
Disks & File Systems

CSE 410, Spring 2009
Computer Systems

<http://www.cs.washington.edu/410>

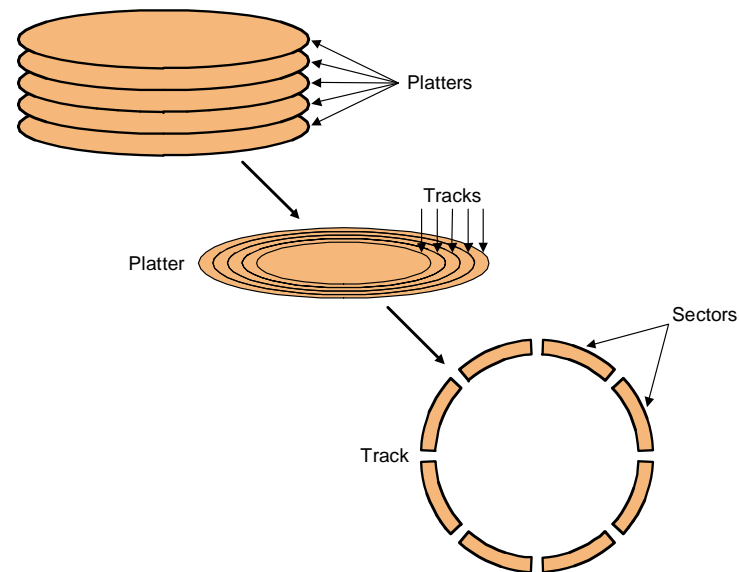
Readings and References

- Reading

- » Sec. 6.3 (disk characteristics), *Computer Organization & Design*, Patterson & Hennessy
- » Sec. 10.1-10.3, 10.6, *Operating System Concepts*, Silberschatz, Galvin, and Gagne. The rest of chs. 10-12 have much useful information if you have time to read them.

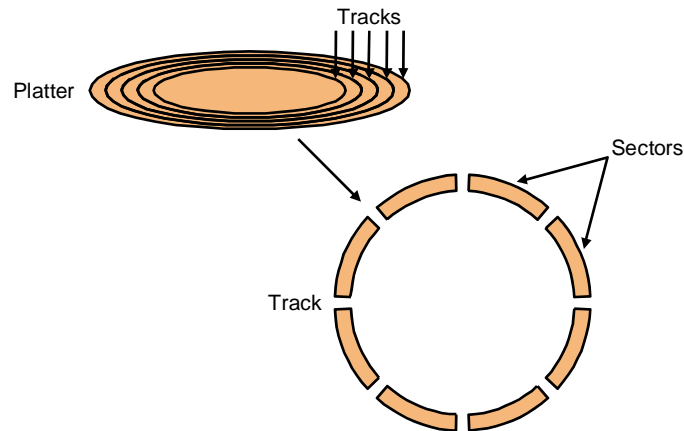
Hard drives

- The ugly guts of a hard disk.
 - » Data is stored on double-sided magnetic disks called **platters**.
 - » Each platter is arranged like a record, with many concentric **tracks**.
 - » Tracks are further divided into individual **sectors**, which are the basic unit of data transfer.
 - » Each surface has a read/write head like the arm on a record player, but all the heads are connected and move together.
- A 75GB IBM Deskstar has roughly:
 - » 5 platters (10 surfaces),
 - » 27,000 tracks per surface,
 - » 512 sectors per track, and
 - » 512 bytes per sector.



Accessing data on a hard disk

- Accessing a sector on a track on a hard disk takes a lot of time!
 - **Seek time** measures the delay for the disk head to reach the track.
 - » A **rotational delay** accounts for the time to get to the right sector.
 - » The **transfer time** is how long the actual data read or write takes.
 - » There may be additional **overhead** for the operating system or the controller hardware on the hard disk drive.
- **Rotational speed**, measured in revolutions per minute or RPM, partially determines the rotational delay and transfer time.



