File System Interface

Files and Directories

features

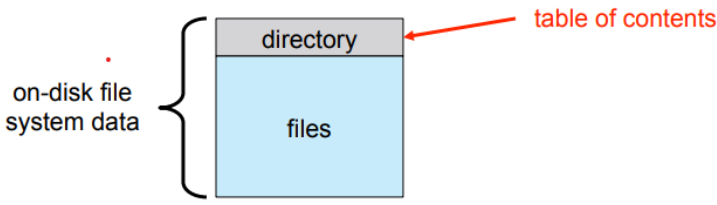
- A file system implements the file abstraction for secondary storage
- It also implements the directory abstraction to organize files logically

Usage

- It is used for users to organize their data
- It is used to permit data sharing among processes and users
- It provides mechanisms for protection

File systems

- component of OS that knows how to do file stuff
- set of algorithms and techniques
- content on disk that describes a set of files
- Remember that a disk can be partitioned arbitrarily into logically independent partitions
- Each partition can contain a file system
- One can have multiple disks, each with arbitrary partitions, each with a different file system on it



File and File Type

- file is data + properties (attributes)
  Content, size, owner, last read/write time, protection, etc.
- file can also have a type
  Understood by the File System or the OS

File Structure

should the OS know about the structure of a file?

- more different structures the OS knows about the more "help" it can provide applications that use particular file types
- the more complicated the OS code is

Modern OSes support very few files structures

- Files are sequences of bytes that the OS doesn't know about but that have meaning to the applications
- Certain files are executables and must have a specific format that the OS knows about
- OS may expect a certain directory structure defining an application

Internal File Structure

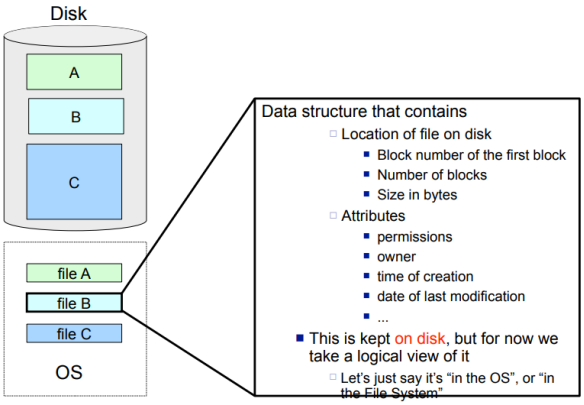
- disk provides the OS with a block abstraction
  ∀ disk I/O is performed in number of blocks
- Each file is stored in a number of blocks

File Operations

- file is an abstraction i.e. abstract data type
- OS defines several file operations
- basic operations include
  - creating
  - writing or reading
    current-file-position pointer is kept per process
  - updated after each write/read operation
  - Re positioning the current-file-position pointer
    called a seek
  - Appending at the end of a file
  - Truncating
  - deleting
  - renaming

Open File

- OS requires that processes open and close files
- After an open, the OS copies the file system's file entry into an open-file table that is kept in RAM in the kernel
- OS keeps two kinds of open-file tables
  one table per process and one global table for all processes
- A process specifies which file the operation is on by giving an index in its local table
- The OS keeps track of a "reference count" for each open file in the global table
  + each time a process open the file
  - each time a process closes the file



File Locking

- file lock can be acquired for a full file or for a portion of a file
- The OS may require mandatory locking for some files
- applications have to implement their own locking
- courses there can be deadlocks and all the messiness of thread synchronization

in Java

```
import java.io.*;
import java.nio.channels.*;
public class LockingExample {
public static final boolean EXCLUSIVE = false;
public static final boolean SHARED = true;
public static void main(String arsg[]) throws IOException {
FileLock sharedLock = null;
FileLock exclusiveLock = null;
try {
 RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");
 FileChannel ch = raf.getChannel();
 // this locks the first half of the file - exclusive (one writer)
 exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);
 /** Now modify the data . . . **/
 // release the lock
 exclusiveLock.release();
 // this locks the second half of the file - shared (multiple readers)
```

```
 sharedLock = ch.lock(raf.length()/2+1, raf.length(), SHARED);
 /** Now read the data . . . **/
 // release the lock
 sharedLock.release();
} catch (java.io.IOException ioe) {
 System.err.println(ioe);
} finally {
 if (exclusiveLock != null)
 exclusiveLock.release();
 if (sharedLock != null)
 sharedLock.release();
}
}
}
```

Access Methods

Sequential Access
- One byte at a time, in order, until the end
- Read next, write next, reset to the beginning

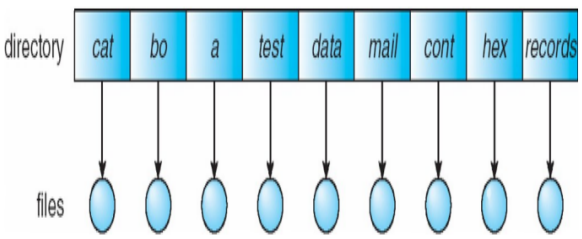
Direct Access
- Ability to position anywhere in the file
- Position to block #n, Read next, write next
- Block number is relative to the beginning of the file

