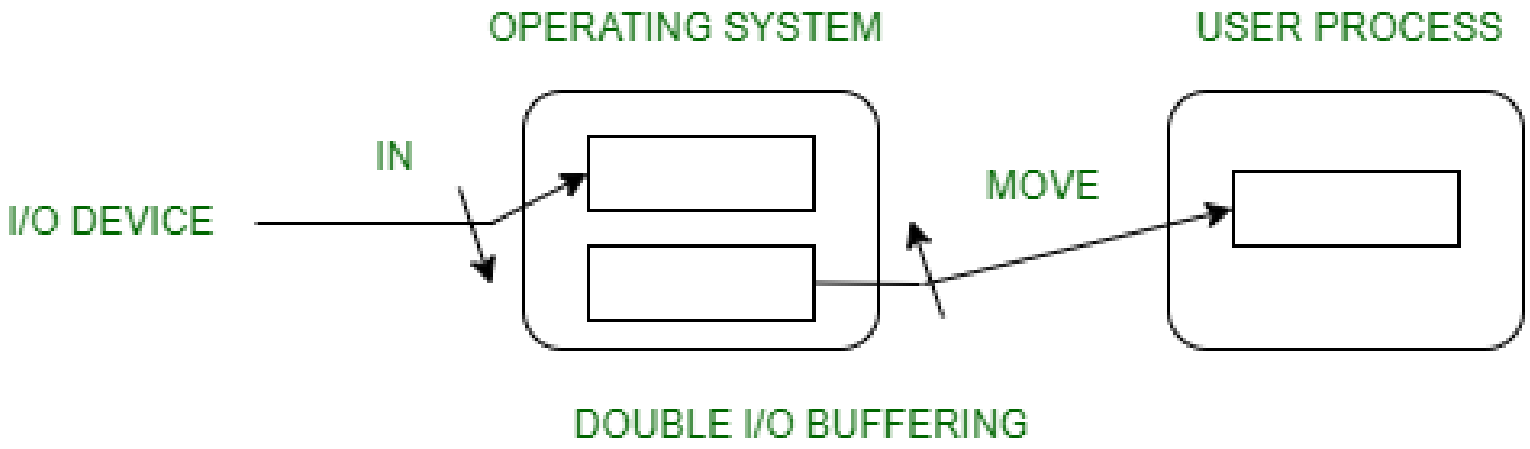
SINGLE I/O BUFFERING

I/O Buffering - Types

Double buffer :

Here are two buffers in the system.

- One buffer is used by the driver or controller to store data while waiting for it to be taken by higher level of the hierarchy.
- Other buffer is used to store data from the lower level module.
- Double buffering is also known as buffer swapping.
- A major disadvantage of double buffering is that the complexity of the process get increased.
- If the process performs rapid bursts of I/O, then using double buffering may be deficient.

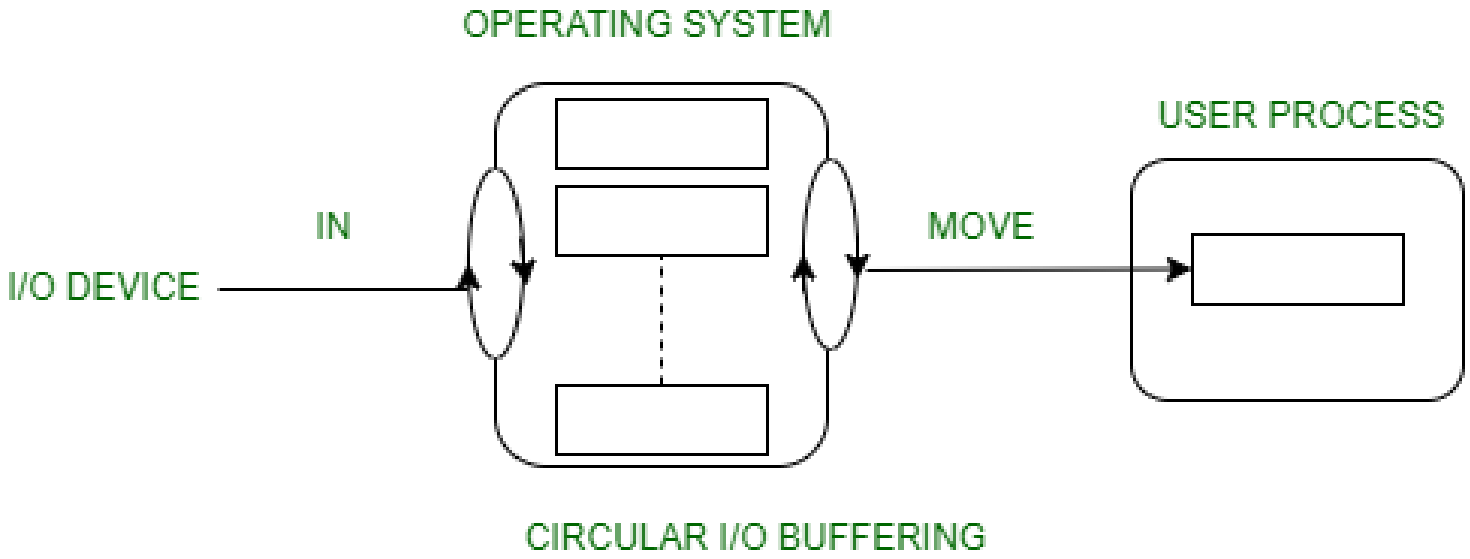


DOUBLE I/O BUFFERING

I/O Buffering - Types

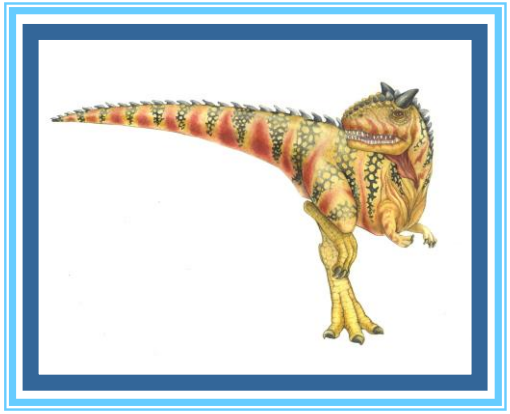
Circular buffer :

- When more than two buffers are used, the collection of buffers is itself referred to as a circular buffer.
- In this, the data do not directly passed from the producer to the consumer because the data would change due to overwriting of buffers before they had been consumed.
- The producer can only fill up to buffer i-1 while data in buffer i is waiting to be consumed.



CIRCULAR I/O BUFFERING

Chapter 14: Protection



Chapter 14: Protection

- Goals of Protection
- Principles of Protection
- Domain of Protection
- Access Matrix
- Implementation of Access Matrix
- Access Control
- Revocation of Access Rights
- Capability-Based Systems
- Language-Based Protection

Objectives

- Discuss the goals and principles of protection in a modern computer system
- Explain how protection domains combined with an access matrix are used to specify the resources a process may access
- Examine capability and language-based protection systems

Goals of Protection

- In one protection model, computer consists of a collection of objects, hardware or software
- Each object has a unique name and can be accessed through a well-defined set of operations
- Protection problem - ensure that each object is accessed correctly and only by those processes that are allowed to do so

Principles of Protection

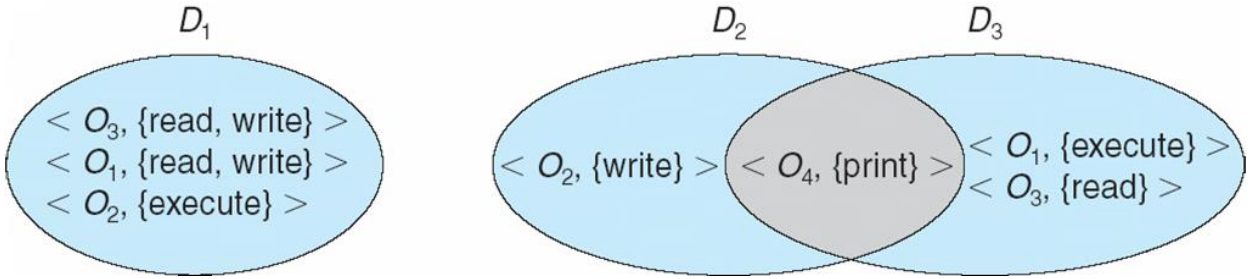
- Guiding principle – **principle of least privilege**
  - Programs, users and systems should be given just enough **privileges** to perform their tasks
  - Limits damage if entity has a bug, gets abused
  - Can be static (during life of system, during life of process)
  - Or dynamic (changed by process as needed) – **domain switching**, **privilege escalation**
  - "Need to know" a similar concept regarding access to data

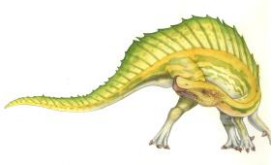
Principles of Protection (Cont.)

