

# CMSC 412

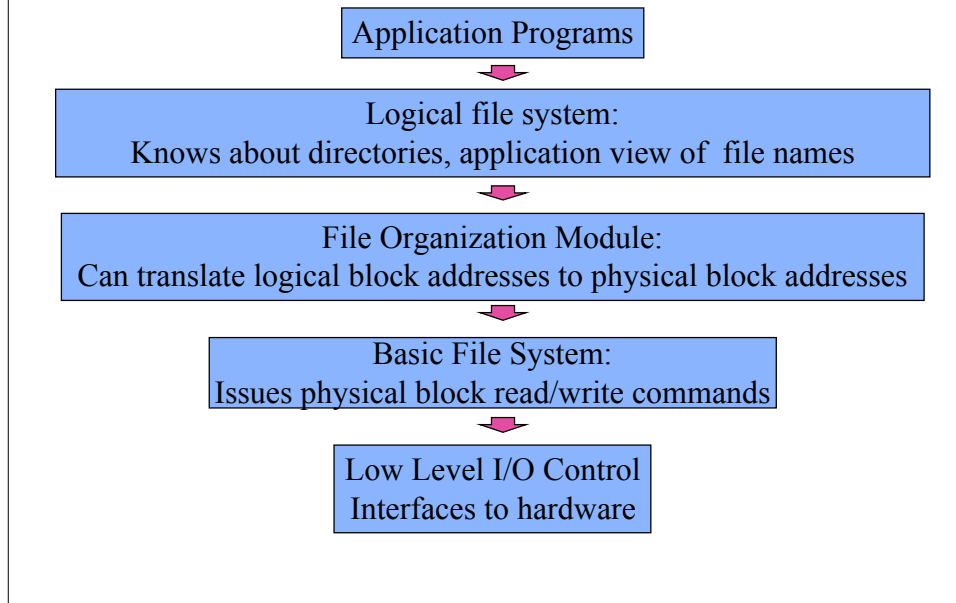
Spring 2007

## File System Implementation

## Announcements

- Reading
  - Chapter 11
  - Chapter 12 next time

## File System in Layers



## File System Structure

- **Self-contained:** all necessary meta-information is stored *on disk* along with the actual file system contents.
  - I.e. must be persistent in case the OS crashes.
- **Cached:** accessing the disk is expensive, so some information is *cached in memory*
  - *Meta-data cache* for file characteristics, and *buffer cache* for file contents
  - Must be kept in sync with on-disk copy

## On-Disk Elements

- Boot control block (per volume)
  - Needed to boot the OS
- Volume control block (per volume)
  - Tracks details like block size, free block counts and pointers, free files, etc.  
*A.k.a.* superblock (UFS), master file table (NTFS)
- Directory structure
- Per-file File Control Block (FCB)

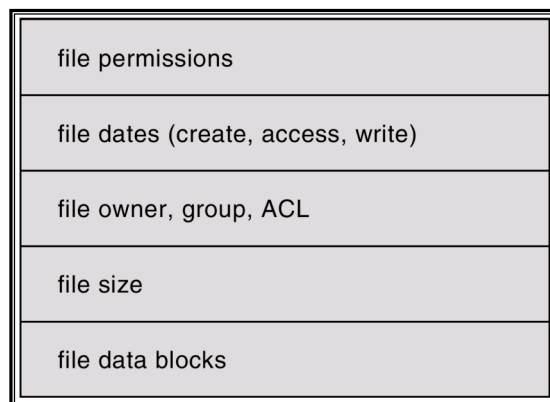
## In-Memory Elements

- Mount table
  - Information about each mounted volume
- Directory structure cache
  - For quicker filename translations
- System-wide open file table
  - Copy of FCB for each open file
- Per-process open file table
  - Points to System table, plus other info