Estimating disk latencies (seek time)

- Manufacturers often report *average* seek times of 8-10ms.
 - » These times average the time to seek from any track to any other track.
- In practice, seek times are often much better.
 - » For example, if the head is already on or near the desired track, then seek time is much smaller. In other words, **locality** is important!
 - » Actual average seek times are often just 2-3ms.

Estimating Disk Latencies (rotational latency)

- Once the head is in place, we need to wait until the right sector is underneath the head.
 - » This may require as little as **no time** (reading consecutive sectors) or as much as **a full rotation** (just missed it).
 - » On **average**, for **random** reads/writes, we can assume that the disk spins halfway.
- Rotational delay depends partly on how fast the disk platters spin.

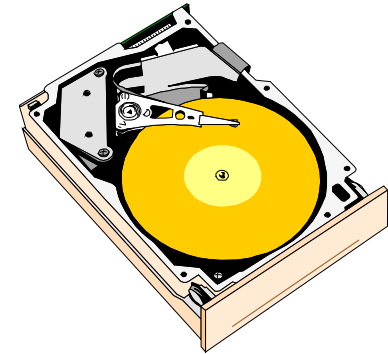
Average rotational delay = 0.5 x rotations x rotational speed

- » For example, a 5400 RPM disk has an average rotational delay of:

