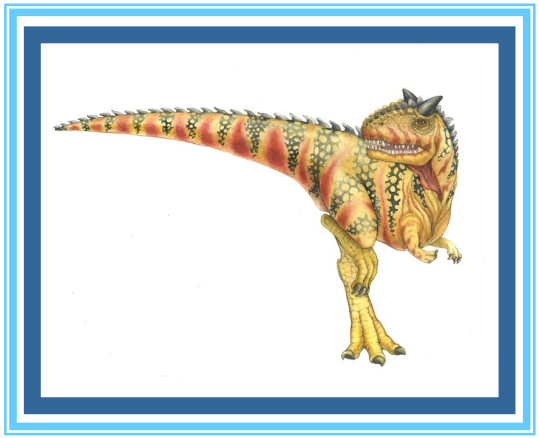
Chapter 13: File System Implementation



Chapter 13: File System Implementation

- File-System Structure
- I/O Hardware
- Polling and Interrupt
- Direct Memory Access
- File-System Implementation
- Allocation Methods
- Unix Inodes
- Efficiency and Performance
- Log-structured File Systems

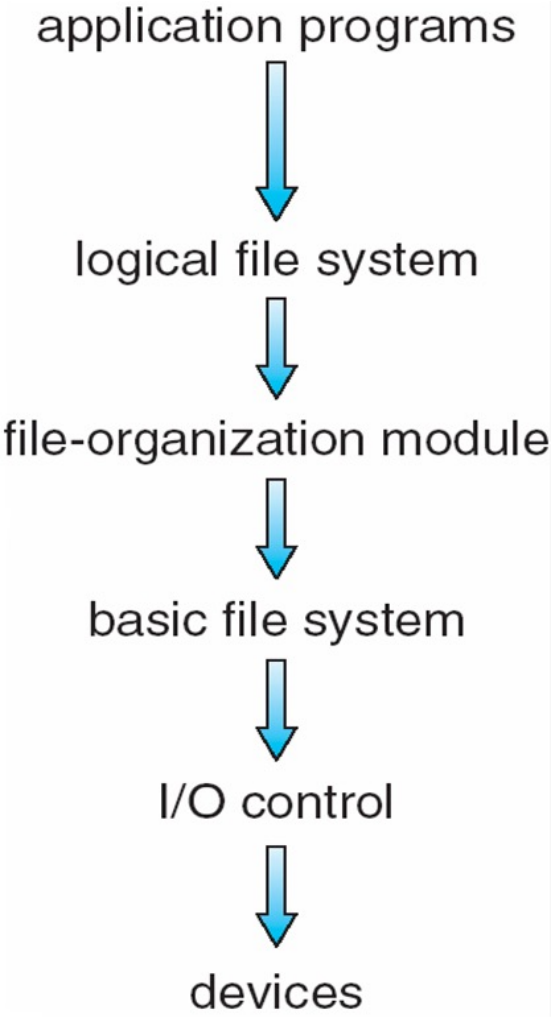
Objectives

- To describe the details of implementing local file systems and directory structures
- 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