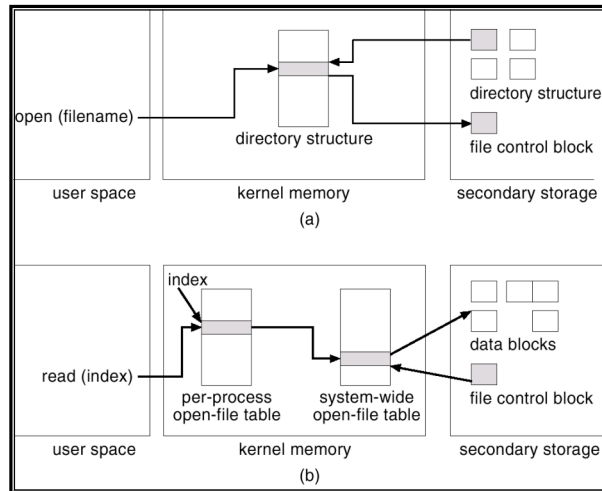
## File Structure

- File contents
  - Composed of one or more **blocks**, stored on disk. Block size usually relates to disk's *sector size*.
- File meta-data (**FCB**)
  - Describes implementation and other characteristics of file, like size, last-modified time, location of contents, etc.

## A Typical File Control Block



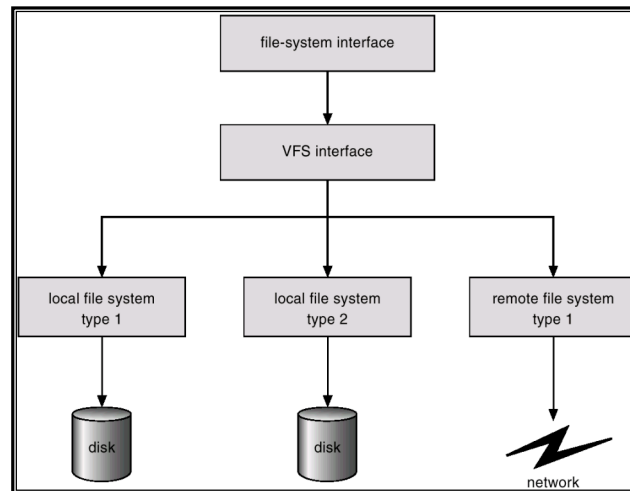
## Caching Metadata



## Virtual File Systems

- **Virtual File Systems (VFS)** 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.
- The API is to the VFS interface, rather than any specific type of file system.

# Virtual File System



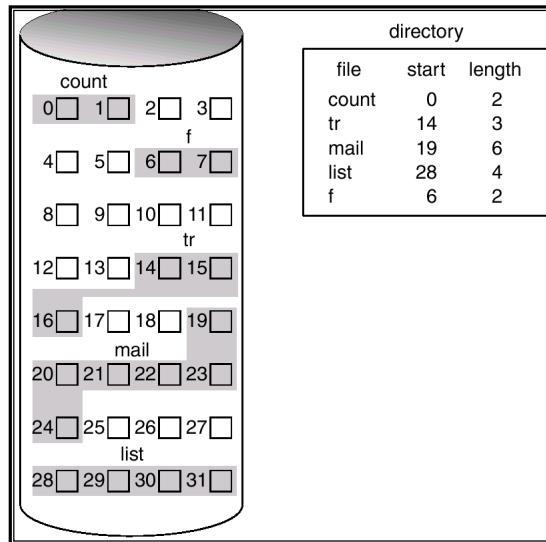
## Allocation Methods

- An allocation method refers to how disk blocks are allocated for files:
- **Contiguous allocation**
- **Linked allocation**
- **Indexed allocation**

# Contiguous Allocation

- Each file is a set of contiguous blocks on disk
- Pros
  - Simple - only starting location (block #) and length (number of blocks) are required.
  - Random access.
- Cons
  - Wasteful of space (external fragmentation).
  - Files cannot grow without some headaches

