Operating Systems (A) (Honor Track)

Lecture 18: File Systems

Yao Guo (郭耀)

Peking University

Fall 2021

Acknowledgements: Prof. Xiangqun Chen & Tao Wang at PKU and Prof. Yuanyuan Zhou at UCSD

Buzz Words

- Files
- Directories
- Hard links
- Soft links
- inode
- Fragmentation
- Partition
- MBR

This Lecture

File Systems

Physical Disks

Files & Directories

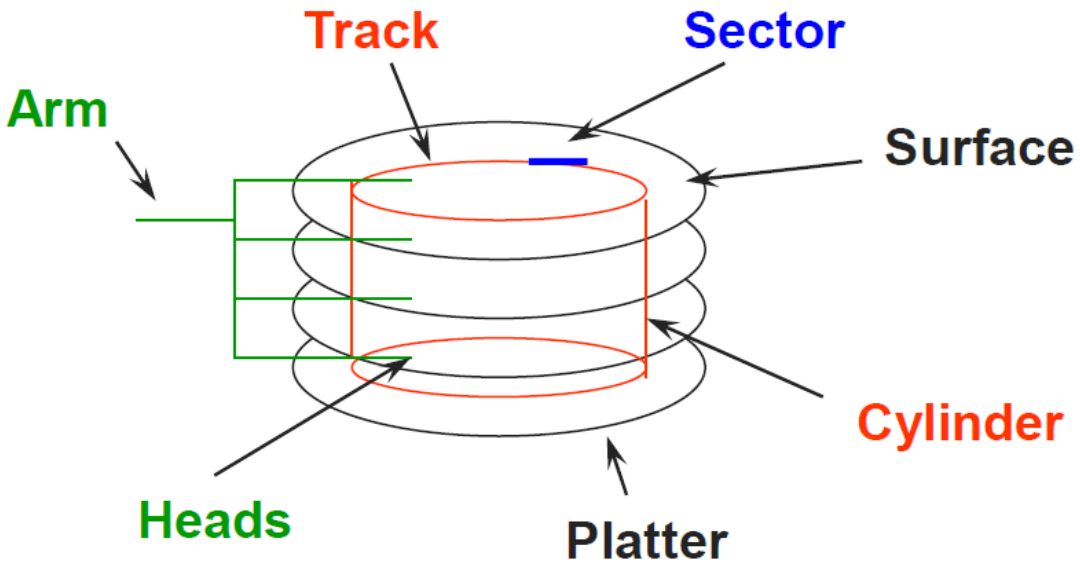
File System Implementation

Disks and the OS

- Disks are messy physical devices:
  - Errors, bad blocks, missed seeks, etc.
- The job of the OS is to hide this mess from higher level software
  - Low-level device control (initiate a disk read, etc.)
  - Higher-level abstractions (files, databases, etc.)
- The OS may provide different levels of disk access to different clients
  - Physical disk (surface, cylinder, sector)
  - Logical disk (block)
  - Logical file (block, record, or byte #)

Physical Disk Structure

- Disk components
  - Platters
  - Surfaces
  - Tracks
  - Sectors
  - Cylinders
  - Arm
  - Heads



Disk Interactions

- Specifying disk requests requires a lot of info:
  - Cylinder #, surface #, track #, sector #, transfer size…
- Older disks required the OS to specify all of these
  - The OS needed to know all disk parameters
- Modern disks are more complicated
  - Not all sectors are the same size, sectors are remapped, etc.
- Current disks provide a higher-level interface
  - The disk exports its data as a logical array of blocks [0…N]
    - Disk maps logical blocks to cylinder/surface/track/sector
    - Block size can be configured via low-level formatting
  - Only need to specify the logical block # to read/write
  - But now the disk parameters are hidden from the OS

Disk Specifications

- Seagate Enterprise Performance 3.5" (server)
  - capacity: 600 GB
  - rotational speed: 15,000 RPM
  - sequential read performance: 233 MB/s (outer) –160 MB/s (inner)
  - seek time (average): 2.0 ms
- Seagate Barracuda 3.5" (workstation)
  - capacity: 3000 GB
  - rotational speed: 7,200 RPM
  - sequential read performance: 210 MB/s -156 MB/s (inner)
  - seek time (average): 8.5 ms
- Seagate Savvio 2.5" (smaller form factor)
  - capacity: 2000 GB
  - rotational speed: 7,200 RPM
  - sequential read performance: 135 MB/s (outer) -? MB/s (inner)
  - seek time (average): 11

Disk Performance

- Disk request performance depends upon three steps
  - Seek – moving the disk arm to the correct cylinder
    - Depends on how fast disk arm can move (increasing very slowly)
  - Rotation – waiting for the sector to rotate under the head
    - Depends on rotation rate of disk (increasing, but slowly)
  - Transfer – transferring data from surface into disk controller electronics, sending it back to the host
    - Depends on density (increasing quickly)
- When the OS uses the disk, it tries to minimize the cost of all these steps
  - Particularly seeks and rotation

Solid State Disks

- SSDs are a relatively new storage technology
  - Memory that does not require power to remember state
- No physical moving parts → faster than hard disks
  - No seek and no rotation overhead
  - But…more expensive, not as much capacity
- Generally speaking, file systems have remained unchanged when using SSDs
  - Some optimizations no longer necessary (e.g., layout policies, disk head scheduling), but basically leave FS code as is
  - Initially, SSDs have the same disk interface (SATA)
  - Increasingly, SSDs used directly over the I/O bus (PCIe)
    - Much higher performance

Non-Volatile Memory

- Under development are new technologies that provide non-volatile memory (NVM)
  - Phase change (PCM), spin-torque transfer (STTM), resistive
- Performance close to DRAM
  - But persistent
- Byte-addressable
  - SSD is in units of a page
- Unlike SSDs, NVM will have a dramatic effect on both operating systems and applications
  - Many research work ongoing

Disk Scheduling

- Because seeks are so slow (milliseconds!), the OS tries to schedule disk requests that are queued waiting for the disk
  - Many algorithms for doing so
- In general, unless there are many requests in queues, disk scheduling does not have much impact
  - Important for servers, less so for PCs
- Modern disks often perform disk scheduling themselves
  - Disks know their layout better than OS, can optimize better
  - Ignores, undoes any scheduling done by OS

