File Systems Overview

Remember the high-level view of the OS as a translator from the user abstraction to the hardware reality.

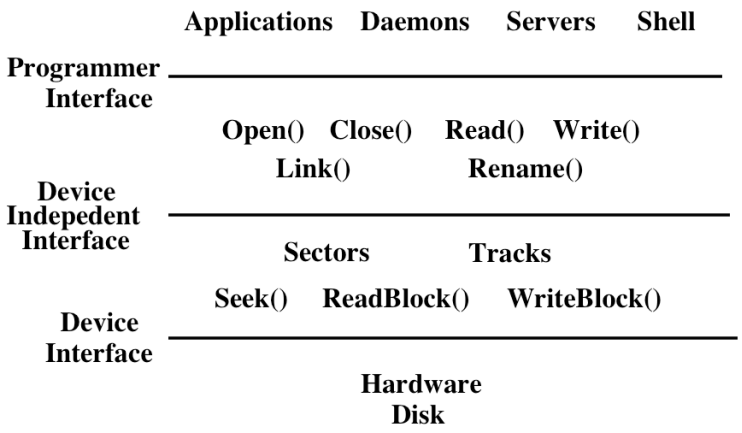
User Abstraction		Hardware Resource
Processes/Threads		CPU
Address Space	<= OS =>	Memory
Files		Disk

Key File System Functionality

- Naming
- Protection
- Persistence
- Fast access

- File System Design involves two aspects
  - How data (files, directories) are stored on disk?
    - Layout on disk, on-disk data structures
  - How the file system is implemented in the OS?
    - Interface, OS implementation

File System Abstraction



User Requirements on Data

- **Persistence:** data stays around between jobs, power cycles, crashes
- **Speed:** can get to data quickly
- **Size:** can store lots of data
- **Sharing/Protection:** users can share data where appropriate or keep it private when appropriate
- **Ease of Use:** user can easily find, examine, modify, etc. data

Hardware/OS Features

- Hardware provides:
  - **Persistence:** Disks provide non-volatile memory
  - **Speed:** Speed gained through random access
  - **Size:** Disks keep getting bigger (typical disk on a PC=500GB - 1TB)
- OS provides:
  - **Persistence:** redundancy allows recovery from some additional failures
  - **Sharing/Protection:** Unix provides read, write, execute privileges for files
  - **Ease of Use**
    - Associating names with chunks of data (files)
    - Organize large collections of files into directories
    - Transparent mapping of the user's concept of files and directories onto locations on disks
    - Search facility in file systems (SpotLight in Mac OS X)

Files

- **File:** Logical unit of storage on a storage device
  - Formally, named collection of related information recorded on secondary storage
  - **Example:** reader.cc, a.out
- Files can contain programs (source, binary) or data
- Files can be structured or unstructured
  - Unix implements files as a series of bytes (unstructured)
  - IBM mainframes implements files as a series of records or objects (structured)
- File attributes: name, type, location, size, protection, creation time

User Interface to the File System

Common file operations (system calls)

Data operations:

Create()	Open()	Read()
Delete()	Close()	Write()
		Seek()

Naming operations: Attributes (owner, protection,...):

HardLink()	SetAttribute()
SoftLink()	GetAttribute()
Rename()	

OS File Data Structures

1. Open file table - shared by all processes with an open file.
   - open count
   - file attributes, including ownership, protection information, access times, ...
   - location(s) of file on disk
   - pointers to location(s) of file in memory

2. Per-process file table - for each file,
   - pointer to entry in the open file table
   - current position in file (offset)
   - mode in which the process will access the file (r, w, rw)
   - pointers to file buffer

- **These are in-memory OS data structures - not on-disk data structures.**

File Operations: Creating a File

- **Create(name)**
  - Allocate disk space (check disk quotas, permissions, etc.)
  - Create a file descriptor for the file including name, location on disk, and all file attributes.
  - Add the file descriptor to the directory that contains the file.
  - Optional file attribute: file type (Word file, executable, etc.)
    - **Advantages:** better error detection, specialized default operations (double-clicking on a file knows what application to start), enables storage layout optimizations
    - **Disadvantages:** makes the file system and OS more complicated, less flexible for user.
    - Unix opts for simplicity (no file types), Macintosh/Windows opt for user-friendliness

File Operations: Deleting a File

- **Delete(name)**
  - Find the directory containing the file.
  - Free the disk blocks used by the file.
  - Remove the file descriptor from the directory.
  - Refcounts and hardlinks?

