Persistent Storage
Persistent storage

just like memory, only different

- Just like diamonds
  - last forever (?)
  - memory is volatile
  - very dense
    - 1 TByte of storage fits here
- ...but much cheaper
  - 1 TByte is less than $100 on Amazon
  - way cheaper than
# How persistent storage affects OS Design

<table>
<thead>
<tr>
<th>Goal</th>
<th>Physical Characteristics</th>
<th>Design Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td>High performance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Named data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Controlled Sharing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reliability</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How persistent storage affects OS Design

<table>
<thead>
<tr>
<th>Goal</th>
<th>Physical Characteristics</th>
<th>Design Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td>High performance</td>
<td>Large cost to initiate I/O</td>
<td></td>
</tr>
<tr>
<td>Named data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Controlled Sharing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reliability</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How persistent storage affects OS Design

<table>
<thead>
<tr>
<th>Goal</th>
<th>Physical Characteristics</th>
<th>Design Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td>High performance</td>
<td>Large cost to initiate I/O</td>
<td>Organize storage to access data in large sequential units Use caching</td>
</tr>
<tr>
<td>Named data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Controlled Sharing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reliability</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How persistent storage affects OS Design

<table>
<thead>
<tr>
<th>Goal</th>
<th>Physical Characteristics</th>
<th>Design Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>High performance</strong></td>
<td>Large cost to initiate I/O</td>
<td>Organize storage to access data in large sequential units</td>
</tr>
<tr>
<td><strong>Named data</strong></td>
<td>Large capacity</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Survives crashes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shared across programs</td>
<td></td>
</tr>
<tr>
<td><strong>Controlled Sharing</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Reliability</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- High performance: Large cost to initiate I/O
- Named data: Large capacity, Survives crashes, Shared across programs
- Controlled Sharing: 
- Reliability: 

Controlled Sharing: 

Reliability: 

# How persistent storage affects OS Design

<table>
<thead>
<tr>
<th>Goal</th>
<th>Physical Characteristics</th>
<th>Design Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>High performance</strong></td>
<td>Large cost to initiate I/O</td>
<td>Organize storage to access data in large sequential units</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use caching</td>
</tr>
<tr>
<td><strong>Named data</strong></td>
<td>Large capacity</td>
<td>Support files and directories with meaningful names</td>
</tr>
<tr>
<td></td>
<td>Survives crashes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shared across programs</td>
<td></td>
</tr>
<tr>
<td><strong>Controlled Sharing</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Reliability</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# How persistent storage affects OS Design

<table>
<thead>
<tr>
<th>Goal</th>
<th>Physical Characteristics</th>
<th>Design Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td>High performance</td>
<td>Large cost to initiate I/O</td>
<td>Organize storage to access data in large sequential units</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use caching</td>
</tr>
<tr>
<td>Named data</td>
<td>Large capacity</td>
<td>Support files and directories with meaningful names</td>
</tr>
<tr>
<td></td>
<td>Survives crashes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shared across programs</td>
<td></td>
</tr>
<tr>
<td>Controlled Sharing</td>
<td>Device may store data from many users</td>
<td></td>
</tr>
<tr>
<td>Reliability</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## How persistent storage affects OS Design

<table>
<thead>
<tr>
<th>Goal</th>
<th>Physical Characteristics</th>
<th>Design Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>High performance</strong></td>
<td>Large cost to initiate I/O</td>
<td>Organize storage to access data in large sequential units</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use caching</td>
</tr>
<tr>
<td><strong>Named data</strong></td>
<td>Large capacity</td>
<td>Support files and directories with meaningful names</td>
</tr>
<tr>
<td></td>
<td>Survives crashes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shared across programs</td>
<td></td>
</tr>
<tr>
<td><strong>Controlled Sharing</strong></td>
<td>Device may store data from many users</td>
<td>Include with files <em>metadata</em> for access control</td>
</tr>
<tr>
<td><strong>Reliability</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# How persistent storage affects OS Design

<table>
<thead>
<tr>
<th>Goal</th>
<th>Physical Characteristics</th>
<th>Design Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>High performance</strong></td>
<td>Large cost to initiate I/O</td>
<td>Organize storage to access data in large sequential units</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use caching</td>
</tr>
<tr>
<td><strong>Named data</strong></td>
<td>Large capacity</td>
<td>Support files and directories with meaningful names</td>
</tr>
<tr>
<td></td>
<td>Survives crashes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shared across programs</td>
<td></td>
</tr>
<tr>
<td><strong>Controlled Sharing</strong></td>
<td>Device may store data from many users</td>
<td>Include with files metadata for access control</td>
</tr>
<tr>
<td><strong>Reliability</strong></td>
<td>Crash can occur during updates</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Storage devices can fail</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Flash memory wears out</td>
<td></td>
</tr>
</tbody>
</table>
### How persistent storage affects OS Design