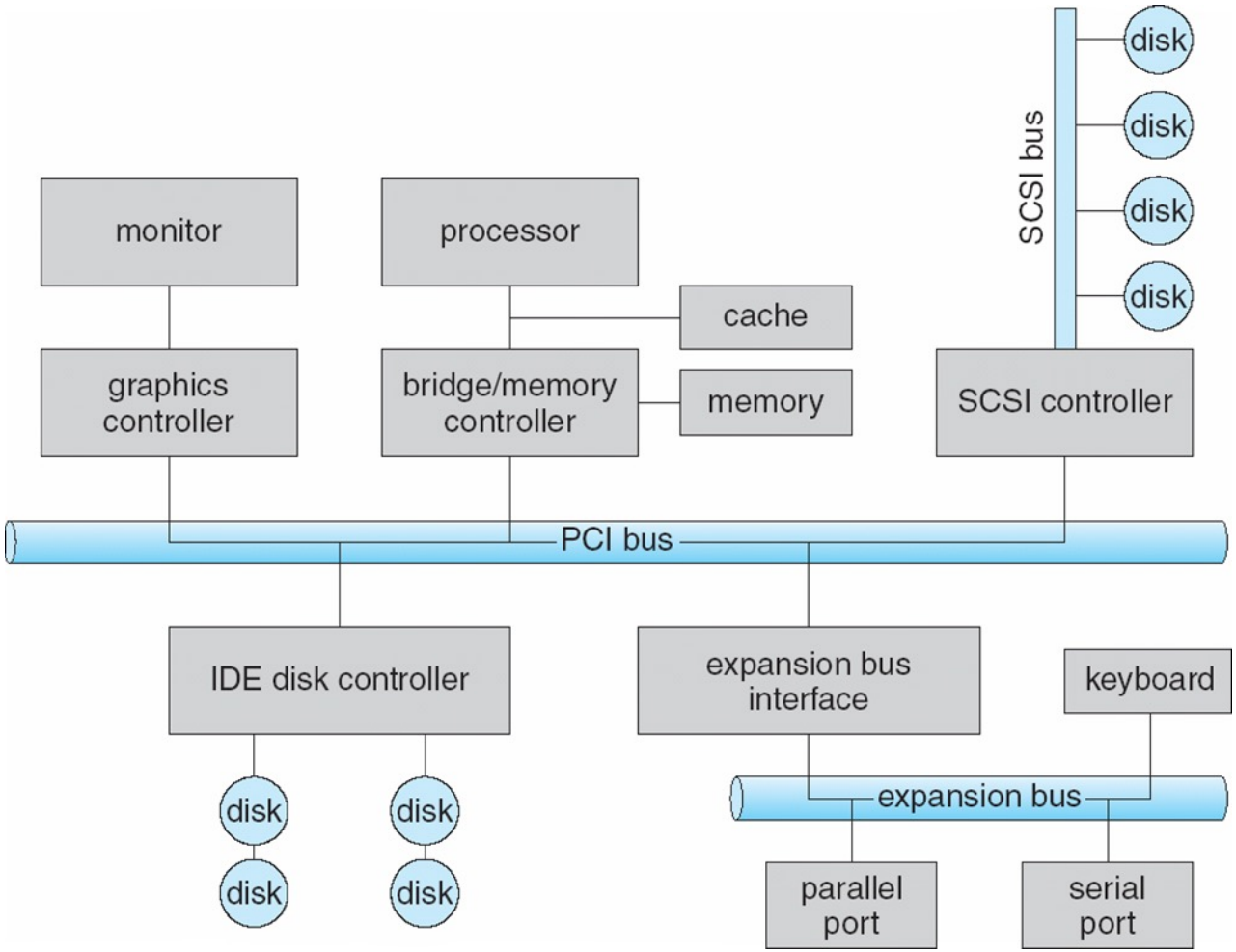
Layered File System



I/O Hardware

- Incredible variety of I/O devices
  - Storage
  - Transmission
  - Human-interface
- Common concepts – signals from I/O devices interface with computer
  - **Port** – connection point for device
  - **Bus** - **daisy chain** or shared direct access
    - **PCI** bus common in PCs and servers, PCI Express (**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

A Typical PC Bus Structure



I/O Hardware (Cont.)

- I/O instructions control devices
- Devices usually have registers where device driver places commands, addresses, and data to write, or read data from registers after command execution
  - Data-in register, data-out register, status register, control register
  - Typically 1-4 bytes, or FIFO buffer
- Devices have addresses, used by
  - Direct I/O instructions
  - **Memory-mapped I/O**
    - Device data and command registers mapped to processor address space
    - Especially for large address spaces (graphics)

Device I/O Port Locations on PCs (partial)

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

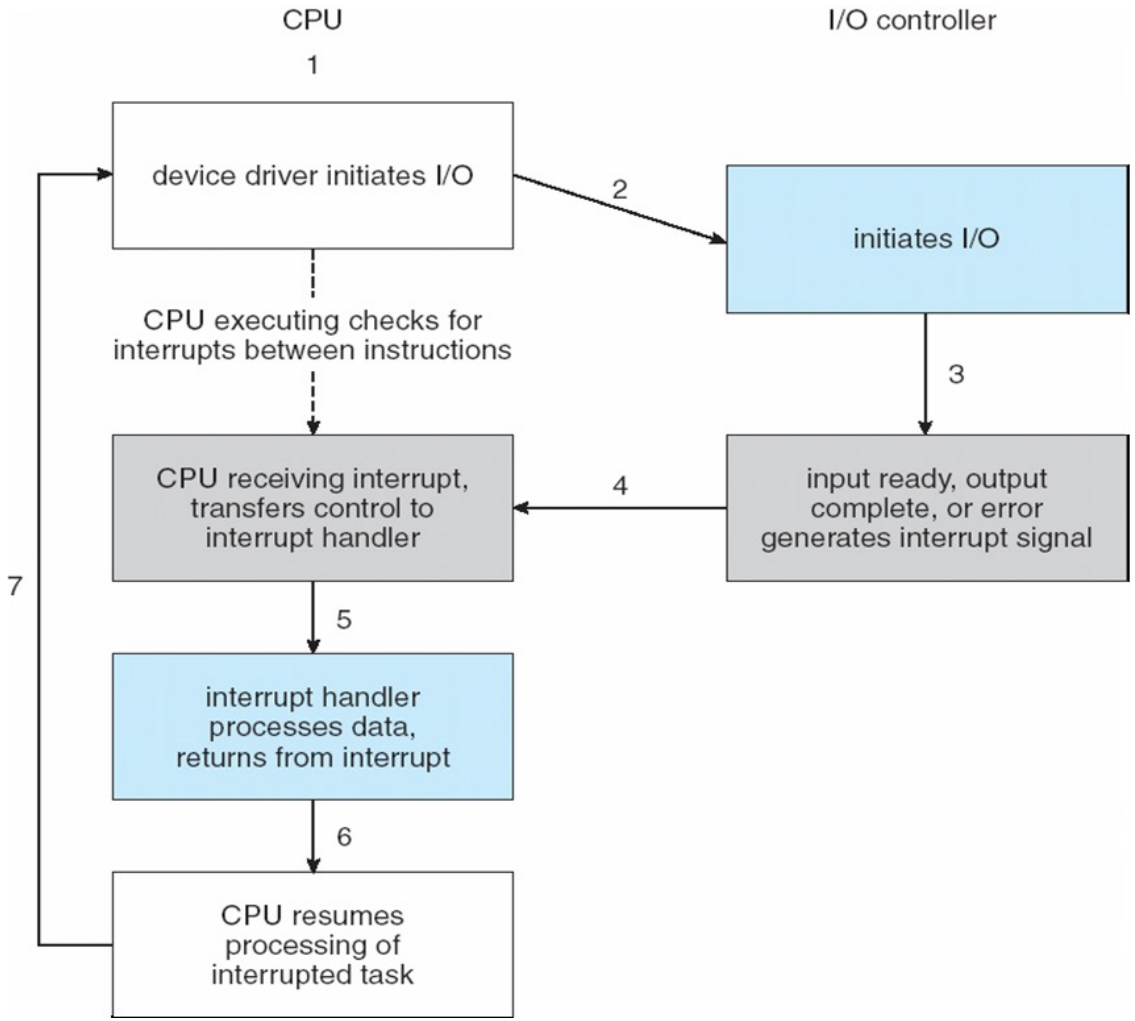
Polling

- For each byte of I/O
  1. Read busy bit from status register until 0
  2. Host sets read or write bit and if write copies data into data-out register
  3. Host sets command-ready bit
  4. Controller sets busy bit, executes transfer
  5. Controller clears busy bit, error bit, command-ready bit when transfer done
- Step 1 is **busy-wait** cycle to wait for I/O from device
  - Read status, logical-and to extract status bit, branch if not zero
  - Reasonable if device is fast
  - But inefficient if device slow
  - How to be more efficient if non-zero infrequently?

