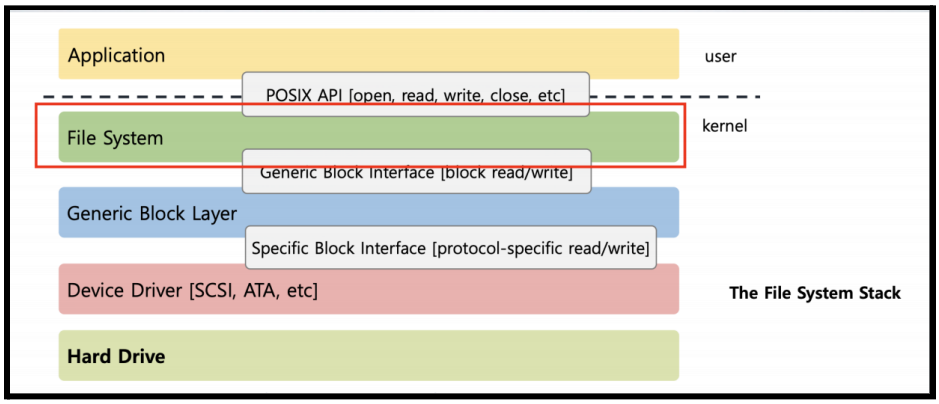
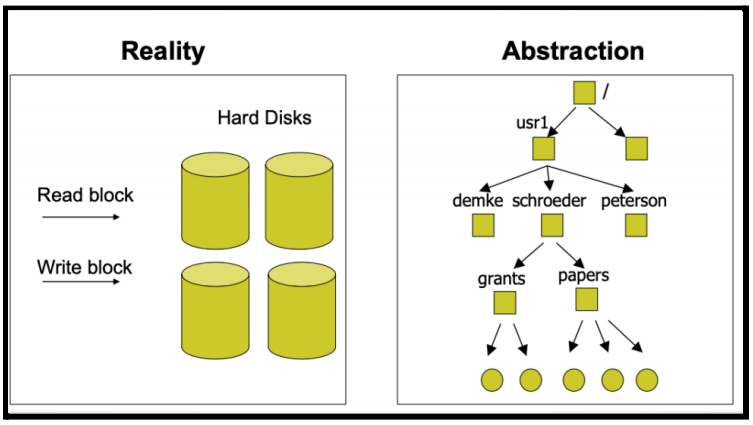
Lecture Notes:

- **File System:**
- The file system provides an abstraction.
- It specifies which disk class it is using.
- It issues block read/write requests to the generic block layer.
- I.e.





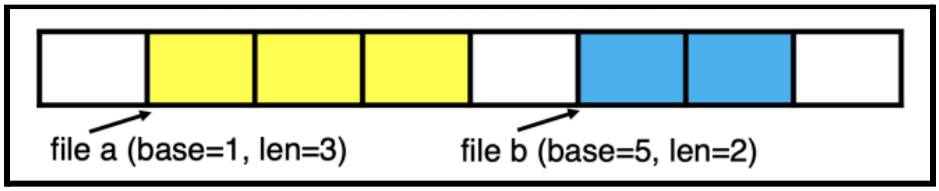
- The goals of a file system are:
  - Implement an abstraction (files) for secondary storage.
  - Organize files logically (directories).
  - Permit sharing of data between processes, people, and machines.
  - Protect data from unwanted access (security).
- **Files:**
- A **file** is named bytes on a disk that encapsulates data with some properties such as: contents, size, owner, last read/write time, protection, etc.
- A file can also have a type:
  - The type is understood by the file system: block device, character device, link, FIFO, socket, etc.
  - The type is also understood by other parts of the OS or runtime libraries: text, image, source, compiled libraries (Unix .so and Windows .dll), executable, etc.
- A file's type can be encoded in its name or contents:
  - Windows encodes type in name: .com, .exe, .bat, .dll, .jpg, etc.
  - Unix encodes type in contents: magic numbers, initial characters (E.g. #! for shell scripts).
- **Note:** In Unix, everything is a file.