<table>
<thead>
<tr>
<th>Goal</th>
<th>Physical Characteristics</th>
<th>Design Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td>High performance</td>
<td>Large cost to initiate I/O</td>
<td>Organize storage to access data in large sequential units</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use caching</td>
</tr>
<tr>
<td>Named data</td>
<td>Large capacity</td>
<td>Support files and directories with meaningful names</td>
</tr>
<tr>
<td></td>
<td>Survives crashes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shared across programs</td>
<td></td>
</tr>
<tr>
<td>Controlled Sharing</td>
<td>Device may store data from many users</td>
<td>Include with files metadata for access control</td>
</tr>
<tr>
<td>Reliability</td>
<td>Crash can occur during updates</td>
<td>Use transactions</td>
</tr>
<tr>
<td></td>
<td>Storage devices can fail</td>
<td>Use redundancy to detect and correct failures</td>
</tr>
<tr>
<td></td>
<td>Flash memory wears out</td>
<td>Migrate data to even the wear</td>
</tr>
</tbody>
</table>
How persistent storage affects applications

Example: Word processor with auto-save feature

If file is large and developer is naive
  □ poor performance
    ▶ may have to overwrite entire file to write a few bytes!
      – clever doc format may transform updates in appends
  □ corrupt file
    ▶ crash while overwriting file
  □ lost file
    ▶ crash while copying new file to old file location
The File System abstraction

- Presents applications with **persistent, named** data
- Two main components:
  - files
  - directories
A **file** is a named collection of data.

A file has two parts

- **data** – what a user or application puts in it
  - array of untyped bytes (in MacOS HFS, multiple streams per file)
- **metadata** – information added and managed by the OS
  - size, owner, security info, modification time
The directory provides names for files
- a list of human readable names
- a mapping from each name to a specific underlying file or directory (hard link)
- a soft link is instead a mapping from a file name to another file name
  - alias: a soft link that continues to remain valid when the (path of) the target file name changes
Path and Volume

- **path**: string that identifies a file or directory
  - absolute (if it starts with “/”, the root directory)
  - relative (w.r.t. the current working directory)

- **volume**: a collection of physical storage resources forming a logical storage device
**Mount**

- **mount**: allows multiple file systems on multiple volumes to form a single logical hierarchy
  - a mapping from some path in existing file system to the root directory of the mounted file system

[Diagram of file system hierarchy with Mount Point, USB Volume, and directories]
File system API

Creating and deleting files
- `create()` creates 1) a new file with some metadata and 2) a name for the file in a directory
- `link()` creates a hard link—a new name for the same underlying file
- `unlink()` removes a name for a file from its directory. If last link, file itself and resources it held are deleted

Open and close
- `open()` provides caller with a file descriptor to refer to file
  - permissions checked at open() time (a capability!)
  - creates per-file data structure, referred to by file descriptor
    - file ID, R/W permission, pointer to process position in file
- `close()` releases data structure

File access
- `read()`, `write()`, `seek()`
  - but can use `mmap()` to create a mapping between region of file and region of memory
- `fsync()` does not return until data is written to persistent storage
File systems: What’s so hard?

- Just map keys to values!
- file name & offset
- block numbers on a device
File systems: What’s so hard?

- Just map keys to values!

- Not so fast!
  - Performance
    - spatial locality
  - Flexibility
    - must handle diverse workloads
  - Reliability
    - must handle OS crashes and HW malfunctions
Implementation: key ideas

- **Directories**
  - file name ➔ file number

- **Index structures**
  - file number ➔ block

- **Free space maps**
  - find a free block; actually, find a free block nearby

- **Locality heuristics**
  - policies enabled by above mechanisms
    - group directories
    - make writes sequential
    - defragment
Directory

- A file that contains a collection of mapping from file name to file number

/Users/lorenzo

<table>
<thead>
<tr>
<th>Directory</th>
<th>File Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documents</td>
<td>45920022</td>
</tr>
<tr>
<td>Music</td>
<td>63092178</td>
</tr>
<tr>
<td>Project1</td>
<td>57294492</td>
</tr>
<tr>
<td>griso.jpg</td>
<td>36271008</td>
</tr>
</tbody>
</table>