Interrupts

- CPU **Interrupt-request line** triggered by I/O device
  - Checked by processor after each instruction
- **Interrupt handler** receives interrupts
  - **Maskable** to ignore or delay some interrupts
- **Interrupt vector** to dispatch interrupt to correct handler
  - Context switch at start and end
  - Based on priority
  - Some **nonmaskable**
  - Interrupt chaining if more than one device at same interrupt number

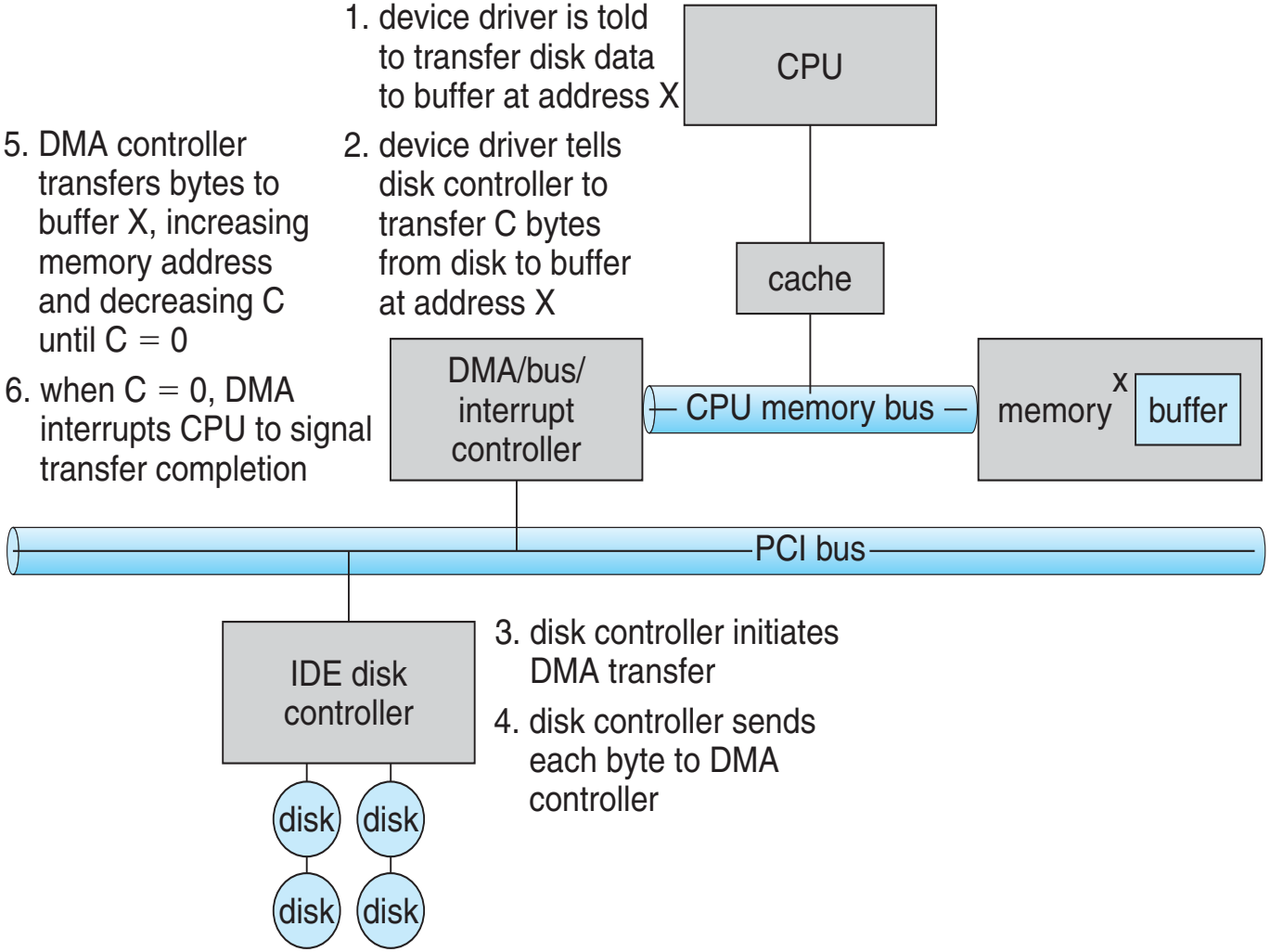
Interrupt-Driven I/O Cycle



Direct Memory Access

- DMA was developed to lessen the burden on the CPU. DMA uses a special-purpose processor called a DMA controller and copies data in chunks. Bypasses CPU to transfer data directly between I/O device and memory
- Used to avoid **programmed I/O** (one byte at a time) for large data movement
- OS writes DMA command block into memory
  - Source and destination addresses
  - Read or write mode
  - Count of bytes
  - Writes location of command block to DMA controller
  - Bus mastering of DMA controller – grabs bus from CPU
    - **Cycle stealing** from CPU but still much more efficient
  - When done, interrupts to signal completion