Indexed Access
- A file contains an index of "file record" locations
- One can then look for the object in the index, and then "jump" directly to the beginning of the record

Directories

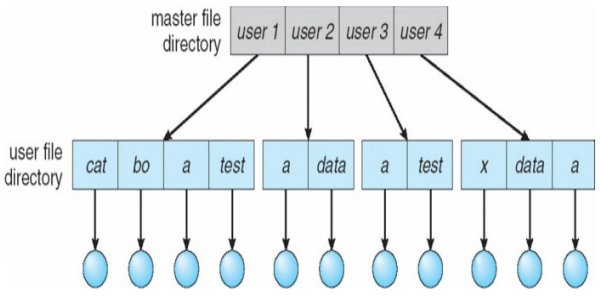
used to file systems that support :
- multiple lvl directories
- notion of current directory

Single Level Directory



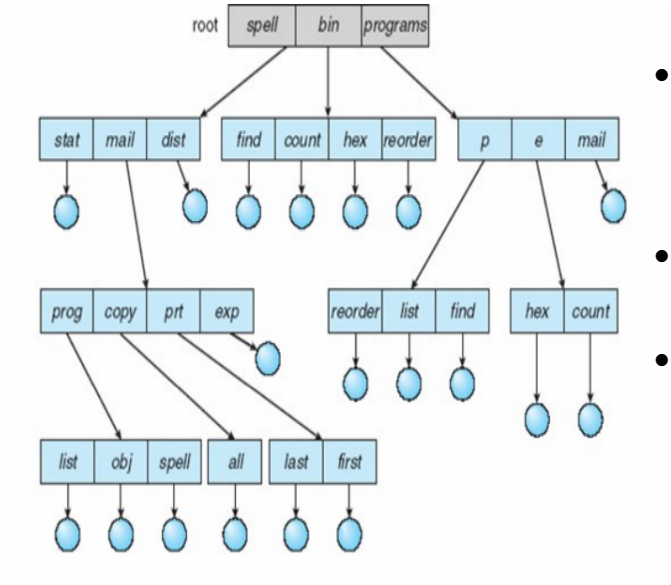
- naming conflicts
- slow searching

Two Lvl



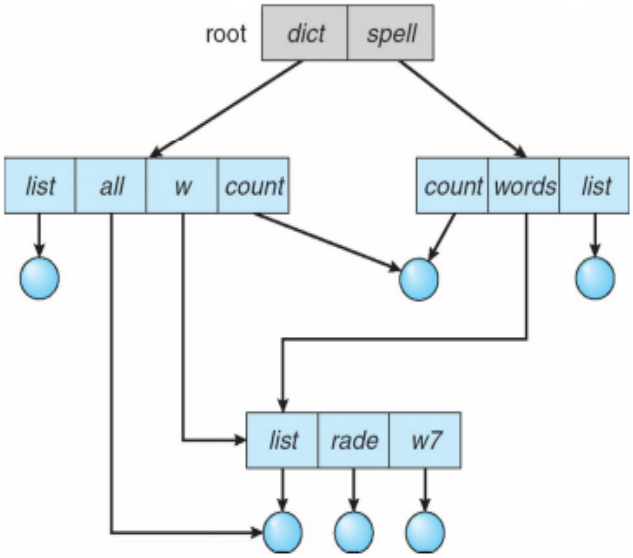
- faster searching
- still conflicts exists for each user

Tree Structured Directories



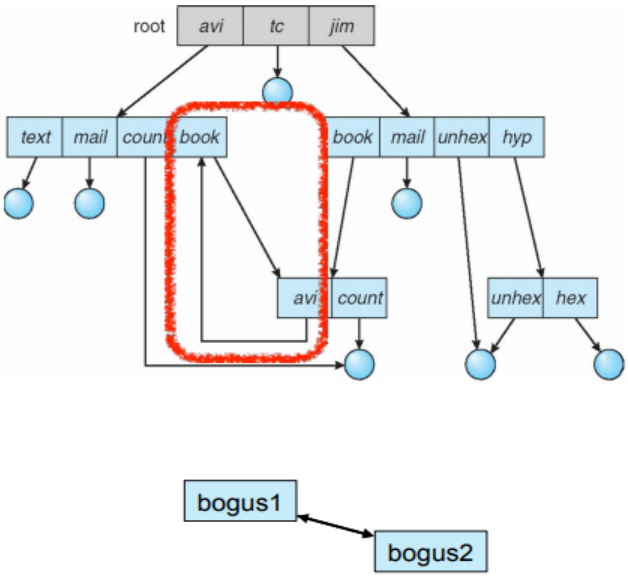
- More general than 1- and 2-level schemes
- Each directory can contain files or directories
  Differentiated internally by a bit set to 0 for files and 1 for directories
- Each process has a current directory
  relative paths and absolute paths
- Path name translation ex. "/one/two/three"
  - Open "/" (the file system knows where that is on disk)
  - Search it for "one" and get its location
  - Open "one", search it for "two" and get its location
  - Open "two", search it for "three" and get its location
  - Open "three"
- OS spends a lot of time walking directory paths
  Another reason why one separates "open" from "read/write"
  The OS attempts to cache "common" path prefixs