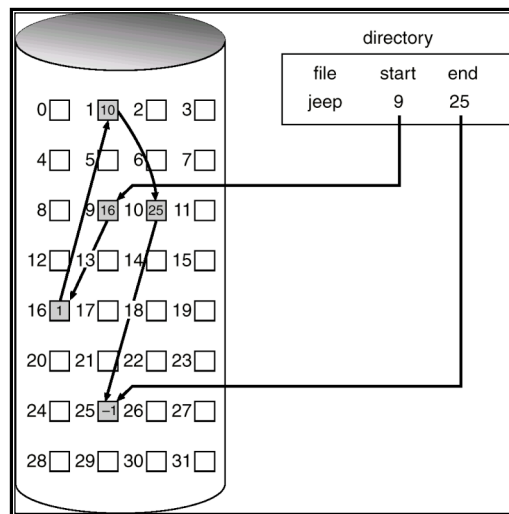
# Contiguous Allocation of Disk



# Linked Allocation

- Each file is a linked list of disk blocks, blocks can be located anywhere
  - FCB points to first (possibly last) block of file
  - Each block contains a pointer to the next block
- Pros:
  - Simple - need only starting address
  - No external fragmentation
- Cons:
  - Best for sequential access data structures
    - requires sequential access whether you want to or not!
  - Reliability - one bad sector and all portions of your file downstream are lost

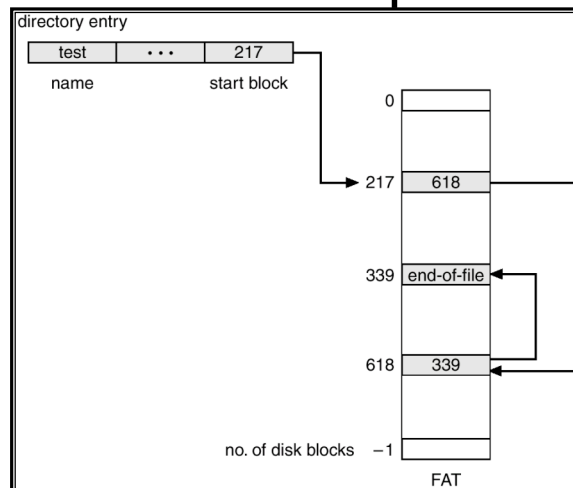
# Linked Allocation



## Modified Linked Allocation (FAT)

- Section of disk contains a table
  - called the file allocate table (FAT)
  - used in MS-DOS
- Directory entry contains the block number of the first block in the file
- Table entry contains the number of the next block in the file
- Last block has a end-of-file value as a table entry

### FAT Example



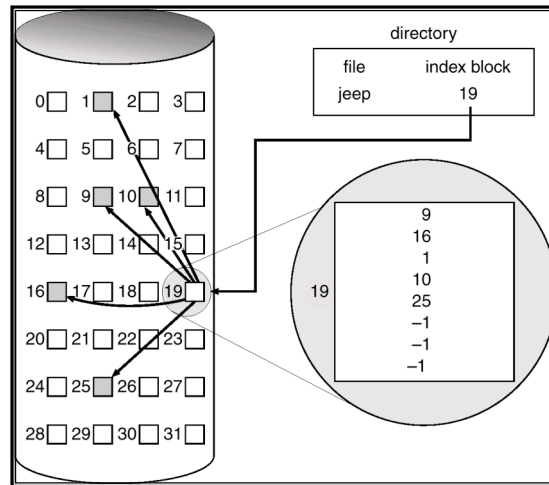
## Extents

- An **extent** combines linked and contiguous allocation:
  - Have a contiguous chunk of data, but then also have a pointer to the next chunk, if such a chunk is needed.
  - In other words, blocks can either be linked to, or adjacent to another block

## Indexed Allocation

- Bring all block pointers together into an **index block**
  - Stored on disk
  - FCB points to the index block, which points to the individual file blocks