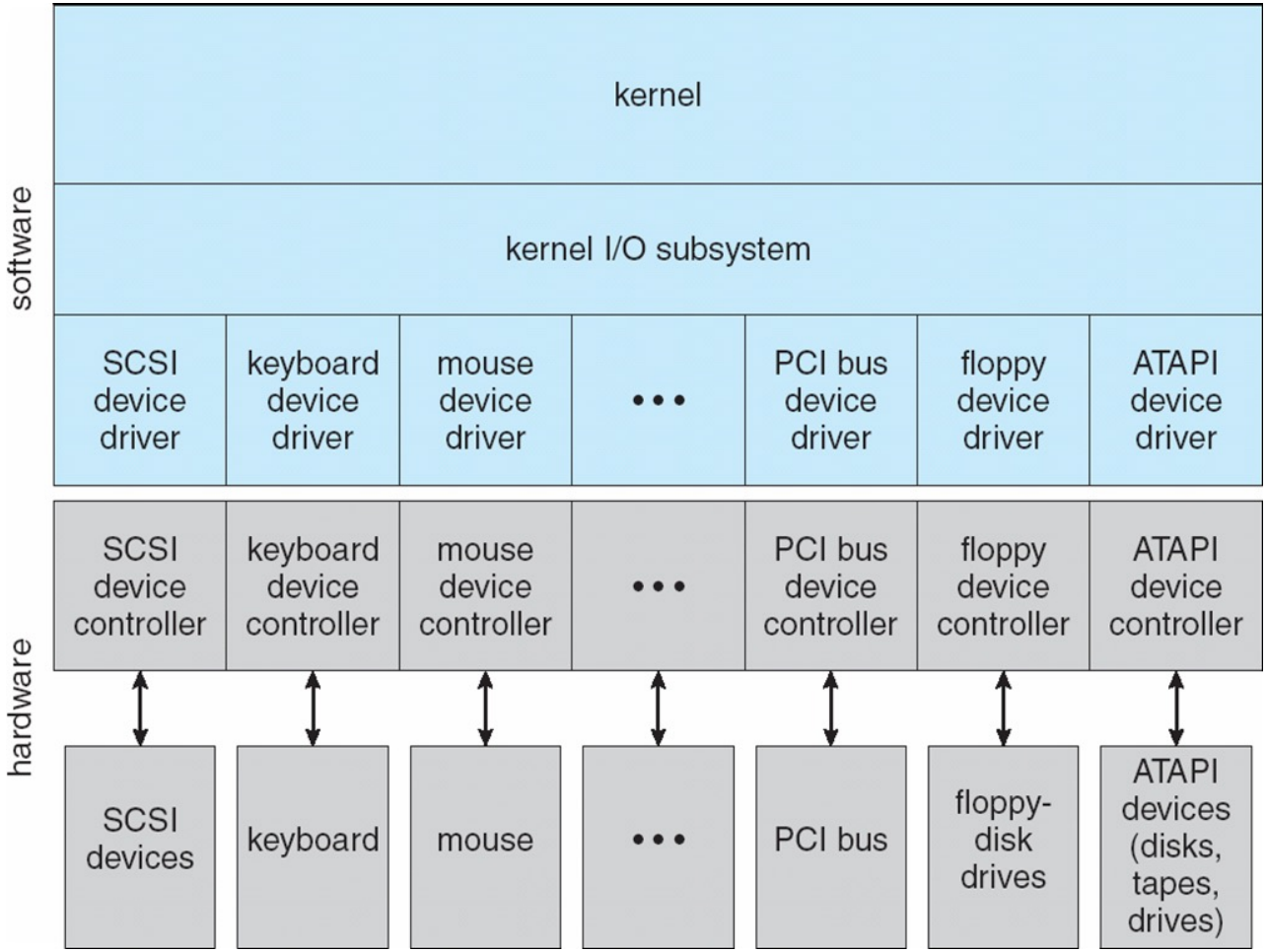
Six Step Process to Perform DMA Transfer



Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes
- Device-driver layer hides differences among I/O controllers from kernel
- New devices talking already-implemented protocols need no extra work
- Each OS has its own I/O subsystem structures and device driver frameworks
- Devices vary in many dimensions
  - **Character-stream** or **block**
  - **Sequential** or **random-access**
  - **Synchronous** or **asynchronous** (or both)
  - **Sharable** or **dedicated**
  - **Speed of operation**
  - **read-write**, **read only**, or **write only**

A Kernel I/O Structure



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 performance
- Translates 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

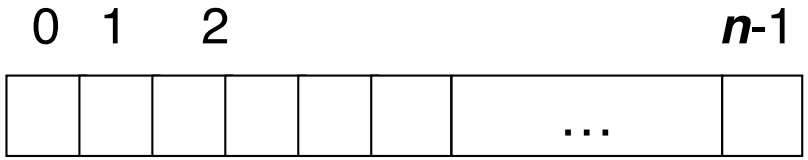
File System Layout Example

How do file systems use the disk to store files?

- File systems define a block size (e.g., 4KB)
  - Disk space is allocated in granularity of blocks
- A “Master Block” determines location of root directory
  - Always at a well-known disk location
  - Often replicated across disk for reliability
- A free map determines which blocks are free, allocated
  - Usually a bitmap, one bit per block on the disk
  - Also stored on disk, cached in memory for performance
- Remaining disk blocks used to store files (and directories)

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)



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

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

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
  - NTFS stores into in master file table using relational DB structures

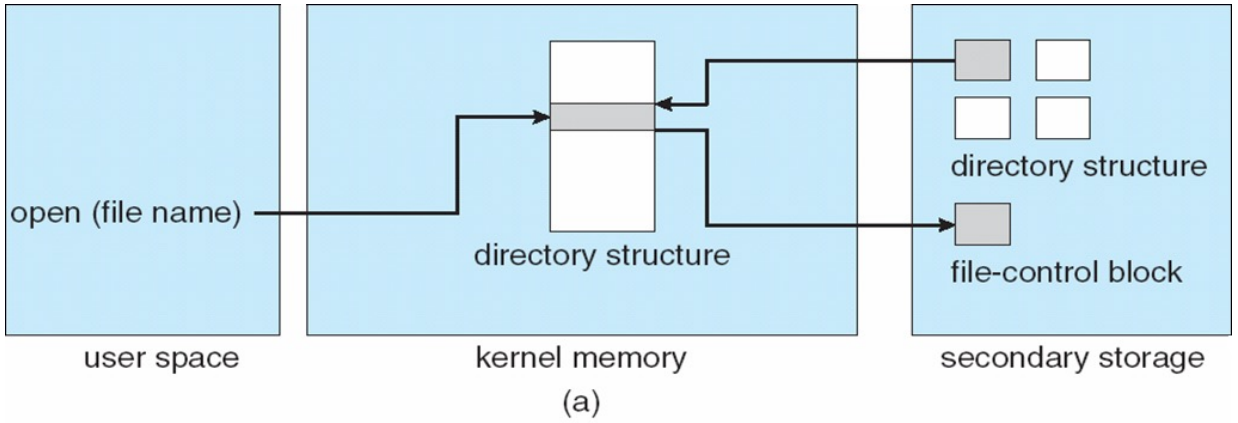
file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

