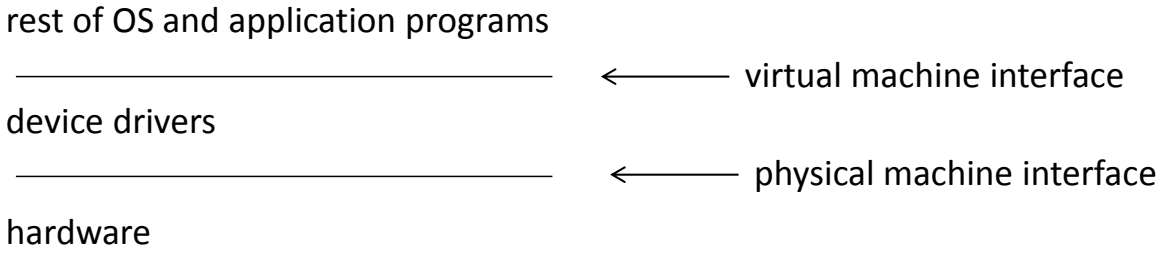
I/O and file systems

- Abstractions provided by operating system for storage devices
  - Heterogeneous -> uniform
  - One/few storage objects (disks) -> many storage objects (files)
  - Simple naming -> rich naming
    - Numeric -> symbolic
    - Flat -> structured
    - Separate -> unified
  - Fixed block assignment -> flexible block assignment
  - Slow -> fast
  - Inconsistency on crashes -> consistency

Dealing with device heterogeneity

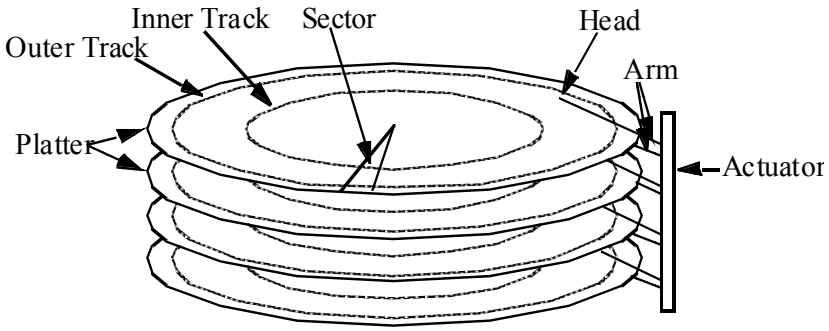
- Problem: different types of disks and disk interfaces. How to manage this diversity?
- Solution: add device-driver abstraction inside OS



  - Hide differences among different brands and interfaces
  - Minimize differences between similar types of devices

I/O device access

- Disk geometry



- Accessing an I/O device
  - Queueing time (wait for device to be free)
  - Overhead
  - Transfer data: size / (transfer rate)

Optimizing I/O performance

- I/O devices are slow. To increase performance:
  - Avoid doing I/O
  - Reduce overhead
  - Amortize overhead over larger request

- Efficiency = transfer time / (positioning time + transfer time)

Disk scheduling

- Reduce overhead by reordering requests
  - FCFS (first come, first served)
  - SSTF (shortest seek time first)
  - SCAN (sort requests by position)

- Does CPU scheduling policy affect throughput?
- Does disk scheduling policy affect throughput?

- How else could OS reduce overhead?

Flash RAM

- Optimizations depend on the specifics of a device
- Flash RAM has different characteristics than magnetic disk
  - Better read performance (but still slow writes)
  - Lower positioning overhead
  - Lower power
  - Better shock resistance

  - But also has some issues: wearout, no overwrite

- OS hides physical characteristics of device from applications

File systems

- What's a file system?
  - A data structure stored on a persistent medium
  - Data persists across what type of events?
  - How to enable data to persist across these events?
- Interface to the file system
  - Create file
  - Delete file
  - Read <file, offset>
  - Write <file, offset>
  - Other

File system workloads

- Optimize data structure for the common case
- Some general rules of thumb
  - Most file accesses are reads
  - Most programs access files sequentially and entirely
  - Most files are small, but most bytes belong to large files

File abstraction

- One (or a few) storage objects (e.g., disks) -> many storage objects (files)
  - How to name files
  - How to find and organize files

How to store a single file

- This is a data structure question
  - Pointers needed for indirection
    - But can't use memory pointers
  - Also need to store metadata (data about the file)
    - File size
    - Owner
    - Access permissions
    - Time of creation, last access, etc.
- File header (inode)
  - Initial structure that describes the file and allows you to find the data

Contiguous allocation

- File = array of blocks ("extent")
- Reserve space in advance. If file grows, move it to a larger free area.