- There are several methods for accessing files:
    1. **Sequential access:**
    - Used by file systems and is the most common method.
    - Read bytes one at a time, in order.
    2. **Random access:**
    - Used by file systems.
    - Random access is given to a block/byte number (read/write bytes at offset n).
    3. **Indexed access:**
    - Used by databases.
    - The file system contains an index to a particular field of each record in a file.
    - Reads specify a value for that field and the system finds the record via the index.
    4. **Record access:**
    - Used by databases.
    - The file is an array of fixed-or-variable-length records.
    - Read/write sequentially or randomly by record number.
- Some basic file operations for Unix & Windows are:

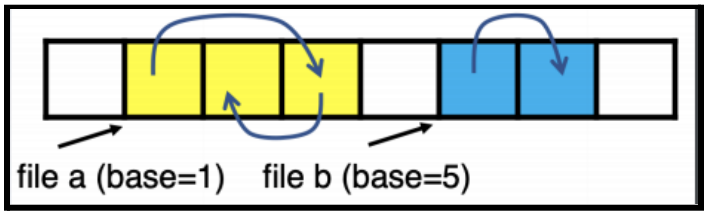
Unix	**Windows**
create(name)	CreateFile(name, CREATE)
open(name, how)	CreateFile(name, OPEN)
read(fd, buf, len)	ReadFile(handle, …)
write(fd, buf, len)	WriteFile(handle, …)
sync(fd)	FlushFileBuffers(handle, …)
seek(fd, pos)	SetFilePointer(handle, …)
close(fd)	CloseHandle(handle, …)
unlink(name)	DeleteFile(name)
	CopyFile(name)
	MoveFile(name)

- How to Track File's Data:
    - Disk management:
        - Need to keep track of where file contents are on disk.
        - Must be able to use this to map byte offset to disk block.
        - Structure tracking a file's blocks is called an index node or inode.
        - Inodes must be stored on disk, too.
    - Things to keep in mind while designing file structure:
        - Most files are small.
        - Much of the disk is allocated to large files.
        - Many of the I/O operations are made to large files.
        - Want good sequential and good random access.

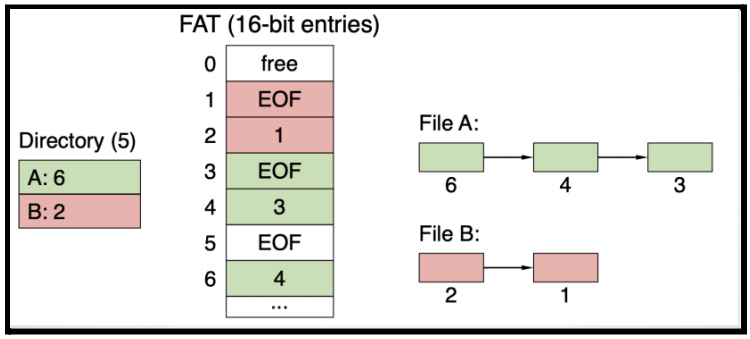
- **Straw Man #1 - Contiguous Allocation:**
- Is **extent-based**, meaning that it allocates files like segmented memory.
- When creating a file, make the user pre-specify its length and allocate all space at once.
- Inode contents: location and size.
- E.g.



- Advantage: Simple, fast access, both sequential and random.
- Disadvantage: External fragmentation (similar to VM).
- **Straw Man #2 - Linked Files:**
- Basically a linked list on disk:
  - Keep a linked list of all free blocks.
  - Inode contents: A pointer to file's first block.
  - In each block, keep a pointer to the next one.
- E.g.

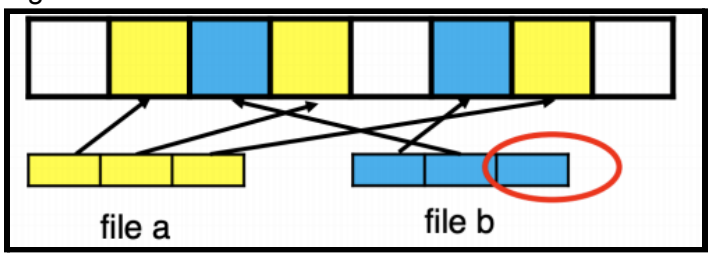


- Advantage: Easy dynamic growth & sequential access, no fragmentation.
- Disadvantage: Linked lists on disk are a bad idea because of access times. Random access is very slow (E.g. You have to traverse the whole file to find the last block). Pointers take up room in blocks, skewing alignment.
- **DOS FAT:**
- Linked files with key optimization: Puts links in fixed-size "file allocation table" (FAT) rather than in the block.
- However, this still does pointer chasing.
- E.g.