Acyclic Graph Directories



- Files/directories can be shared by directories
- hard link created in a directory, to point to or reference another file or directory
  Identified in the file system as a special file
- file system keeps track of reference count for each file, and deletes the file when the last reference is removed
- symbolic link does not toward the reference count
  think of it as an alias for the file & if the target file is removed then the alias simply becomes invalid
- Acyclic is good for quick/simple traversals
- Simple way to prohibit cycles: no hard linking of directories

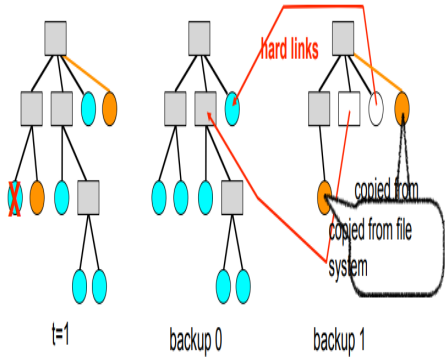
General Graph Directory

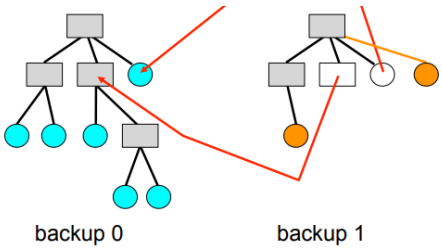


- users can do whatever they want
- Directory traversals algorithms must be smarter to avoid infinite loops
- Garbage collection could be useful because ref counts may never reach zero

Mac OSX Time Machine

- Time Machine is the backup mechanism introduced with Leopard
- uses hard links
  - Every time a new backup is made, a new backup directory is created that contains a snapshot of the current state of the file system
  - Files that haven't been modified are hard links to previously backed up version
    A new backup should be mostly hard links instead of file copies(space saving)
  - When an old backup directory is wiped out, then whatever files have a reference count of zero are removed (no longer part of more recent data)





Advantages

- Extremely simple to implement
- The back up can be navigated in all the normal ways, without Time Machine
- Provided backups are frequent, they are done by creating mostly hard links

Drawbacks

- If you change 1 byte in a 10GB file, then you copy the whole 10GB
- For efficiency, Mac OSX allows hard linking of directories
  - Cycles in the directory hierarchy must be detected, i.e., more complicated file system code
  - Complexity deemed worthwhile by Mac OSX developers

File System Mounting

- There can be multiple file systems
- Each file system is "mounted" at a special location, the mount point
  - Typically seen as an empty directory
- When given a mount point, a volume, a file system type, the OS
  - asks the device driver to read the device's directory
  - checks that the volume does contain a valid file system
  - makes note of the new file system at the specified mount point
- Mac OS X: all volumes are mounted in the /Volumes/ directory
- UNIX: volumes can be mounted anywhere
- Windows: volumes were identified with a letter (e.g., A:, C:), but current versions, like UNIX, allow mounting anywhere
- On Linux the "mount" command lists all mounted volumes

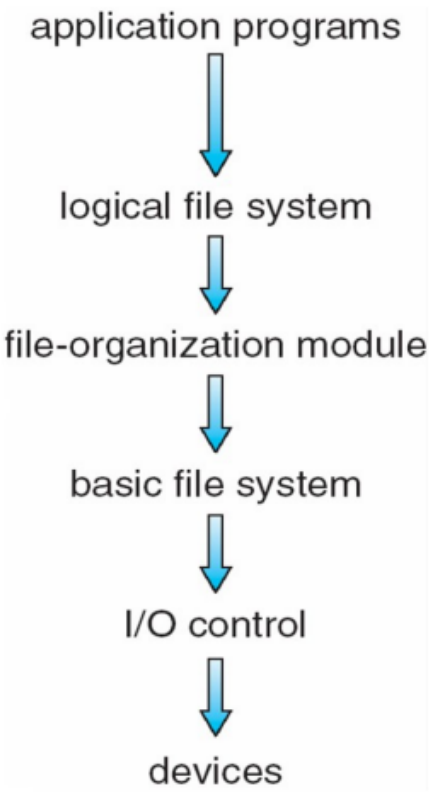
Protection

- file systems provide controlled access
- General approach: Access Control Lists (ACLs)
  - For each file/directory, keep a list of all users and of all allowed accesses for each user - Protection violations are raised upon invalid access
- Problem: ACLs can be very large
  → consider only a few groups of users and only a few possible actions
- UNIX User, Group, Others not in Group, All (ugoa)
  - Read, Write, Execute (rwx)
  - Represented by a few bits