This Lecture

File Systems

Physical Disks

Files & Directories

File System Implementation

File Systems

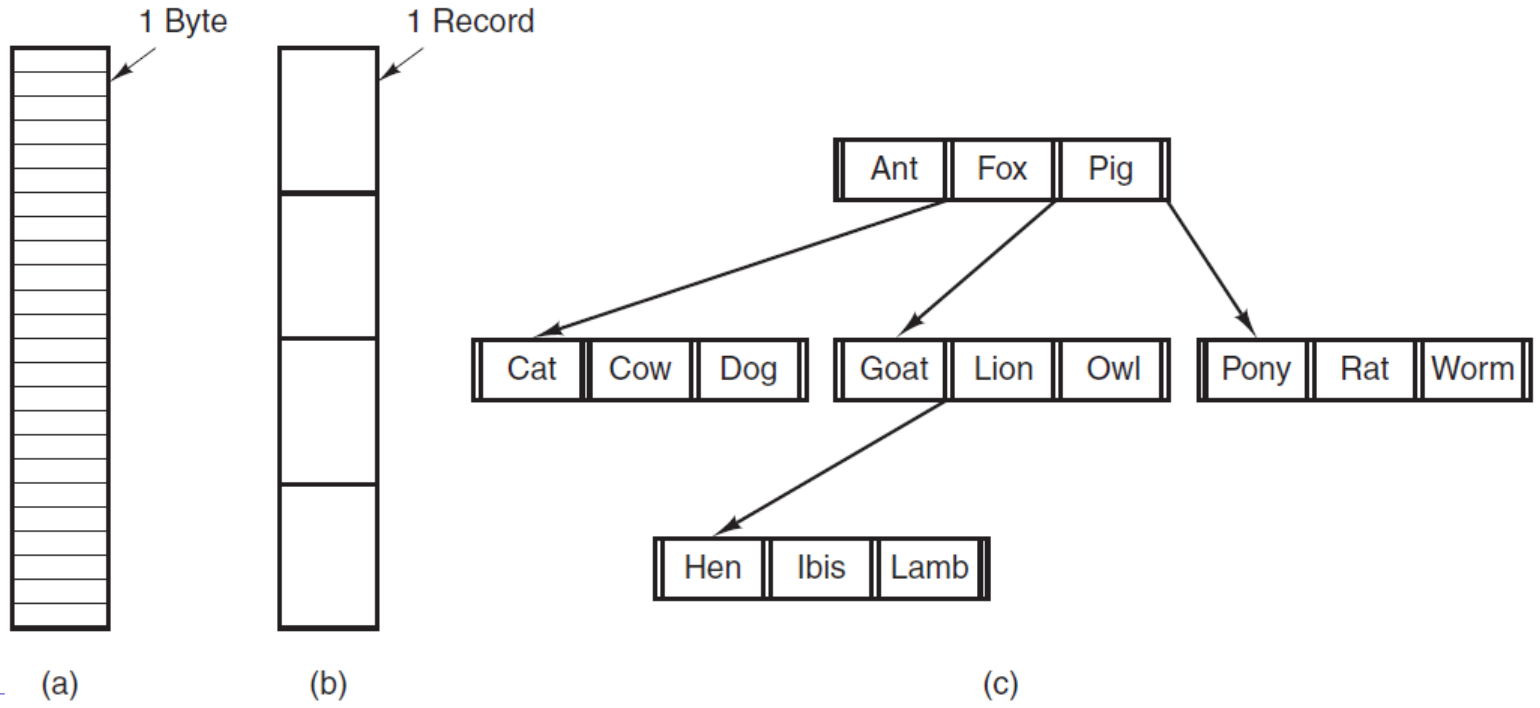
- File systems
  - Implement an abstraction (files) for secondary storage
  - Organize files logically (directories)
  - Permit sharing of data between processes, people, and machines
  - Protect data from unwanted access (protection)

Files

- Files: logical units of information created by processes
  - Contents, size, owner, last read/write time, protection, etc.
- A file can also have a type
  - Understood by the file system
    - Block, character, device, portal, link, etc.
  - Understood by other parts of the OS or runtime libraries
    - Executable, dll, source, object, text, 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)

File Structure

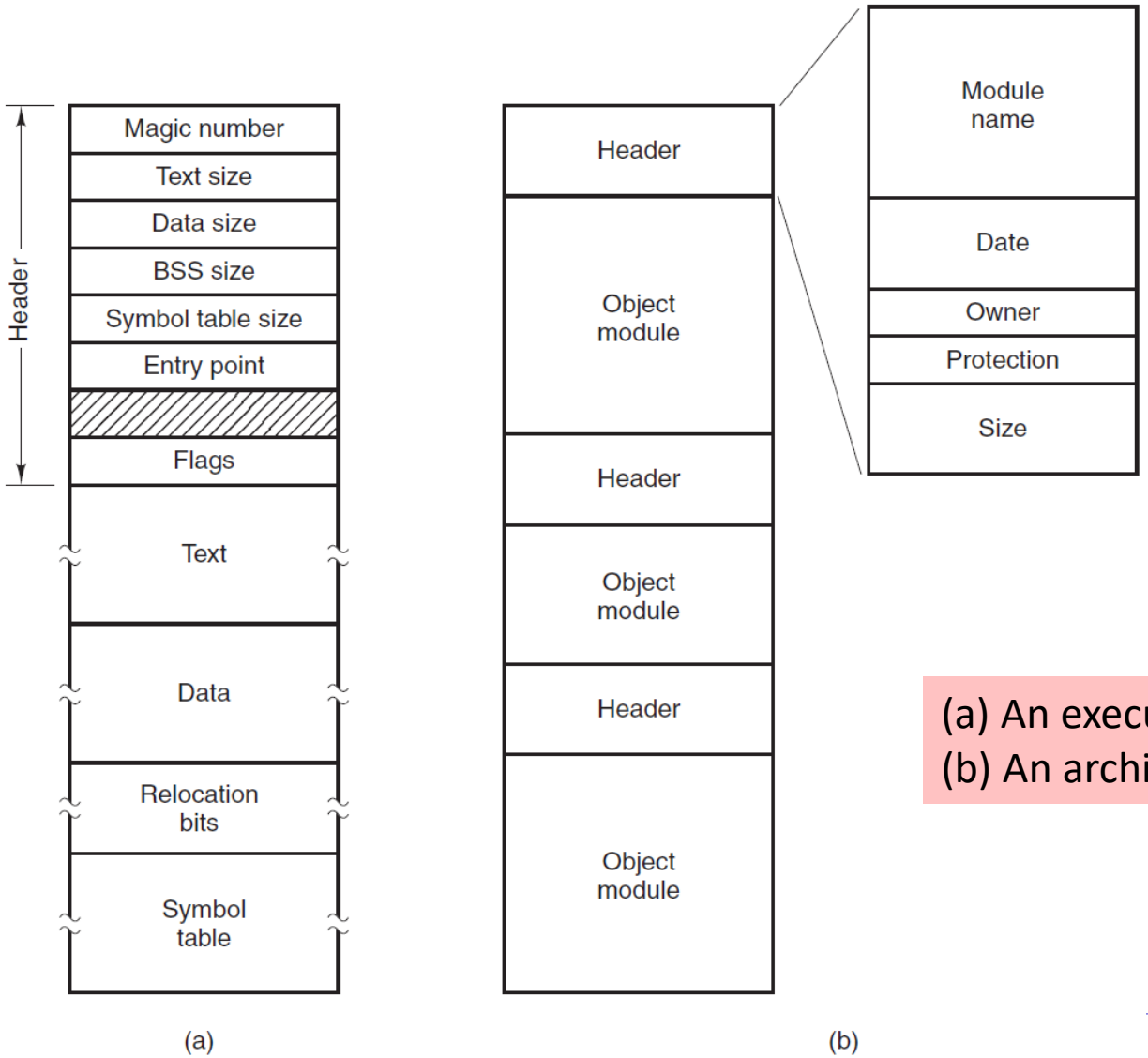
- Files can be structured in any of several ways
  - (a) unstructured sequence of bytes
  - (b) a sequence of fixed-length records
  - (c) a tree of records, each containing a key field



File Types

- Regular files
  - ASCII files
  - Binary files
- Directories
- Character special files
  - related to input/output
  - used to model serial I/O devices, such as terminals, printers, and networks.
- Block special files
  - used to model disks

Structures of Different File Types



(a) An executable file.
(b) An archive.

