Today

- File Systems
  - Files

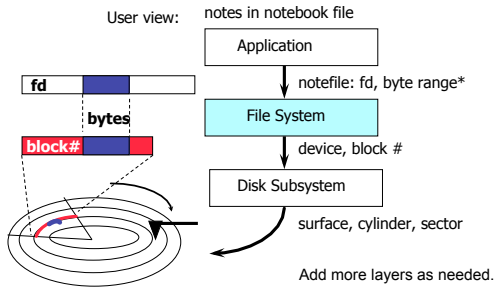
Review

- What is RAID?
  - What level of RAID should you use?

File Systems

- Abstraction on top of persistent storage
  - Magnetic disk
  - Flash memory (e.g., USB thumb drive)
- Devices provide
  - Storage that (usually) survives across machine crashes
  - Block level (random) access
  - Large capacity at low cost
  - Relatively slow performance
    - Magnetic disk read takes 10-20M processor instructions

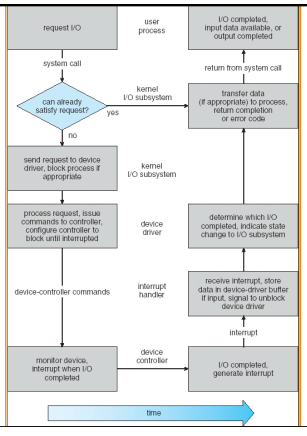
Names and layers



I/O Requests to Hardware Operations

- Consider reading a file from disk for a process:
  - Determine device holding file
  - Translate name to device representation
  - Physically read data from disk into buffer
  - Make data available to requesting process
  - Return control to process

Life Cycle of An I/O Request



The block storage abstraction

- Read/write blocks of size b on a logical storage device ("disk").
- A disk is a numbered array of these basic blocks. Each block is named by a unique number (e.g., logical `BlockID`).
- CPU (typically executing kernel code) forms buffer in memory and issues read or write command to device queue/driver.
- Device DMAs data to/from memory buffer, then interrupts the CPU to signal completion of each request.
- Device I/O is asynchronous: the CPU is free to do something else while I/O in progress.
- Transfer size b may vary, but is always a multiple of some basic block size (e.g., sector size), which is a property of the device, and is always a power of 2.
- Storage blocks containing data/metadata are cached in memory buffers while in active use
  - called buffer cache or block cache

System Performance

- I/O: major factor in system performance:
  - Demands CPU to execute device driver, kernel I/O code
  - Context switches due to interrupts
  - Data copying
  - Network traffic

Peek behind the curtain

FILES

What is a file?

- What does the file represent?
  - What does it abstract?
- What is the state and methods associated with a file?
  - What are the requirements for the methods?

File Abstraction

- A (potentially) large amount of information or data that lives a (potentially) very long time
  - Often much larger than the memory of the computer
  - Often much longer than any computation
  - Sometimes longer than life of machine itself
- (Usually) organized as a linear array of bytes or blocks
  - Internal structure is imposed by application
  - (Occasionally) blocks may be variable length
- (Often) requiring concurrent access by multiple processes
  - Even by processes on different machines!

File = Long-term Information Storage

1. Must store large amounts of data
2. Information stored must survive the termination of the process using it
3. Multiple processes must be able to access the information concurrently

Gap in Perspective of File Systems and Disks

- User view
  - File is a named, persistent collection of data
- OS & file system view
  - File is collection of disk blocks — i.e., a container
  - File System maps file names and offsets to disk blocks

File – a powerful abstraction

- Documents, code
- Databases
  - Very large, possibly spanning multiple disks
- Streams
  - Input, output, keyboard, display
  - Pipes, network connections, …
- Virtual memory backing store
- Temporary repositories of OS information
- Any time you need to remember something beyond the life of a particular process/computation

Fundamental ambiguity

- Is the *file* the "container of the information" or the "information" itself?
- Almost all systems confuse the two.
- Almost all people confuse the two.

Example

- Suppose you email me a document
- Later, how do either of us know that we are using the same version of the document?
- Windows/Outlook/Exchange/MacOS:
  - Time-stamp is a pretty good indication that they are
  - Time-stamps preserved on copy, drag and drop, transmission via e-mail, etc.
- Unix/Linux
  - By default, time-stamps are not preserved on copy, ftp, e-mail, etc.
  - Time-stamp associated with container, not with information

Rule of Thumb

- Almost always, *people* and *applications* think in terms of the *information*
- Many *systems* think in terms of *containers*

Professional Guidance: *Be aware of the distinction, even when the system is not*

Definition — *File Metadata*

- Information *about* a file
- Maintained by the file system
- Separate from file itself
- Usually attached or connected to the file
  - E.g., in block # 1
- Some information visible to user/application
  - Dates, permissions, type, name, etc.
- Some information primarily for OS
  - Location on disk, locks, cached attributes
    - Location is stored in metadata
    - Location can change, even if file does not
    - Location is not visible to user or program