File Operations: Open and Close

- **fileId = Open(name, mode)**
  - Check if the file is already open by another process. If not,
    - Find the file.
    - Copy the file descriptor into the system-wide open file table.
  - Check the protection of the file against the requested mode. If not ok, abort
  - Increment the open count.
  - Create an entry in the process's file table pointing to the entry in the system-wide file table. Initialize the current file pointer to the start of the file.
- **Close(fileId)**
  - Remove the entry for the file in the process's file table.
  - Decrement the open count in the system-wide file table.
  - If the open count == 0, remove the entry in the system-wide file table.

OS File Operations: Reading a File

- **Read(fileID, from, size, bufAddress)** - random access
  - OS reads "size" bytes from file position "from" into "bufAddress"

```
for (i = from; i < from + size; i++)
    bufAddress[i - from] = file[i];
```

- **Read(fileID, size, bufAddress)** - sequential access
  - OS reads "size" bytes from current file position, fp, into "bufAddress" and increments current file position by size

```
for (i = 0; i < size; i++)
    bufAddress[i] = file[fp + i];
fp += size;
```

OS File Operations

- **Write** is similar to reads, but copies from the buffer to the file.
- **Seek** just updates fp.
- **Memory mapping** a file
  - Map a part of the portion virtual address space to a file
  - Read/write to that portion of memory \implies OS reads/writes from corresponding location in the file
  - File accesses are greatly simplified (no read/write call are necessary)

File Access Methods

- Common file access patterns from the programmer's perspective
  - **Sequential:** data processed in order, a byte or record at a time.
    - Most programs use this method
    - **Example:** compiler reading a source file.
  - **Keyed:** address a block based on a key value.
    - **Example:** database search, hash table, dictionary

- Common file access patterns from the OS perspective:
  - **Sequential:** keep a pointer to the next byte in the file. Update the pointer on each read/write.
  - **Random:** address any block in the file directly given its offset within the file.

Naming and Directories

- Need a method of getting back to files that are left on disk.
- OS uses numbers for each files
  - Users prefer textual names to refer to files.
  - **Directory:** OS data structure to map names to file descriptors
- Naming strategies
  - **Single-Level Directory:** One name space for the entire disk, every name is unique.
    1. Use a special area of disk to hold the directory.
    2. Directory contains <name, index> pairs.
    3. If one user uses a name, no one else can.
    4. Some early computers used this strategy. Early personal computers also used this strategy because their disks were very small.
  - **Two Level Directory:** each user has a separate directory, but all of each user's files must still have unique names

Naming Strategies (continued)

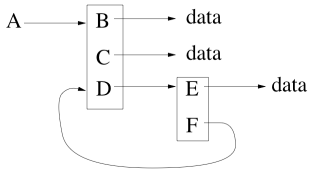
- Multilevel Directories - tree structured name space (Unix, and all other modern operating systems).
  1. Store directories on disk, just like files except the file descriptor for directories has a special flag bit.
  2. User programs read directories just like any other file, but only special system calls can write directories.
  3. Each directory contains <name, fileDesc> pairs in no particular order. The file referred to by a name may be another directory.
  4. There is one special root directory. *Example:* How do we look up name: / usr/local/bin/netscape

Referential naming

- Hard links (Unix: *ln* command)
  - A hard link adds a second connection to a file
  - *Example:* creating a hard link from B to A

Initially:	A → file #100
After "*ln* A B":	A → file #100
	B → file #100

  - OS maintains reference counts, so it will only delete a file after the last link to it has been deleted.
  - *Problem:* user can create circular links with directories and then the OS can never delete the disk space.
  - *Solution:* No hard links to directories



Referential Naming

- Soft links (Unix: *ln -s* command)
  - A soft link only makes a symbolic pointer from one file to another.
  - *Example:* creating a soft link from B to A

Initially:	A → file #100
After "*ln* A B":	A → file #100
	B → A

  - removing B does not affect A
  - removing A leaves the name B in the directory, but its contents no longer exists
  - *Problem:* circular links can cause infinite loops (e.g., trying to list all the files in a directory and its subdirectories)
  - *Solution:* limit number of links traversed.