- Must consider “grain” aspect
  - Rough-grained privilege management easier, simpler, but least privilege now done in large chunks
    - For example, traditional Unix processes either have abilities of the associated user, or of root
  - Fine-grained management more complex, more overhead, but more protective
    - File ACL lists, RBAC
- Domain can be user, process, procedure

Domain Structure

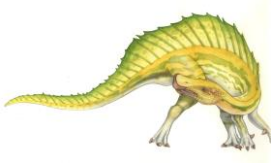
- Access-right = <*object-name*, *rights-set*>
  where *rights-set* is a subset of all valid operations that can be performed on the object
- Domain = set of access-rights





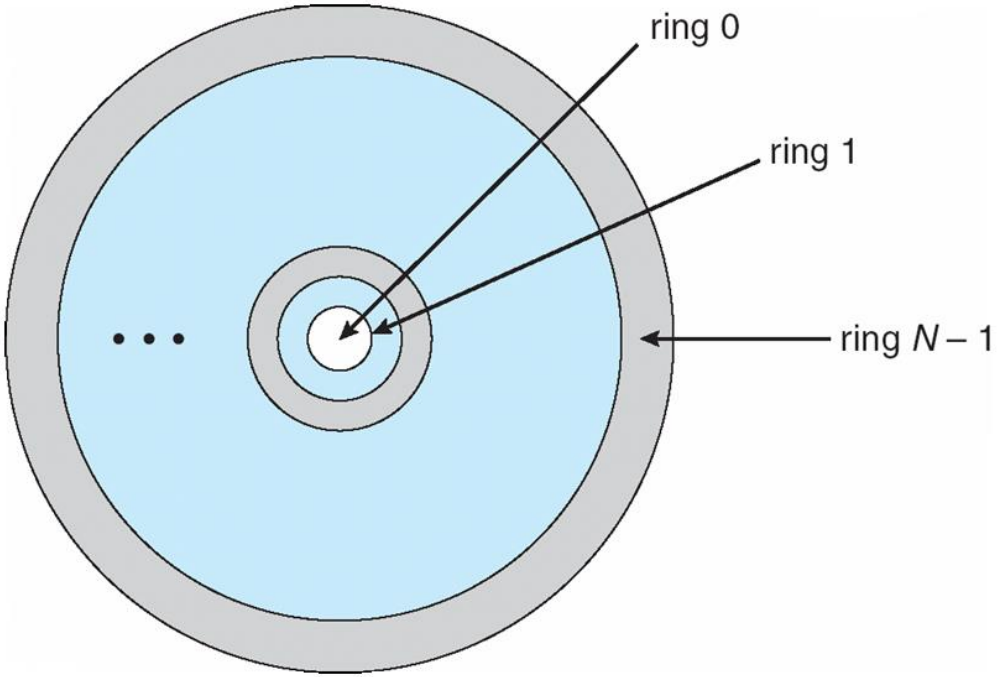
Domain Implementation (UNIX)

- Domain = user-id
- Domain switch accomplished via file system
  - Each file has associated with it a domain bit (setuid bit)
  - When file is executed and setuid = on, then user-id is set to owner of the file being executed
  - When execution completes user-id is reset
- Domain switch accomplished via passwords
  - `su` command temporarily switches to another user's domain when other domain's password provided
- Domain switching via commands
  - `sudo` command prefix executes specified command in another domain (if original domain has privilege or password given)



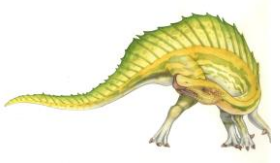
Domain Implementation (MULTICS)

- Let D_i and D_j be any two domain rings
- If $j < I \Rightarrow D_i \subseteq D_j$



Multics Benefits and Limits

- Ring / hierarchical structure provided more than the basic kernel / user or root / normal user design
- Fairly complex -> more overhead
- But does not allow strict need-to-know
  - Object accessible in D_j but not in D_i, then ***j*** must be < ***i***
  - But then every segment accessible in D_i also accessible in D_j



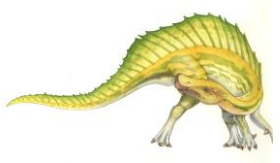
Access Matrix

- View protection as a matrix (**access matrix**)
- Rows represent domains
- Columns represent objects
- **`Access(i, j)`** is the set of operations that a process executing in $Domain_i$ can invoke on $Object_j$

object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Use of Access Matrix

- If a process in Domain D_i tries to do "op" on object O_j, then "op" must be in the access matrix
- User who creates object can define access column for that object
- Can be expanded to dynamic protection
  - Operations to add, delete access rights
  - Special access rights:
    - *owner of* O_i
    - *copy op from* O_i *to* O_j *(denoted by "*")*
    - *control –* D_i *can modify* D_j *access rights*
    - *transfer – switch from domain* D_i *to* D_j
  - *Copy* and *Owner* applicable to an object
  - *Control* applicable to domain object



Use of Access Matrix (Cont.)

- **Access matrix** design separates mechanism from policy
  - Mechanism
    - Operating system provides access-matrix + rules
    - If ensures that the matrix is only manipulated by authorized agents and that rules are strictly enforced
  - Policy
    - User dictates policy
    - Who can access what object and in what mode
- But doesn't solve the general confinement problem

Access Matrix of Figure A with Domains as Objects

domain \ object	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Access Matrix with *Copy* Rights

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

Access Matrix With *Owner* Rights

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

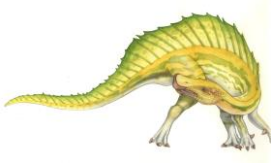
(b)

Modified Access Matrix of Figure B

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Implementation of Access Matrix

- Generally, a sparse matrix
- Option 1 – Global table
  - Store ordered triples **<domain, object, rights-set>** in table
  - A requested operation M on object O_j within domain D_i -> search table for < D_i, O_j, R_k >
    - with M ∈ R_k
  - But table could be large -> won't fit in main memory
  - Difficult to group objects (consider an object that all domains can read)



Implementation of Access Matrix (Cont.)

- Option 2 – Access lists for objects
  - Each column implemented as an access list for one object
  - Resulting per-object list consists of ordered pairs **<domain, rights-set>** defining all domains with non-empty set of access rights for the object
  - Easily extended to contain default set -> If M ∈ default set, also allow access

Implementation of Access Matrix (Cont.)

- Each column = Access-control list for one object
  Defines who can perform what operation

  Domain 1 = Read, Write
  Domain 2 = Read
  Domain 3 = Read

- Each Row = Capability List (like a key)
  For each domain, what operations allowed on what objects

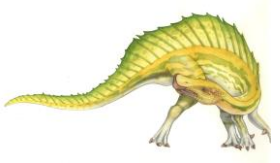
  Object F1 – Read

  Object F4 – Read, Write, Execute

  Object F5 – Read, Write, Delete, Copy

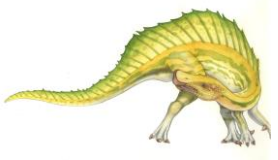
Implementation of Access Matrix (Cont.)

- Option 3 – Capability list for domains
  - Instead of object-based, list is domain based
  - **Capability list** for domain is list of objects together with operations allows on them
  - Object represented by its name or address, called a **capability**
  - Execute operation M on object O_j, process requests operation and specifies capability as parameter
    - Possession of capability means access is allowed
  - Capability list associated with domain but never directly accessible by domain
    - Rather, protected object, maintained by OS and accessed indirectly
    - Like a "secure pointer"
    - Idea can be extended up to applications



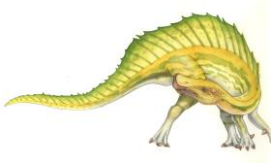
Implementation of Access Matrix (Cont.)

- Option 4 – Lock-key
  - Compromise between access lists and capability lists
  - Each object has list of unique bit patterns, called **locks**
  - Each domain as list of unique bit patterns called **keys**
  - Process in a domain can only access object if domain has key that matches one of the locks



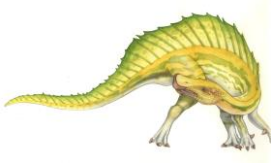
Comparison of Implementations

- Many trade-offs to consider
  - Global table is simple, but can be large
  - Access lists correspond to needs of users
    - Determining set of access rights for domain non-localized so difficult
    - Every access to an object must be checked
      - Many objects and access rights -> slow
  - Capability lists useful for localizing information for a given process
    - But revocation capabilities can be inefficient
  - Lock-key effective and flexible, keys can be passed freely from domain to domain, easy revocation



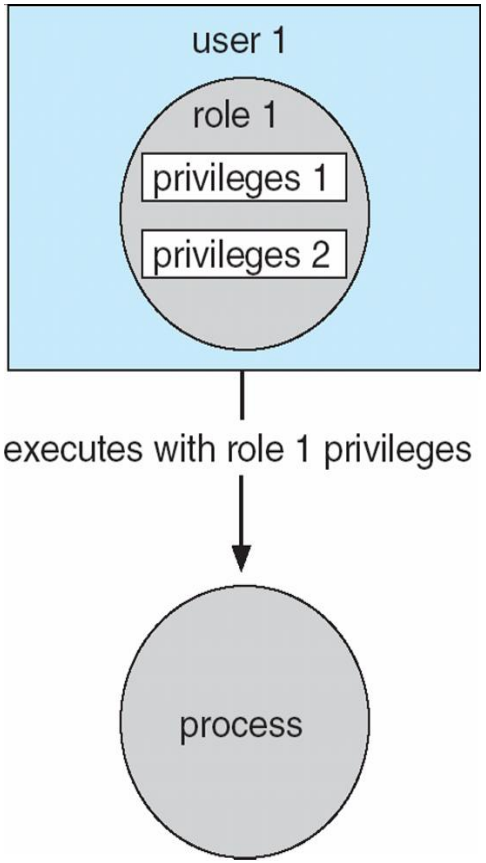
Comparison of Implementations (Cont.)

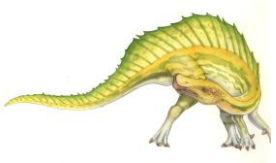
- Most systems use combination of access lists and capabilities
  - First access to an object -> access list searched
    - If allowed, capability created and attached to process
      - Additional accesses need not be checked
    - After last access, capability destroyed
    - Consider file system with ACLs per file



Access Control

- Protection can be applied to non-file resources
- Oracle Solaris 10 provides **role-based access control** (**RBAC**) to implement least privilege
  - ***Privilege*** is right to execute system call or use an option within a system call
  - Can be assigned to processes
  - Users assigned ***roles*** granting access to privileges and programs
    - Enable role via password to gain its privileges
  - Similar to access matrix



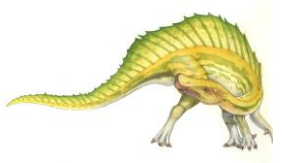


Revocation of Access Rights

- Various options to remove the access right of a domain to an object
  - **Immediate vs. delayed**
  - **Selective vs. general**
  - **Partial vs. total**
  - **Temporary vs. permanent**
- **Access List** – Delete access rights from access list
  - **Simple** – search access list and remove entry
  - **Immediate**, **general** or **selective**, **total** or **partial**, **permanent** or **temporary**

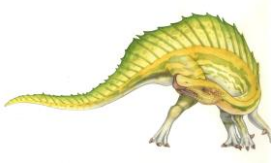
Revocation of Access Rights (Cont.)