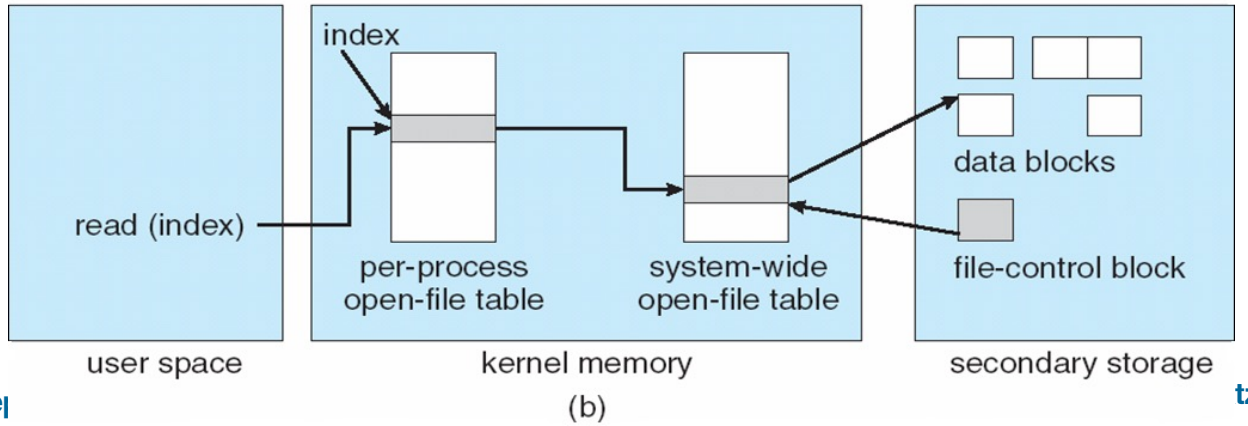
In-Memory File System Structures

- Mount table storing file system mounts, mount points, file system types
- 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

The operating system uses two levels of internal tables: **a per-process table and a system-wide table**. The per-process table tracks all files that a process has open. Stored in this table is information regarding the process's use of the file. Each entry in the per-process table in turn points to a system-wide open-file table. The system-wide table contains process-independent information, such as the location of the file on disk, access dates, and file size. Once a file has been opened by one process, the system-wide table includes an entry for the 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

Disk Layout Strategies

Files span multiple disk blocks. So How do you find all of the blocks for a file?

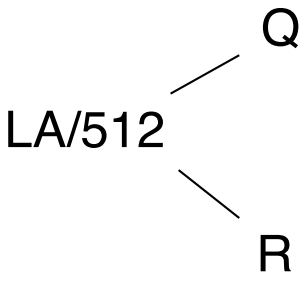
1. Contiguous allocation
   - Like memory
   - Fast, simplifies directory access
   - Inflexible, causes fragmentation, needs compaction
2. Linked structure
   - Each block points to the next, directory points to the first
   - Good for sequential access, bad for all others
3. Indexed structure (indirection, hierarchy)
   - An “index block” contains pointers to many other blocks
   - Handles random better, still good for sequential
   - May need multiple index blocks (linked together)

Allocation Methods - Contiguous

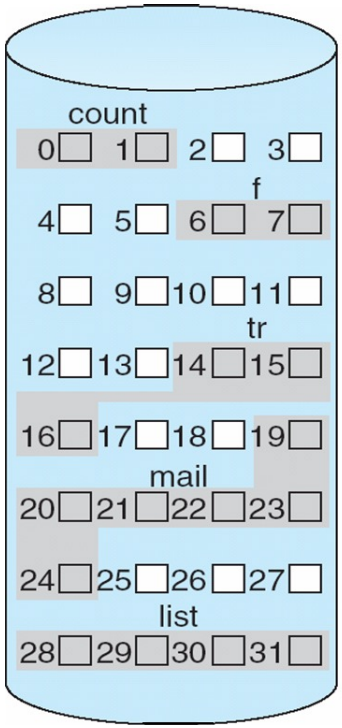
- An allocation method refers to how disk blocks are allocated for files:
- **Contiguous allocation** – each file occupies set of contiguous blocks
  - Best performance in most cases
  - Simple – only starting location (block #) and length (number of blocks) are required
  - Problems include finding 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

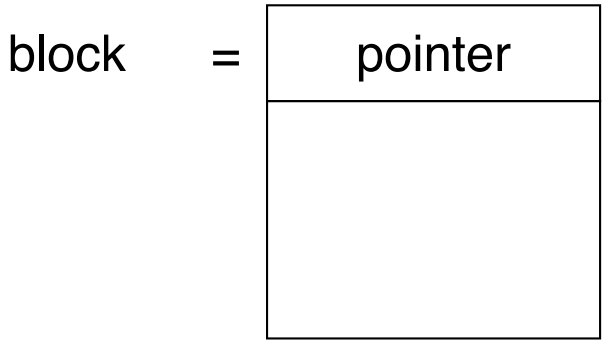
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Allocation Methods - Linked

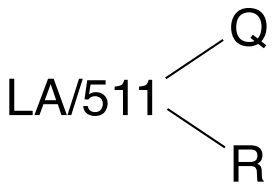
- **Linked allocation** – each file a linked list of blocks
  - File ends at null pointer
  - No external fragmentation
  - Each block contains pointer to next block
  - No compaction, external fragmentation
  - Free space management system called when new block needed
  - Improve efficiency by clustering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