Basic File Operations

Unix

- create(name)
- open(name, how)
- read(fd, buf, len)
- write(fd, buf, len)
- sync(fd)
- seek(fd, pos)
- close(fd)
- link(old, new)
- symlink(old, new)
- unlink(name)

Windows

- CreateFile(name, CREATE)
- CreateFile(name, OPEN)
- ReadFile(handle, …)
- WriteFile(handle, …)
- FlushFileBuffers(handle, …)
- SetFilePointer(handle, …)
- CloseHandle(handle, …)
- DeleteFile(name)
- CopyFile(name)
- MoveFile(name)

File Access

- Some file systems provide different access methods that specify different ways for accessing data in a file
  - Sequential access – read bytes one at a time, in order
  - Direct access – random access given block/byte number
  - Record access – file is array of fixed-or variable-length records, read/written sequentially or randomly by record #
  - Indexed access – 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 (DBs)
- What file access methods does Linux/Windows provide?
- Older systems provide more complicated methods

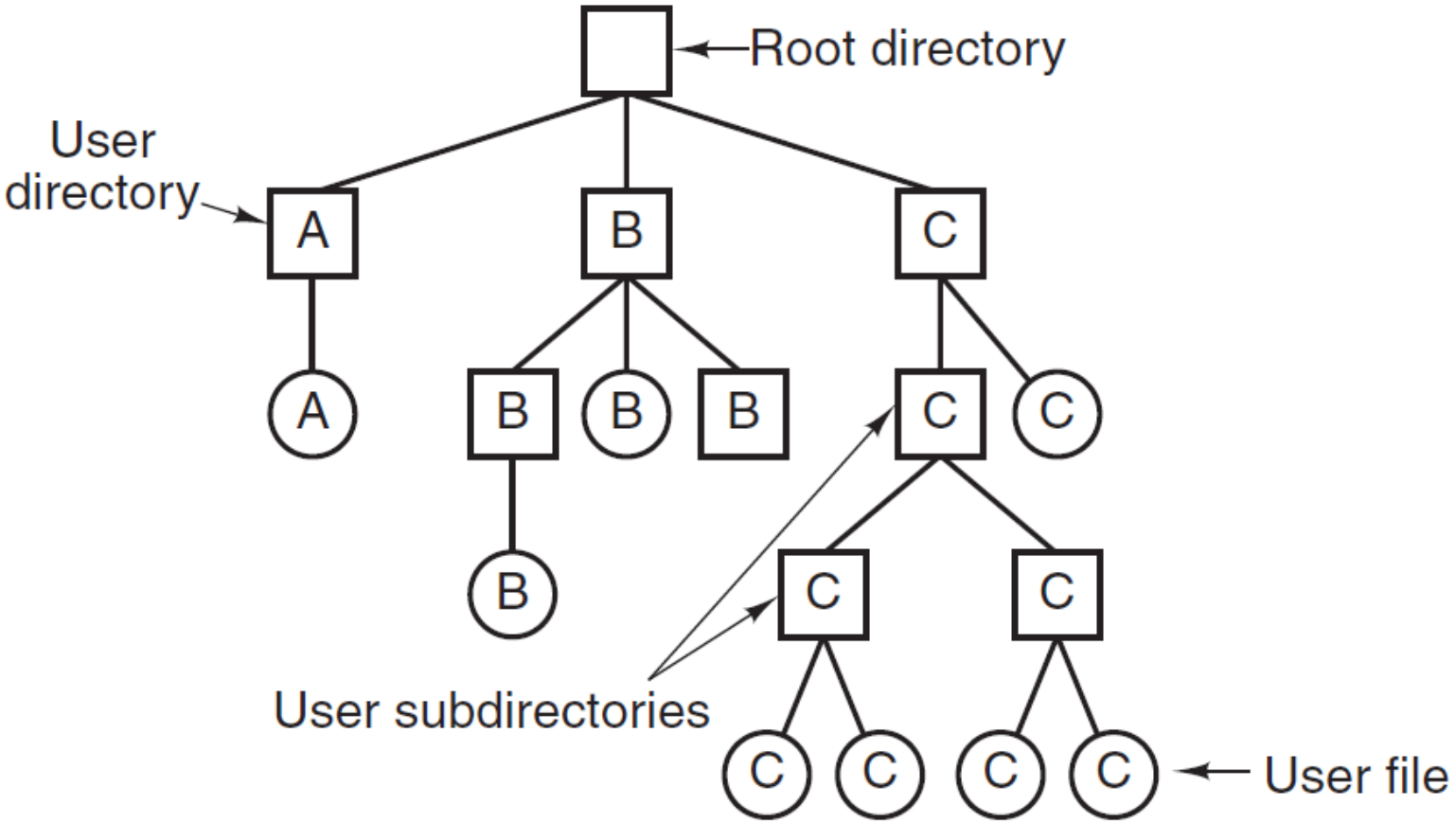
File Attributes/metadata

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Directories

- Directories serve two purposes
  - For users, they provide a structured way to organize files
  - For the file system, they provide a convenient naming interface that allows the implementation to separate logical file organization from physical file placement on the disk
- Most file systems support multi-level directories
  - Naming hierarchies (/, /usr, /usr/local/, …)
- Most file systems support the notion of a current directory (working directory)
  - Relative names specified with respect to current directory
  - Absolute names start from the root of directory tree

Hierarchical Directory System



Directory Internals

- A directory is a list of entries
  - <name, location>
  - Name is just the name of the file or directory
  - Location depends on how file is represented on disk
- List is usually unordered (effectively random)
  - Entries usually sorted by program that reads directory
  - Try "ls –U /bin"
- Directories are typically stored in files
  - Only need to manage one kind of secondary storage unit

Basic Directory Operations

Unix

- Directories implemented in files
  - Use file ops to create dirs
- C runtime library provides a higher-level abstraction for reading directories
  - opendir(name)
  - readdir(DIR)
  - seekdir(DIR)
  - closedir(DIR)