- **Capability List** – Scheme required to locate capability in the system before capability can be revoked
  - **Reacquisition** – periodic delete, with require and denial if revoked
  - **Back-pointers** – set of pointers from each object to all capabilities of that object (Multics)
  - **Indirection** – capability points to global table entry which points to object – delete entry from global table, not selective (CAL)
  - **Keys** – unique bits associated with capability, generated when capability created
    - Master key associated with object, key matches master key for access
    - Revocation – create new master key
    - Policy decision of who can create and modify keys – object owner or others?



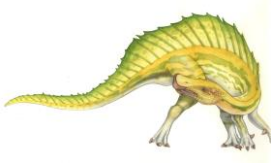
Capability-Based Systems

- Hydra
  - Fixed set of access rights known to and interpreted by the system
    - i.e. read, write, or execute each memory segment
    - User can declare other **auxiliary rights** and register those with protection system
    - Accessing process must hold capability and know name of operation
    - **Rights amplification** allowed by trustworthy procedures for a specific type
  - Interpretation of user-defined rights performed solely by user's program; system provides access protection for use of these rights
  - Operations on objects defined procedurally – procedures are objects accessed indirectly by capabilities
  - Solves the *problem of mutually suspicious subsystems*
  - Includes library of prewritten security routines



Capability-Based Systems (Cont.)

- Cambridge CAP System
  - Simpler but powerful
  - **Data capability** - provides standard read, write, execute of individual storage segments associated with object – implemented in microcode
  - **Software capability** -interpretation left to the subsystem, through its protected procedures
    - Only has access to its own subsystem
    - Programmers must learn principles and techniques of protection



Language-Based Protection

- Specification of protection in a programming language allows the high-level description of policies for the allocation and use of resources
- Language implementation can provide software for protection enforcement when automatic hardware-supported checking is unavailable
- Interpret protection specifications to generate calls on whatever protection system is provided by the hardware and the operating system

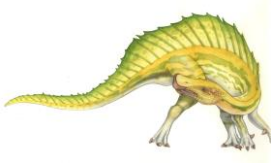
Protection in Java 2

- Protection is handled by the Java Virtual Machine (JVM)
- A **class** is assigned a protection domain when it is loaded by the JVM
- The protection domain indicates what operations the class can (and cannot) perform
- If a library **method** is invoked that performs a privileged operation, the stack is **inspected** to ensure the operation can be performed by the library
- Generally, Java's load-time and run-time checks enforce **type safety**
- Classes effectively **encapsulate** and protect data and methods from other classes

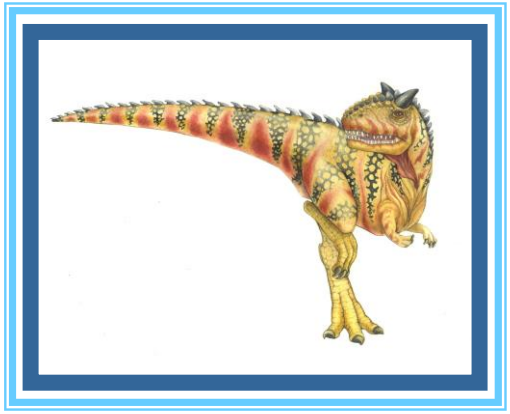
Stack Inspection

protection domain:	untrusted applet	URL loader	networking
socket permission:	none	*.lucent.com:80, connect	any

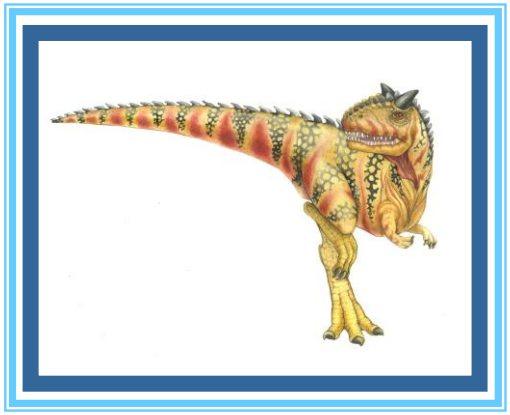
class:	gui:
	...
	get(url);
	open(addr);
	...
		get(URL u):
		...
		doPrivileged {
		open('proxy.lucent.com:80');
		}
		<request u from proxy>
		...
			open(Addr a):
			...
			checkPermission (a, connect);
			connect (a);
			...



End of Chapter 14



Chapter 15: Security



Chapter 15: Security

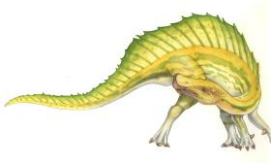
- The Security Problem
- Program Threats
- System and Network Threats
- Cryptography as a Security Tool
- User Authentication
- Implementing Security Defenses
- Firewalling to Protect Systems and Networks
- Computer-Security Classifications
- An Example: Windows 7

Objectives

- To discuss security threats and attacks
- To explain the fundamentals of encryption, authentication, and hashing
- To examine the uses of cryptography in computing
- To describe the various countermeasures to security attacks

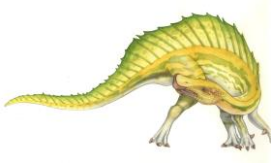
The Security Problem

- System **secure** if resources used and accessed as intended under all circumstances
  - Unachievable
- **Intruders** (**crackers**) attempt to breach security
- **Threat** is potential security violation
- **Attack** is attempt to breach security
- Attack can be accidental or malicious
- Easier to protect against accidental than malicious misuse



Security Violation Categories

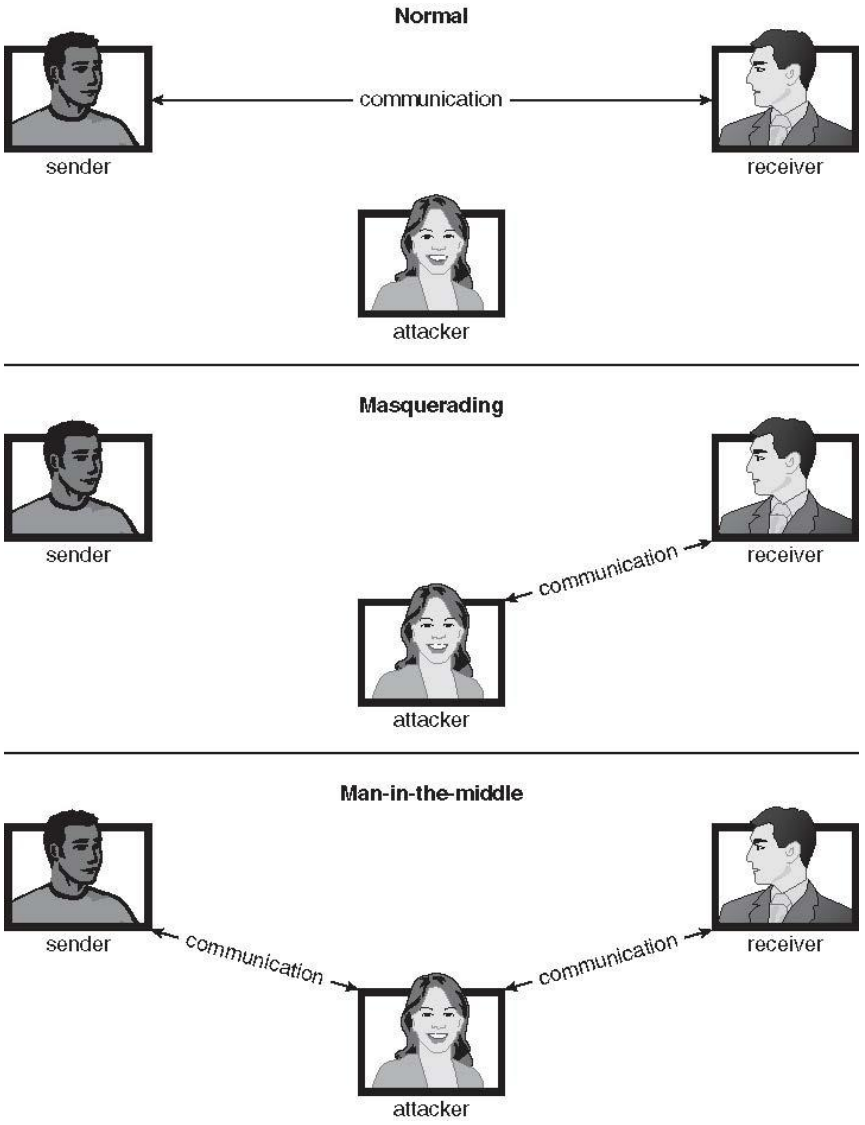
- **Breach of confidentiality**
  - Unauthorized reading of data
- **Breach of integrity**
  - Unauthorized modification of data
- **Breach of availability**
  - Unauthorized destruction of data
- **Theft of service**
  - Unauthorized use of resources
- **Denial of service (DOS)**
  - Prevention of legitimate use



Security Violation Methods

- **Masquerading** (breach **authentication**)
  - Pretending to be an authorized user to escalate privileges
- **Replay attack**
  - As is or with **message modification**
- **Man-in-the-middle attack**
  - Intruder sits in data flow, masquerading as sender to receiver and vice versa
- **Session hijacking**
  - Intercept an already-established session to bypass authentication

Standard Security Attacks



Security Measure Levels

- Impossible to have absolute security, but make cost to perpetrator sufficiently high to deter most intruders
- Security must occur at four levels to be effective:
  - **Physical**
    - Data centers, servers, connected terminals
  - **Human**
    - Avoid **social engineering**, **phishing**, **dumpster diving**
  - **Operating System**
    - Protection mechanisms, debugging
  - **Network**
    - Intercepted communications, interruption, DOS
- Security is as weak as the weakest link in the chain
- But can too much security be a problem?

