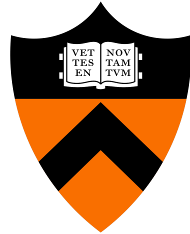
UNIX File System Naming



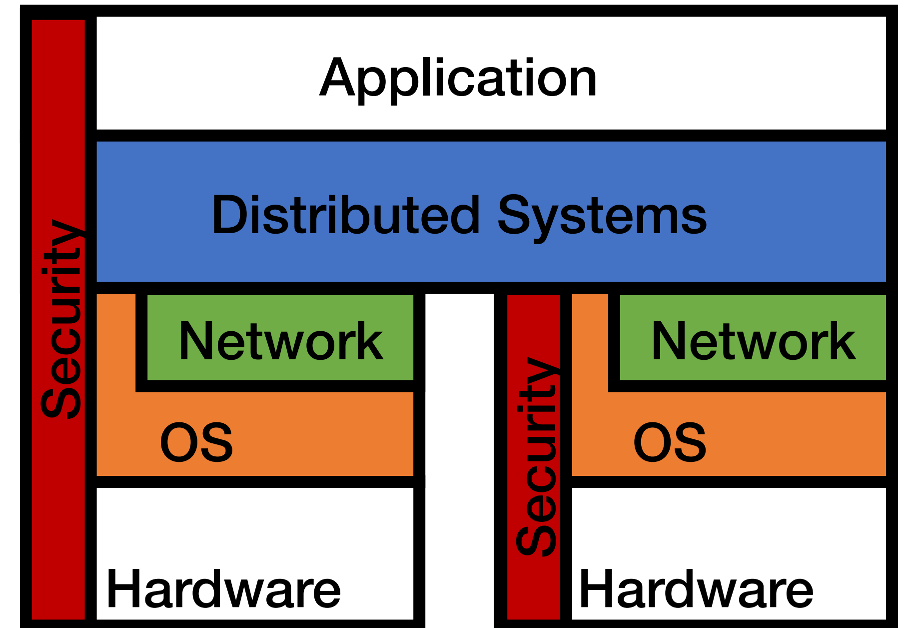
COS 316: Principles of Computer System Design

Lecture 4

Wyatt Lloyd & Rob Fish

Naming Module

- Naming Overview (today)
  - Memory Naming
    - OS, Security
- Unix File System Naming
  - OS
- Git Naming
  - OS, Distributed Systems
- Network Naming
  - Networking



Today's Goals

- Learn specifics of the Unix File System
- See 5 examples of naming within it
- Reason about portability, generality, and isolation
- Get first exposure to layering

Unix File System

- Unix and its descendants around since 1970s
    - (Named by Prof. Brian Kernighan)
- Descendants:
    - Linux
    - BSD
    - Mac OS X
    - iOS
    - Android

Unix File System

- The UNIX operating system’s API has remained relevant since the 1970s
- From “mini”-computers to todays rack-scale servers and personal devices alike!
- The UNIX file system has been even more influential and constant

Why File Systems?

- Common themes in UNIX systems:
  - User oriented
  - Multiple applications
  - Time sharing
- Need a way to store and organize persistent data
- **Key question:** how to let users organize and locate their data on persistent storage?

Key Abstraction

- Data is organized into “files”
  - A linear array of bytes of arbitrary length
  - Meta data about the bytes (modification and creation time, owner, permissions)
- Files organized into “directories”
  - A list of other files or sub-directories
- Common root directory named "/"

UNIX File System Layers

Layer	Purpose
Block	organizes persistent storage into fix-sized blocks
File	organizes blocks into arbitrary-length files
Inode number	names files as uniquely numbered inodes
Directory	human-readable names for files in a directory
Absolute path name	a global root directory

UNIX File System Layers

- For each of these we’ll look at:
  - Values
  - Names
  - Allocation mechanism
  - Lookup mechanism
- And ask:
  - How portable?
  - How general?
  - Can it isolate?

Block Layer

- Underlying resources differ
  - Tape has contiguous magnetic stripe
  - Disk has plates and arms
  - NAND flash (SSDs) even more complex to deal with wear leveling, data striping...
- **Values:** fix-sized “blocks” of contiguous persistent memory
- **Names:** integer block numbers

Block Layer: Allocation

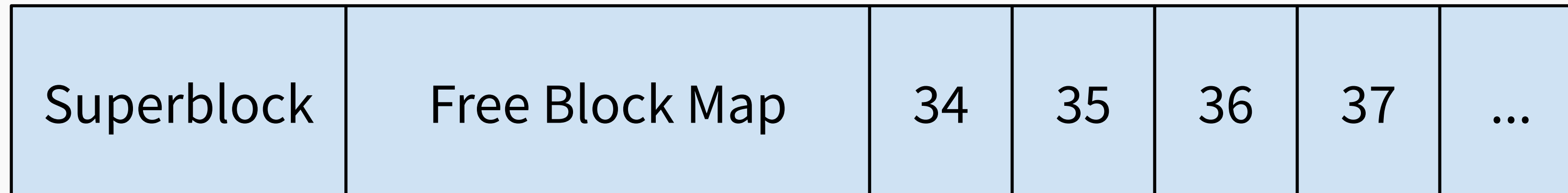
- Hardware specific, but let's just pretend our storage device is in-memory

```
typedef block uint8_t[4096]

# There is some hardware-specific translation from
# blocks to, e.g., plate number and offset
struct device {
  block blocks[N]
}
```

Block Layer: Allocation

- **Super Block:** a special block number to keep a bitmap of occupied blocks



Block Layer: Lookup

```
struct device {
  block blocks[N]
}

def (device *device) block_number_to_block(int32_t block_num) returns bloc
    return device.blocks[block_num + 1]
```

Block Layer: Portable? General? Isolation?