Windows

- Explicit dir operations
  - CreateDirectory(name)
  - RemoveDirectory(name)
- Very different method for reading directory entries
  - FindFirstFile(pattern)
  - FindNextFile()

Path Name Translation

- Let's say you want to open "/one/two/three"
- What does the file system do?
  - Open directory "/" (well known, can always find)
  - Search for the entry "one", get location of "one" (in dir entry)
  - Open directory "one", search for "two", get location of "two"
  - Open directory "two", search for "three", get location of "three"
  - Open file "three"
- Systems spend a lot of time walking thru directory paths
  - This is why open is separate from read/write
  - OS will cache prefix lookups for performance
    - /a/b, /a/bb, /a/bbb, etc., all share the "/a" prefix
- Absolute path name vs. Relative path name
  - Working directory

File Sharing

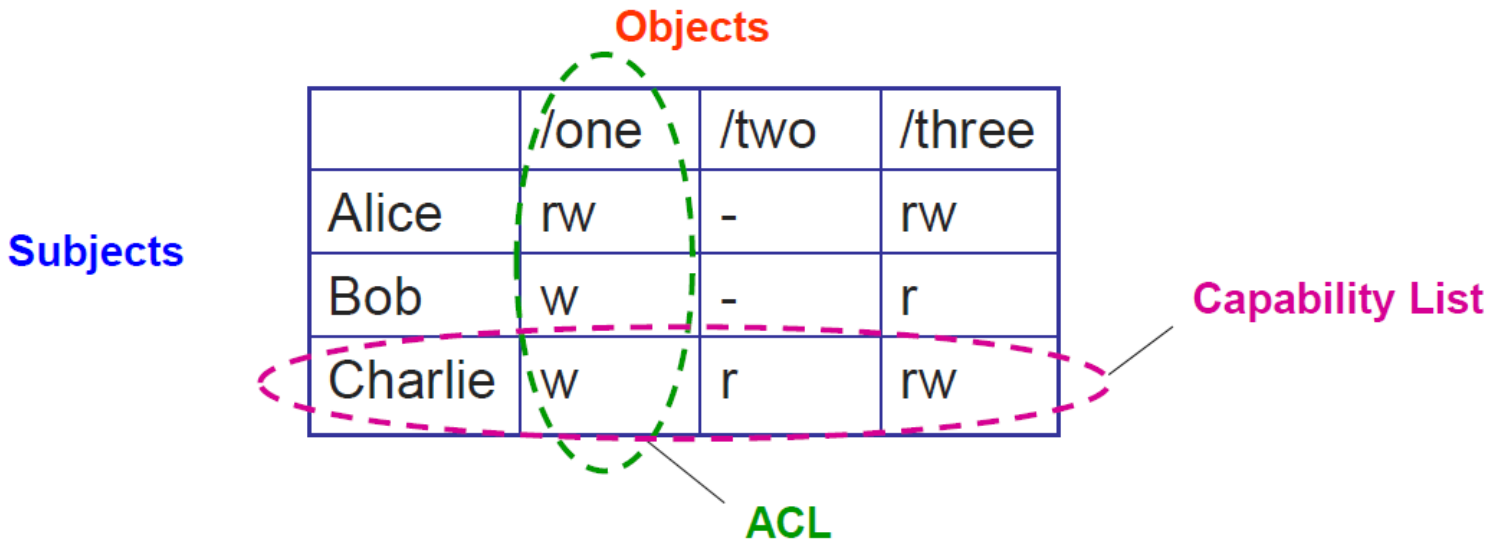
- File sharing has been around since timesharing
  - Easy to do on a single machine
  - PCs, workstations, and networks get us there (mostly)
- File sharing is important for getting work done
  - Basis for communication and synchronization
- Two key issues when sharing files
  - Semantics of concurrent 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?
  - Protection

Protection

- File systems implement a protection system
  - Who can access a file
  - How they can access it
- More generally…
  - Objects are "what", subjects are "who", actions are "how"
- A protection system dictates whether a given action performed by a given subject on a given object should be allowed
  - You can read and/or write your files, but others cannot
  - You can read "/etc/motd", but you cannot write it

Representing Protection

- 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



Root & Administrator

- The user "root" is special on Unix
  - It bypasses all protection checks in the kernel
- Administrator is the equivalent on Windows
- Always running as root can be dangerous
  - A mistake (or exploit) can harm the system
    - "rm" will always remove a file
  - This is why you create a user account on Unix even if you have root access
    - You only run as root when you need to modify the system
  - If you have Administrator privileges on Windows, then you are effectively always running as root (unfortunately)
    - Need additional protection mechanisms (User Account Control)

ACLs and Capabilities

- Approaches differ only in how the table is represented
  - What approach does Unix use in the FS?
- Capabilities are easier to transfer
  - They are like keys, can handoff, does not depend on subject
- In practice, ACLs are easier to manage
  - Object-centric, easy to grant, revoke
  - To revoke capabilities, have to keep track of all subjects that have the capability –a challenging problem
- ACLs have a problem when objects are heavily shared
  - The ACLs become very large
  - Use groups (e.g., Unix)

This Lecture

File Systems

Physical Disks

Files & Directories

File System Implementation

File System Layout

- We start with an empty disk

- Goal for the file system is to manage the disk space to implement the file and directory abstractions that are so convenient for programs and users

Key Questions

- How do we keep track of blocks used by a file?
- Where do we store metadata information?
- How do we (really) do path name translation?
- How do we implement common file operations?
- How can we cache data to improve performance?

- Our discussion will be Unix-oriented
  - Other file systems face the same challenges, with similar approaches and data structures for solving them

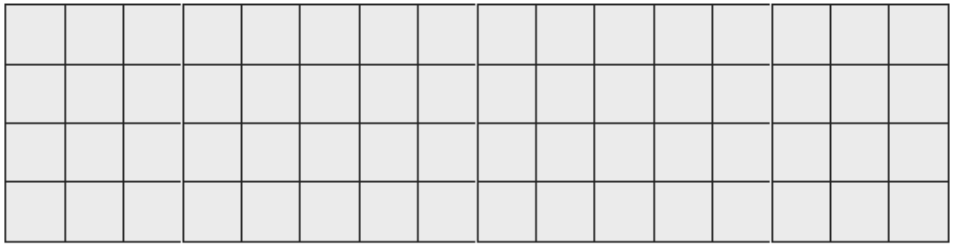
File System Layout

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 dirs)
  - There are many ways to do this

File System Layout

- Partition it into fixed-size file system blocks



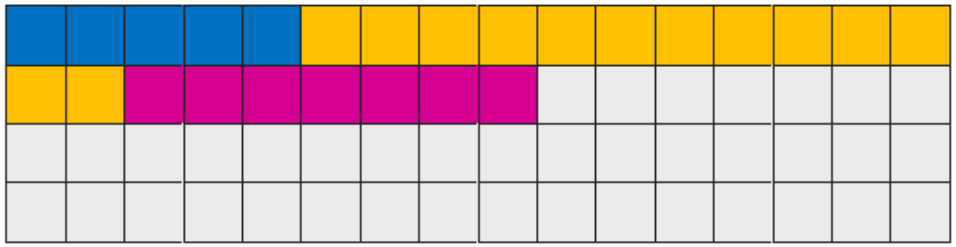
- Typically 4KB in size
  - Block size set when file system is formatted
- Independent of disk physical sector size
  - Sector 512 bytes, file system will use 8 sectors/block

File System Layout

- Files span multiple disk blocks
  - 2MB file uses 2*1024*1024/4096 = 512 blocks (4KB block size)
- A small file still uses an entire block
  - A file of size 4001 bytes uses one block
  - What kind of fragmentation is this, internal or external?
- Challenge: How do we keep track of all of the blocks used by one file?