## Example of Indexed Allocation



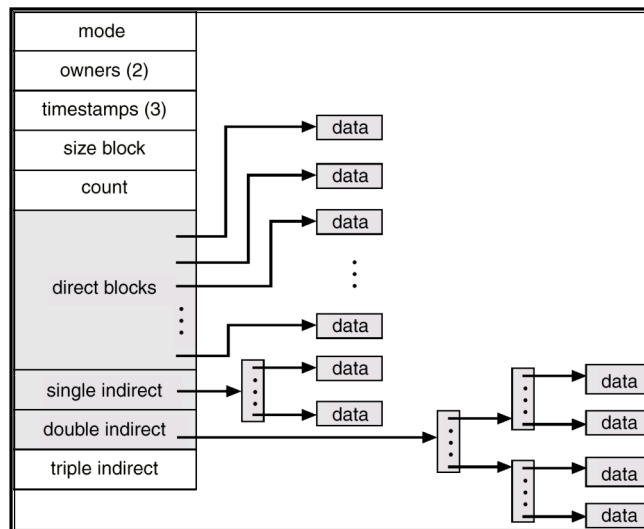
## Indexed Allocation

- Pros
  - Random access
  - No external fragmentation
- Cons
  - Must go through index block on dynamic accesses
  - May need very large index blocks to support large files; depends on the size of file blocks. Imposes per-file overhead.

# Indexed Allocation

- How to make index block extensible?
- Linked scheme:
  - maintain a linked list of indexed blocks
- Multilevel index:
  - Index block can point to other index blocks (which point to index blocks ....), which point to files
- Hybrid multi-level index (UNIX)
  - first n blocks are from a fixed index
  - next m blocks from an indirect index
  - next o blocks from a double indirect index

## UNIX (4K per block)



## UNIX File Sizes

- Assume 4096 byte block:
  - first 12 blocks (48 KB) from a fixed index
  - next 1024 blocks (4 MB) from an indirect index
  - next  $1024^2$  blocks (4 GB) from a double indirect index
  - final  $1024^3$  blocks (4 TB) from a triple indirect index

## Performance Comparison

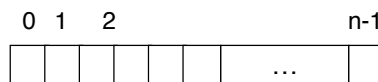
- FAT
  - ✓ simple, easy to implement
  - ✓ faster to traverse than linked allocation
    - random access requires following links
    - files can't have holes in them
- Hybrid indirect
  - ✓ fast access to any part of the file
  - ✓ files can have holes in them
    - more complex

# Free Space Management

- How do we find a disk block to allocate?
- Bit Vectors
  - array of bits (one per block) that indicates if a block is free
  - compact so can keep in memory
    - 100 GB disk, 4K blocks -> 6MB per disk (0.003%)
  - easy to find long runs of free blocks

## Bit Vectors

- Bit vector ( $n$  blocks)



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

Finding a free block offset:

$$\begin{aligned} & (\text{number of bits per word}) * \\ & (\text{number of 0-value words}) + \\ & \text{offset of first 1 bit} \end{aligned}$$

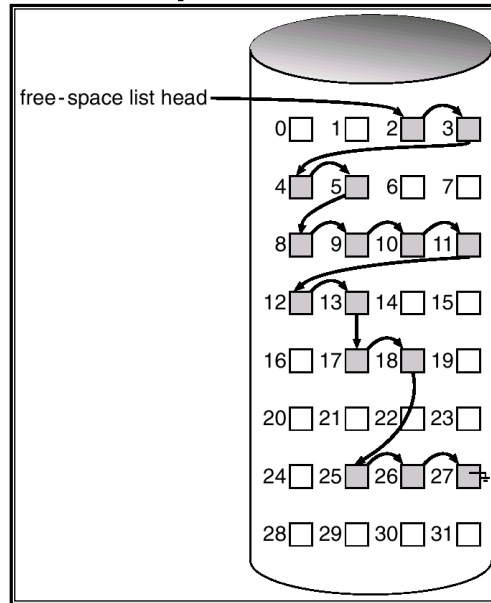
## Preserving Consistency

- Need to “protect”:
  - Pointer to free list
  - Bit map
    - Must be kept on disk
    - Copy in memory and disk may differ
    - Cannot allow for block[i] to have a situation where bit[i] = 0 in memory and bit[i] = 1 on disk
- Solution:
  - Set bit[i] = 0 in disk
  - Allocate block[i]
  - Set bit[i] = 0 in memory