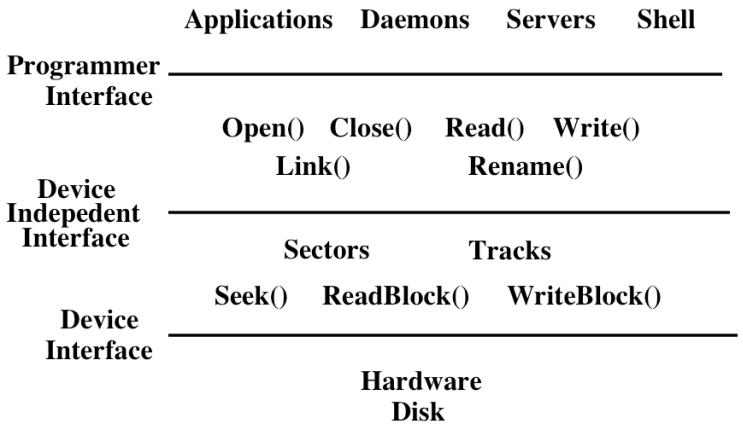
Directory Operations

- Search for a file: locate an entry for a file
- Create a file: add a directory listing
- Delete a file: remove directory listing
- List a directory: list all files (*ls* command in UNIX)
- Rename a file
- Traverse the file system

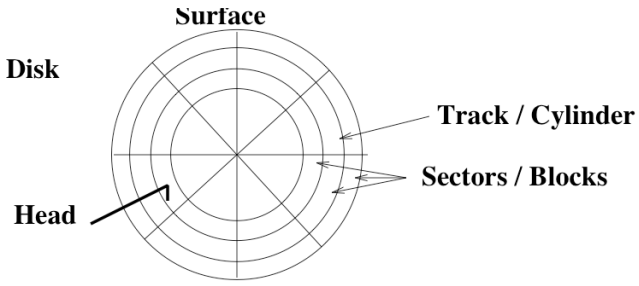
Protection

- The OS must allow users to control sharing of their files => control access to files
- Grant or deny access to file operations depending on protection information
- **Access lists and groups** (Windows NT)
  - Keep an access list for each file with user name and type of access
  - Lists can become large and tedious to maintain
- **Access control bits** (UNIX)
  - Three categories of users (owner, group, world)
  - Three types of access privileges (read, write, execute)
  - Maintain a bit for each combination (111101000 = rwxr-x---)

Next: On-disk File System Structures



How Disks Work



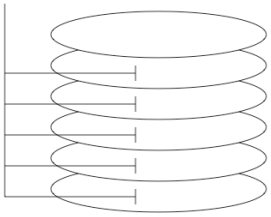
- The disk surface is circular and is coated with a magnetic material. The disk is always spinning (like a CD).
- Tracks are concentric rings on disk with bits laid out serially on tracks.
- Each track is split into *sectors* or *blocks*, the minimum unit of transfer from the disk.

Disk Overheads

- **Overhead:** time the CPU takes to start a disk operation
- **Latency:** the time to initiate a disk transfer of 1 byte to memory.
  - **Seek time:** time to position the head over the correct cylinder
  - **Rotational time:** the time for the correct sector to rotate under the head
- **Bandwidth:** once a transfer is initiated, the rate of I/O transfer

How Disks Work

- CDs come individually, but disks come organized in *disk pack* consisting of a stack of platters.
- Disk packs use both sides of the platters, except on the ends.
- Comb has 2 read/write head assemblies at the end of each arm.
- *Cylinders* are matching sectors on each surface.
- Disk operations are in terms of radial coordinates.
  - Move arm to correct track, waiting for the disk to rotate under the head.
  - Select and transfer the correct sector as it spins by



File Organization on Disk

The information we need:

fileID 0, Block 0 → Platter 0, cylinder 0, sector 0

fileID 0, Block 1 → Platter 4, cylinder 3, sector 8

Recall from last class that we can use Logical Block Addressing (LBA) for disk rather than physical addresses as well.

Key performance issues:

1. We need to support sequential and random access.
2. What is the right data structure in which to maintain file location information?
3. How do we lay out the files on the physical disk?

File Organization: On-Disk Data Structures

- The structure used to describe where the file is on the disk and the attributes of the file is the *file descriptor (FileDesc)*. File descriptors have to be stored on disks just like files.

- Most systems fit the following profile:
  1. Most files are small.
  2. Most disk space is taken up by large files.
  3. I/O operations target both small and large files.