File System 2

File System Implementation

- file system should provide an efficient implementation of the interface it defines
  - storing
  -locating
  - retrieving data
- **The problem: define data structures and algorithms to map the logical FS onto the disk**
- Typical layer organization
  - Good for modularity and code re-use
  - Bad for overhead
- Some layers are hardware, some are software, some are a combination
- **I/O control**
  - Device drivers and interrupt handlers
  - Input from above
  Read physical block #43
  Write physical block #124
  - Output below
  Writes into device controller's memory to enact disk reads and writes; React to relevant interrupts
- **Basic file system**
  - Allocates/maintains various buffers that contain file-system, directory, and data blocks
  - These buffers are caches and are used for enhancing performance
- **File-organization module**
  - Knows about logical file blocks (from 0 to N) and corresponding physical file blocks: it performs translation
  - It also manages free space



- **Logical file system**
  - Keep all the meta-data necessary for the file system
  - It stores the directory structure
  - It stores a data structure that stores the file description **(File Control Block - FCB)**
    - Name, ownership, permissions
    - Reference count, time stamps, pointers to other FBCs
    - Pointers to data blocks on disk

File System Data Structures

file system comprises data structures

On-disk structures

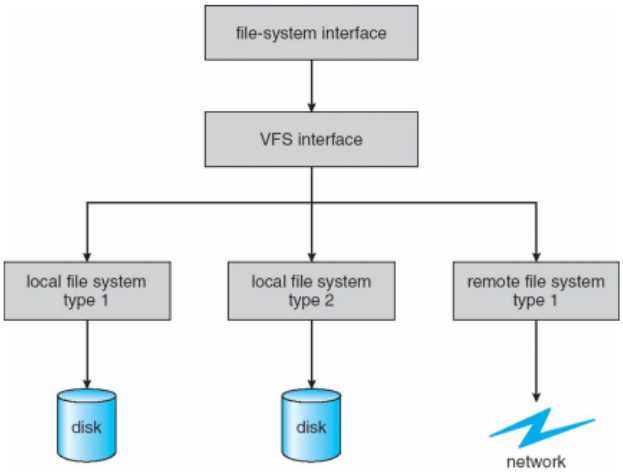
- An optional boot control block
  - First block of a volume that stores an OS
  - boot block in UFS, partition boot sector in NTFS
- volume control block
  - Contains the number of blocks in the volume, block size, free-block count, free-block pointers, free-FCB count, FCB-pointers
  - superblock in UFS, master file table in NTFS
- directory
  - File names associated with an ID, FCB pointers
- per-file FCB
  - In NTFS, the FCB is a row in a relational database

In-memory structures

- mount table
- directory cache
- global open-file table
- per-process open-file table
- Various buffers holding disk blocks "in transit" (performance)

Virtual File System

This is simply about software engineering (modularity and code-reuse)



Directory Implementation

Linear List

- Simply maintain an on-disk doubly-linked list of names and pointers to FCB structures
- The list can be kept sorted according to file name
- Upon deletion of a file, the corresponding entry can be recycled
- **linear search is slow**

Hash Table

- A bit more complex to maintain
- Faster searches

Allocation Methods

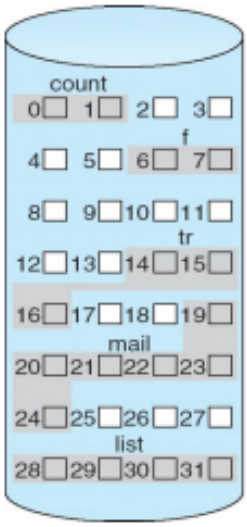
How do we allocate disk blocks to files?

→ **Contiguous Allocation**

- Each file is in a set of contiguous blocks
  Good because sequential access causes little disk head movement, and thus short seek times
- directory keeps track of each file as the address of its first block and of its length in blocks

Problems

- Can be difficult to find free space
  Best Fit, First Fit, etc.
- External fragmentation
  With a big disk, perhaps we don't care
- Difficult to have files grow

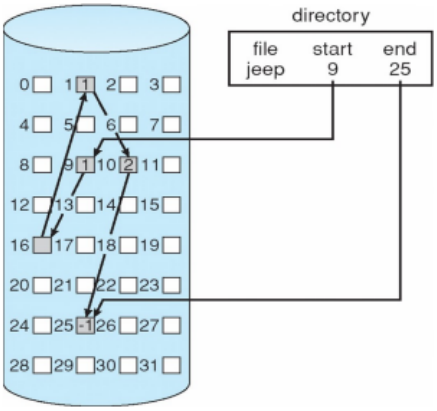


directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Linked Allocation

- A file is a linked-list of disk blocks
- The directory points to the first and last block of the files
- Solves all the problems of contiguous allocation
  - no fragmentation and files can grow