- To look up a file, find the directory that contains the mapping to the file number
- To find that directory, find the parent directory that contains the mapping to that directory's file number...
- Good news: root directory has well-known number (2)
Looking up a file

Find file /Users/lorenzo/griso.jpg
Finding data

- **Index structure** provides a way to locate each of the file's blocks
  - usually implemented as a tree for scalability
- **Free space map** provides a way to allocate free blocks
  - often implemented as a bitmap
- **Locality heuristics** group data to maximize access performance
Case studies

- **FAT** late 70s; Microsoft
  - Key idea: linked list
  - Today: flash sticks

- **Unix FFS** mid 80’s
  - Key idea: tree-based multi-level index
  - Today: Linux ext2 and ext3

- **NTFS** early 1990s; Microsoft.
  - Key idea: variable size extents instead of fixed size blocks
  - Today: Windows 7, Linux ext4, Apple HFS

- **ZFS** early 2000; open source.
  - Key idea: copy on write (COW)
**FAT File system**

Microsoft, late 70s

**File Allocation Table (FAT)**

- started with MSDOS
- in FAT-32, supports $2^{28}$ blocks and files of $2^{32}-1$ bytes

**Index Structures**

- **File Allocation Table (FAT)**
  - array of 32-bit entries
  - file represented as a linked list of FAT entries
  - file # = index of first FAT entry

**Free space map**

- If data block $i$ is free, then FAT[$i$] = 0
- find free blocks by scanning MFT

**Locality heuristics**

- As simple as next fit:
  - scan sequentially from last allocated entry and return next free entry
  - Can be improved through defragmentation
FAT File system

Microsoft, late 70s

- File Allocation Table (FAT)
  - started with MSDOS
  - in FAT-32, supports $2^{28}$ blocks and files of $2^{32}-1$ bytes

### Advantages
- simple!
- used in many USB flash keys
- used even within MS Word!

### Disadvantages
- Poor locality
  - next fit? seriously?
- Poor random access
  - needs sequential traversal
- Limited access control
  - no file owner or group ID metadata
  - any user can read/write any file
- No support for hard links
  - metadata stored in directory entry
- Volume and file size are limited
  - FAT entry is 32 bits, but top 4 are reserved
  - no more than $2^{28}$ blocks
  - with 4kB blocks, at most 1TB volume
  - file no bigger than 4GB
- No support for transactional updates

---

Data blocks:
- file 9 block 0
- file 9 block 1
- file 9 block 2
- file 12 block 0
- file 12 block 1
- file 9 block 4
FFS: Fast File System

Unix, 80s

- Smart index structure
  - multilevel index allows to locate all blocks of a file
    - efficient for both large and small files

- Smart locality heuristics
  - block group placement
    - optimizes placement for when a file data and metadata, and other files within same directory, are accessed together
  - reserved space
    - gives up about 10% of storage to allow flexibility needed to achieve locality
File structure

Each file is a fixed, asymmetric tree, with fixed size data blocks (e.g. 4KB) as its leaves.

The root of the tree is the file’s inode:
- contains file’s metadata
  - owner, permissions (rwx for owner, group other), directory?, etc
  - setuid: file is always executed with owner’s permission
    - add flexibility but can be dangerous
  - setgid: like setuid for groups
- contains a set of pointers
  - typically 15
  - first 12 point to data block
  - last three point to intermediate blocks, themselves containing pointers
    - 13: indirect pointer
    - 14: double indirect pointer
    - 15: triple indirect pointer
Multilevel index

Inode Array

Inode

File metadata

4 Bytes entries

indirect block
contains pointers to data blocks

double indirect block
contains pointers to indirect blocks

triple indirect block
contains pointers to double indirect blocks

Data blocks

12 x 4KB = 48KB

1K x 4KB = 4MB

1K x 1k x 4KB = 4GB

1K x 1k x 1k x 4KB = 4TB

at known location on disk

file number = inode number = index in the array

file number = inode number = index in the array
Multilevel index: key ideas

- Tree structure
  - efficient in finding blocks
- High degree
  - efficient in sequential reads
  - once an indirect block is read, can read 100s of data block
- Fixed structure
  - simple to implement
- Asymmetric
  - supports efficiently files big and small
Example: variations on the FFS theme