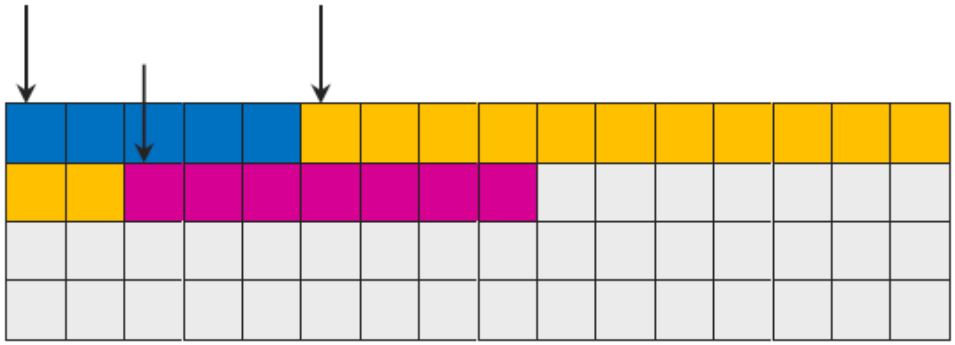
Contiguous Layout

- Can layout file blocks contiguously



Contiguous Layout

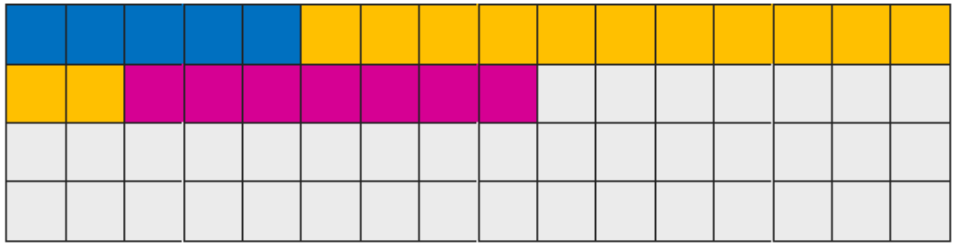
- Simple to keep track of where a file's blocks are



- Directory stores a pointer to the first block
  - All others are a simple offset from the first
  - Makes random access also straightforward
- Enables fast sequential access to disk for reads/writes
  - But there are multiple disadvantages

Contiguous Layout

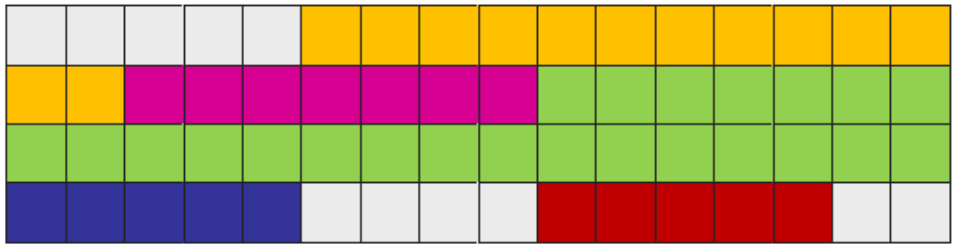
- Difficult to grow a file once it has been written



- If the blue or orange files need to grow, we're stuck

Contiguous Layout

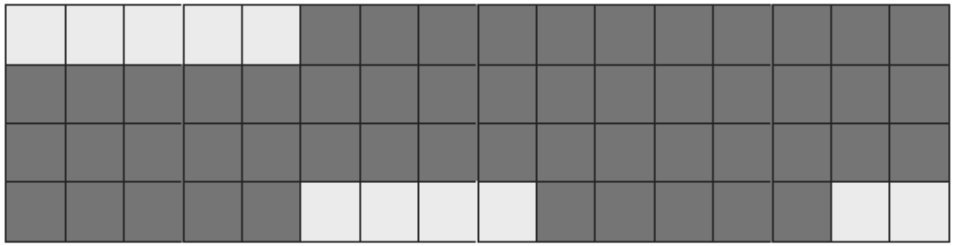
- As files are created and deleted, gaps will occur



- If we need to store a file using 8 blocks, we're stuck
  - What kind of fragmentation is this, internal or external?
  - How do we perform rearranging?

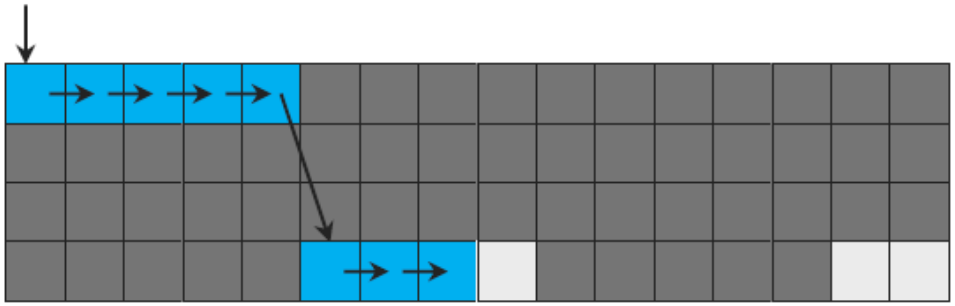
Linked Layout

- Need to store a file with 8 blocks into the “gaps”



Linked Layout

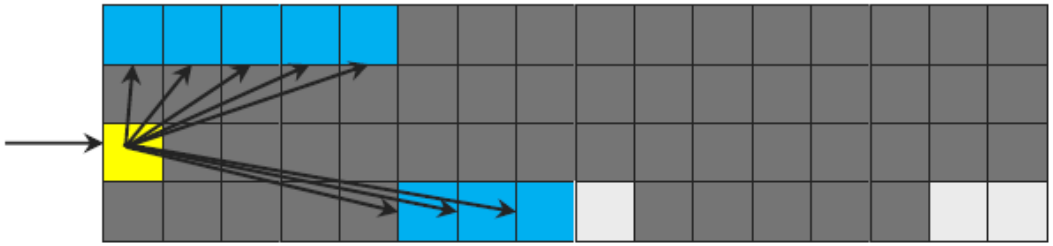
- Another option is to link each block to the next
  - Essentially a linked list on disk for each file