Linked Allocation Problems

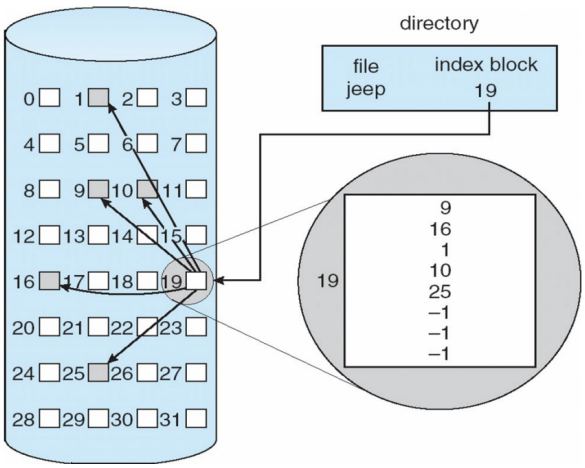
- Great for sequential access but not so much for direct access
  - A direct access requires quite a bit of pointer jumping
- Wastes space
  - Easy to solve by coalescing blocks together, i.e., allowing for bigger blocks But at the cost of larger internal fragmentation
- Poor reliability
  - If a pointer is lost or damaged, then the file is unrecoverable
  - Can be fixed with a double-linked list, but increases overhead

File-Allocation Table (FAT) System

- scheme implements block linking with a separate table that keeps track of all links
- Finding a free block is simple: just find the first 0 entry in the table
- FAT can be cached in memory to avoid disk seeks altogether

Indexed Allocation

- All block pointers are brought to a single location: the **index block**
- one index block per file
- directory contains the address of the index block
  i-th entry points to the i-th block
- directory contains the address of index block
- similar to paging, and **same advantage: easy direct access**
- but same problem: how big is the index block
  - Not good to use all our space for storing index information!
  - Especially if many entries are nil
  - What if the index is bigger than a block?



Linked Index

- To allow for an index block to span multiple disk blocks, we just create a linked list of disk blocks that contain pieces of the full index
- adds complexity, but can accommodate any file size
- **disk space is not as costly as RAM space, and that the disk is very slow**
- ∴ trading off space for performance and allowing for large indices is likely a much better trade off than it would be for RAM

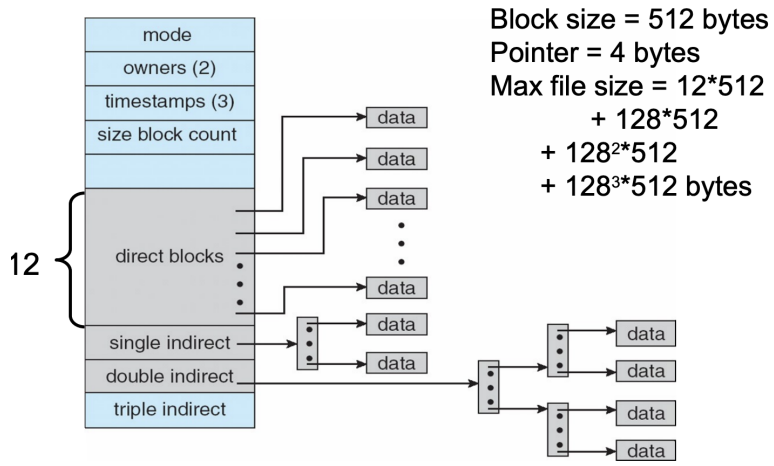
Multilevel index

- like a hierarchical page table
- If we have 512-byte blocks, and 4-byte pointers, then we could store 128 entries in a block
- A 3-level scheme can then allow for 1GB files
- A 4-level scheme can then allow for 128GB files

Combined Index

- a small file it seems a waste to keep a large index
- a medium-sized file it seems a waste to keep multi-level indices
- How about keeping all options open:
  - A few pointers to actual disk blocks
  - A pointer to a single-level index
  - A pointer to a two-level index
  - A pointer to a three-level index

inode



In Class Exercise
Disk blocks are 8KiB, a block pointer is 4 bytes
What is the maximum file size with the i-node structure?