- Given entry size = 16 bits (initial FAT16 in MS-DOS 3.0), what's the maximum size of the FAT → 65,536
- Given a 512 byte block, what's the maximum size of FS → 32MB
- What is the space overhead → 2 bytes / 512 byte block = ~0.4%
- How to protect against errors → Create duplicate copies of FAT on disk (State duplication is a very common theme in reliability).
- Where is the root directory → Fixed location on disk.

- **Indexed Files:**
- Each file has a table holding all of its block pointers:
    - Max file size fixed by table's size.
    - Allocate table to hold file's block pointers on file creation.
    - Allocate actual blocks on demand using the free list.
- E.g.

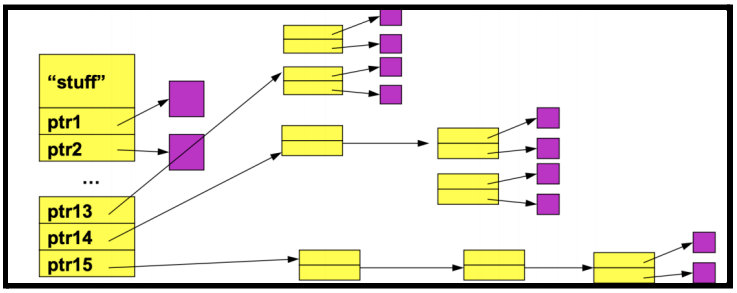


- Adv: Both sequential and random access are easy.
- Disadv: Mapping table requires a large chunk of contiguous space.
- **Unix File System:**
- The disk is (physically) divided into sectors (usually 512 bytes per sector).
- The file system is (logically) divided into blocks (E.g. 4 KB per block).
- Disk space is allocated in granularity of blocks:
    1. The data blocks "D" stored files (and directories) content.
    2. The inodes block "I" stores the inode table.
    3. The data bitmap "d" block d tracks which data block is free or allocated (one bit per block on the disk).
    4. The inode bitmap "i" block i tracks which inode is free or allocated (one bit per inode).
    5. The Superblock "S" (a.k.a Master Block or partition control block) contains:
        a. A magic number to identify the file system type.
        b. The number of blocks dedicated to the two bitmaps and inodes.
- **The Inode Table:**
- Physical Disk capacity in our example (64 blocks of 4KB each): 4 x 64 = 256 KB
- Logical capacity (8 blocks are reserved): 4 x 56 = 224 KB (the actual data storage space)
- Maximum number of inodes (each inode is 256 bytes): (5 * 4 * 1024) / 256 = 80 inodes (i.e max number of files)
- Size of the inode bitmap (1 bit per inode): 1 x 80 inodes = 80 bits (out of 32K bits)
- Size of the data bitmap (1 bit per storage block): 1 bit x 56 blocks = 56 bits (out of 32K bits, max data storage 128 MB)
- Decoding inodes: E.g. What disk sector to read to retrieve inode 32?
    1. Calculate the offset (each inode is 256 bytes) 32 x 256 = 8,192
    2. Add the start of the address of the inode table (12K) 8,192 + 12 x 1,024 = 20,480 (20 KB)
    3. Find the corresponding disk sector (each sector is 512 bytes) (20 x 1,024) / 512 = 40

- Simplified Unix Inode Table:

Size	Name	Description
2	mode	can the file be read/written/executed
2	uid	file owner id
4	size	the file size in bytes
4	time	time the file was last accessed
4	ctime	time when the file created
4	mtime	time when the file was last modified
4	dtime	time when the inode was deleted
2	gid	file group owner id
2	links_count	number of hard links pointing to this file
4	blocks	the number of blocks allocated to this file
60	block	disk pointers (15 in total)
4	file_acl	ACL permissions
4	dir_acl	ACL permissions

- So far, each inode has 15 block pointers. This means that the maximum file size can be 60KB (15 * 4 KB = 60 KB). Remember, 1 block is 4KB. Should we increase the number of block pointers to increase the file size?
    - Large file size with lots of unused entries means the mapping table requires a large chunk of contiguous space. A solution is to use a mapping table structured as a multi-level index array. This is called **multi-level indexed files**.
    - Unix uses 12 pointers that are direct blocks, which solves the problem of the first blocks' access being slow. Then it uses single, double, and triple indirect block pointers.



