CS162
Operating Systems and
Systems Programming
Lecture 20

Filesystems 1: Filesystem Design
Filesystem Case Studies

November 2\textsuperscript{nd}, 2021
Prof. Ion Stoica
http://cs162.eecs.Berkeley.edu
Recall: I/O Performance (Network Example)

- Consider a 1 Gb/s link \( (B = 125 \text{ MB/s}) \) with startup cost \( S = 1 \text{ ms} \)
- Latency: \( L(b) = S + \frac{b}{B} \)
- Effective Bandwidth:
  \[
  E(b) = \frac{b}{S + \frac{b}{B}} = \frac{B \cdot b}{B \cdot S + b} = \frac{B}{B \cdot S + b} + 1
  \]
- Half-power Bandwidth: \( E(b) = \frac{B}{2} \)
- For this example, half-power bandwidth occurs at \( b = 125 \text{ KB} \)
Recall: A Few Queuing Theory Results

- **Assumptions:**
  - System in equilibrium; No limit to the queue
  - Time between successive arrivals is random and memoryless

- **Parameters that describe our system:**
  - \( \lambda \): mean number of arriving customers/second
  - \( T_{ser} \): mean time to service a customer (“m1”)
  - \( C \): squared coefficient of variance = \( \sigma^2 / m1^2 \)
  - \( \mu \): service rate = \( 1 / T_{ser} \)
  - \( \rho \): server utilization (\( 0 \leq \rho \leq 1 \)): \( \rho = \lambda / \mu = \lambda \times T_{ser} \)

- **Parameters we wish to compute:**
  - \( T_q \): Time spent in queue
  - \( L_q \): Length of queue = \( \lambda \times T_q \) (by Little’s Law)

- **Results:**
  - Memoryless service distribution (\( C = 1 \)): (an “M/M/1 queue”, (an “M/M/1 queue”):
    » \( T_q = T_{ser} \times \rho / (1 - \rho) \)
  - General service distribution (no restrictions), 1 server (an “M/G/1 queue”, (an “M/G/1 queue”):
    » \( T_q = T_{ser} \times \frac{1}{2} \left( 1 + C \right) \times \rho / (1 - \rho) \)

Why does response/queueing delay grow unboundedly even though the utilization is < 1?
Optimize I/O Performance

- How to improve performance?
  - Speed: make everything faster 😊
  - Parallelism: More Decoupled systems
    » multiple independent buses or controllers
  - Overlap: do other useful work while waiting
  - Optimize the bottleneck to increase service rate
    » Use the queue to optimize the service
- Queues absorb bursts and smooth the flow
- Admissions control (finite queues)
  - Limits delays, but may introduce unfairness and livelock

Response Time = Queue + I/O device service time
When is Disk Performance Highest?

• When there are big sequential reads, or ....
• … when there is so much work to do so that they can be piggy backed (reordering queues—one moment)

• OK to be inefficient when things are mostly idle
• Bursts are both a threat and an opportunity
  – Treat: they can increase latency
  – Opportunity: enable piggyback (e.g., reordering of requests) & batching (e.g., one context switch to handle multiple requests*)

• Waste space for speed?

• Other techniques:
  – Reduce overhead through user level drivers (e.g., avoid context switching)
  – Reduce the impact of I/O delays by doing other useful work in the meantime
Disk Scheduling (1/3)

• Disk can do only one request at a time; What order do you choose to do queued requests?

User Requests

- FIFO Order
  - Fair among requesters, but order of arrival may be to random spots on the disk ⇒ Very long seeks

- SSTF: Shortest seek time first
  - Pick the request that’s closest on the disk
  - Although called SSTF, today must include rotational delay in calculation, since rotation can be as long as seek
  - Con: SSTF good at reducing seeks, but may lead to starvation
Disk Scheduling (2/3)

- Disk can do only one request at a time; What order do you choose to do queued requests?

  - SCAN: Implements an Elevator Algorithm: take the closest request in the direction of travel
    - No starvation, but retains flavor of SSTF

```
2,2 5,2 7,2 3,10 2,1 2,3
```

User Requests

![Diagram showing disk scheduling with user requests and head movement]

14 37 53 65 67 98 122 124 183
Disk Scheduling (3/3)

- Disk can do only one request at a time; What order do you choose to do queued requests?
  - C-SCAN: Circular-Scan: only goes in one direction
    - Skips any requests on the way back
    - Fairer than SCAN, not biased towards pages in middle
Recall: How do we Hide I/O Latency?