$$0.5 \text{ rotations} / (5400 \text{ rotations/minute}) = 5.55\text{ms}$$

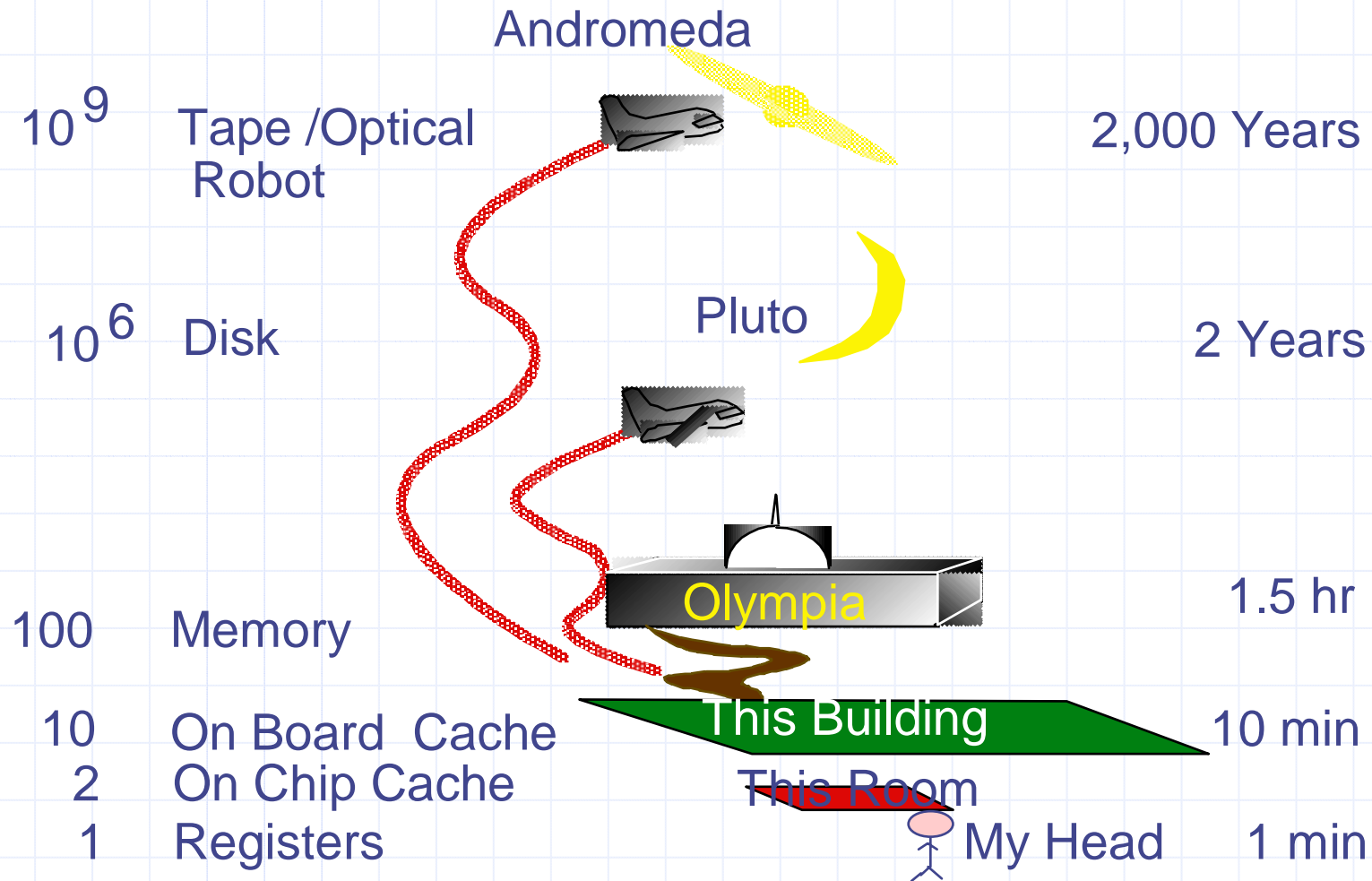
Estimating disk times

- The overall **response time** is the sum of the seek time, rotational delay, transfer time, and overhead.
- Assume a disk has the following specifications.
 - » An average seek time of 9ms
 - » A 5400 RPM rotational speed
 - » A 10MB/s average transfer rate
 - » 2ms of overheads
- How long does it take to read a random 1,024 byte sector?
 - » The average rotational delay is 5.55ms.
 - » The transfer time will be about $(1024 \text{ bytes} / 10 \text{ MB/s}) = 0.1\text{ms}$.
 - » The response time is then $9\text{ms} + 5.55\text{ms} + 0.1\text{ms} + 2\text{ms} = 16.7\text{ms}$. That's 16,700,000 cycles for a 1GHz processor!
- One possible measure of throughput would be the number of random sectors that can be read in one second.



$$(1 \text{ sector} / 16.7\text{ms}) \times (1000\text{ms} / 1\text{s}) = 60 \text{ sectors/second.}$$

Storage Latency: How Far Away is the Data?



File systems

- The concept of a file system is simple
 - » the implementation of the abstraction for secondary storage
 - abstraction = files
 - » logical organization of files into directories
 - the directory hierarchy
 - » sharing of data between processes, people and machines
 - access control, consistency, ...

Files

- A file is a collection of data with some properties
 - » contents, size, owner, last read/write time, protection ...
- Files may also have types
 - » understood by file system
 - device, directory, symbolic link
 - » understood by other parts of OS or by runtime libraries
 - executable, dll, source code, object code, text file, ...
- Type can be encoded in the file's name or contents
 - » windows encodes type in name
 - .com, .exe, .bat, .dll, .jpg, .mov, .mp3, ...
 - » old Mac OS stored the name of the creating program along with the file
 - » unix has a smattering of both
 - in content via magic numbers or initial characters (e.g., #!)

Basic operations

Unix

- create(name)
- open(name, mode)
- read(fd, buf, len)
- write(fd, buf, len)
- sync(fd)
- seek(fd, pos)
- close(fd)
- unlink(name)
- rename(old, new)

NT

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

File access methods

- Some file systems provide different **access methods** that specify ways the application will access data
 - » sequential access
 - read bytes one at a time, in order
 - » direct access
 - random access given a block/byte #
 - » record access
 - file is array of fixed- or variable-sized records
 - » indexed access
 - FS contains an index to a particular field of each record in a file
 - apps can find a file based on value in that record (similar to DB)
- Why do we care about distinguishing sequential from direct access?
 - » what might the FS do differently in these cases?