Attributes of Files

- Name
- Identifier
  - Unique number identifies file within file system
- Type:
  - May be encoded in the name (e.g., .cpp, .txt)
- Dates:
  - Creation, updated, last accessed, etc.
  - (Usually) associated with container
  - Better if associated with content
- Location
  - pointer to file location on device
- Size:
  - Length in number of bytes; occasionally rounded up
- Protection:
  - Owner, group, etc.
  - Authority to read, update, extend, etc.
- Locks:
  - For managing concurrent access

Example – Location

- Example 1:

```
mv ~/Project4.pdf ~/public_html/cs330/projects/
```

- Example 2:
  - System moves file from disk block 10,000 to disk block 20,000
  - System restores a file from backup
- May or may not be reflected in metadata

File Operations

- Create
- Write – at write pointer location
- Read – at read pointer location
- Reposition within file - seek
- Delete
- Truncate
- Open(f_i)
  - search the directory structure on disk for entry f_i, and move the content of entry to memory
- Close (f_i)
  - move the content of entry f_i in memory to directory structure on disk

Open Files

- Data to manage open files:
  - Open-file table: tracks open files
  - File pointer: pointer to last read/write location, per process that has the file open
  - File-open count: counter of number of times a file is open
    - Allows removal open-file table entry when last processes closes it
  - Disk location of the file: cache of data access information
  - Access rights: per-process access mode information

Representing files: inodes

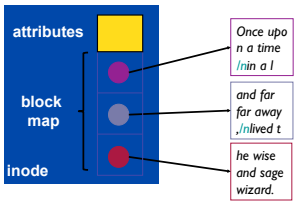
- There are many, many file system implementations.
- Most of them use a **block map** to represent each file.
- Each file is represented by a corresponding data object, which is the root of its block map, and holds other information about the file (the file's "metadata").
- In classical Unix and many other systems, this per-file object is called an ***inode***, "index node"
- The inode for a file is stored "on disk": the OS/FS reads it in and keeps it in memory while the file is in active use.
- When a file is modified, the OS/FS writes any changes to its inode/maps back to the disk.

Inodes

A file's data blocks could be "anywhere" on disk. The file's **inode** maps them. Each entry of the map gives the disk location for the corresponding logical block.

A fixed-size inode has a fixed-size block map.

How to represent large files that have more logical blocks than can fit in the inode's map?



An inode could be "anywhere" on disk.
How to find the inode for a given file?
Assume: inodes are uniquely numbered:
we can find an inode from its number.

Classical Unix inode

A classical Unix **inode** has a set of **file attributes** in addition to the root of a hierarchical block map for the file.
The inode structure size is fixed, e.g., total size is 128 bytes: 16 inodes fit in a 4KB block.

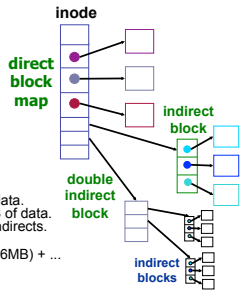
```
/* Metadata returned by the stat and fstat functions */
struct stat {
    dev_t         st_dev;      /* device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* protection and file type */
    nlink_t       st_nlink;    /* number of hard links */
    uid_t         st_uid;      /* user ID of owner */
    gid_t         st_gid;      /* group ID of owner */
    dev_t         st_rdev;     /* device type (if inode device) */
    off_t         st_size;     /* total size, in bytes */
    unsigned long st_blksize;  /* blocksize for filesystem I/O */
    unsigned long st_blocks;   /* number of blocks allocated */
    time_t        st_atime;    /* time of last access */
    time_t        st_mtime;    /* time of last modification */
    time_t        st_ctime;    /* time of last change */
};
```


Representing Large Files

Classical Unix file systems

inode == 128 bytes

Each inode has 68 bytes of attributes and 15 block map entries that are the root of a tree-structured block map.



Suppose block size = 8KB
12 direct block map entries: map 96KB of data.
One indirect block pointer in inode: + 16MB of data.
One double indirect pointer in inode: +2K indirects.

Maximum file size is 96KB + 16MB + (2K*16MB) + ...

The numbers are for illustration only.

Skewed tree block maps

- Inodes are the root of a tree-structured block map.
- These maps are skewed.
  - Low branching factor at the root.
  - "The further you go, the bushier they get."
  - Small files are cheap: just need the inode to map it.
    - **…and most files are small.**
- Use indirect blocks for large files.
  - Requires another fetch for another level of map block
  - But the shift to a high branching factor covers most large files.
- Double indirect blocks allow very large files.

Looking Ahead

- Project 4
- File systems continuing
  - File system design
  - NSF