Inodes and Directories

- inode can describe a file or a directory
- inode for a directory also points to data blocks
- these data blocks happen to contain <name, pointer to inode> pairs
- data blocks are searched for names when doing pathname resolution
- system keeps an in-memory cache of recent pathname resolutions

Free Space Management

How do we keep track of free blocks?
→ Bitmap

- Keep an array of bits, one bit per disk block
- 1 means free, 0 means not free
- Good
  - simple and Easy to find a free block
- Bad
  - bitmap can get huge so it may not be fully cachable in memory

Linked List

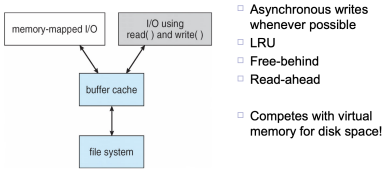
- Maintain a chain of free blocks, keeping a pointer to the first block
- Traversing the list could take time, but we rarely need to do it
- Remember that FAT deals with free blocks in the data structure that keeps the "linked-list" of non-free blocks

Counting

- Simply keep the address of a free block and the number of free blocks immediately after it
- saves space
  - entries are longer but we have fewer of them
- These entries can be stored in an efficient data structure so that chunks of contiguous free space can be identified

Efficiency and Performance

- many efficiency and performance issues for file systems
- Each aspect of the design impacts performance, hence many clever implementation tricks
- One well-known example: inode allocation
  - inodes are pre-allocated
  - inodes are spread all over the disk
- Caching of disk blocks to take advantage of temporal locality



Consistency Checking

- File System should not lose data or become inconsistent
  - a fragile affair, with data structure pointers all over the place, with parts of it cached in memory
- abrupt shutdown can leave an inconsistent state
- Consistency can be checked by scanning all the metadata
- **Bottom line: We allow the system to be corrupted, and we later attempt repair**
- **Problems with consistency checking**
  - **Some data structure damaged may not be repairable**
  - **Human intervention is needed to repair the data structure**
  - **Checking a large file system takes forever**

Log-based transaction-oriented FS (Journaling)

- Whenever the file system metadata needs to be modified, the sequence of actions to perform is written to a circular log and all actions are marked as "pending"
- Then the system proceeds with the actions asynchronously
  - Marking them as completed along the way
- Once all actions in a transaction are completed, the transaction is "committed"
- If the system crashes, we know all the pending actions in all noncommitted transactions, so we can undo all committed actions

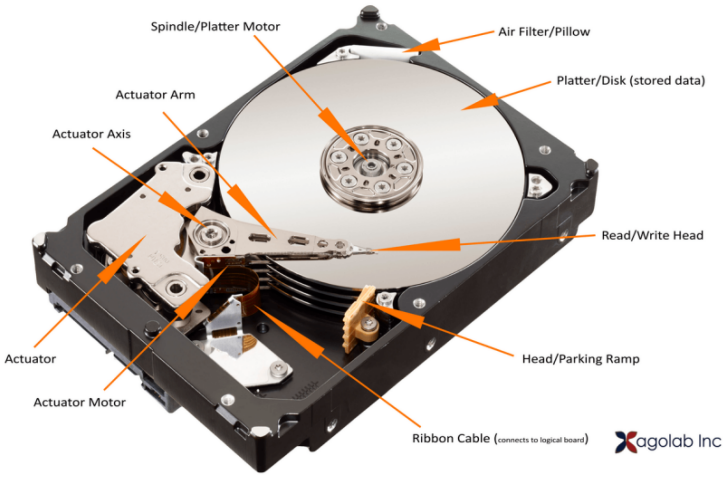
Lesson Summary

- File Systems can be seen as part of or outside the OS
- File Systems are a complex and active topic

Mass Storage

Magnetic Disks

- (still) the most common secondary storage devices today
- they are messy - Errors, bad blocks, missed seeks, moving parts
- yet, the data they hold is critical
- OS used to hide all the "messiness" from higher-level software
- This has been done increasingly with help from the hardware



Hard Drive Data Access

- needs several pieces of information for each data access
  - platter, track, sector etc.
- Hard drives today are more complicated than the simple picture
- Nowadays, hard drives comply with standard interfaces
  - EIDE, ATA, SATA, USB, Fiber Channel, SCSI
- hard drives, in these interfaces, is seen as an array of logical blocks (512 bytes)
- e device, in hardware, does the translation between the block and platter, sector, track, etc.
- good because
  - kernel code to access the disk is straightforward
  - controller can do a lot of work, e.g., transparently hiding bad blocks

Hard drive performance

- Data request performance depends on three steps
  1. **seek** moving the disk arm to the correct cylinder
     Depends on how fast disk arm can move (increasing very slowly over the years)
  2. **rotation** waiting for the sector to rotate under the head
     Depends on rotation rate of disk (increasing slowly over the years)
  3. **transfer** transferring data from surface into disk controller
     Depends on density
- **must minimize seek and rotation time**

Disk Scheduling

- simplest policy FIFO

Shortest Seek Time First (SSTF)

- Select the request that's the closest to the current
- may cause starvation
  If there is a constant stream of requests for the current track, the other tracks will not be visited
- one sol'n is elevator algorithm
  - head just goes through the tracks back and forth and serves requests as they come
  - Requests that come it for the current track may have to wait until the "elevator" comes back
  - But this is really not good to minimize request service time

SATF: Shortest Access Time First

- Account for seek time and rotation time to pick the next request
- All these algorithms are implemented in the disk controller