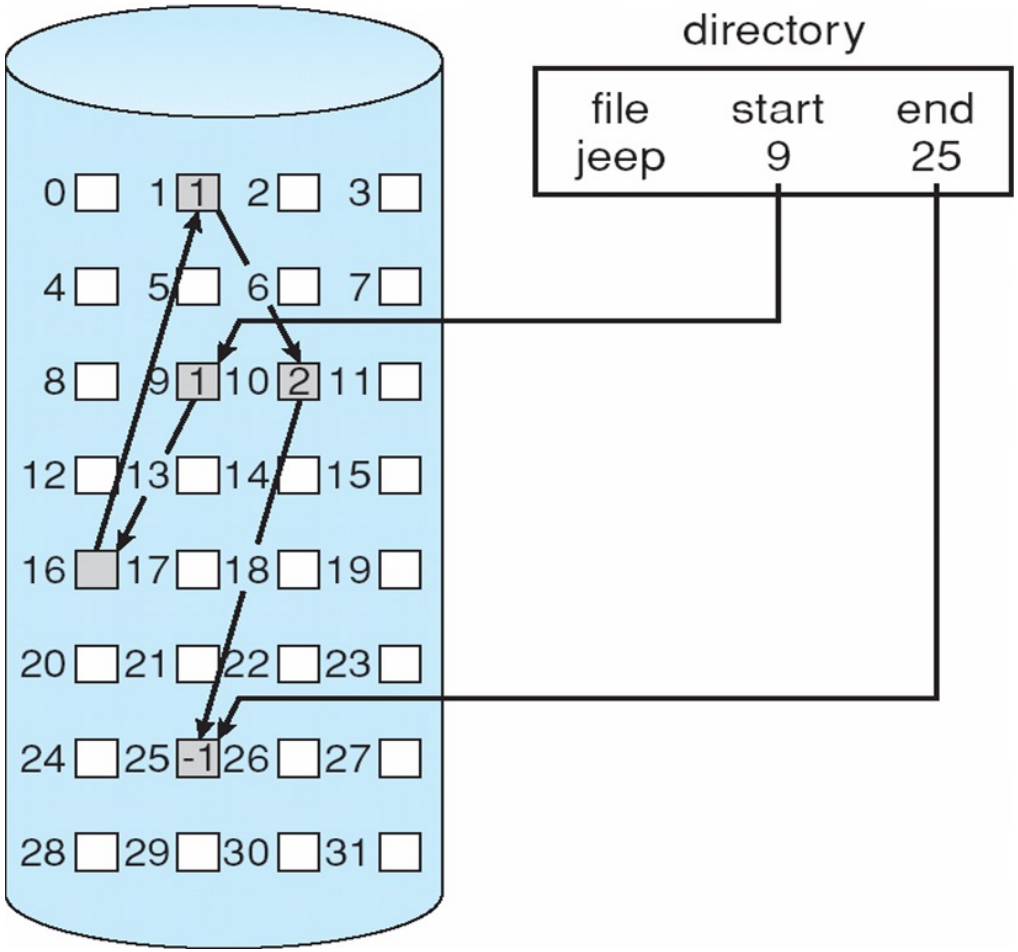


- Mapping

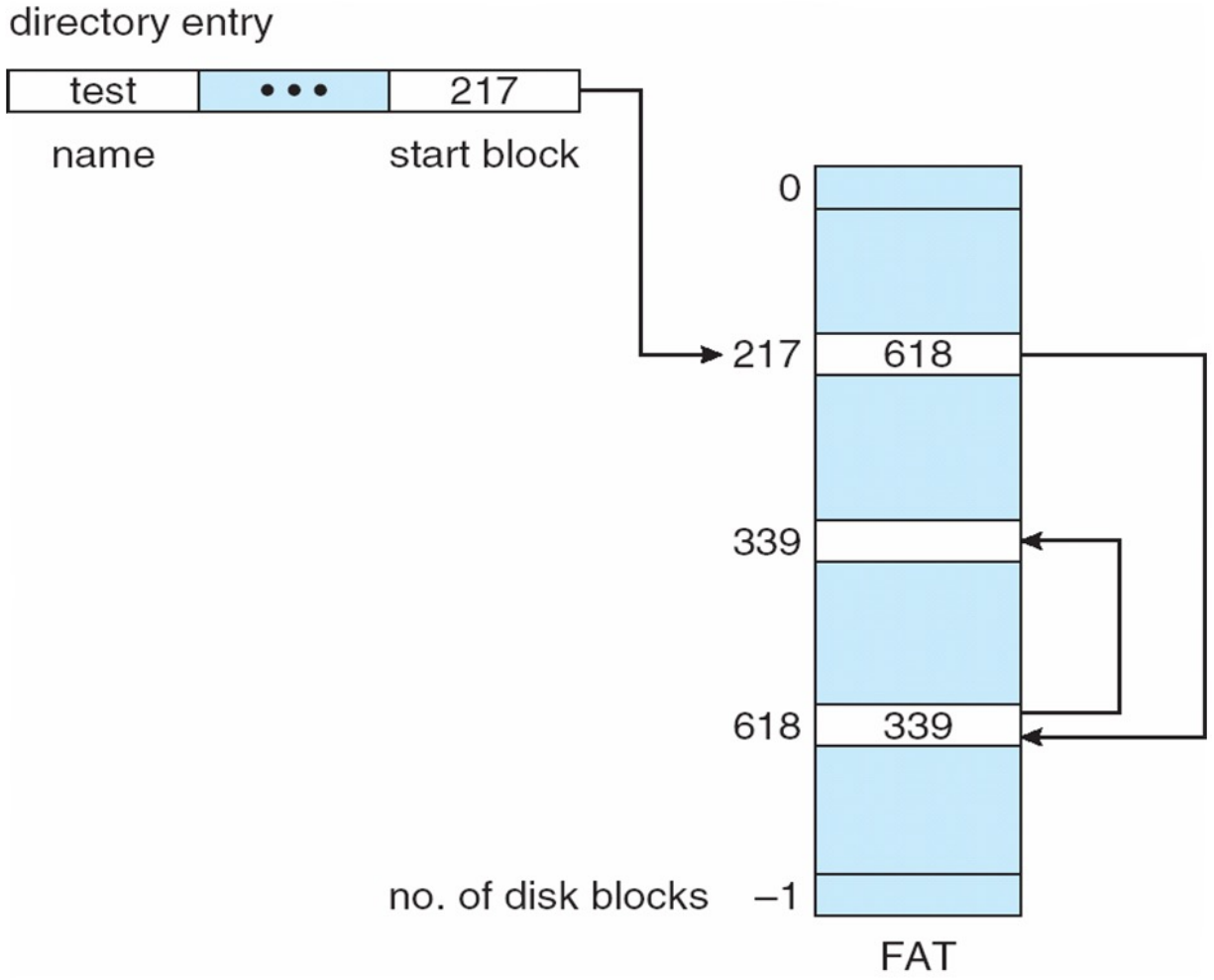


Block to be accessed is the Qth block in the linked chain of blocks representing the file.