In BigFS an inode stores
4kb blocks, 8 byte pointers
- 12 direct pointers
- 1 indirect pointer
- 1 double indirect
- 1 triple indirect
- 1 quadruple indirect

What is the maximum size of a file?
- Through direct pointers
  - $12 \times 4\text{kb} = 48\text{KB}$
- Indirect pointer
  - $512 \times 4\text{kb} = 2\text{MB}$
- Double indirect pointer
  - $512^2 \times 4\text{kb} = 1\text{GB}$
- Triple indirect pointer
  - $512^3 \times 4\text{kb} = 512\text{GB}$
- Quadruple indirect pointer
  - $512^4 \times 4\text{kb} = 256\text{TB}$

Total $= (256 + .5 \times 10^{-6} + 2 \times 10^{-9} + 4.8 \times 10^{-11}) \approx 256.5 \text{ TB}$
Free space management

- Easy
  - a bitmap with one bit per storage block
  - bitmap location fixed at formatting time
  - i-th bit indicates whether i-th block is used or free
Locality heuristics: block group placement

- Divide disk in **block groups**
  - sets of nearby tracks

- Distribute metadata
  - old design: free space bitmap and inode map in a single contiguous region
    - lots of seeks when going from reading metadata to reading data
  - FFS: distribute free space bitmap and inode array among block groups

- Place file in block group
  - when a new file is created, FFS looks for inodes in the same block as the file's directory
  - when a new directory is created, FFS places it in a different block from the parent's directory

- Place data blocks
  - first free heuristics
  - trade short term for long term locality
Locality heuristics: block group placement

- Divide disk in block groups
  - sets of nearby tracks

- Distribute metadata
  - old design: free space bitmap and inode map in a single contiguous region
  - lots of seeks when going from reading metadata to reading data
  - FFS: distribute free space bitmap and inode array among block groups

- Place file in block group
  - when a new file is created, FFS looks for inodes in the same block as the file's directory
  - when a new directory is created, FFS places it in a different block from the parent's directory

- Place data blocks
  - first free heuristics
  - trade short term for long term locality
Locality heuristics: block group placement

- Divide disk in **block groups**
  - sets of nearby tracks

- Distribute metadata
  - old design: free space bitmap and inode map in a single contiguous region
    - lots of seeks when going from reading metadata to reading data
  - FFS: distribute free space bitmap and inode array among block groups

- Place file in block group
  - when a new file is created, FFS looks for inodes in the same block as the file's directory
  - when a new directory is created, FFS places it in a different block from the parent's directory

- Place data blocks
  - first free heuristics
  - trade short term for long term locality

- Free space bitmap
- Inodes
- Data blocks for files in
  - `/a`
  - `/d`
  - `/b/c`

- Start of block group

Small file
Locality heuristics: block group placement

- Divide disk in **block groups**
  - sets of nearby tracks

- Distribute metadata
  - old design: free space bitmap and inode map in a single contiguous region
  - lots of seeks when going from reading metadata to reading data
  - FFS: distribute free space bitmap and inode array among block groups

- Place file in block group
  - when a new file is created, FFS looks for inodes in the same block as the file's directory
  - when a new directory is created, FFS places it in a different block from the parent's directory

- Place data blocks
  - first free heuristics
  - trade short term for long term locality
Locality heuristics: reserved space

- When a disk is full, hard to optimize locality
  - file may end up scattered through disk

- FFS presents applications with a smaller disk
  - about 10% smaller
  - user write that encroaches on reserved space fails
  - super user still able to allocate inodes to clean things up
NTFS: flexible tree with extents

Index structure: extents and flexible tree

- extents
  - track ranges of contiguous blocks rather than single blocks

- flexible tree
  - file represented by variable depth tree
    - large file with few extents can be stored in a shallow tree

- MFT (Master File Table)
  - array of 1 KB records holding the trees’ roots
  - similar to inode table (but 1 file can have multiple MFT entries)
  - each record stores sequence of variable-sized attribute records
    - both data and metadata are attributes
    - attributes can be resident (fits in the record) or nonresident
Example of NTFS index structure

Basic file with two data extents
Example of NTFS index structure

Master File Table

MFT Record

<table>
<thead>
<tr>
<th>Std. Info</th>
<th>File Name</th>
<th>Data (resident)</th>
<th>free</th>
</tr>
</thead>
</table>

- file name and number of parent directory
- one file name attribute per hard link

- file creation time
- access time
- owner ID
- security specifier

Small file where data is resident
Example of NTFS index structure

A file’s attributes can span multiple records
Small, normal, and big files

Master File Table

...and for really huge (or really badly fragmented) files, even the attribute list can become nonresident!

