POSIX File Systems

COS 316 Lecture 6

Amit Levy

Figure 1: PDP-11/40 [3]

UNIX History: How did UNIX become a de-facto standard?

- 1970, “Unics” began as a rewrite of “Multics” (Multiplexed Information and Computer Services)
  - “Uniplexed Information and Computing Service”, a pun because original UNIX was single-tasking
  - Name credit: Brian Kerninghan!
- Developed at first by Bell Labs, first release was “Research Unix” to UIUC
- Mostly used in research until the late 1980’s
- 1990s: Linux and BSD offshoots (after copyright dispute settled in 1994)
  - Linux serves a primary role for server based web applications in the 90’s dot-com boom
- 2000: Apple uses BSD as basis for Darwin, core of OS X and iOS
- BSD (Berkeley Software Distribution) and System V UNIX co-developed over years

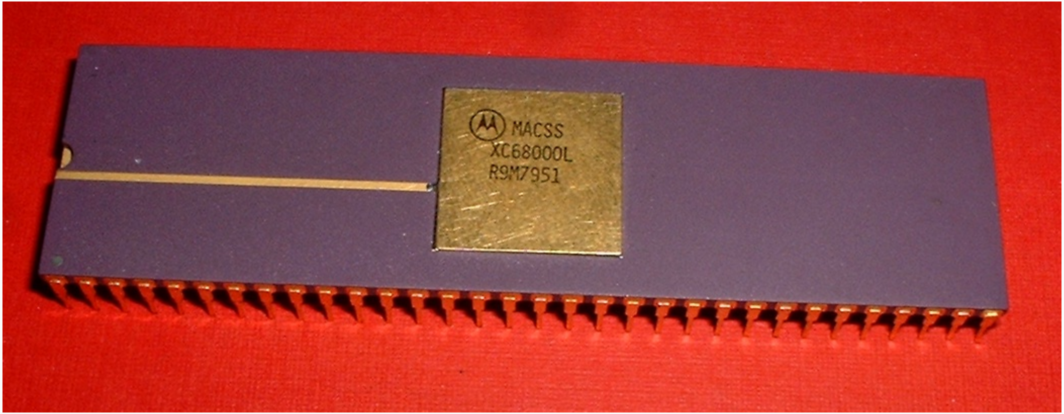


Figure 2: Motorola 68000 [2]

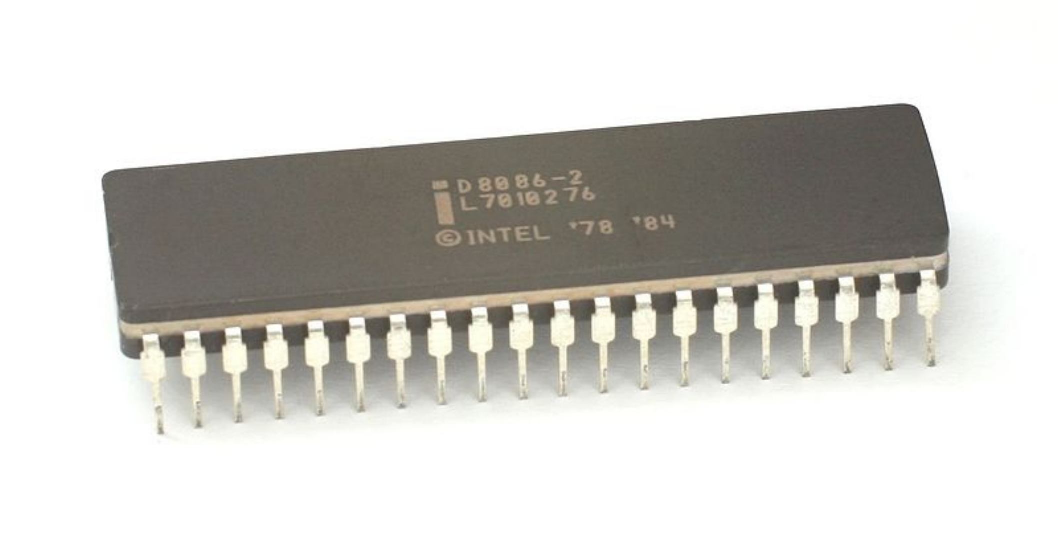


Figure 3: Intel 8086 [1]

Why File Systems?