Program Threats

- Many variations, many names
- **Trojan Horse**
  - Code segment that misuses its environment
  - Exploits mechanisms for allowing programs written by users to be executed by other users
  - **Spyware**, **pop-up browser windows**, **covert channels**
  - Up to 80% of spam delivered by spyware-infected systems
- **Trap Door**
  - Specific user identifier or password that circumvents normal security procedures
  - Could be included in a compiler
  - How to detect them?

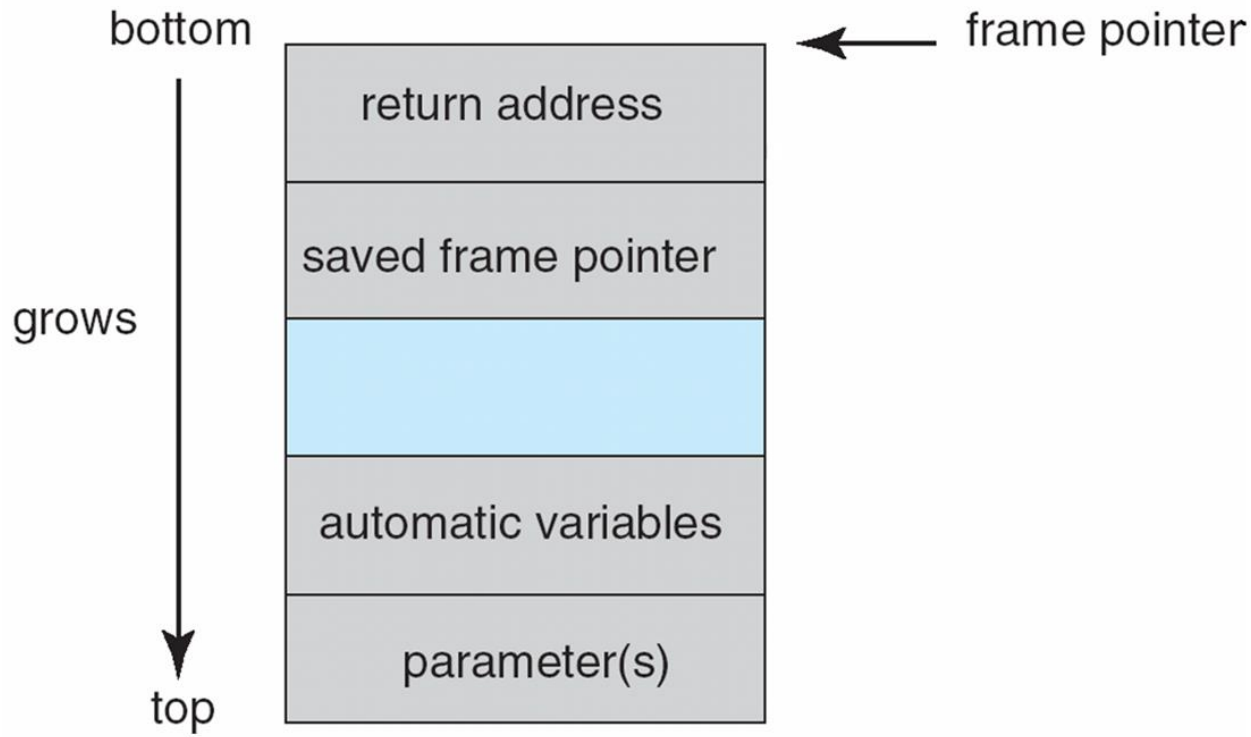
Program Threats (Cont.)

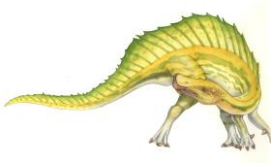
- **Logic Bomb**
  - Program that initiates a security incident under certain circumstances
- **Stack** and **Buffer Overflow**
  - Exploits a bug in a program (overflow either the stack or memory buffers)
  - Failure to check bounds on inputs, arguments
  - Write past arguments on the stack into the return address on stack
  - When routine returns from call, returns to hacked address
    - Pointed to code loaded onto stack that executes malicious code
  - Unauthorized user or privilege escalation

C Program with Buffer-overflow Condition

```
#include <stdio.h>
#define BUFFER SIZE 256
int main(int argc, char *argv[])
{
   char buffer[BUFFER SIZE];
   if (argc < 2)
       return -1;
   else {
       strcpy(buffer,argv[1]);
       return 0;
   }
}
```

Layout of Typical Stack Frame

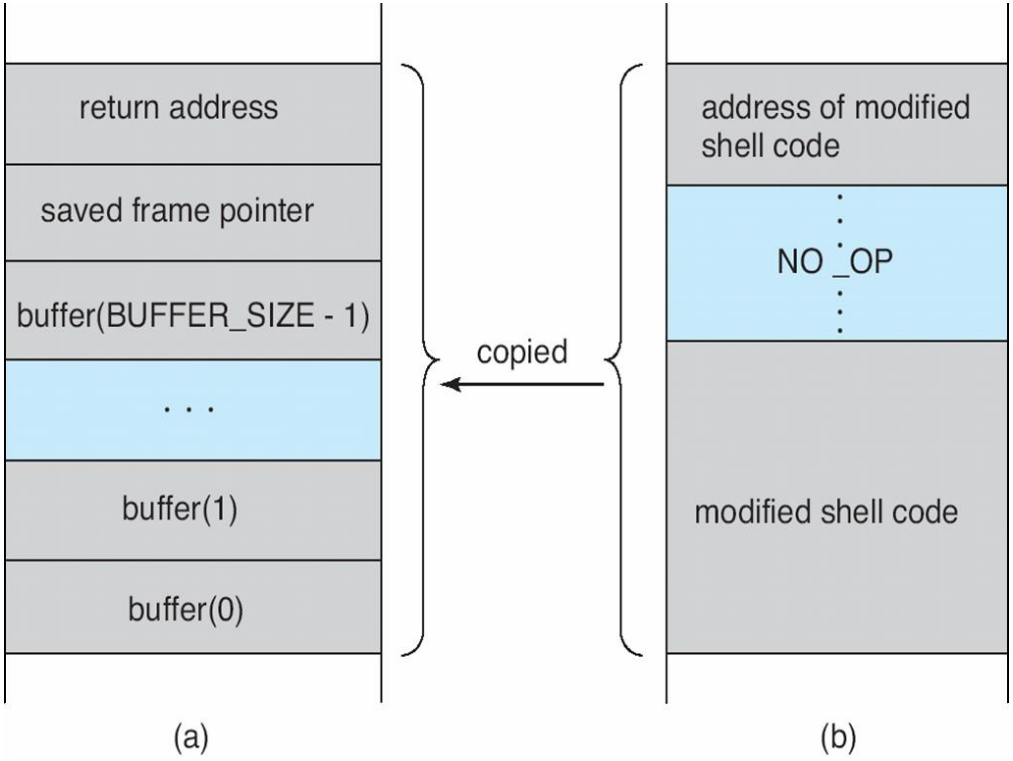




Modified Shell Code

```
#include <stdio.h>
int main(int argc, char *argv[])
{
   execvp(‘‘\bin\sh’’,‘‘\bin \sh’’, NULL);
   return 0;
}
```

Hypothetical Stack Frame



Before attack

After attack

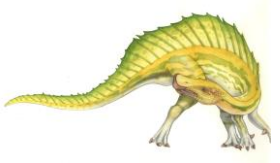
Great Programming Required?

- For the first step of determining the bug, and second step of writing exploit code, yes
- **Script kiddies** can run pre-written exploit code to attack a given system
- Attack code can get a shell with the processes' owner's permissions
  - Or open a network port, delete files, download a program, etc
- Depending on bug, attack can be executed across a network using allowed connections, bypassing firewalls
- Buffer overflow can be disabled by disabling stack execution or adding bit to page table to indicate "non-executable" state
  - Available in SPARC and x86
  - But still have security exploits

Program Threats (Cont.)

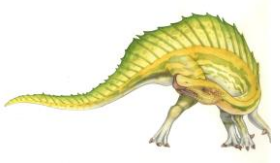
- **Viruses**
  - Code fragment embedded in legitimate program
  - Self-replicating, designed to infect other computers
  - Very specific to CPU architecture, operating system, applications
  - Usually borne via email or as a macro
  - Visual Basic Macro to reformat hard drive

```
Sub AutoOpen()
Dim oFS
  Set oFS = CreateObject(''Scripting.FileSystemObject'')
  vs = Shell(''c:command.com /k format c:'',vbHide)
End Sub
```