- attribute list split in separate extents
Metadata files

- NTFS stores most metadata in ordinary files with well-known numbers
  - 5 (root directory); 6 (free space bitmap); 8 (list of bad blocks)
- $Secure (file no. 9)
  - stores access control list for every file
  - indexed by fixed-length key
  - files store appropriate key in their MFT record
- $MFT (file no. 0)
  - stores Master File Table
  - to read MFT, need to know first entry of MFT
    - a pointer to it stored in first sector of NTFS
  - MFT can start small and grow dynamically
  - To avoid fragmentation, part of start of volume reserved to MFT expansion
    - when full, halves reserved MFT area
Locality heuristics

- **Best fit**
  - finds smallest region large enough to fit file
  - NTFS caches allocation status for a small area of disk
    - writes that occur together in time get clustered together
  - `SetEnfOfFile()` lets specify expected length of file at creation
COW File Systems (copy-on-write)

Data and metadata not updated in place, but written to new location

- transforms random writes into sequential writes

Adding a block to a file

- Traditional
- COW
COW File Systems

Why?

- Small writes are expensive
- Small writes are expensive on RAID
  - expensive to update a single block (4 disk I/O) but efficient for entire stripes
- Caches filter reads
- Widespread adoption of flash storage
  - wear leveling, which spreads writes across all cells, important to maximize flash life
  - COW techniques used to virtualize block addresses and redirect writes to cleared erasure blocks
- Large storage capacities enable versioning
  - versioning is easy with COW!
The core idea

Inode Array → Indirect Blocks → Data Blocks

Traditional:
- Fixed Location
- Anywhere

COW:
- Fixed Location
- Anywhere

COW (Copy On Write)
The core idea
The core idea

- Inode Array
- Indirect Blocks
- Data Blocks

Traditional

- Fixed Location
- Anywhere

COW

- Fixed Location
- Anywhere
The core idea

Inode Array  Indirect Blocks  Data Blocks

Traditional  Fixed Location  Anywhere

COW  Fixed Location  Anywhere
The core idea

Inode

Array

Indirect Blocks

Data Blocks

Fixed Location

Anywhere

Traditional

COW

Fixed Location

Anywhere
File access in FFS

What it takes to read /Users/lorenzo/wisdom.txt

- Read Inode for “/” (root) from a fixed location
- Read first data block for root
- Read Inode for /Users
- Read first data block of /Users
- Read Inode for /Users/lorenzo
- Read first data block for /Users/lorenzo
- Read Inode for /Users/lorenzo/wisdom.txt
- Read data blocks for /Users/lorenzo/wisdom.txt

“A cache is a man’s best friend”
Caching and consistency

- File systems maintain many data structures
  - bitmap of free blocks
  - bitmap of inodes
  - directories
  - inodes
  - data blocks

- Data structures cached for performance
  - works great for read operations...
  - ...but what about writes?

- Solutions:
  - write-back caches: delay writes for higher performance at the cost of potential inconsistencies
  - write through caches: write synchronously but poor performance
    - do we get consistency at least?
Example: a tiny ext2

- 6 blocks, 6 inodes

Suppose we append a data block to the file

- add new data block D2

```
inode bitmap: 010000  
data bitmap: 000010  

Iv1  --  --  --  --  

i-nodes  data blocks

owner: lorenzo  
permissions: read-only  
size: 1  
pointer: 4  
pointer: null  
pointer: null  
pointer: null  
```
Example: a tiny ext2

- 6 blocks, 6 inodes

Suppose we append a data block to the file
- add new data block D2
- update inode

<table>
<thead>
<tr>
<th>inode bitmap</th>
<th>data bitmap</th>
<th>i-nodes</th>
<th>data blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 0</td>
<td>0 0 0 0 1 0</td>
<td>Iv1</td>
<td>D1 D2</td>
</tr>
</tbody>
</table>

- owner: lorenzo
- permissions: read-only
- size: 1
- pointer: 4
- pointer: null
- pointer: null
- pointer: null
Example: a tiny ext2

- 6 blocks, 6 inodes

- Suppose we append a data block to the file
  - add new data block D2
  - update inode
  - update data bitmap

<table>
<thead>
<tr>
<th>inode bitmap</th>
<th>data bitmap</th>
<th>i-nodes</th>
<th>data blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 0</td>
<td>0 0 0 1 1 0</td>
<td>- - - - -</td>
<td>D1 D2</td>
</tr>
</tbody>
</table>

- owner: lorenzo
- permissions: read-only
- size: 2
- pointer: 4
- pointer: 5
- pointer: null
- pointer: null
Example: a tiny ext2

6 blocks, 6 inodes

Suppose we append a data block to the file

- add new data block D2
- update inode
- update data bitmap

What if a crash or power outage occurs between writes?
What if only a single write succeeds?