Redundant Array of Independent Disks (RAID)

- Disks are unreliable, slow, and cheap
- use redundancy
  - Increases reliability
  If one fails, you have another one (increased perceived MTTF)
  - Increases speed
  Aggregate disk bandwidth if data is split across disks
- The OS can implement it with multiple bus-attached disks
- intelligent RAID controller in hardware
- RAID array" as a stand-alone box

RAID techniques

Data Mirroring	Data Striping	Parity Bits
- Keep the same data on multiple disks - Every write is to each mirror, which takes time	- Keep data split across multiple disks to allow parallel reads **ex** read bits of a byte from 8 disks	- Keep information from which to reconstruct lost bits due to a drive failing

These techniques are combined at will

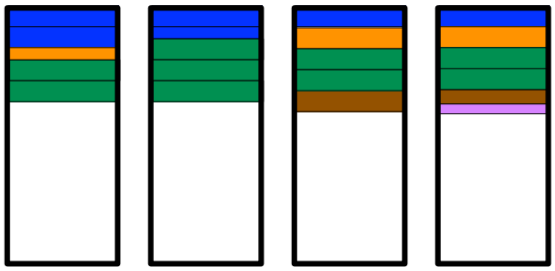
RAID levels

Combinations of the techniques are called "levels"

RAID 0

EX

- Data is striped across multiple disks
- Gives the illusion of a larger disk with high bandwidth when reading/writing a file
- Improves performance, but not reliability
- Useful for high-performance applications

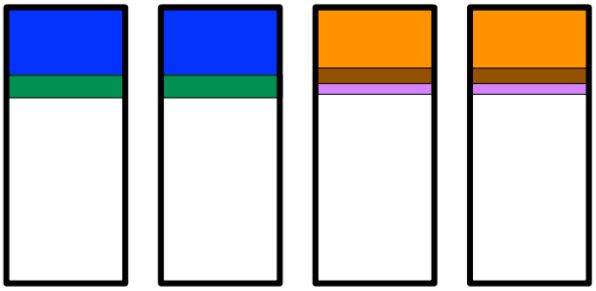


- Fixed strip size
- 5 files of various sizes
- 4 disks

RAID 1

EX

- Mirroring
- Write every written byte to 2 disks
- Reliability is ensured unless you have (extremely unlikely) simultaneous failures
- Performance can be boosted by reading from the disk with the fastest seek time



- 5 files of various sizes
- 4 disks

RAID 3

- Bit-interleaved parity
  - Each write goes to all disks, with each disk storing one bit
  - A parity bit is computed, stored, and used for data recovery
- XOR overhead for each write (done in hardware)
- Time to recovery is long (a bunch of XOR's)

OS is responsible for
-Formatting the disk
- Booting from disk
- Bad-block recovery

RAID 4 and 5

- RAID 4
  Basically like RAID 3, but interleaving it with strip
- RAID 5
  Like RAID 4, but parity is spread all over the disks as opposed to having just one parity disk, as s

Physical disk formatting

- Divides the disk into sectors
- Fills the disk with a special data structure for each sector
- In the header and trailer is the sector number, and extra bits for error-correcting code (ECC)
  - The ECC data is updated by the disk controller on each write and checked on each read
  - If only a few bits of data have been corrupted, the controller can use the ECC to fix those bits
  - Otherwise the sector is now known as "bad" which is reported to the OS

  Typically all done at the factory before shipping

Logical formatting

- **the partitions**
  OS first partitions the disk into one or more groups of cylinders
- OS then treats each partition as a separate disk
- file system information is written to the partitions

Boot blocks

- The full bootstrap is stored in the boot blocks at a fixed location on a boot disk/partition
  so-called master boot record
- program then loads the OS

sector sparing

- Upon reboot, the disk controller can be told to replace a bad block by a spare
  - Each time the OS asks for the bad block, it is given the spare instead
  - The controller maintains an entire block map

Problem: the OS's view of disk

Bad Blocks

- Sometimes, data on the disk is corrupted and the ECC can't fix it, due to occurring error
  - Damage to the platter's surface - Defect in the magnetic medium due to wear
  - Temporary mechanical error (e.g., head touching the platter)
  - Temporary thermal fluctuation

locality may be very different from the physical locality

- **Solution** #1**:** Spares in each cylinders and a spare cylinder
  - Always try to find spares "close" to the bad block
- **Solution** #2**:** Shuffle sectors to bring the spare block next to the bad block
  - called **sector splitting**

Solid state drives (SSDs)

- Purely based on solid-state memory
  - Flash-based: persistent but slower (the common case)
  - DRAM-based: faster but volatile

Advantage

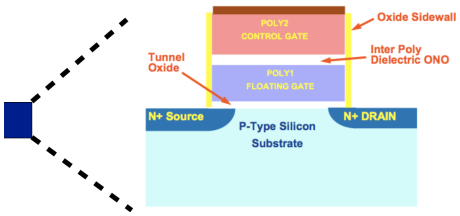
- No moving parts
- SSDs are becoming more and more mainstream