- Directory still just stores pointer to first block of file
- Fragmentation no longer a problem, can fill in gaps
- Random access now expensive
  - Need to traverse pointers to access a random block
- Potentially many disk reads just to get to desired block

Indexed Layout

- Indexed layouts use a special block (index block) to store pointers to the data blocks



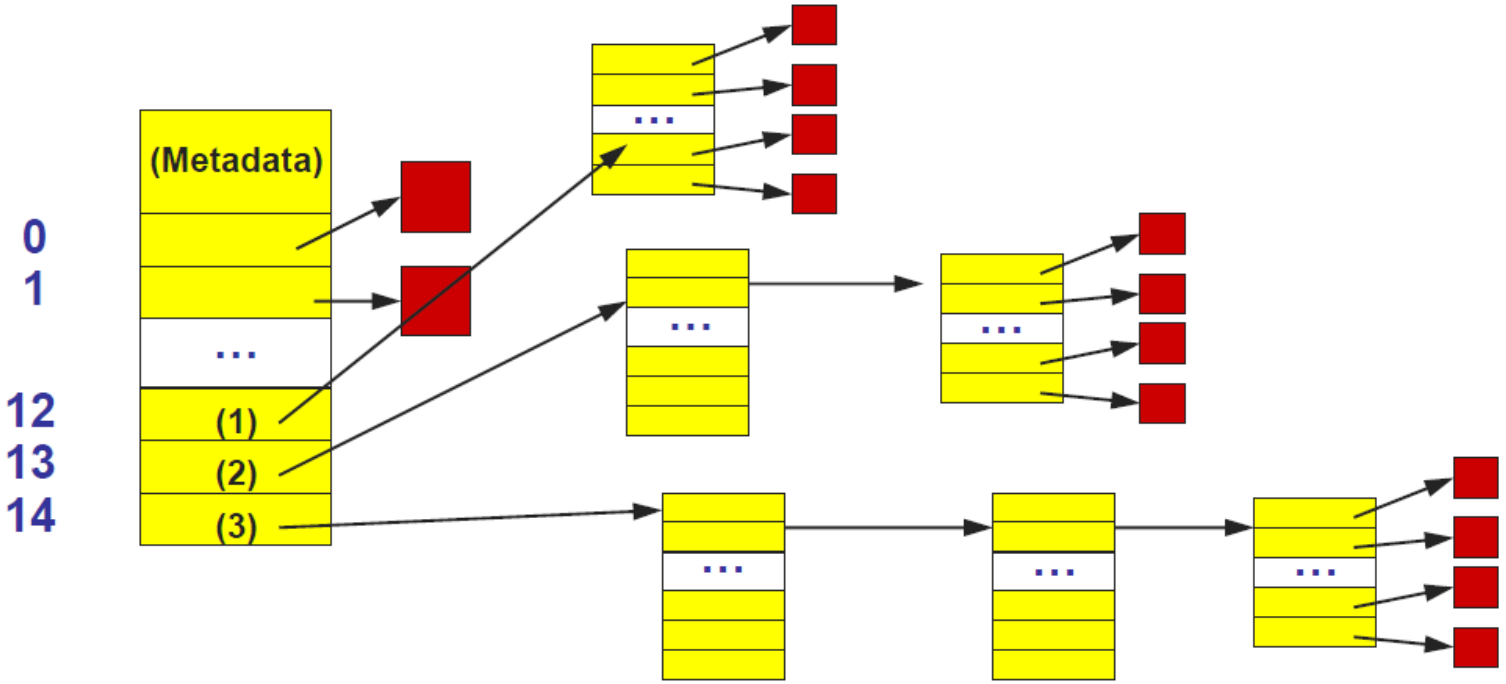
- Directory points to the index block
- Can solve the fragmentation problem (can fill in gaps)
- Also solves the random access problem
  - After reading the index block, the locations of all blocks are known
- For large files, need multiple index blocks

Disk Layout Summary

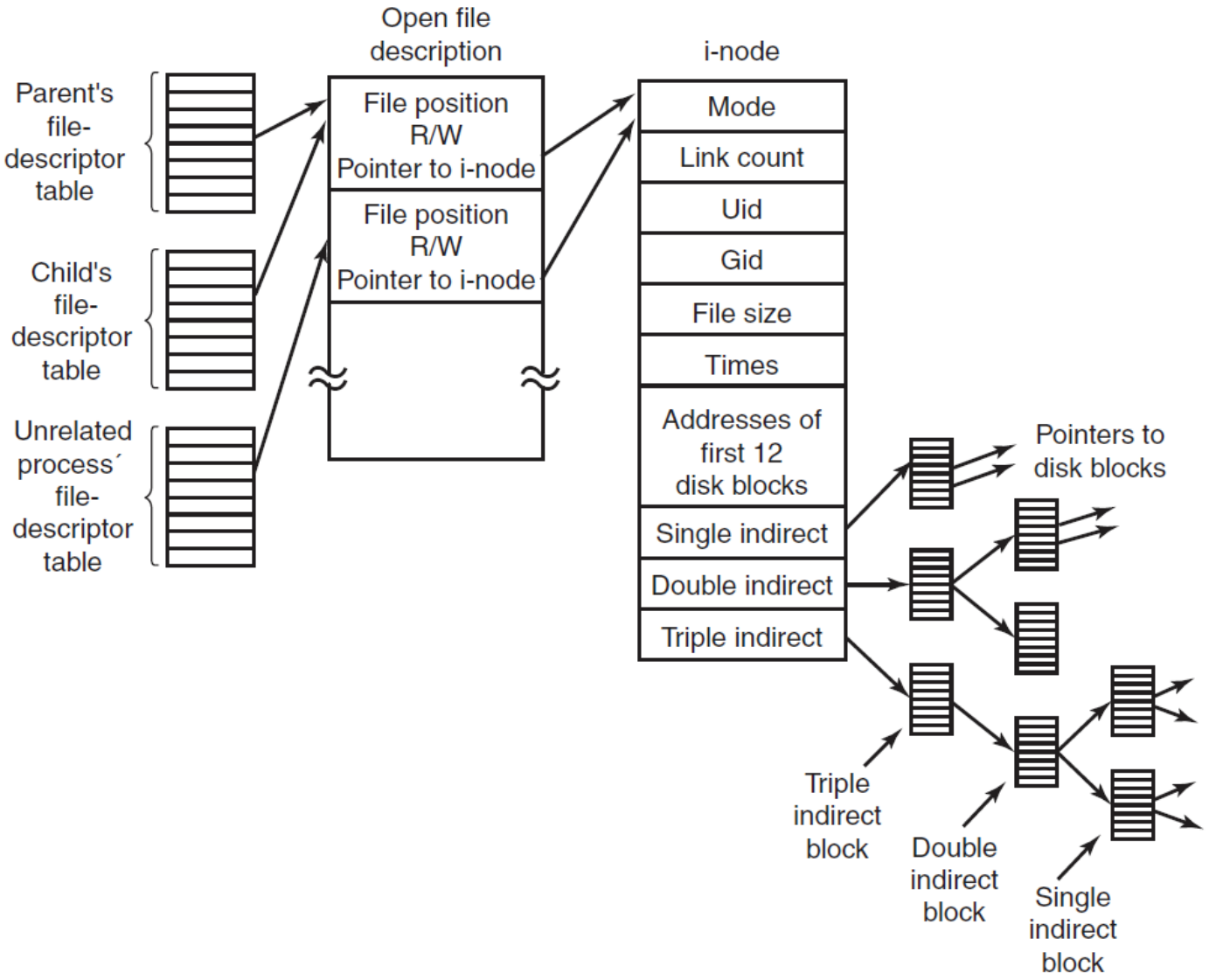
- Files span multiple disk blocks
- How do you find all of the blocks for a file?
  1. Contiguous allocation
     - Like memory
     - Fast, simplifies directory access
     - Inflexible, causes fragmentation, needs compaction
  2. Linked structure
     - Each block points to the next, directory points 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)