## Free space as Linked List

- Each disk block contains the pointer to the next free block
- Pointer to first free block is kept in a special location on disk

## Linked Free Space List on Disk



## Run Length Encoding

- Pointer to first free block is kept in a special location on disk
- Each free block also includes a count of the number of consecutive blocks that are free
- Called **counting** in book
  - Similar to *extents* for allocated files
  - Could also store many pointers in a free block to the next N free blocks

## Implementing Directories

- Linear List
  - array of names for files
  - must search entire list to find or allocate a filename
  - sorting can improve search performance, but adds complexity
- Hash table
  - use hash function to find filenames in directory
  - needs a good hash function
  - need to resolve collisions
  - must keep table small and expand on demand since many directories are mostly empty

## DOS Directories

- Root directory
  - immediately follows the FAT
- Directory is a table of 32 byte entries
  - 8 byte file name, 3 byte filename extension
  - size of file, data and time stamp, starting cluster number of the file, file attribute codes
  - Fixed size and capacity
- Subdirectory
  - This is just a file
  - Record of where the subdirectory is located is stored in the FAT

## Unix Directories

- Space for directories are allocated in units called *chunks*
  - Size of a chunk is chosen so that each allocation can be transferred to disk in a single operation
  - Chunks are broken into variable-length directory entries to allow filenames of differing lengths
  - No directory entry can span more than one chunk
  - Directory entry contains
    - pointer to inode (file data-structure)
    - size of entry
    - length of filename contained in entry (up to 255)
    - remainder of entry is variable length - contains file name

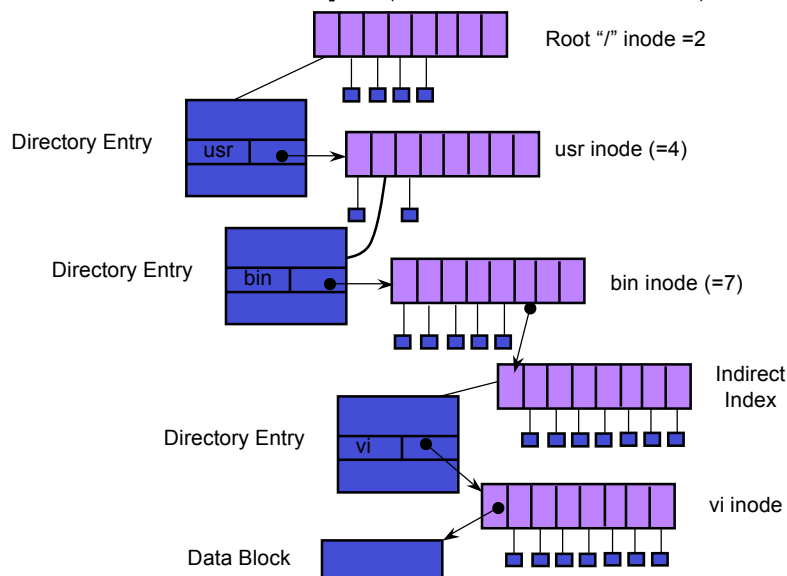
## Inodes

- File *index node*
- Contains:
  - Pointers to blocks in a file (direct, single indirect, double indirect, triple indirect)
  - Type and access mode
  - File's owner
  - Number of references to file
  - Size of file
  - Number of physical blocks

## Unix directories - links

- Each file has unique inode but it may have multiple directory entries in the same filesystem to reference the inode
- Each directory entry creates a hard link of a filename to the file's inode
  - Number of links to file are kept in a reference count variable in the inode
  - If links are removed, file is deleted when number of links becomes zero
- Symbolic or soft link
  - Implemented as a file that contains a pathname
  - Symbolic links do not have an effect on inode reference count