- File header
  - Starting location of the file
  - File size
  - Owner, permissions, etc.

- Pros and cons
  - Fast sequential access
  - Easy random access
  - But difficult to grow file; external fragmentation (or wastes space).

Indexed files

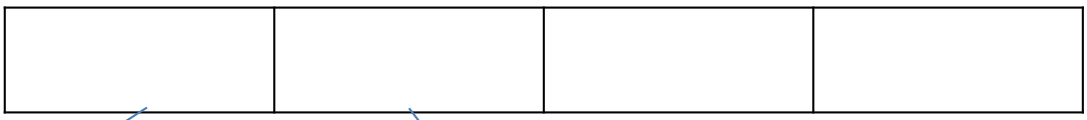
- File = array of block pointers
  - Just like page table

File block #	Disk block #
0	18
1	50
2	8
3	15

- Pros and cons
  - Easy to grow file
  - Easy random access
  - But potentially slow for sequential access
- How to allow large files?
  - Large page table
  - Large block size

Multi-level indexed files

- File = tree of block pointers



level 1 node

level 2 node

File block #	Disk block #
0	18
1	50
2	8
3	15

File block #	Disk block #
4	20
5	11
6	3
7	43

- Allows large file without wasting header space for small files
- But potentially many accesses to read a file block

Naming and directories

- How to specify file to be accessed?
  - Start with file name, or click on icon, or describe contents
  - Eventually map from file name to the disk location of that file's header
- File name is usually hierarchical
  - E.g., /home/pmchen/482/notes
  - Allows users to group related files into one directory/folder
  - Allows easy searching, e.g., "ls /home/pmchen/482"

- Must translate from file name to disk block # of the file header
  - Another data structure. Examples:
  - Call this data structure a "directory"
  - Like files, the directory data structure must be persistent

Directories

- A directory contains the mapping information for a set of files
  - Name of file -> file header's disk block # for that file
  - Often a simple array of (name, file header's disk block #) entries
- Directories are stored on disk
- We can often deal directories and files in the same way
  - Same storage structure
  - Directory entry can point to file or directory
- Can we allow an application to read/write a directory, just as an application can read/write a file?

Tree of directories

- Connect directories into a tree of directories
  - Natural match for hierarchical naming structure
- To build a tree, we need
  - Root node (/)
  - Pointers between nodes. Directory stores disk block # of header for files and directories

Example: /home/pmchen/482/notes

- / is root directory
  - Contains list of the files and directories in /. For each entry, contains mapping from name -> header's disk block #
  - One of those entries is "home"
- /home is a directory within the / directory
  - Contains list of the files and directories in /home
  - One of those entries is "pmchen"
- /home/pmchen is a directory within the /home directory
  - Contains list of files and directories in /home/pmchen
  - One of those entries is "482"
- /home/pmchen/482 is a directory within the /home/pmchen directory
  - Contains list of files and directories in /home/pmchen/482
  - One of those entries is notes
- /home/pmchen/482/notes is a file within the /home/pmchen/482 directory

How many disk I/Os to read /home/pmchen/482/notes?

- Improving performance through caching (but not in Project 4)

Unified view of multiple storage devices

- Combine (mount) multiple storage devices into one file system
- Each storage device contains its own file system (starting with its root)
- An entry in a directory can point to the root of a different storage device
- E.g., loginlinux.engin.umich.edu

  / (root)

    bin (same device as /)

    etc (same device as /)

    tmp (separate storage device)

    afs (network storage "device")
- Directory now can map name to:
  - File
  - Directory
  - Device

File caching

- File systems store lots of data structures on disk
  - Data blocks
  - Directories
  - File headers (inodes)
  - Indirect blocks
  - Free lists
- Caching is main way to improve performance
  - Memory throughput is higher than sequential I/O
  - Response time can be faster by many orders of magnitude
- Is file cache in virtual memory or physical memory?

