Review: Flash
Flash Architecture

- Bank/plane: 1024—4096 blocks
  - Banks accessed in parallel
Flash Architecture

- Bank/plane: 1024—4096 blocks
  - Banks accessed in parallel

- Block: 64—256 pages

- Page: 2—8 KB
Flash Architecture

- **Bank/plane**: 1024—4096 blocks
  - Banks accessed in parallel

- **Block**: 64—256 pages
  - Unit of *erase*

- **Page**: 2—8 KB
  - Unit of *read*
  - Unit of *program*
Flash Architecture

- **Bank/plane**: 1024—4096 blocks
  - Banks accessed in parallel

- **Block**: 64—256 pages
  - Unit of **erase**

- **Page**: 2—8 KB
  - Unit of read
  - Unit of **program (write ‘0’s)**
Flash Architecture

- **Bank/plane**: 1024—4096 blocks
  - Banks accessed in parallel

- **Block**: 64—256 pages
  - Unit of **erase**

- **Page**: 2—8 KB
  - Unit of read
  - Unit of **program** (write ‘0’s)

Flash does not support overwrite on valid pages (pages that contain 0s)!
Flash Architecture

- Bank/plane: 1024—4096 blocks
  - Banks accessed in parallel

- Block: 64—256 pages
  - Unit of erase (write ‘1’s)

- Page: 2—8 KB
  - Unit of read
  - Unit of program (write ‘0’s)

To write to those valid pages, flash needs to erase the whole block
Flash Performance

- **Throughput**
  - Disk: \(~130\text{MB/s}\) (sequential)
  - **Flash**: \(~400\text{MB/s}\)

- **Latency**
  - Disk: \(~10\text{ms}\) (one op)
  - **Flash**:
    - **Read**: \(10\text{-}50\text{us}\)
    - **Program**: \(200\text{-}500\text{us}\)
    - **Erase**: \(2\text{ms}\)

Asymmetric read and write cost
File System Implementation
Review: File Names

- Three types of file names
  - inode number
  - path
  - file descriptor
Review: File Names

- **Inode**
  - Unique name
  - Remember file size, permissions, etc.

- **Path**
  - Easy to remember
  - Hierarchical

- **File descriptor**
  - Avoid frequent traversal
  - Remember runtime status (e.g., offsets)
Review: File Read and Write APIs

```c
int fd = open(char *path, int flag, mode_t mode);
-OR-
int fd = open(char *path, int flag);

ssize_t sz = read(int fd, void *buf, size_t count);

ssize_t sz = write(int fd, void *buf, size_t count);

int ret = close(int fd);
```
Review: Special APIs

```c
fsync(int fd)
```

```c
rename(char *oldpath, char *newpath)
```

```c
unlink(char *path)
```
File System API Examples

prompt> vim file.txt
File System API Examples

```c
int fd = open(".file.txt.swp");
```

```
prompt> vim file.txt
```

Under the hood...
File System API Examples

```
prompt> vim file.txt
...
...
```

```
int fd = open(".file.txt.swp");
write(fd, buffer, size); // editing
```

Under the hood...
File System API Examples

```c
int fd = open("./file.txt.swp");
write(fd, buffer, size);    // editing
fsync(fd);                // :w
close(fd);                // :q
rename("./file.txt.swp", "file.txt");  // :q
```

Under the hood…
Implementation
File System Implementation

- On-disk structures
  - How do we represent files and directories?

- File system efficiency
  - How to avoid excessive I/O operations?
On-Disk Structures
A Naïve Flat Persistent Store

- **Given**: big array of on-disk bytes/blocks
- **Want**: to support reads and writes
A Naïve Flat Persistent Store

- **Given**: big array of on-disk bytes/blocks
- **Want**: to support reads and writes

- Build a **flat** persistent store where each file is associated with a unique key
  - Uses a flat table to track files
  - Uses offsets for non-sequential I/O
Flat Persistent Store vs. File System

- What features does a file system provide beyond what a naïve flat persistent store would provide?
Flat Persistent Store vs. File System

- What features does a file system provide beyond what a naïve flat persistent store would provide?
  - Human readable string names
  - Hierarchy (names within names)
  - Changeable file sizes
  - Sharing across processes
  - ...
Flat Persistent Store vs. File System

- What features does a file system provide beyond what a naïve flat persistent store would provide?
  - Human readable string names
  - Hierarchy (names within names)
  - Changeable file sizes
  - Sharing across processes
  - …

All these features require a variety of on-disk data structures!
On-Disk Structures

- Common file system structures
  - Data block
  - Inode table
  - Directories
  - Data bitmap
  - Inode bitmap
  - Superblock
On-Disk Structure: Empty Disk
On-Disk Structure: Data Blocks
On-Disk Structures

- Common file system structures
  - Data block
  - Inode table
  - Directories
  - Data bitmap
  - Inode bitmap
  - Superblock
On-Disk Structure: Inodes
On-Disk Structure: Inodes