Directories

- Directories provide:
 - » a way for users to organize their files
 - » a convenient file name space for both users and FS's
- Most file systems support multi-level directories
 - » naming hierarchies (`/`, `/usr`, `/usr/local`, `/usr/local/bin`, ...)
- Most file systems support the notion of current directory
 - » absolute names: fully-qualified starting from root of FS
`bash$ cd /usr/local`
 - » relative names: specified with respect to current directory
`bash$ cd /usr/local` (absolute)
`bash$ cd bin` (relative, equivalent to `cd /usr/local/bin`)

Directory internals

- A directory is typically just a file that happens to contain special metadata
 - » directory = list of (name of file, file attributes)
 - » attributes include such things as:
 - size, protection, location on disk, creation time, access time, ...
 - » the directory list is usually unordered (effectively random)
 - when you type “ls”, the “ls” command sorts the results for you
 - » Key difference from ordinary files: system will not allow user process to write a directory with ordinary I/O calls, even if the user created/owns it. Why?

Path name translation

- Let's say you want to open “/one/two/three”

```
fd = open("/one/two/three", O_RDWR);
```
- What goes on inside the file system?
 - » open directory “/” (well known, can always find)
 - » search the directory for “one”, get location of “one”
 - » open directory “one”, search for “two”, get location of “two”
 - » open directory “two”, search for “three”, get loc. of “three”
 - » open file “three”
 - » (of course, permissions are checked at each step)
- FS spends lots of time walking down directory paths
 - » this is why open is separate from read/write (session state)
 - » OS will cache prefix lookups to enhance performance
 - /a/b, /a/bb, /a/bbb all share the “/a” prefix

Protection systems

- FS must implement some kind of protection system
 - » to control who can access a file (user)
 - » to control how they can access it (e.g., read, write, or exec)
- More generally:
 - » generalize files to **objects** (the “what”)
 - » generalize users to **principals** (the “who”, user or program)
 - » generalize read/write to **actions** (the “how”, or operations)
- A protection system dictates whether a given action performed by a given principal on a given object should be allowed
 - » e.g., you can read or write your files, but others cannot
 - » e.g., you can read `/etc/motd` but you cannot write to it

The original Unix file system

- Dennis Ritchie and Ken Thompson, Bell Labs, 1969
- “UNIX rose from the ashes of a multi-organizational effort in the early 1960s to develop a dependable timesharing operating system” -- Multics
- Designed for a “workgroup” sharing a single system
- Did its job exceedingly well
 - » Although it has been stretched in many directions and made ugly in the process
- A wonderful study in engineering tradeoffs



All Unix disks are divided into five parts

- Boot block
 - » can boot the system by loading from this block
- Superblock
 - » specifies boundaries of next 3 areas, and contains head of freelists of inodes and file blocks
- i-node area
 - » contains descriptors (i-nodes) for each file on the disk; all i-nodes are the same size; head of freelist is in the superblock
- File contents area
 - » fixed-size blocks; head of freelist is in the superblock
- Swap area
 - » holds processes that have been swapped out of memory

So ...

- You can attach a disk to a dead system ...
- Boot it up ...
- Find, create, and modify files ...
 - » because the superblock is at a fixed place, and it tells you where the i-node area and file contents area are
 - » by convention, the second i-node is the root directory of the volume

i-node format

- User number
- Group number
- Protection bits
- Times (file last read, file last written, inode last written)
- File code: specifies if the i-node represents a directory, an ordinary user file, or a “special file” (typically an I/O device)
- Size: length of file in bytes
- Block list: locates contents of file (in the file contents area)
 - » more on this soon!
- Link count: number of directories referencing this i-node

The flat (i-node) file system

- Each file is known by a number, which is the number of the i-node
 - » seriously – 1, 2, 3, etc.!
 - » why is it called “flat”?
- Files are created empty, and grow when extended through writes