Unix inodes

- Unix inodes implement an indexed structure for files
- Each inode contains 15 block pointers
  - First 12 are direct blocks (e.g., 4 KB blocks)
  - Then single, double, and triple indirect



Linux inode implementation



Questions on inode

Assume block size 4KB, each pointer 4B

- What is the size limit of a file?
- What is the size limit of a file system?

Draw the resulting inode structure of the file

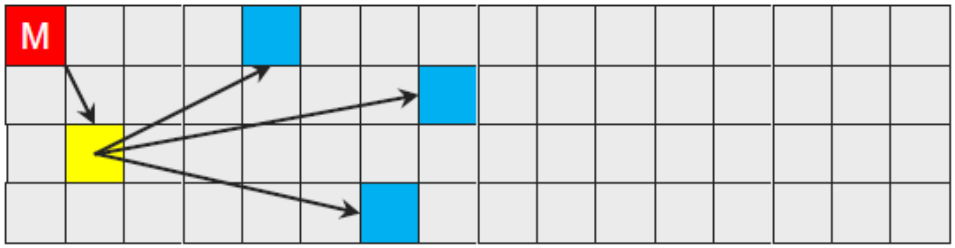
- write one byte
- seek 1MB(from beginning of file) and write another byte
- seek 1GB (from beginning of file) and write another byte.

File Metadata

- Unix inodes also store all of the metadata for a file
- File size
  - In bytes (actual file size) and blocks (data blocks allocated)
- User & group of file owner
- Protection bits
  - User/group/other, read/write/execute
- Reference count
  - How many directory entries point to this inode
- Timestamps
  - Created, modified, last accessed, any change
- "ls –l" reads this info from the inode (syscall: stat())

Master Block (Superblock)

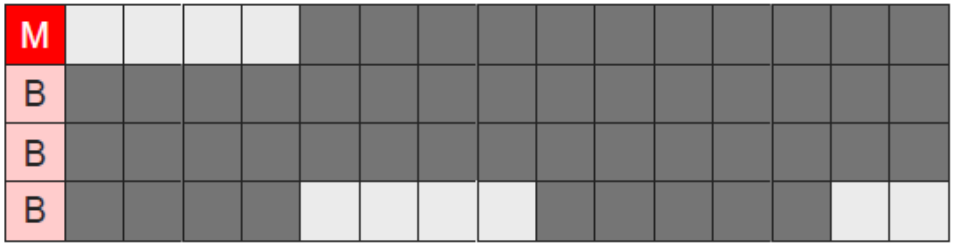
- "/" is the directory that is the root of the file system
- How do we find the inode for "/"?



- The master block M stores a pointer to the inode of "/"
  - The inode for "/" has pointers to all of the blocks storing the directory entries for "/"
- It is the basis for translating all path names
- It is stored at a fixed, pre-defined location on disk
  - Replicated deterministically across the FS for redundancy

Block Allocation

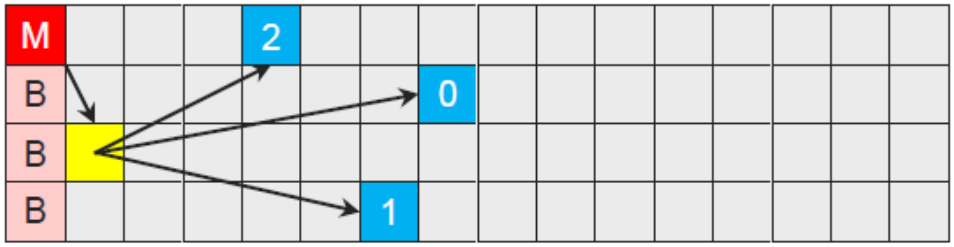
- The file system needs to keep track of which blocks have been allocated and which are free



- Free map blocks B store a bitmap, one bit per block
  - Bit is set → block is allocated
  - One bitmap for data blocks
  - Another for inode blocks

Types of Blocks on Disk

- Four basic kinds of blocks on disk
  - Only data blocks store file data and directory data



- Master block M
- Bitmap blocks B
- Inode blocks
- Data blocks 0 1 2

Unix Inodes != Directories

- Unix inodes are not directories
  - inodes describe where on the disk the blocks of a file are
- Directories are files, so inodes also describe where the blocks for directories are placed on the disk
  - Every directory and file on disk has an associated inode for it

Path Name Translation (inodes)

- 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

Symbolic (Soft) Links

- It is convenient to be able to create aliases in the FS
  - A file can be referred to by multiple names
- Soft links are the most familiar form in Unix

  % ln –s *file alias* (ln –s /a/b/c /tmp/softlink)
  - Syscall:symlink()
- Soft links create aliases via path name translation
  - Path name translation starts again when hitting a soft link
  - /tmp/softlink → /a/b/c
- Implemented simply by storing the alias as a string in a file and marking the inode as a soft link
  - FS reads the path alias from the file and restarts translation

Hard Links

- Hard links are another form of aliasing

  % ln *file alias* (ln /a/b/c /tmp/hardlink)

  - Syscall:link()
- Hard links create aliases via inode pointers in dirs
  - Recall that a directory entry maps a name to an inode
  - Creating a hard link adds another directory entry mapping the new name to the same inode as the old name
  - It adds a new pointer, or link, to the inode
  - Reference count in the inode is also incremented
- The "." and ".." names are hard links to directories
  - /a/b/c and /a/b/c/. point to the same inode

Soft vs. Hard Links

- Do the link and the original file use the same inode number?
- Can we link across different file systems?
- How about their performance?
- Is relative path allowed?
- What happens if you try to delete the link?
- What will happen if the original file is deleted?