- Inodes are typically 128 or 256 bytes (depends on the file system)
  - 16—32 inodes per inode block
On-Disk Structure: Inodes

Inode Block

Inode

- type
- uid
- rwx
- size
- blocks
- time
- ctime
- links_count
- addrs[N]
On-Disk Structure: Inodes

Inode Block

Inode

Inode Block

type
uid
rwx
size
blocks
time
cmtime
links_count
addr[0-31]
On-Disk Structure: Inodes
On-Disk Structure: Inodes
On-Disk Structure: Inodes

- Inodes
- Data Region

inode Block

Inode

<table>
<thead>
<tr>
<th>type</th>
<th>uid</th>
<th>rwx</th>
<th>size</th>
<th>blocks</th>
<th>time</th>
<th>ctime</th>
<th>links_count</th>
<th>addr[N]</th>
</tr>
</thead>
</table>

access time and create time
On-Disk Structure: Inodes

Inode Block

Inode

inode 16
inode 17
inode 18
inode 19
inode 20
inode 21
inode 22
inode 23
inode 24
inode 25
inode 26
inode 27
inode 28
inode 29
inode 30
inode 31

Inode

type
uid
rwx
size
blocks
time
ctime
links_count
addr[N]

how many links (directories)
On-Disk Structure: Inodes

- Inode Block
- Inode
- type
- uid
- rwx
- size
- blocks
- time
- ctime
- links_count
- addr[N]
On-Disk Structure: Inodes

Each inode points to a file stored on disk, as one or multiple data blocks.
On-Disk Structures

- Common file system structures
  - Data block
  - Inode table
  - Directories
  - Data bitmap
  - Inode bitmap
  - Superblock
On-Disk Structure: Directories

- Common directory design: just store directory entries in files
  - Different file systems vary

- Various data structures (formats) could be used
  - Lists
  - B-trees
On-Disk Structures

- Common file system structures
  - Data block
  - Inode table
  - Directories
  - Data bitmap
  - Inode bitmap
  - Superblock
Allocation

- How does file system find free data blocks or free inodes?
Allocation

- How does file system find free data blocks or free inodes?
  - Free list
  - Bitmaps

- What are the tradeoffs?
Free List
Bitmap

Each bit of the bitmap is used to indicate whether the corresponding object/block is free (0) or in-use (1)

\[ \text{bit}[i] = \begin{cases} 
1 & \Rightarrow \text{object}[i] \text{ in use} \\
0 & \Rightarrow \text{object}[i] \text{ free} 
\end{cases} \]
Allocation

- How does file system find free data blocks or free inodes?
  - Free list
  - Bitmaps

- What are the tradeoffs?
  - Free list: Cannot get contiguous space easily
  - Bitmap: Easy to allocate contiguous space for files
On-Disk Structure: Data Bitmaps

Data bitmap
On-Disk Structure: Inode Bitmaps

Inode bitmap
On-Disk Structures

- Common file system structures
  - Data block
  - Inode table
  - Directories
  - Data bitmap
  - Inode bitmap
  - Superblock
On-Disk Structure: Superblock

- Need to know basic file system configuration and runtime status, such as:
  - Block size
  - How many inodes are there
  - How much free space

- Store all these *metadata* info in a superblock
On-Disk Structure: Superblock
On-Disk Structure: Superblock
On-Disk Structure Overview

The Inode Table (Closeup)

Super i-bmap d-bmap

Inodes Data Region

Data Region

0KB 4KB 8KB 12KB 16KB 20KB 24KB 28KB 32KB

Sid

0 7 8 15 16 23 24 31

32 39 40 47 48 55 56 63

DDDDDDDDDDDD DDDDDDDDDDD DDDDDDDDDDD
File System Efficiency
File System Efficiency and Performance

How does file system avoid excessive I/O for basic operations?
File System Efficiency and Performance

How does file system avoid excessive I/O for basic operations?

File system in-memory cache
  – A separate memory region reserved for frequently used blocks
  – Cache for
    • Reads
    • Write buffering (asynchronous writes)
Unified Page Cache

- Many modern OS’s integrate virtual memory pages and file system blocks into a unified page cache
  - Caches memory-mapped I/O pages (virtual memory)
  - Caches file system blocks (file system I/O)
I/O Paths Using a Unified Page Cache

- Memory-mapped I/O
- FS read() and write() I/O
- Unified page cache
- File system
Write Buffering

Why does procrastination (laziness) help?
Write Buffering

Why does procrastination (laziness) help?

1. Overwrites
   - By delaying writes, file system can batch updates of small I/Os
Write Buffering

Why does procrastination (laziness) help?

1. Overwrites
   - By delaying writes, file system can batch updates of small I/Os

2. Scheduling
   - OS can always schedule buffered writes to improve performance
Write Buffering

Why does procrastination (laziness) help?

1. Overwrites
   – By delaying writes, file system can batch updates of small I/Os

2. Scheduling
   – OS can always schedule buffered writes to improve performance

3. Deletes
   – Avoid delayed writes