Comparing file caching and virtual memory

- Both use physical memory as a cache for disk
  - VM: started with memory, then added disk for increased capacity
  - File systems: started with disk, then added memory for faster performance
- But why have two mechanisms that both cache disk data in memory?

- Memory-mapped files
  - Use the VM paging system to cache both virtual address space **and** disk
  - Map file into a virtual address space, then point the backing store for that part of the address space at the file's data blocks
  - VM knows how to cache address spaces. File cache knows how to cache files. Memory-mapped files makes files look like part of the address space
  - Example: how to load a program executable from disk to memory?

File cache design

- Normal design issues for caches, e.g., cache size, block size, replacement policy, etc.
- Should the file cache use write back or write through?

Multiple updates and reliability

- Reliability (durability) is especially important for file systems
  - Data in a process's address space need not survive system crashes or power outage
  - Data in file system must survive system crashes and power outage (and device failure?)
- Multi-step updates cause problems if crash happens in the middle
- E.g., transfer $100 from Peter's account to Janet's account
  1. Deduct $100 from Peter
  2. Add $100 to Janet
- E.g., move file to new directory
  1. Delete file from old directory
  2. Add file to new directory
- E.g., create new (empty) file
  1. Update directory to point to new file header
  2. Write new file header to disk
- What happens if you crash between steps 1 and 2?

Careful ordering

- How to fix problem in prior example (file creation)?
- Let's say I also want to update the list of free disk blocks. How do I do this?
- Can careful ordering solve the problem of transferring money from Peter's account to Janet's account?
  - What EECS 482 concept does this remind you of?

Transactions

- Commonly used in databases and file systems. Main aspect for file systems is atomicity/durability (all or nothing)

```
begin
    write disk
    write disk
    write disk
end (this "commits" the transaction)
```

- Basic atomic unit provided by hardware is writing a single sector to disk
- How to make an arbitrary sequence of updates atomic, using single-sector update?

Implementing transactions with shadowing

- Keep two versions of file system (old and new), and store a persistent pointer to the current version
- Write updates to the new version, then switch the pointer to the new version when you want to commit the changes
- Indirection shrinks the size of the write, so it fits in a single sector

- Principle: a series of changes can be committed with a single-sector write
- Optimizations
  - Don't need to copy entire file system
  - Sector can store more than just a 1-bit pointer
  - E.g., rename /home/pmchen/482/f13/notes to /home/pmchen/482/f14/notes
  - What needs to be written? Remember that each file/directory has a header, which points to the data block(s) for that file/directory

Implementing transactions with logging

- Write-ahead logging
  - Write new data to append-only log
  - Write commit sector to end of log to commit the set of changes
- Eventually, copy new data from log to the in-place version of the file system

- What if system crashes after writing log records and commit, but before copying the changes to the in-place version of the file system?

- Most recent file systems use transactions (via logging) to make atomic a group of updates to the file system metadata (but not necessarily the data). This is called "journalling".

Case study: log-structured file system

- Goal: make (almost) all I/Os sequential
  - In general, not possible for reads
  - Maybe possible for writes. File system can write to any free disk block.
- Basic idea: treat disk as an append-only log
  - Append data to log
  - Append new inode (which points to new data) to log

- Is writing sequentially to disk enough to achieve good performance?

Recursive update

- After writing the new inode, what else must be updated?

- Solution: store constant “inode number”, rather than disk block # of inode

How to find inode location?

- Another data structure: the inode map
  - Must update the inode map when you write a new version of the inode
  - Solves recursive update problem
  - But where to write the inode map?

- Have we made progress?

What if inode map is large?

- Could append inode map to log

- Could divide inode map into several pieces, and write updated portions to the log

- Other data structures needed:

What happens when you run out of log space?

- Defragment the log
  - All free space at end of log?

- Treat log as set of large chunks of disk space; don't worry about making all I/O sequential