- File size with multi-level indexed files:
- File size using 12 direct blocks: 12 x 4 KB = 48 KB
    - Adding single indirect block: (12 + 1024) x 4 KB ~ 4 MB
    - Adding a double indirect block: (12 + 1024 + 1024^2) × 4 KB) ~ 4 GB
    - Adding a triple indirect block: (12 + 1024 + 1024^2 + 1024^3) × 4 KB) ~ 4 TB
- Rationale behind multi-level index files:
    - Most files are small: ~2K is the most common size
    - Average file size is growing: Almost 200K is the average
    - Most bytes are stored in large files: A few big files use most of space
    - File systems contains lots of files: Almost 100K on average
    - File systems are roughly half full: Even as disks grow, file systems remain ~50% full
    - Directories are typically small: Many have few entries; most have 20 or fewer

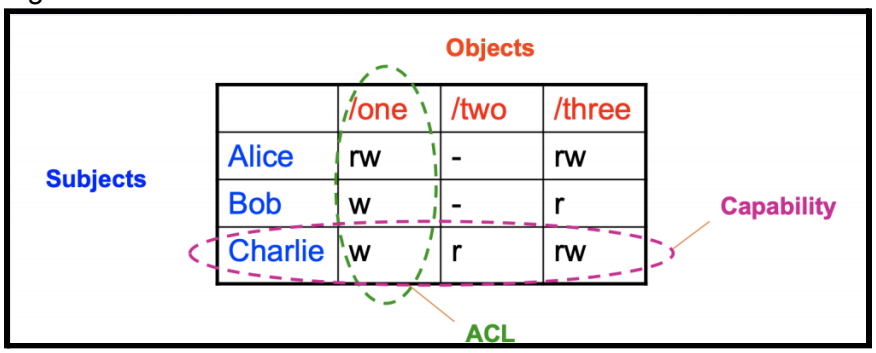
- **Directories:**
- Directories serve two purposes:
    1. For users, they provide a structured way to organize files by using digestible names rather than inode numbers directly.
    2. For the File System, they provide a convenient naming interface that allows the separation of logical file organization from physical file placement on the disk.
- Basic Directory Operations:

Unix	**Windows**
Directories implemented in file and a C runtime library provides a higher-level abstraction for reading directories.	Explicit directory operations.
opendir(name)	CreateDirectory(name)
readdir(DIR)	RemoveDirectory(name)
seekdir(DIR)	FindFirstFile(pattern)
closedir(DIR)	FindNextFile()

- A Short History of Directories:
    1. Approach 1: Single directory for the entire system.
        - Puts a directory at a known location on the disk.
        - Directories contain hname, inumberi pairs.
        - If one user uses a name, no one else can.
        - Many ancient personal computers work this way.
    2. Approach 2: Single directory for each user.
        - Still clumsy, and ls on 10,000 files is a real pain.
    3. Approach 3: Hierarchical name spaces.
        - Allow the directory to map names to files or other directories.
        - File system forms a tree (or graph, if links are allowed).
        - Large name spaces tend to be hierarchical (ip addresses, domain names, scoping in programming languages, etc.)
- Hierarchical Directory:
    - Used since CTSS (1960s) Unix picked up and used really nicely.
    - Directories stored on disk just like regular files:
        - A special inode type byte set to directory.
        - Users can read just like any other file.
        - Only special syscalls can write.
        - Inodes are at a fixed disk location.
        - A file pointed to by the index may be another directory.
        - Makes the file system into a hierarchical tree.
    - Simple, plus speeding up file ops speeds up dir ops.
- **Note:** Unix inodes are not directories. Inodes describe where on the disk the blocks for a file are placed whereas directories are files. So, inodes also describe where the blocks for directories are placed on the disk.