- Just the data block (D2) is written to disk
  - data is written, but no way to get to it – in fact, D2 still appears as a free block
  - as if write never occurred

- Just the updated inode (Iv2) is written to disk
  - if we follow the pointer, we read garbage
  - file system inconsistency: data bitmap says block is free, while inode says it is used. Must be fixed

- Just the updated bitmap is written to disk
  - file system inconsistency: data bitmap says data block is used, but no inode points to it.
  - No idea which file the data block was to belong to!
What if two writes succeed?

- Inode and data bitmap updates succeed
  - file system is consistent
  - but reading new block returns garbage

- Inode and data block updates succeed
  - file system inconsistency. Must be fixed

- Data bitmap and data block succeed
  - file system inconsistency
  - no idea to which file data block should belong to!
The Consistent Update problem

Several file systems operations update multiple data structures

- Move a file between directories
  - delete file from old directory
  - add file to new directory
- Create new file
  - update inode bitmap and data bitmap
  - write new inode
  - add new file to directory file

Even with write through we have a problem!
Ad hoc solutions: metadata consistency

- Synchronous write through for metadata
- Updates performed in a specific order
  - File create
    - write data block
    - update inode
    - update inode bitmap
    - update data bitmap
    - update directory
    - if directory grew: 1) update data bitmap; 2) update directory inode
- On file crash
  - fsck
    - scans entire disk for inconsistencies, starting with superblock
    - scans inodes, indirect blocks, double indirect, etc to understand which blocks are allocated
    - uses scan results to update bitmaps
    - file created but not in any directory: delete file
- Issues
  - need to get ad-hoc reasoning exactly right
  - synchronous writes lead to poor performance
  - recovery is sloooow: must scan entire disk
Ad hoc solutions: user data consistency

- Asynchronous write back
  - forced after a fixed interval (e.g. 30 sec)
  - can lose up to 30 sec of work

- Rely on metadata consistency
  - updating a file in vi
    - delete old file
    - write new file
Ad hoc solutions: user data consistency

- Asynchronous write back
  - forced after a fixed interval (e.g. 30 sec)
  - can lose up to 30 sec of work

- Rely on metadata consistency
  - updating a file in vi
    - write new version to temp
    - move old version to other temp
    - move new version to real file
    - unlink old version
      - if crash, look in temp area and send “there may be a problem” email to user
Ad hoc solutions: implementation tricks

- **Block I/O Barriers**
  - allow a block device user to enforce ordering among I/O issued to that block device
  - client need not block waiting for write to complete
  - instead, OS builds a dependency graph
    - no write goes to disk unless all writes it depends on have
A principled approach: Transactions

- Group together actions so that they are
  - Atomic: either all happen or none
  - Consistent: maintain invariants
  - Isolated: serializable (schedule in which transactions occur is equivalent to transactions executing sequentially)
  - Durable: once completed, effects are persistent

- Critical sections are ACI, but not Durable

- Transaction can have two outcomes:
  - Commit: transaction becomes durable
  - Abort: transaction never happened
    - may require appropriate rollback
Journaling
(write ahead logging)

- Turns multiple disk updates into a single disk write
  - "write ahead" a short note to a "log", specifying changes about to be made to the FS data structures
  - if a crash occurs while updating the FS data structure, consult log to determine what to do
    - no need to scan entire disk!
Data Jounaling: an example

We start with

<table>
<thead>
<tr>
<th>inode bitmap</th>
<th>data bitmap</th>
<th>i-nodes</th>
<th>data blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000000</td>
<td>00000100</td>
<td>Iv1</td>
<td>D1</td>
</tr>
</tbody>
</table>

We want to add a new block to the file

Three easy steps

- Write to the log 5 blocks: TxBegin | Iv2 | B2 | D2 | TxEnd
  - write each record to a block, so it is atomic
- Write the blocks for Iv2, B2, D2 to the FS proper
- Mark the transaction free in the journal

What happens if we crash before the log is updated?