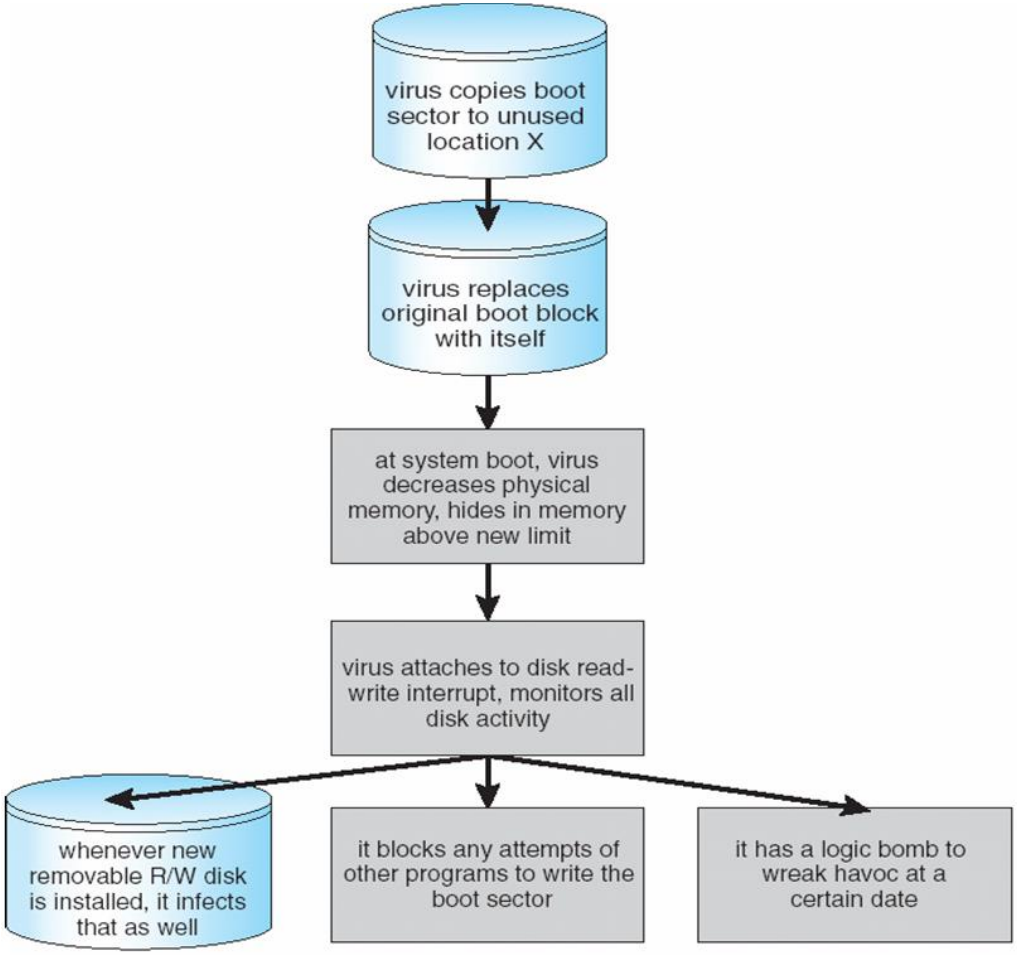


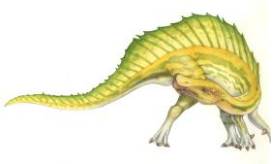
Program Threats (Cont.)

- **Virus dropper** inserts virus onto the system
- Many categories of viruses, literally many thousands of viruses
  - File / parasitic
  - Boot / memory
  - Macro
  - Source code
  - Polymorphic to avoid having a **virus signature**
  - Encrypted
  - Stealth
  - Tunneling
  - Multipartite
  - Armored



A Boot-sector Computer Virus



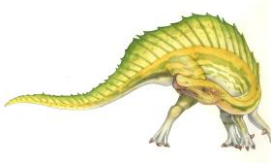


The Threat Continues

- Attacks still common, still occurring
- Attacks moved over time from science experiments to tools of organized crime
  - Targeting specific companies
  - Creating botnets to use as tool for spam and DDOS delivery
  - **Keystroke logger** to grab passwords, credit card numbers
- Why is Windows the target for most attacks?
  - Most common
  - Everyone is an administrator
    - Licensing required?
  - **Monoculture** considered harmful

System and Network Threats

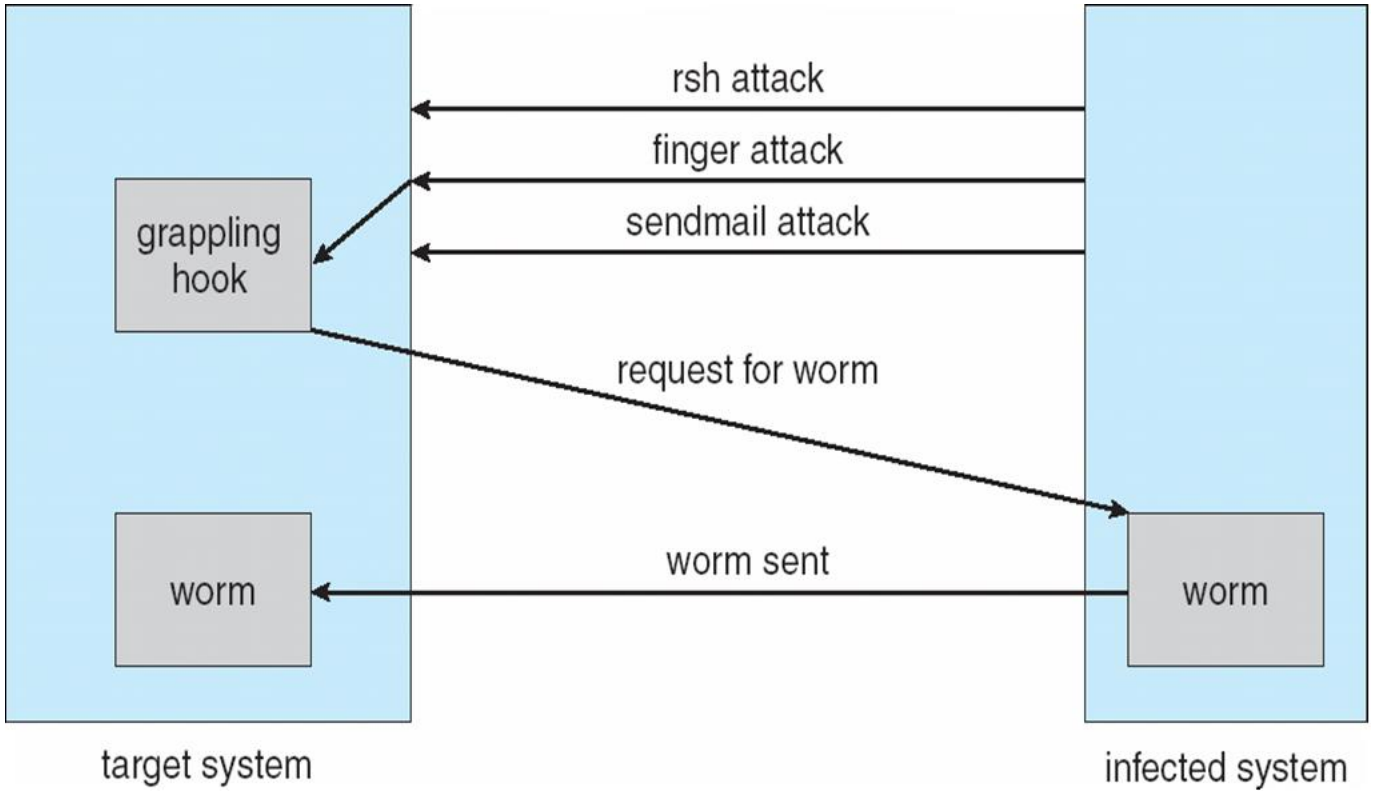
- Some systems “open” rather than **secure by default**
  - Reduce **attack surface**
  - But harder to use, more knowledge needed to administer
- Network threats harder to detect, prevent
  - Protection systems weaker
  - More difficult to have a shared secret on which to base access
  - No physical limits once system attached to internet
    - Or on network with system attached to internet
  - Even determining location of connecting system difficult
    - IP address is only knowledge

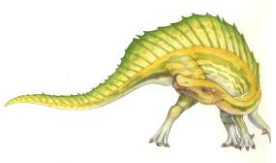


System and Network Threats (Cont.)

- **Worms** – use **spawn** mechanism; standalone program
- Internet worm
  - Exploited UNIX networking features (remote access) and bugs in *finger* and *sendmail* programs
  - Exploited trust-relationship mechanism used by *rsh* to access friendly systems without use of password
  - **Grappling hook** program uploaded main worm program
    - 99 lines of C code
  - Hooked system then uploaded main code, tried to attack connected systems
  - Also tried to break into other users accounts on local system via password guessing
  - If target system already infected, abort, except for every 7th time

The Morris Internet Worm



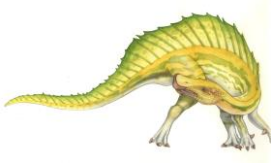


System and Network Threats (Cont.)

- **Port scanning**
  - Automated attempt to connect to a range of ports on one or a range of IP addresses
  - Detection of answering service protocol
  - Detection of OS and version running on system
  - `nmap` scans all ports in a given IP range for a response
  - `nessus` has a database of protocols and bugs (and exploits) to apply against a system
  - Frequently launched from **zombie systems**
    - To decrease trace-ability

System and Network Threats (Cont.)

