UNIX File Systems

COS 316: Principles of Computer System Design

Amit Levy & Jennifer Rexford

A Brief History of UNIX

A Brief History of UNIX

Figure 1: PDP-11/40 @pdp1140

A Brief History of UNIX: 1970s

- Developed at AT&T Bell Labs following demise of the “Multics” project
- “Unics” began as a rewrite of “Multics” (Multiplexed Information and Computer Services)
  - “Uniplexed Information and Computing Service”, because early versions were single-tasking
  - Naming credit: Prof. Brian Kernighan
- Berkeley Software Distribution (BSD) follows Ken Thompson’s sabbatical at UC Berkeley

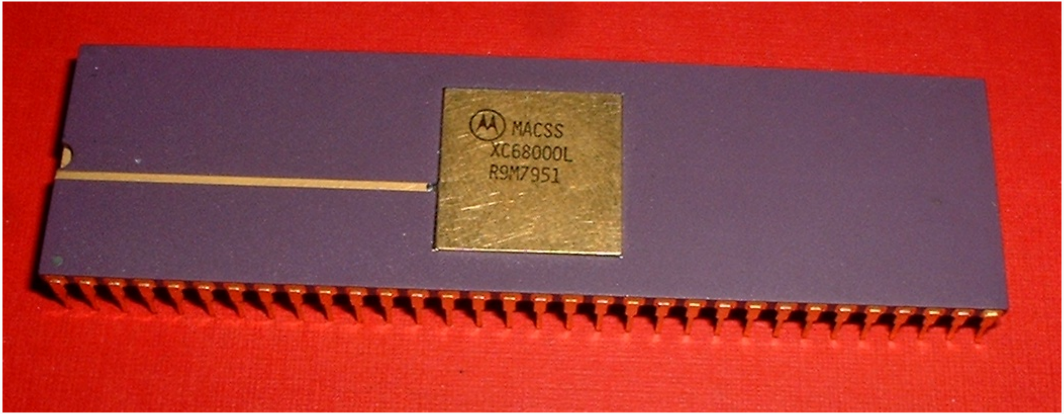

Figure 2: Motorola 68000 @motorola68000

A Brief History of UNIX: 1980s

- AT&T free to sell computers after Bell Systems breakup
    - AT&T UNIX versions turn proprietary
- Flurry of non-AT&T UNIX variants
    - Academic: Minix, Mac microkernels
    - GNU – “free” alternative to UNIX
    - NeXTStep (OS X predecessor), SunOS, Xenix

A Brief History of UNIX: 1990s & Beyond

- BSD rewritten following copyright claims, emerges as various offshoots
    - (FreeBSD, NetBSD, OpenBSD, DragonflyBSD, ...)
- Linux + GNU, fill void during BSD copyright dispute
- Apple uses NeXTSTEP & BSD as basis for OS X
- Android, iOS

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 `"/"`
  - Contrast with drive letters in Windows

UNIX File System Layers

Block layer	organizes persistent storage 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

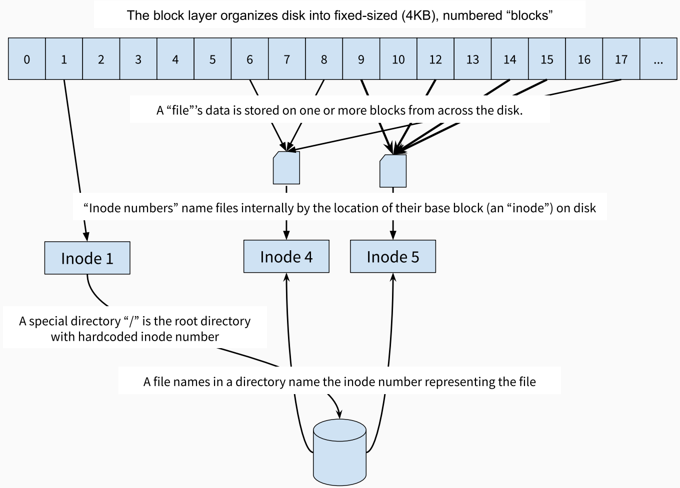

Figure 3: The UNIX File System’s Naming Hierarchy

UNIX File System Layers

For each of these we'll look at:

- Values
- Names
- Allocation mechaniem
- Lookup mechanism

And ask:

- How portable?
- How general?
- Can it isolation?

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

```
struct super_block {
  int32_t total_size
  int32_t free_block_map
}
```

Superblock	Free Block Map	34	35	36	37	...

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

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

Block layer: Portable? General? Isolation?

How portable?

Block layer: Portable? General? Isolation?