- **Blocking Interface:** “Wait”
  - When request data (e.g., read() system call), put process to sleep until data is ready
  - When write data (e.g., write() system call), put process to sleep until device is ready for data
- **Non-blocking Interface:** “Don’t Wait”
  - Returns quickly from read or write request with count of bytes successfully transferred to kernel
  - Read may return nothing, write may write nothing
- **Asynchronous Interface:** “Tell Me Later”
  - When requesting data, take pointer to user’s buffer, return immediately; later kernel fills buffer and notifies user
  - When sending data, take pointer to user’s buffer, return immediately; later kernel takes data and notifies user
Recall: I/O and Storage Layers

Application / Service

High Level I/O

Streams

File Descriptors
open(), read(), write(), close(), ...
Open File Descriptions

Low Level I/O

Syscall

Files/Directories/Indexes

File System

I/O Driver

Commands and Data Transfers

Disks, Flash, Controllers, DMA

What we covered in Lecture 4

What we will cover next…

What we just covered…
From Storage to File Systems

I/O API and syscalls

Variable-Size Buffer

Memory Address

Logical Index, Typically, 4 KB

File System

Block

Hardware Devices

Sector(s)

Phys Index., 4KB

Sector(s)

Flash Trans. Layer

HDD

SSD

Physical Index, 512B or 4KB

Phys. Block

Erasure Page
Building a File System

- **File System**: Layer of OS that transforms block interface of disks (or other block devices) into Files, Directories, etc.

- Classic OS situation: Take limited hardware interface (array of blocks) and provide a more convenient/useful interface with:
  - Naming: Find file by name, not block numbers
  - Organization:
    - File names in directories
    - Map files to blocks
  - Protection: Enforce access restrictions
  - Reliability: Keep files intact despite crashes, hardware failures, etc.
Recall: User vs. System View of a File

• User’s view:
  – Durable Data Structures

• System’s view (system call interface):
  – Collection of Bytes (UNIX)
  – Doesn’t matter to system what kind of data structures you want to store on disk!

• System’s view (inside OS):
  – Collection of blocks (a block is a logical transfer unit, while a sector is the physical transfer unit)
  – Block size ≥ sector size; in UNIX, block size is 4KBs
Translation from User to System View

• What happens if user says: “give me bytes 2 – 12?”
  – Fetch block corresponding to those bytes
  – Return just the correct portion of the block

• What about writing bytes 2 – 12?
  – Fetch block, modify relevant portion, write out block

• Everything inside file system is in terms of whole-size blocks
  – Actual disk I/O happens in blocks
  – read/write smaller than block size needs to translate and buffer
Disk Management

• Basic entities on a disk:
  – File: user-visible group of blocks arranged sequentially in logical space
  – Directory: user-visible index mapping names to files

• The disk is accessed as linear array of sectors
• How to identify a sector?
  – Physical position
    » Sectors is a vector [cylinder, surface, sector]
    » Not used anymore
    » OS/BIOS must deal with bad sectors
  – Logical Block Addressing (LBA)
    » Every sector has integer address
    » Controller translates from address ⇒ physical position
    » Shields OS from structure of disk
What Does the File System Need?

• Track which blocks contain data for which files
  – Need to know where to read a file from

• Track files in a directory
  – Find list of file's blocks given its name

• Track free disk blocks
  – Need to know where to put newly written data

• Where do we maintain all of this?
  – Somewhere on disk
Data Structures on Disk

• Data structure on disk different than data structures in memory

• Access a block at a time
  – Can't efficiently read/write a single word
  – Have to read/write full block containing it
  – Ideally want sequential access patterns

• Durability
  – Ideally, file system is in meaningful state upon shutdown
  – This obviously isn't always the case…
Announcements

• Midterm 2 Wednesday, 11/3, 7-9PM
  – Please see Piazza for more details, including proctoring guide, scope, and past exam threads.

• Project 2 deadline pushed to Sunday, 11/7!
FILE SYSTEM DESIGN
Critical Factors in File System Design

• (Hard) Disks Performance !!!
  – Maximize sequential access, minimize seeks

• Open before Read/Write
  – Can perform protection checks and look up where the actual file resource are, in advance

• Size is determined as files are used !!!
  – Can write (or read zeros) to expand the file
  – Start small and grow, need to make room

• Organized into directories
  – What data structure (on disk) for that?

• Need to carefully allocate / free blocks
  – Such that access remains efficient
Components of a File System

File path
Directory Structure

File number
"inode"

File Header Structure

"inumber"

Data blocks

One Block = multiple sectors
Ex: 512 sector, 4K block
Recall: Abstract Representation of a Process

Suppose that we execute `open("foo.txt")` and that the result is 3.

Next, suppose that we execute `read(3, buf, 100)` and that the result is 100.
Components of a File System

Open file description is better described as remembering the inumber (file number) of the file, not its name.

Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)
Components of a File System

- Open performs *Name Resolution*
  - Translates path name into a “file number”
- Read and Write operate on the file number
  - Use file number as an “index” to locate the blocks

- 4 components:
  - directory, index structure, storage blocks, free space map
How to get the File Number?

• Look up in *directory structure*

• A directory is a file containing `<file_name : file_number>` mappings
  – File number could be a file or another directory
  – Operating system stores the mapping in the directory in a format it interprets
  – Each `<file_name : file_number>` mapping is called a *directory entry*

• Process isn’t allowed to read the raw bytes of a directory
  – The *read* function doesn’t work on a directory
  – Instead, see *readdir*, which iterates over the map without revealing the raw bytes

• Why shouldn’t the OS let processes read/write the bytes of a directory?
### Directories

A screenshot of a macOS file browser showing a directory named `website` with various subdirectories and files. The browser window includes a search bar and a list of directories and files with their respective names, modification dates, sizes, and kinds.

#### Key Features:
- **Favorites** section with options like Dropbox, iCloud Drive, Google Drive, and others.
- **Devices** section showing Removable Disk, Shared, and other options.
- **Tags** section.
- **Files** section showing directories like `static`, `cas`, `exams`, and others.
- **Search** bar at the top of the window.

#### Example File List:
- **static**
  - `cas.pdf`
  - `exams.pdf`
  - `fonts.pdf`
  - `hw1.pdf`
  - `hw1_draft.pdf`
  - `hw2.pdf`
  - `hw3.pdf`

- **lectures**
  - `Assignments 2016.pdf`
  - `example_01.pdf`
  - `example_02.pdf`

- **projects**
  - `Project A.pdf`
  - `Project B.pdf`

- **readings**
  - `example_03.pdf`
  - `example_04.pdf`

#### Date and Time:
- The screenshot includes a date and time stamp at the bottom left corner: `11/2/21 12:39 PM`.
Directory Abstraction

- Directories are specialized files
  - Contents: List of pairs
    `<file name, file number>`

- System calls to access directories
  - `open` / `creat` / `readdir` traverse the structure
  - `mkdir` / `rmdir` add/remove entries
  - `link` / `unlink` (`rm`)

- libc support
  - `DIR * opendir (const char *dirname)`
  - `struct dirent * readdir (DIR *dirstream)`
  - `int readdir_r (DIR *dirstream, struct dirent *entry, struct dirent **result)`

```
/usr
    /usr/lib
    /usr/lib4.3
    /usr/lib4.3/foo
```

```
/usr
    /usr/lib
    /usr/lib4.3
    /usr/lib4.3/foo
```
Directory Structure

• How many disk accesses to resolve “/my/book/count”?
  – Read in file header for root (fixed position on disk)
  – Read in first data block for root
    » Table of file name/index pairs.
    » Search linearly – ok since directories typically very small
  – Read in file header for “my”
  – Read in first data block for “my”; search for “book”
  – Read in file header for “book”
  – Read in first data block for “book”; search for “count”
  – Read in file header for “count”

• Current working directory: Per-address-space pointer to a directory used for resolving file names
  – Allows user to specify relative filename instead of absolute path (say CWD=“/my/book” can resolve “count”)
In-Memory File System Structures

- Open syscall: find inode on disk from pathname (traversing directories)
  - Create “in-memory inode” in system-wide open file table
  - One entry in this table no matter how many instances of the file are open
- Read/write syscalls look up in-memory inode using the file handle
A Five-Year Study of File-System Metadata

NITIN AGRAWAL
University of Wisconsin, Madison
and
WILLIAM J. BOLOSKY, JOHN R. DOUCEUR, and JACOB R. LORCH
Microsoft Research

Published in FAST 2007
Observation #1: Most Files Are Small

Fig. 2. Histograms of files by size.
Observation #2: Most Bytes are in Large Files

Fig. 4. Histograms of bytes by containing file size.
CASE STUDY:
FAT: FILE ALLOCATION TABLE

• MS-DOS, 1977
• Still widely used!
FAT (File Allocation Table)

• Assume (for now) we have a way to translate a path to a “file number”
  – i.e., a directory structure
• Disk Storage is a collection of Blocks
  – Just hold file data (offset o = < B, x >)
• Example: file_read 31, < 2, x >
  – Index into FAT with file number
  – Follow linked list to block
  – Read the block from disk into memory
FAT (File Allocation Table)

- File is a collection of disk blocks
- FAT is linked list 1-1 with blocks
- File number is index of root of block list for the file
- File offset: block number and offset within block
- Follow list to get block number
- Unused blocks marked free
  - Could require scan to find
  - Or, could use a free list
FAT (File Allocation Table)