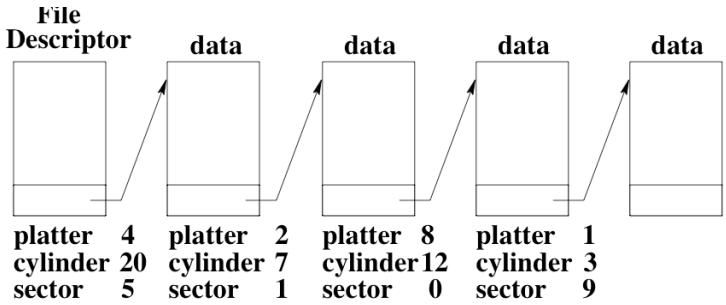
  => The per-file cost must be low, but large files must also have good performance.

Contiguous Allocation

- OS maintains an ordered list of free disk blocks
- OS allocates a contiguous chunk of free blocks when it creates a file.
- Need to store only the start location and size in the file descriptor
- **Advantages**
  - Simple,
  - Access time? Number of seeks?
- **Disadvantages**
  - Changing file sizes
  - Fragmentation? Disk management?
- **Examples:** IBM OS/360, write-once disks, early personal computers

Linked files

- Keep a list of all the free sectors/blocks.
- In the file descriptor, keep a pointer to the first sector/block.
- In each sector, keep a pointer to the next sector.



Linked files

- **Advantages:**
  - Fragmentation?
  - File size changes?
  - Efficiently supports which type of access?
- **Disadvantages:**
  - Does not support which type of access? Why?
  - Number of seeks?
- **Examples:** MS-DOS

Indexed files

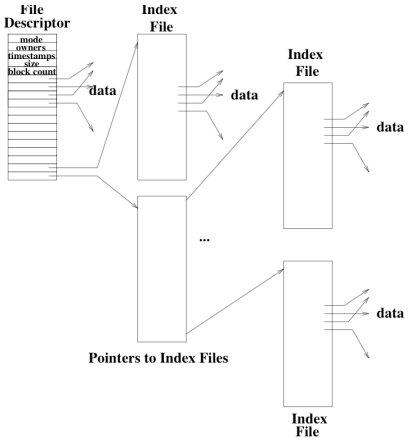
- OS keeps an array of block pointers for each file.
- The user or OS must declare the maximum length of the file when it is created.
- OS allocates an array to hold the pointers to all the blocks when it creates the file, but allocates the blocks only on demand.
- OS fills in the pointers as it allocates blocks.

Indexed files

- **Advantages**
  - Not much wasted space.
  - Both sequential and random accesses are easy.
- **Disadvantages**
  - Sets a maximum file size.
  - Lots of seeks because data is not contiguous.
-

Multilevel indexed files

- Each file descriptor contains 14 block pointers.
- First 12 pointers point to data blocks.
- 13*th* pointer points to a block of 1024 pointers to 1024 more data blocks. (One indirection)
- 14*th* pointer points to a block of pointers to indirect blocks. (Two indirections)



Multilevel indexed files: BSD UNIX 4.3

- **Advantages**
  - Simple to implement
  - Supports incremental file growth
  - Small files?
- **Disadvantages**
  - Indirect access is inefficient for random access to very large files.
  - Lots of seeks because data is not contiguous.
- Is the file size bounded?
- What could the OS do to get more contiguous access and fewer seeks?

Free-Space Management

- Need a free-space list to keep track of which disk blocks are free (just as we need a free-space list for main memory)
- Need to be able to find free space quickly and release space quickly => use a bitmap
  - The bitmap has one bit for each block on the disk.
  - If the bit is 1, the block is free. If the bit is 0, the block is allocated.
- Can quickly determine if any page in the next 32 is free, by comparing the word to 0. If it is 0, all the pages are in use. Otherwise, you can use bit operations to find an empty block.

  11000010010001111110...
- Marking a block as freed is simple since the block number can be used to index into the bitmap to set a single bit.

Free-Space Management

- **Problem:** Bitmap might be too big to keep in memory for a large disk. A 2 GB disk with 512 byte sectors requires a bitmap with 4,000,000 entries (500,000 bytes).
- If most of the disk is in use, it will be expensive to find free blocks with a bitmap.
- An alternative implementation is to link together the free blocks.
  - The head of the list is cached in kernel memory. Each block contains a pointer to the next free block.
  - How expensive is it to allocate a block?
  - How expensive is it to free a block?
  - How expensive is it to allocate consecutive blocks?

Super Block

- In unix file systems, the super block is a special disk block(s) for meta-data
- Super blocks are stored on known locations (e.g., on track 0)
- Super block store inodes sequentially
- Inodes have a fixed size and are preallocated
  - Loc. of inode i: block # = [size of inode * i] /blk size