How portable?

- Can be (and has been!) implemented efficiently for most persistent storage media
  - Tape, HDDs, floppy disks, optical drives... 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

Block layer: Portable? General? Isolation?

How portable?

- Can be (and has been!) implemented efficiently for most persistent storage media
  - Tape, HDDs, floppy disks, optical drives… 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?

Block layer: Portable? General? Isolation?

How portable?

- Can be (and has been!) implemented efficiently for most persistent storage media
  - Tape, HDDs, floppy disks, optical drives... 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

Block layer: Portable? General? Isolation?

How portable?

- Can be (and has been!) implemented efficiently for most persistent storage media
  - Tape, HDDs, floppy disks, optical drives... 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 layer: Portable? General? Isolation?

How portable?

- Can be (and has been!) implemented efficiently for most persistent storage media
    - Tape, HDDs, floppy disks, optical drives... 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?

- Could have different meanings for the same block number
- But typically no: the block number usually expresses a particular physical location
- Still, isolation at entire drive or partition
    - E.g. virtual machines may only get access to some block devices from hypervisor

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?

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: References to inode structs

Values: arrays of bytes up to size `N`

Allocation: reuse block layer to store new inode structs in blocks

File Layer

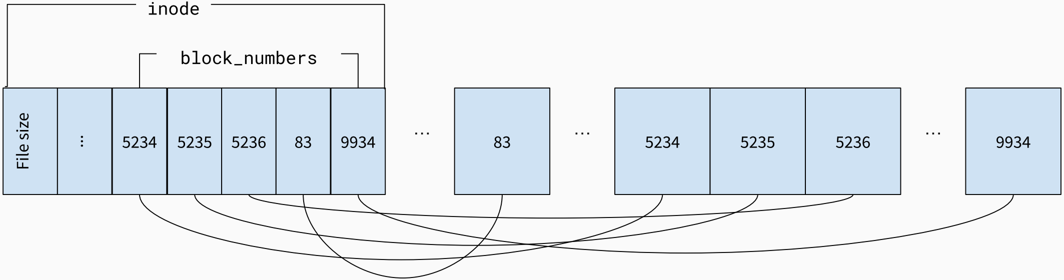

Figure 4: The inode struct is stored in a block and points to blocks containing file data

File layer: Lookup

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

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: 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]
```

What's the maximum file size this scheme can support? Assume `BLOCKSIZE == 4KiB`

```
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? Assume `BLOCKSIZE == 4KiB`

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

File layer: Portable? General? Isolation?

How portable?

File layer: Portable? General? Isolation?

How portable?

- Can implement for any block device
- Can *also* implement for other kinds of devices (e.g. non-block networked storage) . . .

How general?

File layer: Portable? General? Isolation?

How portable?

- Can implement for any block device
- Can *also* implement for other kinds of devices (e.g. non-block networked storage) . . .

How general?

- Applications completely lose locality information
- Fine for most applications, but not for specific use cases, e.g., databases

File layer: Portable? General? Isolation?

How portable?

- Can implement for any block device
- Can *also* implement for other kinds of devices (e.g. non-block networked storage) . . .

How general?

- Applications completely lose locality information
- Fine for most applications, but not for specific use cases, e.g., databases

Isolation?

File layer: Portable? General? Isolation?

How portable?

- Can implement for any block device
- Can *also* implement for other kinds of devices (e.g. non-block networked storage) . . .

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 here.

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
- 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 a inode number, we can get an ordered byte array.

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 a inode number, we can get an ordered byte array.

Remaining issues:

1. Numbers are convenient names for machines, but not 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

Figure 5: A directory is a special type of file that maps filenames to inode numbers

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 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:
    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, you tried to traverse a file")
        return next_dir.lookup(rest as directory)
  return -1
```

Directory layer: Portable? General? Isolation?

How portable?

Directory layer: Portable? General? Isolation?

How portable?

Can implement for any inode & file layer—simply uses file layer for storage

Directory layer: Portable? General? Isolation?

How portable?

Can implement for any inode & file layer—simply uses file layer for storage

How general?

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 struture to file system.
- Works poorly for relational or structured data ("please find all YAML files wit the field `foo`")
  - Alternate approaches: relational model: WinFS, GNOME Storage (both defunct) . . .

Isolation?

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 struture to file system.
- Works poorly for relational or structured data (“please find all YAML files wit the field `foo`”)
    - Alternate approaches: relational model: WinFS, GNOME Storage (both defunct) . . .

Isolation?

- All lookups are relative to some base directory!
- Can isolate applications by giving them different starting points (e.g. working directory)

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

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

References