Linked Allocation



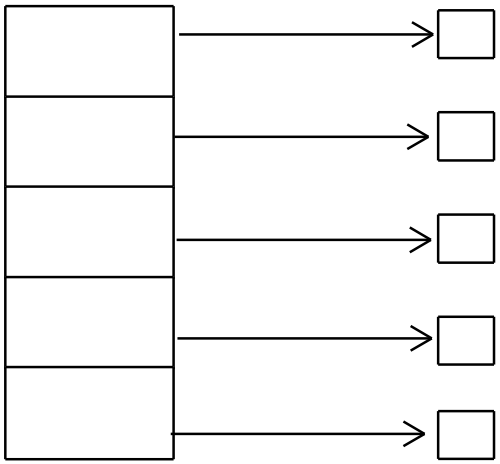
File-Allocation Table



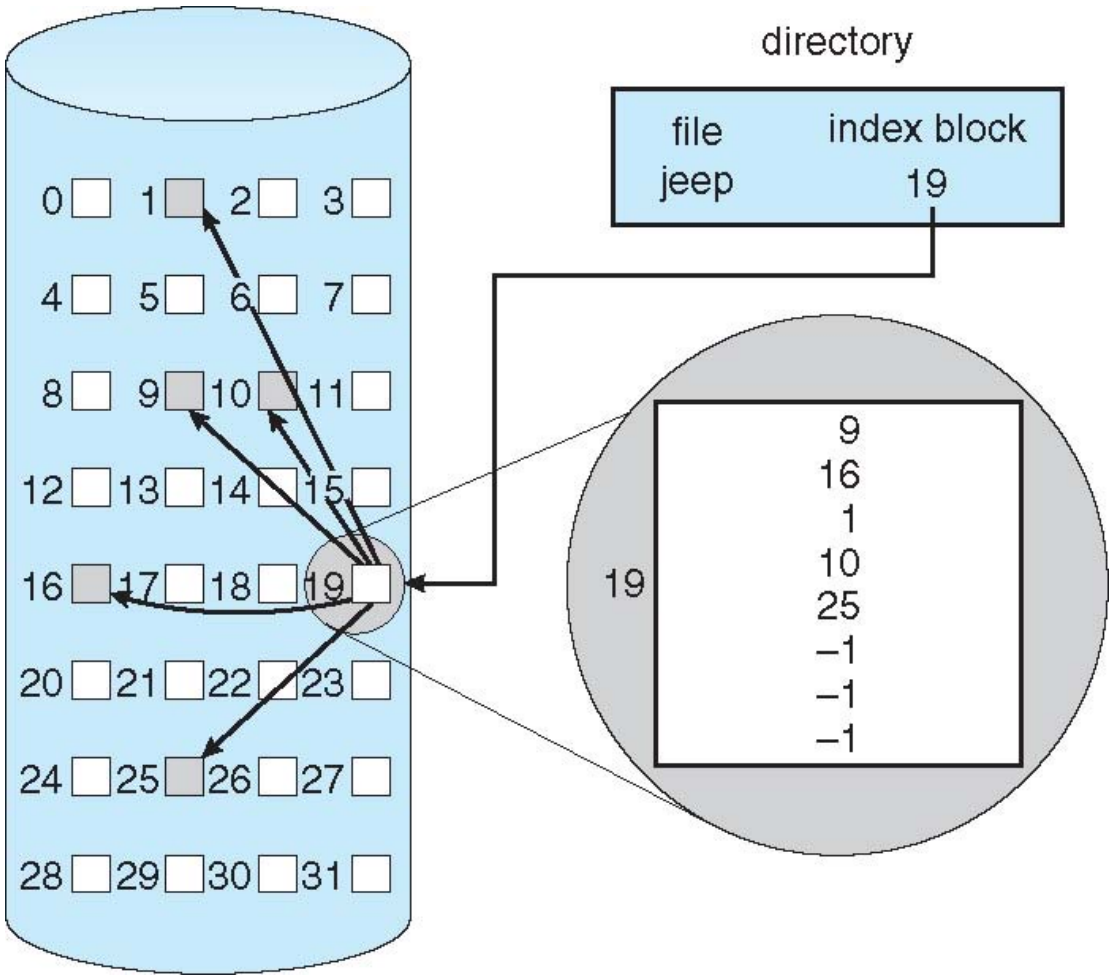
Allocation Methods - Indexed

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

- Logical view

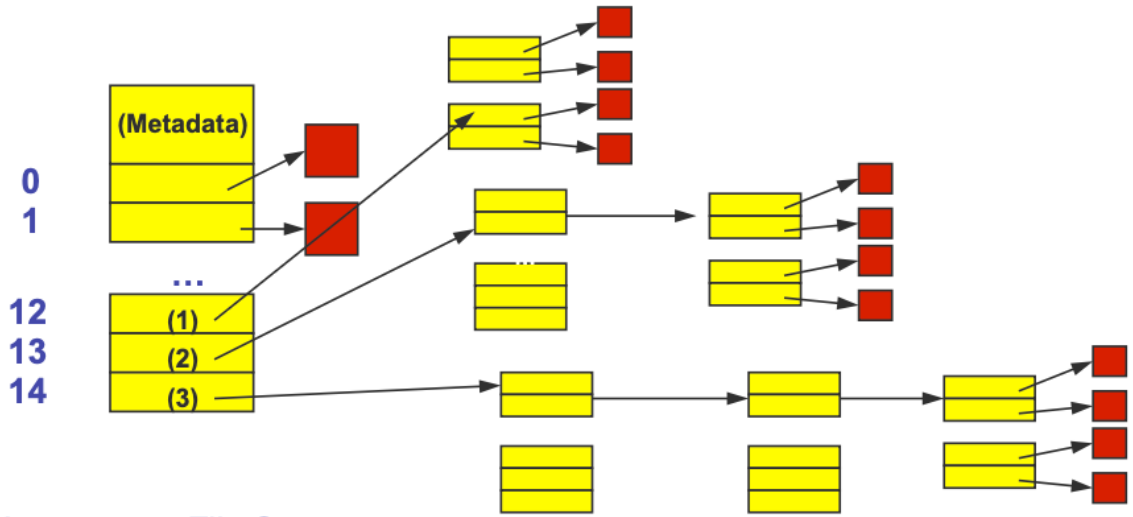


Example of Indexed Allocation



Unix Inodes

- Unix inodes implement an indexed structure for files
  - Also store metadata info (protection, timestamps, length, ref count...)
- Each inode contains 15 block pointers
  - First 12 are direct blocks (e.g., 4 KB blocks), then up to 48 KB of data can be accessed directly.
  - The next three pointers point to indirect blocks. The first points to a single indirect block, which is an index block containing not data but the addresses of blocks that do contain data. The second points to a double indirect block, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer contains the address of a triple indirect block.