- Directory entries map file names to inodes:
    1. To open "/one", use Master Block to find the inode for "/" on the disk.
    2. Open "/", look for the entry for "one".
    3. This entry gives the disk block number for the inode for "one".
    4. Read the inode for "one" into memory.
    5. The inode says where the first data block is on disk.
    6. Read that block into memory to access the data in the file.
- In Unix, each process has a "current working directory" (cwd). File names not beginning with "/" are assumed to be relative to cwd. Otherwise the translation happens as before.
- Shells track a default list of active contexts.
    - Given a search path A:B:C, the shell will check in A, then B, then C.
    - We can escape using explicit paths. For example: "./foo".
- Hard and Soft Links (synonyms):
    - More than one directory entry can refer to a given file.
    - A **hard link** creates a synonym for file.
    - Unix stores count of pointers ("hard links") to inode.
    - If one of the links is removed, the data is still accessible through any other link that remains.
    - If all links are removed, the space occupied by the data is freed.
    - **Soft symbolic links** are synonyms for names.
    - Soft links point to a file/dir name, but objects can be deleted from underneath it (or never exist).
    - Unix implements soft links like directories. The inode has a special "symlink" bit set and contains the name of the link target.
    - When the file system encounters a soft link it automatically translates it if possible.
- **File Buffer Cache:**
- Applications exhibit significant locality for reading and writing files.
- Idea: Cache file blocks in memory to capture locality 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 small cache can be very effective.
- Issues:
    - The file buffer cache competes with VM (tradeoff here).
    - Like VM, it has limited size.
    - Need replacement algorithms again (LRU is usually used).
- Caching Writes:
    - On a write, some applications assume that data makes it through the buffer cache and onto the disk. As a result, writes are often slow even with caching.
    - OSes typically do write back caching:
        - Maintain a queue of uncommitted blocks.
        - Periodically flush the queue to disk (30 second threshold).
        - If blocks have changed many times in 30 secs, we only need one I/O.
        - If blocks are deleted before 30 secs (e.g., /tmp), no I/Os needed.
    - This is unreliable, but practical:
        - On a crash, all writes within the last 30 secs are lost.
        - Modern OSes do this by default; too slow otherwise.
        - System calls (Unix: fsync) enable apps to force data to disk.

- **Read Ahead:**
- Many file systems implement "read ahead".
    - The FS predicts that the process will request the next block.
    - The FS goes ahead and requests it from the disk.
    - This can happen while the process is computing on the previous block.
    - Overlap I/O with execution.
    - When the process requests a block, it will be in cache.
    - Compliments the disk cache, which also is doing read ahead.
- For sequentially accessed files read ahead can be a big win. Read ahead won't work if the blocks for the file are scattered across the disk, but file systems try to prevent that, though during allocation.
- **File Sharing:**
- File sharing is important for getting work done. It is the basis for communication and synchronization.
- There are two key issues when sharing files:
    1. Semantics of concurrent access:
        - What happens when one process reads while another writes?
        - What happens when two processes open a file for writing?
        - What are we going to use to coordinate?
    2. Protection
- **Protection:**
- File systems need to implement a protection system.
    - Who can access a file?
    - How can they access it?
- A protection system dictates whether a given action performed by a given subject on a given object should be allowed.
    - I.e. You can read and/or write your files, but others cannot.
    - I.e. You can read "/etc/motd", but you cannot write it.
- **Access Control Lists (ACL):** For each object, maintain a list of subjects and their permitted actions.
- **Capabilities:** For each subject, maintain a list of objects and their permitted actions.
- E.g.



- An ACL is a list for each object consisting of the domains with a nonempty set of access rights for that object. A capability list is a list of objects and the operations allowed on those objects for each domain.
- E.g. Consider this scenario: There is a list of buildings and a list of people and we want to have a list of which people can enter which buildings. There are 2 ways to do it:
    1. ACL: Each building has a list of people that can enter it.
    2. Capability: Each person has a list of buildings that they can enter.

- Approaches between ACL and capability differ only in how the table is represented.
- Capabilities are easier to transfer. They are like keys, can handoff, and do not depend on the subject.
- ACLs are easier to manage in practice. They are object-centric, easy to grant, and revoke. To revoke capabilities, we have to keep track of all subjects that have the capability, which is a challenging problem.
- However, ACLs have a problem when objects are heavily shared, they become very large.