- How portable?
  - Can be (and has been!) implemented efficiently for most persistent storage media
    - Tape, HDDs, floppy disks, ... even network attached storage!
  - SSDs not a great fit due to need for wear leveling
    - Flash controllers are complex and obscure computers that hide flash behind block interface
- How general?
  - Lose some expressiveness: block size, performance characteristics
  - But not much
- Isolation?
  - Block numbers are global, they always represent the same physical location

File Layer

- A file is a linear array of bytes of arbitrary length:
  - May span multiple blocks
  - May grow or shrink over time

- How do we keep track of which blocks belong to which file?

- **Values:** arrays of bytes up to size N
- **Names:** References to inode structs
- **Allocation:** reuse block layer to store new inode structs in blocks

File layer: Lookup

```
struct inode {
  int32_t block_numbers[N];
  int32_t filesize
}

def (inode *inode) offset_to_block(int offset) returns block:
  block_idx = offset / BLOCKSIZE
  block_num = inode.block_numbers[block_idx]
  return device.block_number_to_block[block_num]
```

File layer: Portable? General? Isolation?

- How portable?
  - Can implement for any block device …
- How general?
  - Applications completely lose locality information
  - Fine for most applications, but not for specific use cases, e.g., databases
- Isolation?
  - A name always refers to particular data, so no inherent isolation

Inode number layer

- **Names:** Inode *numbers*
- **Values:** Inode structs
- **Allocation**
  - Can re-use block allocation and block numbers
- **Lookup**
  - if re-using block allocation:
  - $inode_number_to_inode \equiv block_number_to_block$

Recap so far

- Name files by inode number (e.g., 43982), translate to inode structs
- Inodes translate to a list of ordered block numbers that store the file's data
- Block numbers translate to blocks—the actual file data
- Given an inode number, we can get an ordered byte array.
- Remaining issues:
  - Numbers are convenient names for machines, but not for humans
  - How do we discover files?

Directory layer

- Structure files into collections called “directories”. Each file in a directory gets a human readable name—i.e., an (almost) arbitrary ASCII string
- **Names:** Human readable names within a “directory”
  - resume.docx
  - a.out
  - profile.jpg
- **Values:** Inode numbers
- Directories can contain files as well as other sub-directories

Directory layer: Allocation

```
struct dirent {
  char[MAX_NAME_LENGTH] filename;
  int    inode_number;
}

// Add type field to inode
struct inode {
  ...
  bool directory;
}

typedef directory inode; // Only when directory == true
```

Directory layer: Lookup

```
def (dir *directory) lookup(string filename) returns inode_number:
  for block_num in dir.block_numbers:
    directory = block_number_to_block(block_num) as struct dirent[]
    file_inode = directory.find(|dirent| dirent.filename == filename)
    if file_inode >= 0:
      return file_inode
  return -1
```

Directory layer: Lookup

Paths name files by joining directory and file names with /: path/to/file.txt

```
def (dir *directory) lookup(string path) returns inode_number:
  let (next_path, rest) = path.split_first('/')
  for block_num in dir.block_numbers:
    directory = block_number_to_block(block_num) as struct dirent[]
    if inode = directory.find(|dirent| dirent.filename == filename):
      if rest.empty():
        return inode
      else
        next_dir = block_number_to_block(inode)
        if !next_dir.directory: panic("Uh oh, can't traverse a file")
        return next_dir.lookup(rest as directory)
  return -1
```

Directory layer: Portable? General? Isolation?

- How portable?
  - Can implement for any inode & file layer—simply uses file layer for storage
- How general?
  - Assumes a hierarchical structure to file system.
  - Works poorly for relational or structured data
    - “please find all JSON files with the field foo”
- Isolation?
  - All lookups are relative to some base directory!
  - Can isolate applications by giving them different starting points
    - `$: man chroot`

Absolute path name layer

- Each running UNIX program has a “working directory” (wd)
- File lookups are relative to the wd
- What if we want to name files outside of our wd’s directory hierarchy?
  - E.g., share files between users
- What if we want globally meaningful paths?

Absolute path name layer

- Solution:
  - Special name /, hardcoded to a specific inode number
  - All directories are part of a global file system tree rooted at /
    - the “root” directory

- **Names:** One name, /
- **Values:** Hardcoded inode number, e.g., 2
- **Allocation:** nil
- **Lookup:** $\lambda_ \rightarrow 2$

Naming in UNIX File System: Recap

1. Absolute paths translate to paths starting from the “root” directory
2. Paths translate to recursive lookup for human-readable names in each directory
3. Human readable names translate to inode numbers
4. Inode numbers translate to inode structs
5. Inode structs translate to an ordered list of block numbers
6. Block numbers translate to blocks—the actual file data

Summary

- Learned specifics of the Unix File System
- Saw 5 examples of naming within it
- Reasoned about portability, generality, and isolation
- Got first exposure to layering