Flash SSDs competitive vs. hard drives

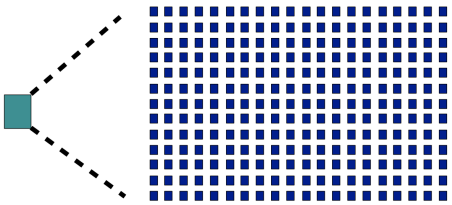
- faster startups and reads silent, low-heat, low-power
- more reliable
- less heavy
- getting larger and cheaper, close to HDD
- lower lifetime due to write wear of
- slower writes

SSD Structure

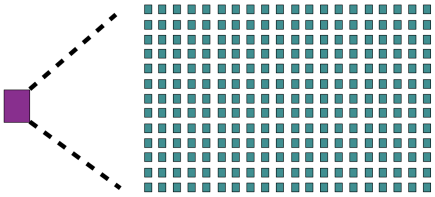
flash cell



The page



block

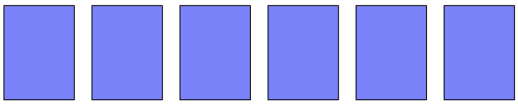


Have slow writes

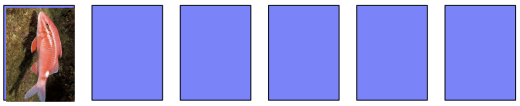
- slow because of **write amplification**: as time goes on, a write of x bytes of data in fact entails writing y $>$ x bytes of data
- The smallest unit that can be read: a 4-KiB page
- The smallest unit that can be erased: a 512-KiB block

Example

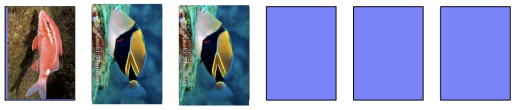
- Let's say we have a 6-page block (I don't want to draw 128 pages)



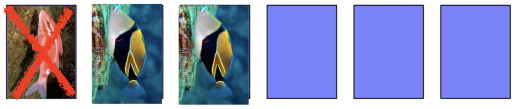
- Let's write a 4-KiB file

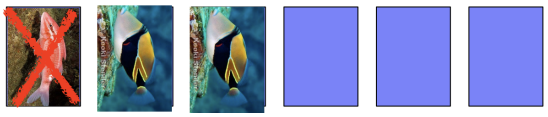


- Let's now write an 8-KiB file

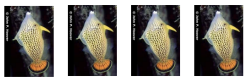


- Let's "erase" the first file
  - We can't erase the file without erasing the whole block, so the SSD controller only marks the block(s) that contains the file as "invalid"

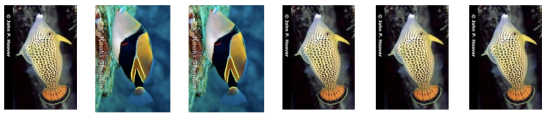




- Let's now write an 16-KiB file:



- We have to
  - Load the whole block into the SSD controller cache
  - Modify the in-cache block
  - Write back the whole block



- To write 4KiB + 8KiB + 16KiB = 28KiB of application data, we had to write 4KiB + 8KiB + 24KiB = 36KiB of data to the SSD
- As the drive fills up and files get written/modified/deleted, writes end up amplified
- The controller keeps writing on the SSD until full, before it attempts any rewrite

- In the end, performance is still good relative to that of an HDD
- The OS can, in the background, clean up block with invalid pages so that they're easily writable

SSDs. vs. HDDs

- SSDs have many advantages of HDDs
  - Random read latency much smaller
  - SSDs are great at parallel read/write
  - SSDs are great at small write
  - SSDs are great for random access in general
  - Which is typically the bane of HDDs
- Note that not all SSDs are made equal

In Class Exercises

Disk blocks are 8KiB, a block pointer is 4 bytes
What is the maximum file size with the i-node structure?

Direct indirect: 12 * 8KiB
Single indirect: (8KiB / 4) * 8KiB
Double indirect: (8KiB / 4) * (8KiB / 4) * 8KiB
Triple indirect: (8KiB / 4) * (8KiB / 4) * (8KiB / 4) * 8KiB

Total: $12 * 2^{13} + 2^{11} * 2^{13} + 2^{11} * 2^{11} * 2^{13} + 2^{11} * 2^{11} * 2^{11} * 2^{13}$

Conclusion

- HDDs are slow, large, unreliable, and cheap
- Disk scheduling helps with performance
- Redundancy is a way to cope with slow and unreliable HDDS (RAID)
- SSDs provide a radically different approach that may replace HDDs in the future
  The two are likely to coexist for years to come
- The OS is involved minimally in disk management functions as the drive controllers do most of the work