The tree (directory, hierarchical) file system

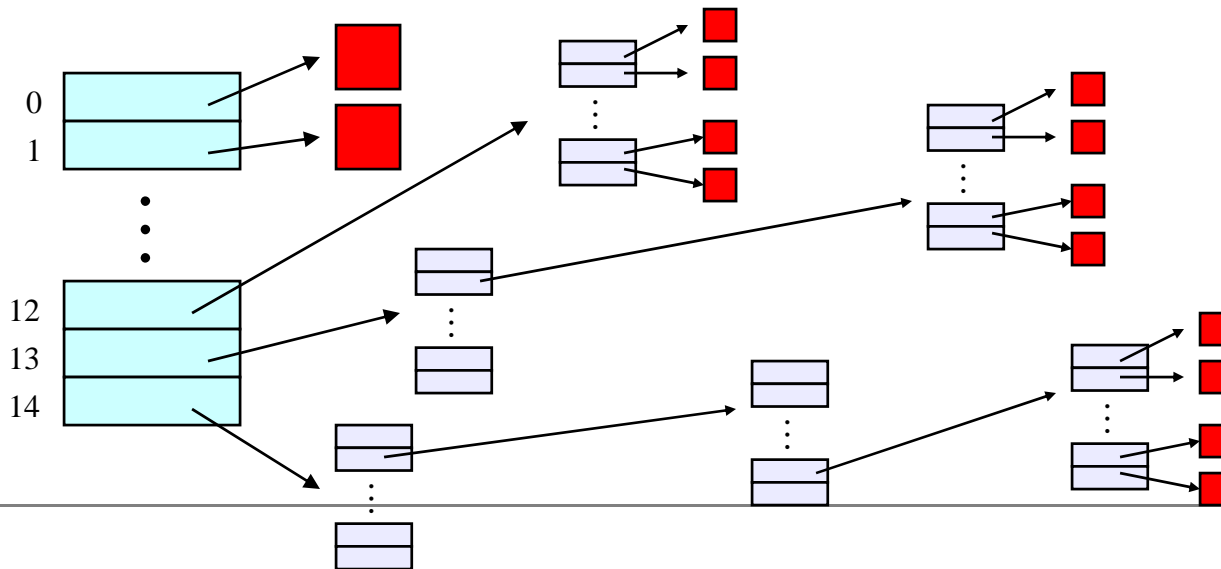
- A directory is a flat file of fixed-size entries
- Each entry consists of an i-node number and a file name

i-node number	File name
152	.
18	..
216	my_file
4	another_file
93	oh_my_god
144	a_directory

- It's as simple as that!

The “block list” portion of the i-node

- Clearly it points to blocks in the file contents area
- Must be able to represent very small and very large files.
How?
- Each inode contains 15 block pointers
 - » first 12 are direct blocks (i.e., 4KB blocks of file data)
 - » then, single, double, and triple indirect indexes



So ...

- Only occupies 15 x 4B in the i-node
- Can get to 12 x 4KB = a 48KB file directly
 - » (12 direct pointers, blocks in the file contents area are 4KB)
- Can get to 1024 x 4KB = an additional 4MB with a single indirect reference
 - » (the 13th pointer in the i-node gets you to a 4KB block in the file contents area that contains 1K 4B pointers to blocks holding file data)
- Can get to 1024 x 1024 x 4KB = an additional 4GB with a double indirect reference
 - » (the 14th pointer in the i-node gets you to a 4KB block in the file contents area that contains 1K 4B pointers to 4KB blocks in the file contents area that contain 1K 4B pointers to blocks holding file data)
- Maximum file size is 4TB

File system consistency

- Both i-nodes and file blocks are cached in memory
- The “sync” command forces memory-resident disk information to be written to disk
 - » system does a sync every few seconds
- A crash or power failure between sync’s can leave an inconsistent disk
- You could reduce the frequency of problems by reducing caching, but performance would suffer big-time

i-check: consistency of the flat file system

- Is each block on exactly one list?
 - » create a bit vector with as many entries as there are blocks
 - » follow the free list and each i-node block list
 - » when a block is encountered, examine its bit
 - If the bit was 0, set it to 1
 - if the bit was already 1
 - if the block is both in a file and on the free list, remove it from the free list and cross your fingers
 - if the block is in two files, call support!
 - » if there are any 0's left at the end, put those blocks on the free list

d-check: consistency of the directory file system

- Do the directories form a tree?
- Does the link count of each file equal the number of directories links to it?
 - » I will spare you the details
 - uses a zero-initialized vector of counters, one per i-node
 - walk the tree, then visit every i-node

Protection

- **Objects:** individual files
- **Principals:** owner/group/world
- **Actions:** read/write/execute

- This is pretty simple and rigid, but it has proven to be about what we can handle!