- **Denial of Service**
  - Overload the targeted computer preventing it from doing any useful work
  - **Distributed denial-of-service** (**DDOS**) come from multiple sites at once
  - Consider the start of the IP-connection handshake (SYN)
    - How many started-connections can the OS handle?
  - Consider traffic to a web site
    - How can you tell the difference between being a target and being really popular?
  - Accidental – CS students writing bad `fork()` code
  - Purposeful – extortion, punishment



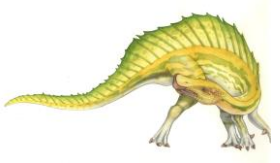
Sobig.F Worm

- More modern example
- Disguised as a photo uploaded to adult newsgroup via account created with stolen credit card
- Targeted Windows systems
- Had own SMTP engine to mail itself as attachment to everyone in infect system's address book
- Disguised with innocuous subject lines, looking like it came from someone known
- Attachment was executable program that created **`WINPPR23.EXE`** in default Windows system directory
  Plus the Windows Registry

```
[HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run]
   "TrayX" = %windir%\winppr32.exe /sinc
[HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run]
   "TrayX" = %windir%\winppr32.exe /sinc
```

Cryptography as a Security Tool

- Broadest security tool available
  - Internal to a given computer, source and destination of messages can be known and protected
    - OS creates, manages, protects process IDs, communication ports
  - Source and destination of messages on network cannot be trusted without cryptography
    - Local network – IP address?
      - Consider unauthorized host added
    - WAN / Internet – how to establish authenticity
      - Not via IP address

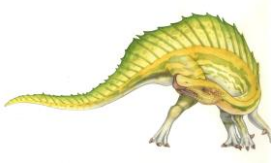


Cryptography

- Means to constrain potential senders (*sources*) and / or receivers (*destinations*) of *messages*
  - Based on secrets (**keys**)
  - Enables
    - Confirmation of source
    - Receipt only by certain destination
    - Trust relationship between sender and receiver

Encryption

- Constrains the set of possible receivers of a message
- **Encryption** algorithm consists of
  - Set *K* of keys
  - Set *M* of Messages
  - Set *C* of ciphertexts (encrypted messages)
  - A function $E : K \rightarrow (M \rightarrow C)$. That is, for each $k \in K$, E_k is a function for generating ciphertexts from messages
    - Both *E* and E_k for any *k* should be efficiently computable functions
  - A function $D : K \rightarrow (C \rightarrow M)$. That is, for each $k \in K$, D_k is a function for generating messages from ciphertexts
    - Both *D* and D_k for any *k* should be efficiently computable functions

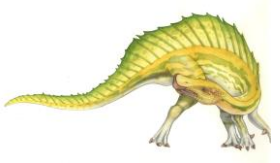


Encryption (Cont.)

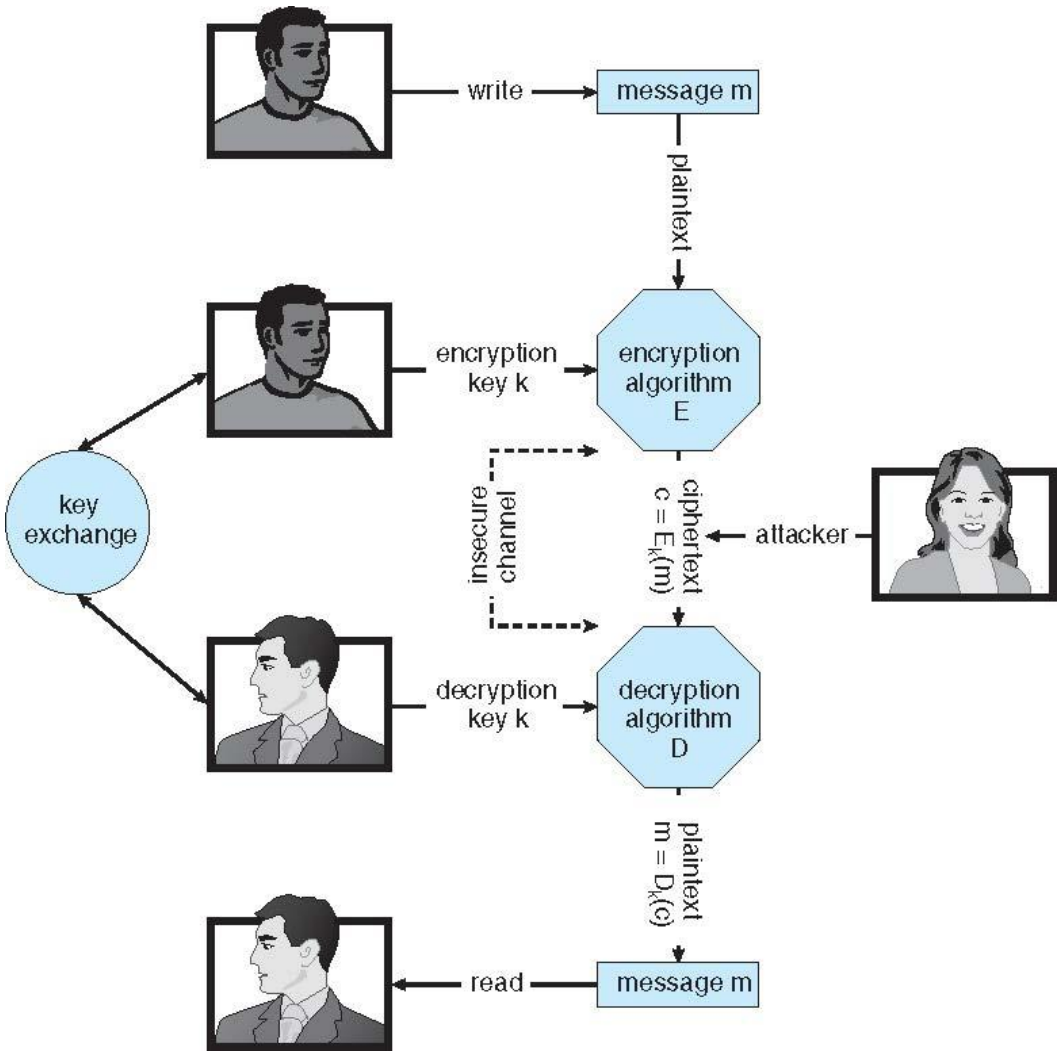
- An encryption algorithm must provide this essential property: Given a ciphertext $c \in C$, a computer can compute m such that $E_k(m) = c$ only if it possesses *k*
  - Thus, a computer holding *k* can decrypt ciphertexts to the plaintexts used to produce them, but a computer not holding *k* cannot decrypt ciphertexts
  - Since ciphertexts are generally exposed (for example, sent on the network), it is important that it be infeasible to derive *k* from the ciphertexts

Symmetric Encryption

- Same key used to encrypt and decrypt
  - Therefore *k* must be kept secret
- DES was most commonly used symmetric block-encryption algorithm (created by US Govt)
  - Encrypts a block of data at a time
  - Keys too short so now considered insecure
- Triple-DES considered more secure
  - Algorithm used 3 times using 2 or 3 keys
  - For example $c = E_{k3}(D_{k2}(E_{k1}(m)))$
- 2001 NIST adopted new block cipher - Advanced Encryption Standard (**AES**)
  - Keys of 128, 192, or 256 bits, works on 128 bit blocks
- RC4 is most common symmetric stream cipher, but known to have vulnerabilities
  - Encrypts/decrypts a stream of bytes (i.e., wireless transmission)
  - Key is a input to pseudo-random-bit generator
    - Generates an infinite **keystream**



Secure Communication over Insecure Medium



Asymmetric Encryption

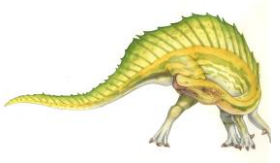
- **Public-key encryption** based on each user having two keys:
  - **public key** – published key used to encrypt data
  - **private key** – key known only to individual user used to decrypt data
- Must be an encryption scheme that can be made public without making it easy to figure out the decryption scheme
  - Most common is **RSA** block cipher
  - Efficient algorithm for testing whether or not a number is prime
  - No efficient algorithm is know for finding the prime factors of a number

Asymmetric Encryption (Cont.)

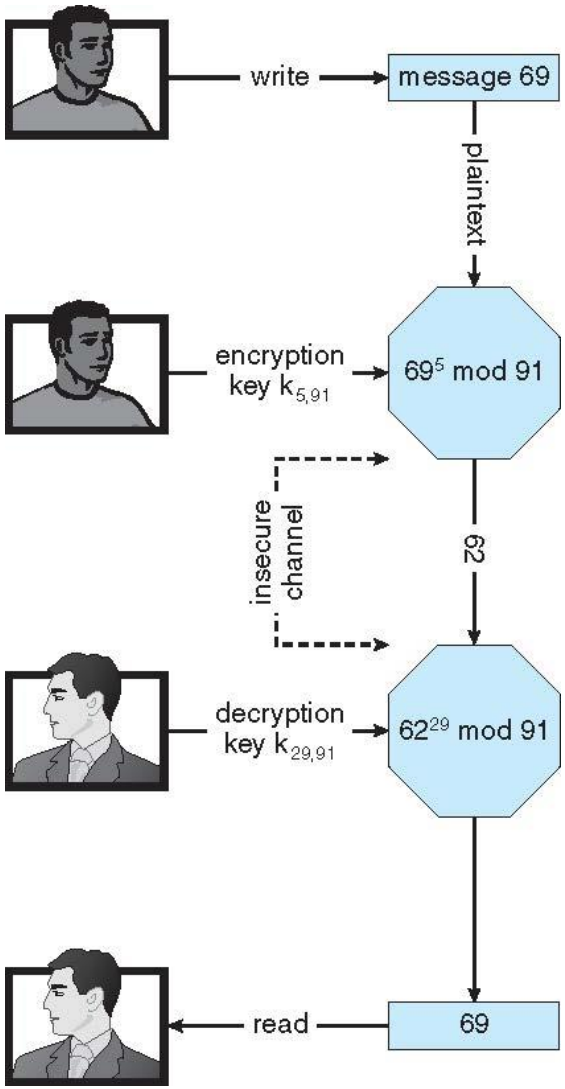
- Formally, it is computationally infeasible to derive $k_{d,N}$ from $k_{e,N}$, and so k_e need not be kept secret and can be widely disseminated
  - k_e is the **public key**
  - k_d is the **private key**
  - N is the product of two large, randomly chosen prime numbers p and q (for example, p and q are 512 bits each)
  - Encryption algorithm is $E_{ke,N}(m) = m^{k_e} \bmod N$, where k_e satisfies $k_e k_d \bmod (p-1)(q-1) = 1$
  - The decryption algorithm is then $D_{kd,N}(c) = c^{k_d} \bmod N$

Asymmetric Encryption Example

- For example. make $p = 7$and $q = 13$
- We then calculate $N = 7*13 = 91$ and $(p-1)(q-1) = 72$
- We next select k_e relatively prime to 72 and< 72, yielding 5
- Finally, we calculate k_d such that $k_e k_d$ mod 72 = 1, yielding 29
- We how have our keys
  - Public key, $k_{e,N} = 5, 91$
  - Private key, $k_{d,N} = 29, 91$
- Encrypting the message 69 with the public key results in the cyphertext 62
- Cyphertext can be decoded with the private key
  - Public key can be distributed in cleartext to anyone who wants to communicate with holder of public key

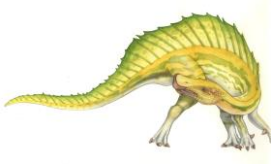


Encryption using RSA Asymmetric Cryptography



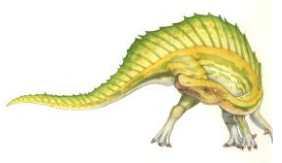
Cryptography (Cont.)