- no commit, nothing to disk - ignore changes!

What happens if we crash after the log is updated?

- replay changes in log back to disk
Journaling and Write Order

- Issuing the 5 writes to the log sequentially is slow

- Issue at once, and transform in a single sequential write

- Problem: disk can schedule writes out of order
  - first write TxBegin, Iv2, B2, TxEnd
  - then write D2

- Log contains: TxBegin | Iv2 | B2 | ?? | TxEnd
  - syntactically, transaction log looks fine, even with nonsense in place of D2!

- Set a Barrier before TxEnd
  - TxEnd must block until data on disk
What about performance?

* All data is written twice... surely it is horrible?

100 1KB random writes vs. log + write-back

- Direct write: \(100 \times T_{rw} \approx 100 \times 10ms \approx 1s\)
- Pessimistic log
  - \(100 \times T_{sw} + 100 \times T_{rw} \approx 100/(50 \times 10^3) + 1s = 2ms + 1s\)
- Realistic (write-back performed in the background)
  - more opportunities for disk scheduling
  - 100 random writes may take less time than in direct write case
Back to
The early 90s

Growing memory sizes
- file systems can afford large block caches
- most reads can be satisfied from block cache
- performance dominated by write performance

Growing gap in random vs sequential I/O performance
- transfer bandwidth increases 50%-100% per year
- seek and rotational delay decrease by 5%-10% per year
- using disks sequentially is a big win

Existing file system perform poorly on many workloads
- 6 writes to create a new file of 1 block
  - new inode | inode bitmap | directory data block that includes file | directory inode (if necessary) | new data block storing content of new file | data bitmap
- lots of short seeks
Log structured file systems

- Use disk as a log
  - buffer all updates (including metadata!) into a segment
  - when segment is full, write to disk in a long sequential transfer to unused part of disk
- Virtually no seeks
  - much improved disk throughput
- But how does it work?
  - suppose we want to add a new block to a 0-sized file
  - LFS paces both data block and inode in its in-memory segment

Fine. But how do we find the inode?
Finding inodes

in UFS, just index into inode array

Super Block | Inodes | Data blocks

Super Block
b0

Inodes
b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11 ...
0 4 8 12 16 20 24 28 32 36
1 5 9 13 17 21 25 29 33 37
2 6 10 14 18 22 26 30 34 38
3 7 11 15 19 23 27 31 35 39

Data blocks

512 bytes/block
128 bytes/inode

To find address inode 11:
addr(b1)+ #inode x size(inode)

Same in FFS (but Inodes are at divided (at known locations) between block groups
Finding inodes in LFS

- **inode map**: a table indicating where each inode is on disk
  - inode map blocks written as part of the segment
  - ... so need not seek to write to imap

- but how do we find the inode map?
  - table in a fixed **checkpoint region**
    - updated periodically (every 30 seconds)

- The disk then looks like

```
+---+---+---+---+---+
| CR | seg1 | free | seg2 | seg3 | free |
+---+---+---+---+---+
```
LFS vs UFS

Log-structured File System

blocks written to create two 1-block files: dir1/file1 and dir2/file2 in UFS and LFS
Suppose nothing in memory...

- read checkpoint region
- from it, read and cache entire inode map
- from now on, everything as usual
  - read inode
  - use inode's pointers to get to data blocks

When the imap is cached, LFS reads involve

**virtually** the same work as reads in traditional file systems

modulo an imap lookup
Garbage collection

- As old blocks of files are replaced by new, segment in log become fragmented
- **Cleaning** used to produce contiguous space on which to write
  - compact M fragmented segments into N new segments, newly written to the log
  - free old M segments
- Cleaning mechanism:
  - How can LFS tell which segment blocks are live and which dead?
- Cleaning policy
  - How often should the cleaner run?
  - How should the cleaner pick segments?
Segment summary block

- For each data block, stores
  - the file it belongs (inode#)
  - the offset (block#) within file

- During cleaning
  - allows to determine whether data block D is live
    - use inode# to find in imap where inode is currently on disk
    - read inode (if not already in memory)
    - check whether pointer for block block# refers to D's address
  - allows to update file's inode with correct pointer if D is live and compacted to new segment
Which segments to clean, and when?