- Can you link a directory?

Why can't you hard link a directory?

- It may create loops

```
mkdir -p /tmp/a/b
cd /tmp/a/b
ln -d /tmp/a l
```

  - A file system with a directory loop has infinite depth:

```
cd /tmp/a/b/l/b/l/b/l/b/l/b
```

- It breaks the unambiguity of parent directories
  - multiple parent directories exist (for /temp/a/b):

```
cd /tmp/a/b
cd /tmp/a/b/l/b
```

- A file may have multiple paths

```
/tmp/a/b/foo.txt
/tmp/a/b/l/b/foo.txt
```

Create a File

- Creating a file "new" is relatively straightforward
- Allocate an inode
  - Initialize metadata (owner, protection, timestamps, …)
  - Update inode bitmap
- Allocate a directory entry in the directory for the file
  - Entry maps "new" to the inode allocated for "new"
- When process starts writing to file, allocate data blocks
  - Update inode to point to data blocks allocated
  - Update data block bitmap
  - Continue to allocate data blocks on demand
    - Pre-allocating blocks in "extents" helps keep blocks contiguous

Extent Allocation

- Another allocation method besides block allocation
- An extent is a contiguous area of storage reserved for a file in a file system, represented as a range of block numbers.
  - A file can consist of zero or more extents; one file fragment requires one extent.
- Benefits
  - Storing each range compactly as two numbers, instead of canonically storing every block number in the range
  - Resulting in less file fragmentation
  - Extent-based file systems can also eliminate most of the metadata overhead of large files that would traditionally be taken up by the block-allocation tree.
- Example FS: ext4, JFS, btrfs, NTFS (called runs)

Rename a File

- One way to rename a file is to simply create a new one with the new name, copy the contents, and delete the old file
  - Method used in original version of Unix
- More efficient to implement in FS

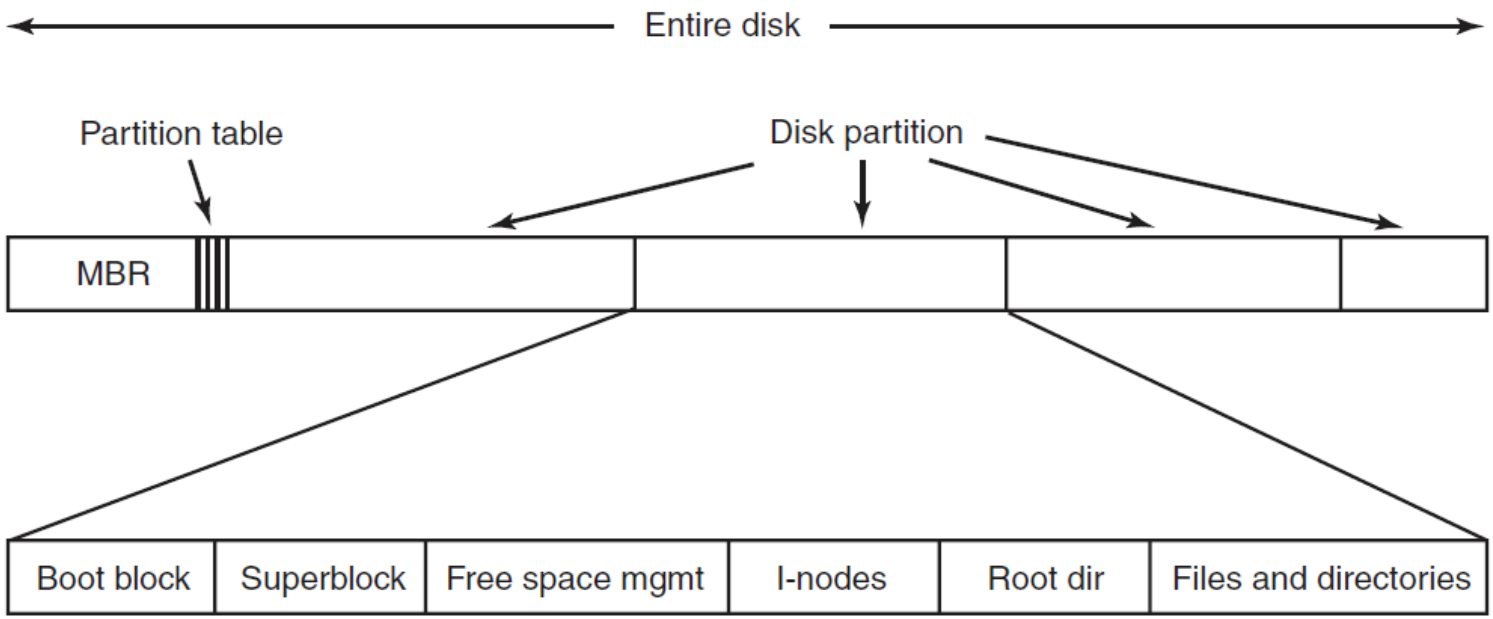
  % mv old new
  - Syscall: rename()
- Rename creates a new directory entry with the new name that points (links) to the same inode as the old
  - Then it deletes the entry directory for the old name
  - Only directories are modified, file and inode stay the same

Delete a File

- Deleting a file has a few steps
  - Remove the directory entry for the name being deleted
    - Hence the syscall name unlink()
  - Decrement the reference count in the inode
  - If the file still has links to it, nothing else happens
- If there are no remaining links
  - Free up the data blocks (update the data block bitmap)
  - Free up the inode blocks (update the inode bitmap)
  - Block data is not erased
- If the file is still open in any process, the directory entry is removed but the file blocks are not
  - Until the last process with the file open finally closes it

Partitions

- What if we want multiple file systems on one disk?
  - Split up physical disk into multiple partitions
    - MBR (Master Boot Record) is used to boot the computer
  - Each partition has an entire file system inside of it
    - Superblock contains all the key parameters about the FS



Mounting File Systems

- Mounting is the mechanism used to piece together multiple file systems into a single global name space
- One file system is mounted as "/" (root)
- Other file systems attached at mount points
  - An empty directory in file system, e.g., /home
  - Mounting the "home" file system attaches the root for "home" to /home in the name space
  - Opening "/home/username/file" starts path name translation in the "root" file system, continues in the "home" file system when crossing the mount point
- Mostly invisible to users and processes
  - Some exceptions(e.g., cannot hard link across file systems)

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: More physical memory for file cache, less for VM
  - Like VM, it has limited size
  - Need replacement algorithms again (LRU 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 changed many times in 30 secs, only need one I/O
  - If blocks deleted before 30 secs(e.g., /tmp), no I/Os needed
- Unreliable, but practical
  - On a crash, all writes within 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

Block 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
  - Can happen while the process is computing on a previous block
    - Overlap I/O with execution
  - 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)

Summary

- File systems are the solution to persistent storage
  - Disk allocation
  - Files & directories
  - File system implementation
  - File system optimization

- Next Lecture: FS Examples