Unix Inodes and Path Search

- Inodes describe where on the disk the blocks for a file are placed
  - Unix Inodes are *not* directories
  - Directories are files, so inodes also describe where the blocks for directories are placed on the disk
- Directory entries map file names to inodes
  - To open “/one”, use Master Block to find inode for “/” on disk
  - Open “/”, look for entry for “one”
  - This entry gives the disk block number for the inode for “one”
  - Read the inode for “one” into memory
  - The inode says where first data block is on disk
  - Read that block into memory to access the data in the file

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

Being Smarter

- So far, we've discussed how file systems work
  - Files, directories, Inodes, data blocks, etc.
  - Didn't focus much on where the data was coming from
- Now we'll focus on how to make them *perform*
  - By being smart about how we lay out the data on disk
- 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
- Three (somewhat dated, but illustrative) case studies:
  - BSD Unix Fast File System (FFS)
  - Log-structured File System (LFS)
  - Redundant Array of Inexpensive Disks (RAID, covered)

Data and Inode Placement

- The original Unix file system had a simple, straightforward implementation
  - Easy to implement and understand
  - But very poor utilization of disk bandwidth (lots of seeking)

Original Unix FS had two placement problems:

- Data blocks allocated randomly in aging file systems
  - Blocks for the same file allocated sequentially when FS is new
  - As FS “ages” and fills, need to allocate into blocks freed up when other files are deleted
  - Problem: Deleted files essentially randomly placed
  - So, blocks for new files become scattered across the disk
- Inodes allocated far from blocks
  - All Inodes at beginning of disk, far from data
  - Traversing file name paths, manipulating files, directories requires going back and forth from Inodes to data blocks

Both of these problems generate many long seeks

Cylinder Group

- BSD FFS addressed these problems using the notion of a cylinder group
  - Disk partitioned into groups of cylinders
  - Data blocks in same file allocated in same cylinder
  - Files in same directory allocated in same cylinder
  - Inodes for files allocated in same cylinder as file data blocks
- Free space requirement
  - To be able to allocate according to cylinder groups, the disk must have free space scattered across cylinders
  - 10% of the disk is reserved just for this purpose

Other Problems

- Small blocks (1K) caused two problems:
  - Low bandwidth utilization
  - Small max file size (function of block size)
- Fix using a larger block (4K)
  - Very large files, only need two levels of indirection for 2^32
  - Problem: internal fragmentation
  - Fix: Introduce “fragments” (1K pieces of a block)
- Problem: Media failures
  - Replicate master block (superblock)

File Buffer Cache

- Applications exhibit significant locality for reading and writing files
- Idea: Cache file blocks in memory to capture locality
  - This is called the file buffer cache
  - Cache is system wide, used and shared by all processes
  - Reading from the cache makes a disk perform like memory
  - Even a 4 MB cache can be very effective
- Issues
  - The file buffer cache competes with virtual memory (tradeoff here)
  - Like virtual memory, it has limited size
  - Need replacement algorithms again (LRU usually used)

Caching Write

- Applications assume writes make it to disk
  - As a result, writes are often slow even with caching
- Several ways to compensate for this
  - “write-behind”
    - Maintain a queue of uncommitted blocks
    - Periodically flush the queue to disk
    - Unreliable
- Battery backed-up RAM (NVRAM)
  - As with write-behind, but maintain queue in NVRAM
  - Expensive
- Log-structured file system
  - Always write next block after last block written
  - Complicated

Read Ahead

- Many file systems implement “read ahead”
  - FS predicts that the process will request next block
  - FS goes ahead and requests it from the disk. . . while the process is computing on previous block!
  - When the process requests block, it will be in cache
  - Compliments the disk cache, which also is doing read ahead
- For sequentially accessed files can be a big win
  - Unless blocks for the file are scattered across the disk
  - File systems try to prevent that, though (during allocation)
- Unfortunately, this doesn’t do anything for writes

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

LFS Approach

Treat the disk as a single log for appending

- Collect writes in disk cache, write out entire collection in one large disk request
  - Leverages disk bandwidth
  - No seeks (assuming head is at end of log)
- All info written to disk is appended to log
  - Data blocks, attributes, inodes, directories, etc.

Log-structured File Systems (LFS)

- The Log-structured File System (LFS) was designed in response to two trends in workload and technology:
  - Disk bandwidth scaling significantly (40% a year)
    - Latency is not
  - Large main memories in machines
    - Large buffer caches
    - Absorb large fraction of read requests
    - Can use for writes as well
    - Coalesce small writes into large writes
- LFS takes advantage of both of these to increase FS performance
  - Rosenblum and Ousterhout (Berkeley, ’91)

LFS: Locating Data

- LFS uses Inodes to locate data blocks
  - Inodes pre-allocated in each cylinder group
  - Directories contain locations of Inodes
- LFS appends Inodes to end of the log just like data
  - Makes them hard to find
- Approach
  - Use another level of indirection: Inode maps
  - Inode maps map file #s to Inode location
  - Location of Inode map blocks kept in checkpoint region
  - Checkpoint region has a fixed location
  - Cache Inode maps in memory for performance

LFS Challenges

LFS has two challenges it must address for it to be practical

- Locating data written to the log
  - FFS places files in a location, LFS writes data “at the end”
- LFS append-only quickly runs out of disk space
  - Need to recover deleted blocks
  - Disk is finite, so log is finite, cannot always append
  - Cleaning is a big problem (Costly overhead)

End of Chapter 13