- Note symmetric cryptography based on transformations, asymmetric based on mathematical functions
  - Asymmetric much more compute intensive
  - Typically not used for bulk data encryption



Authentication

- Constraining set of potential senders of a message
  - Complementary to encryption
  - Also can prove message unmodified
- Algorithm components
  - A set K of keys
  - A set M of messages
  - A set A of authenticators
  - A function $S : K \rightarrow (M \rightarrow A)$
    - That is, for each $k \in K$, S_k is a function for generating authenticators from messages
    - Both S and S_k for any k should be efficiently computable functions
  - A function $V : K \rightarrow (M \times A \rightarrow \{\text{true, false}\})$. That is, for each $k \in K$, V_k is a function for verifying authenticators on messages
    - Both V and V_k for any k should be efficiently computable functions

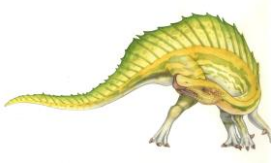


Authentication (Cont.)

- For a message *m*, a computer can generate an authenticator $a \in A$ such that $V_k(m, a)$ = `true` only if it possesses *k*
- Thus, computer holding *k* can generate authenticators on messages so that any other computer possessing *k* can verify them
- Computer not holding *k* cannot generate authenticators on messages that can be verified using V_k
- Since authenticators are generally exposed (for example, they are sent on the network with the messages themselves), it must not be feasible to derive *k* from the authenticators
- Practically, if $V_k(m,a)$ = **`true`** then we know *m* has not been modified and that send of message has *k*
  - If we share *k* with only one entity, know where the message originated

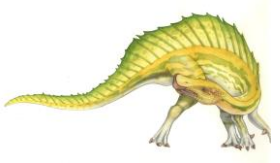
Authentication – Hash Functions

- Basis of authentication
- Creates small, fixed-size block of data **message digest** (**hash value)** from *m*
- Hash Function *H* must be collision resistant on *m*
  - Must be infeasible to find an $m' \neq m$ such that $H(m) = H(m')$
- If $H(m) = H(m')$, then $m = m'$
  - The message has not been modified
- Common message-digest functions include **MD5**, which produces a 128-bit hash, and **SHA-1**, which outputs a 160-bit hash
- Not useful as authenticators
  - For example *H(m)* can be sent with a message
    - But if *H* is known someone could modify *m* to *m'* and recompute *H(m')* and modification not detected
    - So must authenticate *H(m)*



Authentication - MAC

- Symmetric encryption used in **message-authentication code** (**MAC**) authentication algorithm
- Cryptographic checksum generated from message using secret key
  - Can securely authenticate short values
- If used to authenticate *H(m)* for an *H* that is collision resistant, then obtain a way to securely authenticate long message by hashing them first
- Note that *k* is needed to compute both S_k and V_k, so anyone able to compute one can compute the other



Authentication – Digital Signature

- Based on asymmetric keys and digital signature algorithm
- Authenticators produced are **digital signatures**
- Very useful – ***anyone*** can verify authenticity of a message
- In a digital-signature algorithm, computationally infeasible to derive k_s from k_v
  - V is a one-way function
  - Thus, k_v is the public key and k_s is the private key
- Consider the RSA digital-signature algorithm
  - Similar to the RSA encryption algorithm, but the key use is reversed
  - Digital signature of message $S_{ks}(m) = H(m)^{ks} \bmod N$
  - The key k_s again is a pair (d, N), where N is the product of two large, randomly chosen prime numbers p and q
  - Verification algorithm is $V_{kv}(m, a)$ $(a^{kv} \bmod N \stackrel{?}{=} H(m))$
    - Where k_v satisfies $k_v k_s \bmod (p-1)(q-1) = 1$

Authentication (Cont.)

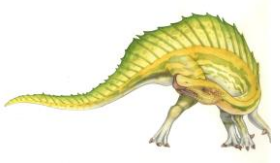
- Why authentication if a subset of encryption?
  - Fewer computations (except for RSA digital signatures)
  - Authenticator usually shorter than message
  - Sometimes want authentication but not confidentiality
    - Signed patches et al
  - Can be basis for **non-repudiation**

Key Distribution

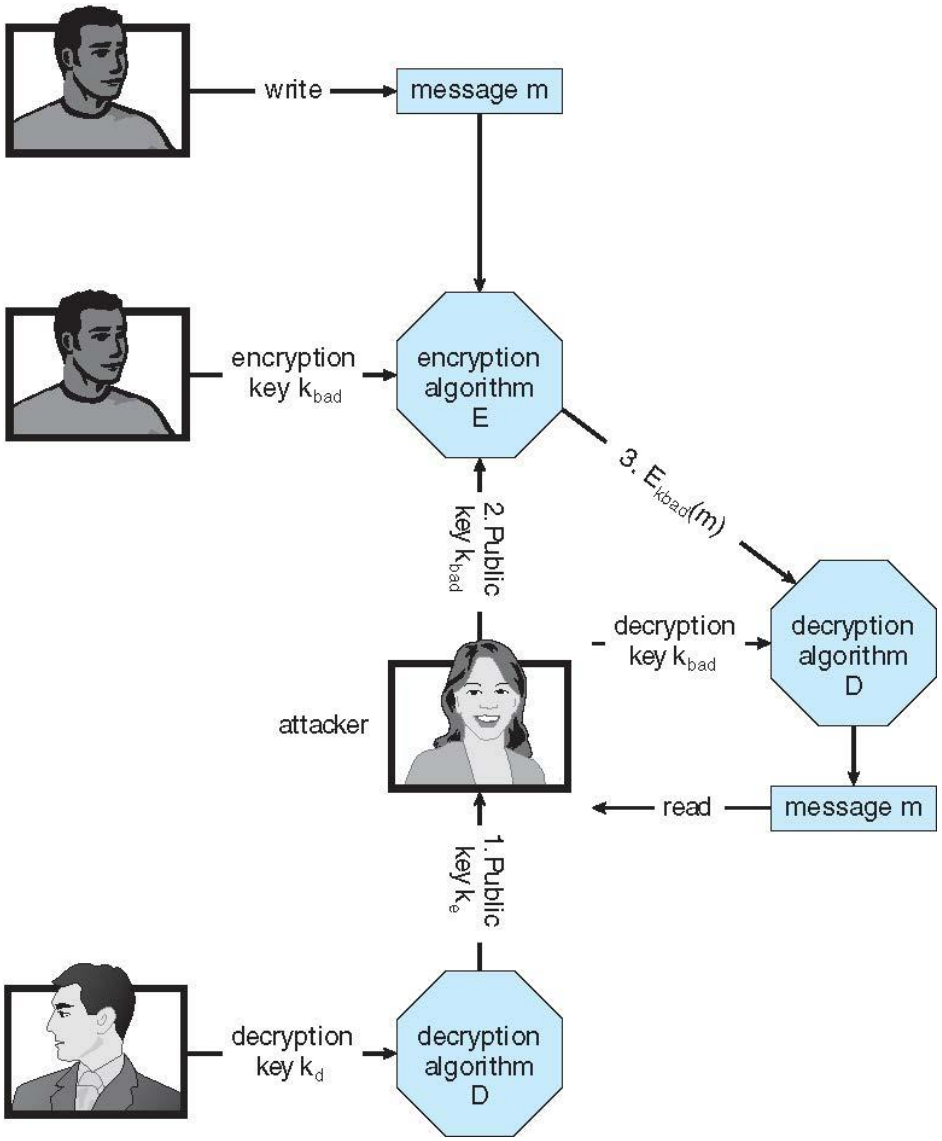
- Delivery of symmetric key is huge challenge
  - Sometimes done **out-of-band**
- Asymmetric keys can proliferate – stored on **key ring**
  - Even asymmetric key distribution needs care – man-in-the-middle attack

Digital Certificates

- Proof of who or what owns a public key
- Public key digitally signed a trusted party
- Trusted party receives proof of identification from entity and certifies that public key belongs to entity
- **Certificate authority** are trusted party – their public keys included with web browser distributions
  - They vouch for other authorities via digitally signing their keys, and so on



Man-in-the-middle Attack on Asymmetric Cryptography



Implementation of Cryptography

- Can be done at various **layers** of ISO Reference Model
  - SSL at the Transport layer
  - Network layer is typically **IPSec**
    - **IKE** for key exchange
    - Basis of **Virtual Private Networks (VPNs)**
- Why not just at lowest level?
  - Sometimes need more knowledge than available at low levels
    - i.e. User authentication
    - i.e. e-mail delivery

OSI model

7. Application Layer

NNTP · SIP · SSI · DNS · FTP · Gopher · HTTP · NFS · NTP · SMPP · SMTP · SNMP · Telnet · Netconf · (more)

6. Presentation Layer

MIME · XDR · TLS · SSL

5. Session Layer

Named Pipes · NetBIOS · SAP · L2TP · PPTP · SPDY

4. Transport Layer

TCP · UDP · SCTP · DCCP · SPX

3. Network Layer

IP (IPv4, IPv6) · ICMP · IPsec · IGMP · IPX · AppleTalk

2. Data Link Layer

ATM · SDLC · HDLC · ARP · CSLIP · SLIP · GFP · PLIP · IEEE 802.3 · Frame Relay · ITU-T G.hn DLL · PPP · X.25 · Network Switch · DHCP