- File is a collection of disk blocks
- FAT is linked list 1-1 with blocks
- File number is index of root of block list for the file
- File offset: block number and offset within block
- Follow list to get block number
- Unused blocks marked free
  - Could require scan to find
  - Or, could use a free list
FAT (File Allocation Table)

- File is a collection of disk blocks
- FAT is linked list 1-1 with blocks
- File number is index of root of block list for the file
- File offset: block number and offset within block
- Follow list to get block number
- Unused blocks marked free
  - Could require scan to find
  - Or, could use a free list
- Ex: file_write(31, < 3, y >)
  - Grab free block
  - Linking them into file
FAT (File Allocation Table)

- Where is FAT stored?
  - On disk

- How to format a disk?
  - Zero the blocks, mark FAT entries “free”

- How to quick format a disk?
  - Mark FAT entries “free”

- Simple: can implement in device firmware
FAT: Directories

- A directory is a file containing `<file_name: file_number>` mappings
- Free space for new/deleted entries
- In FAT: file attributes are kept in directory (!!!)
  - Not directly associated with the file itself
- Each directory a linked list of entries
  - Requires linear search of directory to find particular entry
- Where do you find root directory (`"/"`)?
  - At well-defined place on disk
  - For FAT, this is at block 2 (there are no blocks 0 or 1)
  - Remaining directories
Suppose you start with the file number:
- Time to find block?
- Block layout for file?
- Sequential access?
- Random access?
- Fragmentation?
- Small files?
- Big files?
CASE STUDY:
UNIX FILE SYSTEM (BERKELEY FFS)
Inodes in Unix (Including Berkeley FFS)

• File Number is index into set of inode arrays
• Index structure is an array of *inodes*
  – File Number (inumber) is an index into the array of inodes
  – Each inode corresponds to a file and contains its metadata
    » So, things like read/write permissions are stored with *file*, not in directory
    » Allows multiple names (directory entries) for a file
• Inode maintains a multi-level tree structure to find storage blocks for files
  – Great for little and large files
  – Asymmetric tree with fixed sized blocks

• Original *inode* format appeared in BSD 4.1 (more following)
  – Berkeley Standard Distribution Unix!
  – Part of your heritage!
  – Similar structure for Linux Ext 2/3
Inode Structure
9 basic access control bits
- UGO x RWX
SetUID bit
- execute at owner permissions rather than user
SetGID bit
- execute at group's permissions
Small Files: 12 Pointers Direct to Data Blocks

Direct pointers
4kB blocks ⇒ sufficient for files up to 48KB

File Metadata

Indirect Pointers
Dbl. Indirect Ptr.
Tripl. Indirect Ptr.

Indirect Blocks

Double Indirect Blocks

Triple Indirect Blocks

Data Blocks

Fig. 2. Histograms of files by size.
Indirect pointers
- point to a disk block containing only pointers
- 4 kB blocks => 1024 ptrs
  => 4 MB @ level 2
  => 4 GB @ level 3
  => 4 TB @ level 4

Large Files: 1-, 2-, 3-level indirect pointers

A Five-Year Study of File-System Metadata
Putting it All Together: On-Disk Index

- Sample file in multilevel indexed format:
  - 10 direct ptrs, 1K blocks
  - How many accesses for block #23? (assume file header accessed on open)?
    » Two: One for indirect block, one for data
  - How about block #5?
    » One: One for data
  - Block #340?
    » Three: double indirect block, indirect block, and data
UNIX 4.2 BSD FFS

• Pros
  – Efficient storage for both small and large files
  – Locality for both small and large files
  – Locality for metadata and data
  – No defragmentation necessary!

• Cons
  – Inefficient for tiny files (a 1 byte file requires both an inode and a data block)
  – Inefficient encoding when file is mostly contiguous on disk
  – Need to reserve 10-20% of free space to prevent fragmentation
Conclusion

• File System:
  – Transforms blocks into Files and Directories
  – Optimize for access and usage patterns
  – Maximize sequential access, allow efficient random access

• File (and directory) defined by header, called “inode”

• Naming: translating from user-visible names to actual sys resources
  – Directories used for naming for local file systems
  – Linked or tree structure stored in files

• File Allocation Table (FAT) Scheme
  – Linked-list approach
  – Very widely used: Cameras, USB drives, SD cards
  – Simple to implement, but poor performance and no security

• Look at actual file access patterns
  – Many small files, but large files take up all the space!

• 4.2 BSD Fast File System: Multi-level inode header to describe files
  – Inode contains ptrs to actual blocks, indirect blocks, double indirect blocks, etc.
  – Optimizations for sequential access: start new files in open ranges of free blocks, rotational optimization