- Common themes in UNIX systems:
  - User oriented
  - Multiple applications
  - Time sharing
- Need a way to store and organize persistent data
  - Other ways we might organize persistent data?

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
  - Contrast with drive letters in Windows

UNIX File System Layers

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

UNIX File System Layers

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

- For each of these we'll look at:
    - Names
    - Values
    - Allocation algorithm
    - Translation algorithm

Block layer

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

Block layer

```
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

```
struct super_block {
  int32_t total_size
  int32_t free_block_map_block_num
}

def (device *device) allocate_new_block() returns block_number:
  superblock = (device[SUPERBLOCK_INDEX] as super_block
  for i, b in device[superblock.fre_block_map_block_num]:
    if b != 0xffff:
      empty_block_bit = b.lowest_zero_bit()
      b |= 1 << empty_block_bit
      return i * 8 + empty_block_bit
```

Block layer: Translation

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

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

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?

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?

Names: Inode structs

Values: Files, arrays of linear bytes

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?

Names: Inode structs

Values: Files, arrays of linear bytes

Reuse block allocation for inode allocation

File layer

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

File layer

```
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]
```

```
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]
```

Block size: 4096

int32_t → 4 bytes

what is N?

4B bytes ≈ 4GB

What's the maximum file size this scheme can support?

N * Blocksize = max file size

4096 → 4092 / 4 ≈ 1023 · 4096 ≈ 4M

File layer

```
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]
```

What’s the maximum file size this scheme can support?

$(4096-4)/4 * 4k \approx 4MB$

What's a scheme that would allow larger files?

1. Spn more block per inode: 1→32

2. Change the struct

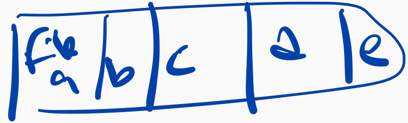
↳ Linked list of 'inodes'

hash table?

tree

inode {
int32 direct[N-M]
int32 indirect[M]

3 ↳



Inode number layer

- Names: Inode *numbers*
- Values: Inode structs

Inode number layer

- Names: Inode *numbers*
- Values: Inode structs
- Allocation
  - Can re-use block allocation and block numbers
  - File systems often use special inode allocation to avoid slow seeks on disk for common operations
- Translation
  - 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

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

Remaining issues:

1. Numbers are convenient names for machines, but not so much for humans
2. 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 {
  string filename
  int    inode_number
}

struct inode {
  int32_t filesize
  bool directory
  union {
    int32_t block_numbers[N]
    dirent  files[M]
  }
}
```

Directory Layer: Translation

```
typedef directory inode when inode.directory

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

Directory Layer: Translation

Paths name files by concatenating directory and file names with /: `path/to/a/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:
    block = block_number_to_block(block_num) as files[]
    if inode = files.find(next_path):
      if rest.empty():
        return inode
      else
        next_dir = block_number_to_block(inode) as directory
        return next_dir.lookup(rest)
  return -1
```

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., 1

Allocation: nil

Translation: $\lambda\_ \rightarrow 1$

Naming in UNIX File System: Recap

1. Absolute paths translate to paths starting from the “root” directory

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

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

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

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

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

Up Next

- Problems with location-addressed naming (e.g. UNIX file system)
  - Transactions
  - Versioning
  - Data corruption
- We'll look at Git's content addressable store
- Please read chapter 10 of the Git book: Git Internals

Problem set 1 due Thursday

Assignment 2 due next Tuesday

References

[1] *Intel 8086*. Wikimedia Commons.

[2] *Motorola 68000 microprocessor, pre-release version XC68000L with R9M mask*. Wikimedia Commons.

[3] *PDP11/40 as exhibited in Vienna Technical Museum*. Wikimedia Commons.