**When?**
- periodically
- when you have nothing better to do
- when disk is full

**Which segments?**
- utilization: how much it is gained by cleaning
  - segment usage table tracks how much live data in segment
- age: how likely is the segment to change soon
  - better to wait on cleaning a hot block
Crash recovery

The journal is the file system!

On recovery

- read checkpoint region
  - may be out of date (written periodically)
  - may be corrupted
    - 1) two CR blocks at opposite ends of disk / 2) timestamp blocks before and after CR
    - use CR with latest consistent timestamp blocks

- roll forward
  - start from where checkpoint says log ends
  - read through next segments to find valid updates not recorded in checkpoint
    - when a new inode is found, update imap
    - when a data block is found that belongs to no inode, ignore
Towards Distributed Systems
What is a distributed system?

“A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.”

Leslie Lamport
The Client/Server paradigm

- **Server**
  - offers some service (e.g., file server)
  - may exist in more than one node

- **Client**
  - uses the service

- **The basic pattern**
  - clients binds (i.e., connects) to the server
  - client sends **request** (with parameters) to perform services
  - server returns **response**
How to communicate?

Messages
- very flexible
- leave programmers to worry about
  - message format
  - how to pack and unpack messages
  - how to decode at the server
  - error handling

Procedure calls
- an old friend!
- server is a module that exports a set of procedures
Remote Procedure Call

Birrell & Nelson, Xerox PARC, '80s

Procedure calls as basis for distributed communication

How can we make RPC look like LPC?
- how can we make it invisible to the programmer?
- what are the semantics of parameter passing?
- how do we locate the server (binding)?
- how do we support heterogeneity (OS, architecture, programming language)?

Three-part solution
- user program (client or server)
- set of stub procedures
- runtime support
Building a server

Define server’s interface in an interface definition language (IDL)
- specifies names, parameters, and types for all server procedures that clients can invoke

Stubs compiler
- reads IDL
- produces a client and a server stub for each server procedure
  - manage all details of remote client-server communication

Server and client developers link their code to respective stub
RPC Stubs

**Client-side stub**
- Looks to client as callable server procedure
- Client program thinks it is calling the server

**Server-side stub**
- Server program thinks it is called by client
- foo actually called by server's stub

---

```
client program

client stub
call foo(x,y)
proc foo(a,b)

proc foo(a,b)
begin foo(a,b)
end foo(a,b)

server program

server stub
call foo(x,y)
call foo
```

stubs communicate with one another
RPC Stubs

Client Program
- Call foo(x,y)
  - call foo
  - Proc foo(a,b)
    - Send msg
      - Runtime sends msg to remote node

Client Stub
- Proc foo(a,b)
  - Stub builds msg packet, inserts params
  - Send msg
    - Runtime sends msg to remote node

Server Program
- Call foo
  - Proc foo(a,b)
    - Begin foo(a,b)
    - End foo(a,b)

Server Stub
- Call foo(x,y)
  - Stub unpacks params and makes call
  - Msg received
    - Runtime receives msg and calls stub

RPC Runtime
- Call
  - Runtime receives msg and calls stub
  - Msg received
    - Runtime receives msg and calls stub
RPC Stubs

client program

```
call foo(x,y)
```

```
proc foo(a,b)
begin foo(a,b)
end foo(a,b)
```

server program

server stub

```
proc foo(a,b)
begin foo(a,b)
end foo(a,b)
```

```
call foo(x,y)
```

server runtime

RPC runtime

```
msg received
```

```
return
```

```
stub unpacks msg, returns to caller
```

```
stub builds result msg with output args
```

```
runtime receives msg, calls stub
```

```
runtime responds to original message
```

```
send msg
```

```
return
```

Return
RPC Binding

- Server at startup exports its interface
  - identifies itself to a network name server
  - tells local runtime address of the dispatch routine that will perform requested services

- Client, before issuing calls, imports server
  - RPC runtime looks up service though network name server
  - contacts server to set up a connection

- Import and export are explicit calls in the code
RPC Marshalling

- Packing of procedure parameters in a message packet
  - notion close to pickling (Python), serialization (Java)
- RPC stubs call type-specific procedures to marshal/unmarshal call parameters
RPC Marshalling

- Packing of procedure parameters in a message packet
  - notion close to pickling (Python), serialization (Java)

- RPC stubs call type-specific procedures to marshal/unmarshal call parameters

- Roles reverse on return
  - server stub marshals return parameters; client stub unmarshals