## File Lookup (/usr/bin/vi)



## UNIX data structures

- Example: find /usr/bin/vi
  - (from Leffler, McKusick, Karels and Quarterman)
  - Search root directory of filesystem to find /usr
    - root directory inode is, by convention, stored in inode #2
    - inode shows *where data blocks are* for root directory - *these blocks* (not the inode itself) *must* be retrieved and searched for directory entry **usr**
    - from its entry we discover that **usr's** inode is #4
  - Search **usr** for **bin**
    - access blocks pointed to by inode #4 and search contents of blocks for entry that gives us **bin's** inode
    - we discover that **bin's** inode is #7
  - Search **bin** for **vi**
    - access blocks pointed to by inode #7 and search contents of block for an entry that gives us **vi's** inode
    - we discover that **vi's** inode is inode #9
  - Access inode #9 - this is **vi's** inode

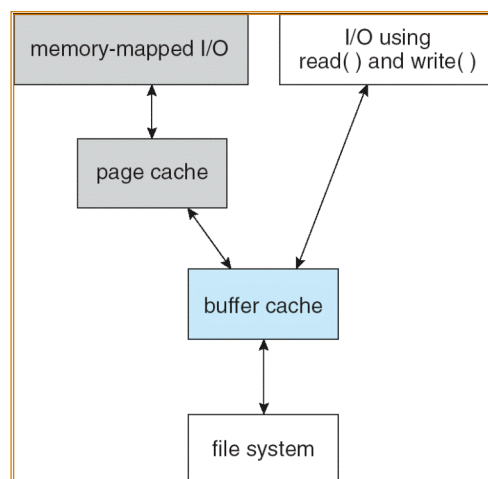
## Efficiency and Performance

- Efficiency dependent on:
  - disk allocation and directory algorithms
  - types of data kept in file's directory entry
- Performance
  - disk cache - separate section of main memory for frequently used blocks
  - free-behind and read-ahead - techniques to optimize sequential access
  - improve PC performance by dedicating section of memory as virtual disk, or RAM disk

## Page Cache

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

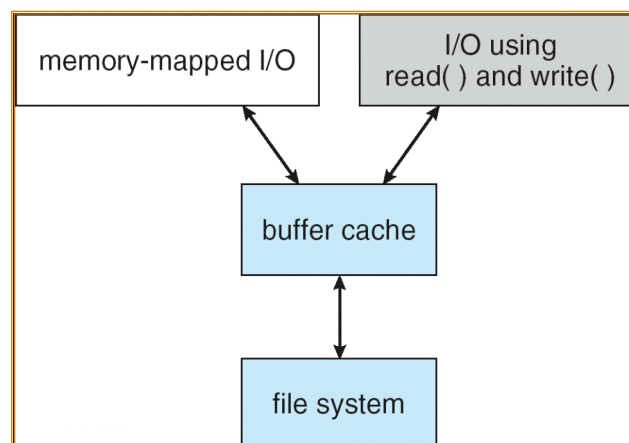
## Page and Buffer Cache



## Unified Buffer Cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O

## Unified Buffer Cache



## UNIX Shell and Current Directory

- Current Directory
  - Maintained on a per process basis by kernel
  - System Calls: get/set the current directory
  - Open system Call
    - File name checked and if it lacks a leading /, pre-pend cwd onto path
- Shell (file path)
  - Entirely implemented in user space
  - PATH Environment variable
    - Lists directories to search
  - Hash table of commands and their location (file, or internal)

## Log-Structured File Systems

- Key Idea
  - Use transactions-like model for filesystem updates
- Write data to a log (also called a journal)
  - Records meta data changes
  - Records data blocks written
  - File operation is committed when added to the log
  - Partial updates to log are lost on failure
- Next Step
  - Eliminate the filesystem and just keep the log
  - Requires a process called a cleaner
    - Copies old data from log to head of log to allow compaction

## NFS : Network File System

- Provides a set of remote procedure calls for remote file operations.
- NFS servers are *stateless*; each request has to provide a full set of arguments.
- Modified data must be committed to the server's disk before results are returned to the client (lose advantages of caching).

## 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.

# NFS Architecture