1. Physical Layer

EIA/TIA-232 · EIA/TIA-449 · ITU-T V-Series · I.430 · I.431 · POTS · PDH · SONET/SDH · PON · OTN · DSL · IEEE 802.3 · IEEE 802.11 · IEEE 802.15 · IEEE 802.16 · IEEE 1394 · ITU-T G.hn PHY · USB · Bluetooth · Hubs

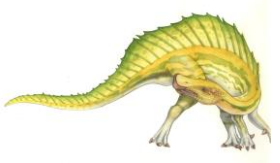
This box: view · talk · edit

OSI Model			
	Data unit	Layer	Function
Host layers	Data	7. Application	Network process to application
		6. Presentation	Data representation, encryption and decryption, convert machine dependent data to machine independent data
		5. Session	Interhost communication
	Segments	4. Transport	End-to-end connections and reliability, flow control
Media layers	Packet/Datagram	3. Network	Path determination and logical addressing
	Frame	2. Data Link	Physical addressing
	Bit	1. Physical	Media, signal and binary transmission

Source: http://en.wikipedia.org/wiki/OSI_model

Encryption Example - SSL

- Insertion of cryptography at one layer of the ISO network model (the transport layer)
- SSL – Secure Socket Layer (also called TLS)
- Cryptographic protocol that limits two computers to only exchange messages with each other
  - Very complicated, with many variations
- Used between web servers and browsers for secure communication (credit card numbers)
- The server is verified with a **certificate** assuring client is talking to correct server
- Asymmetric cryptography used to establish a secure **session key** (symmetric encryption) for bulk of communication during session
- Communication between each computer then uses symmetric key cryptography
- More details in textbook

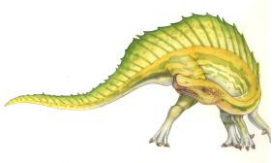


User Authentication

- Crucial to identify user correctly, as protection systems depend on user ID
- User identity most often established through **passwords**, can be considered a special case of either keys or capabilities
- Passwords must be kept secret
  - Frequent change of passwords
  - History to avoid repeats
  - Use of “non-guessable” passwords
  - Log all invalid access attempts (but not the passwords themselves)
  - Unauthorized transfer
- Passwords may also either be encrypted or allowed to be used only once
  - Does encrypting passwords solve the exposure problem?
    - Might solve **sniffing**
    - Consider **shoulder surfing**
    - Consider Trojan horse keystroke logger
    - How are passwords stored at authenticating site?

Passwords

- Encrypt to avoid having to keep secret
  - But keep secret anyway (i.e. Unix uses superuser-only readably file `/etc/shadow`)
  - Use algorithm easy to compute but difficult to invert
  - Only encrypted password stored, never decrypted
  - Add “salt” to avoid the same password being encrypted to the same value
- One-time passwords
  - Use a function based on a seed to compute a password, both user and computer
  - Hardware device / calculator / key fob to generate the password
    - Changes very frequently
- Biometrics
  - Some physical attribute (fingerprint, hand scan)
- Multi-factor authentication
  - Need two or more factors for authentication
    - i.e. USB “dongle”, biometric measure, and password



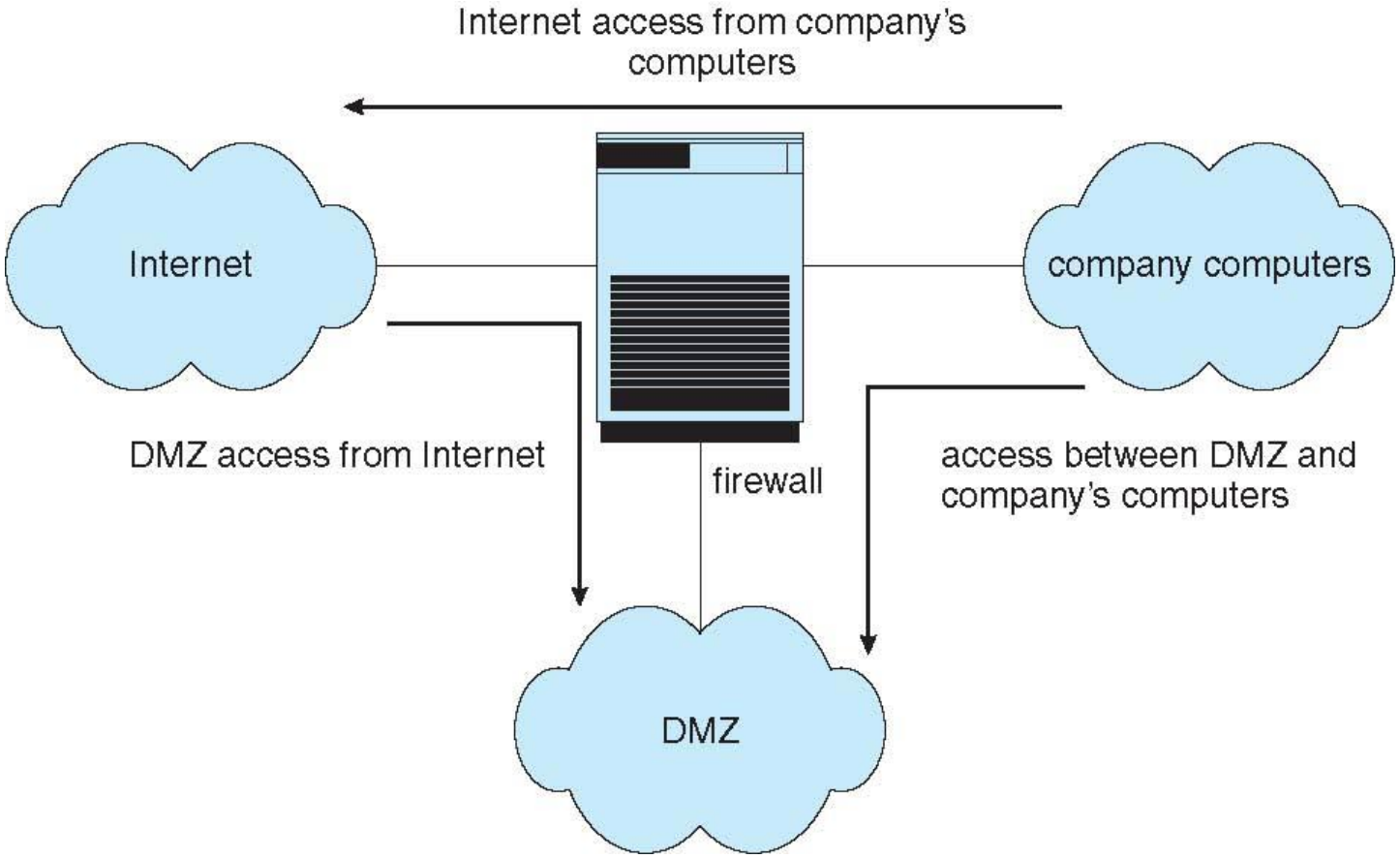
Implementing Security Defenses

- **Defense in depth** is most common security theory – multiple layers of security
- **Security policy** describes what is being secured
- Vulnerability assessment compares real state of system / network compared to security policy
- Intrusion detection endeavors to detect attempted or successful intrusions
  - **Signature-based** detection spots known bad patterns
  - **Anomaly detection** spots differences from normal behavior
    - Can detect **zero-day** attacks
  - **False-positives** and **false-negatives** a problem
- Virus protection
  - Searching all programs or programs at execution for known virus patterns
  - Or run in **sandbox** so can't damage system
- Auditing, accounting, and logging of all or specific system or network activities
- Practice **safe computing** – avoid sources of infection, download from only "good" sites, etc

Firewalling to Protect Systems and Networks

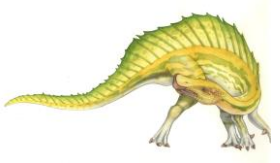
- A network **firewall** is placed between trusted and untrusted hosts
  - The firewall limits network access between these two **security domains**
- Can be tunneled or spoofed
  - Tunneling allows disallowed protocol to travel within allowed protocol (i.e., telnet inside of HTTP)
  - Firewall rules typically based on host name or IP address which can be spoofed
- **Personal firewall** is software layer on given host
  - Can monitor / limit traffic to and from the host
- **Application proxy firewall** understands application protocol and can control them (i.e., SMTP)
- **System-call firewall** monitors all important system calls and apply rules to them (i.e., this program can execute that system call)

Network Security Through Domain Separation Via Firewall



Computer Security Classifications

- U.S. Department of Defense outlines four divisions of computer security: **A**, **B**, **C**, and **D**
- **D** – Minimal security
- **C** – Provides discretionary protection through auditing
  - Divided into **C1** and **C2**
    - **C1** identifies cooperating users with the same level of protection
    - **C2** allows user-level access control
- **B** – All the properties of **C**, however each object may have unique sensitivity labels
  - Divided into **B1**, **B2**, and **B3**
- **A** – Uses formal design and verification techniques to ensure security



Example: Windows 7

- Security is based on **user accounts**
  - Each user has unique security ID
  - Login to ID creates **security access token**
    - Includes security ID for user, for user's groups, and special privileges
    - Every process gets copy of token
    - System checks token to determine if access allowed or denied
- Uses a **subject** model to ensure access security
  - A subject tracks and manages permissions for each program that a user runs
- Each object in Windows has a security attribute defined by a security descriptor
  - For example, a file has a security descriptor that indicates the access permissions for all users

Example: Windows 7 (Cont.)

- Win added mandatory integrity controls – assigns **integrity label** to each securable object and subject
  - Subject must have access requested in discretionary access-control list to gain access to object
- Security attributes described by security descriptor
  - Owner ID, group security ID, discretionary access-control list, system access-control list

End of Chapter 15