File Systems

Part 1

Charts: Augmented from MIT's Adam Belay
https://pdos.csail.mit.edu/6.828/2023/lec/l-fs1.pdf

File Systems

- Provide persistent storage across restarts and crashes
- Provide naming and organization
- Provide sharing of data among users and processes

File System Attributes that are interesting to study/solve

- Crash recovery
- Performance + concurrency
- Sharing + security
- Powerful abstractions (e.g., proc, afs, 9P, pipes, etc.)

File System Software Layers

System Calls

Names and File Descriptors

Inodes

Inode Cache and Buffer Cache

Log

Virtual Disk Driver (QEMU)

Today

High-level design choices in system calls

- Objects: Use files (not virtual disks or databases)
- Content: Use byte arrays (not structured)
- Naming: Human-readable (not ID numbers)
- Organization: Name hierarchy
- Synchronization: None (no locking, no versions)
- link() and unlink() can change names concurrently w/ open()

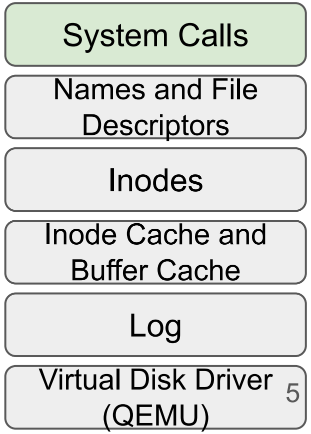
System Call Layer

```
fd = open("x/y", flags); // creates a
fd
write(fd, "abc", 3);     // wr 3 bytes
link("x/y", "x/z");      // create link
unlink("x/y");           // removes x/y
write(fd, "def", 3);     // wr 3 more
bytes
close(fd);               // close the
fd
```

File `x/z` contains `abcdef`

See file user/user.h.

```
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
```



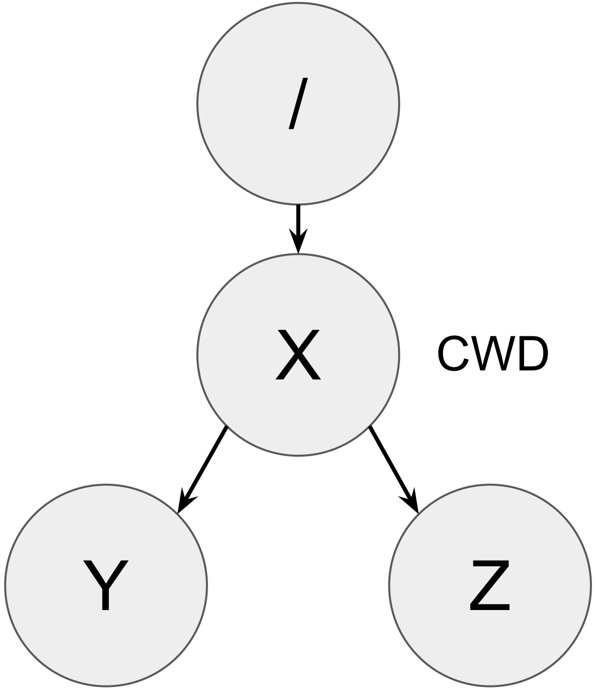
Linux File System Call API

https://www.gnu.org/software/libc/manual/html_node/File-System-Interface.html

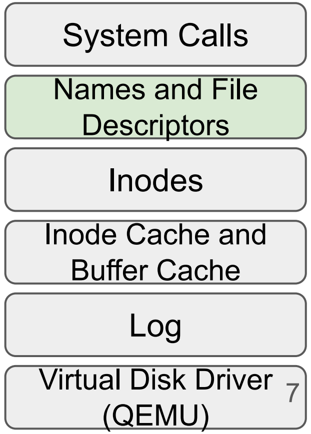
```
int fd = open(char *path, int flag, mode_t mode)
int close(int fd)
ssize_t read(int fd, void *buf, size_t nbyte)
ssize_t write(int fd, void *buf, size_t nbyte)
int fsync(int fd)
off_t lseek(int fd, off_t offset, int whence)
int rename(const char *old_filename, const char *new_filename)
int stat(const char *path,struct stat *statbuf)
int fstat(int fd, struct stat *statbuf)
int dup(int old) - copies old to first available fd
int dup2(int old, int new) - copies old to new, replacing new
int link(const char *oname, const char *nname)
int symlink(const char *oname, const char *nname)
int unlink(const char *pathname)
int mkdir(const char *path, mode_t mode)
int chdir(const char *path)
int fchdir(int filedes)
char *getcwd(char *buffer, size_t size)
DIR *opendir(char *dirname)
struct dirent *readdir(DIR *dirp)
```

- System Calls
- Names and File Descriptors
- Inodes
- Inode Cache and Buffer Cache
- Log
- Virtual Disk Driver (QEMU)

Name Layer

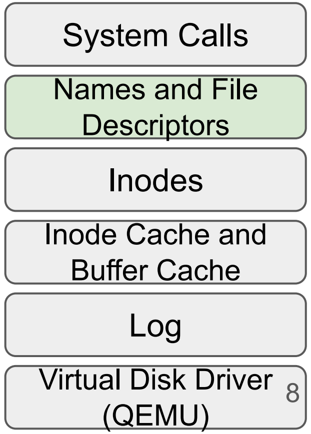


- Path names are organized as a tree
- No cycles, but multiple names can refer to the same file (i.e., via link())
- Processes share the namespace
- But each process has a current working directory (CWD)
- Absolute path: /x/y
- Relative path: x/y

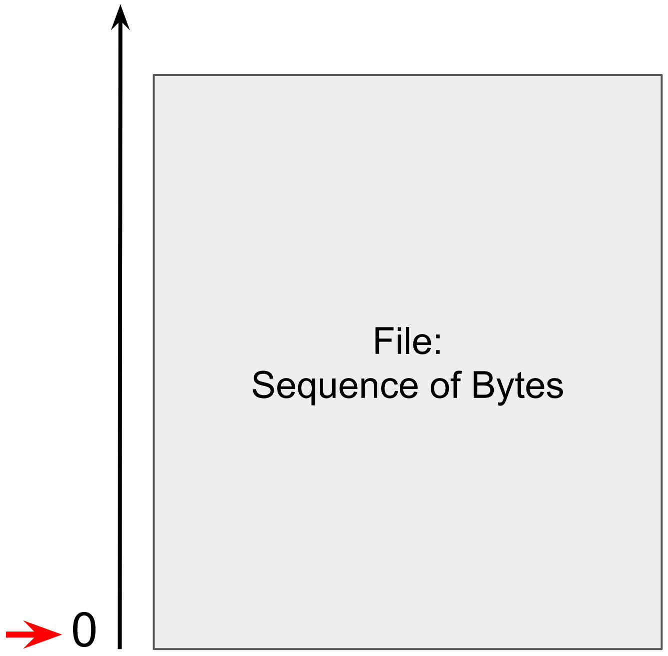


File Description Layer

- Each process has its own FD number namespace
- Each FD identifies a file created by open()
- By convention STDIN, STDOUT, STDERR are file descriptors 0, 1, and 2
- Lowest available FD number is allocated during open()
- FD returned during open() is usable by the process even if the file is unlinked (i.e., deleted)
- A file is an object that you can read and write to like a stream

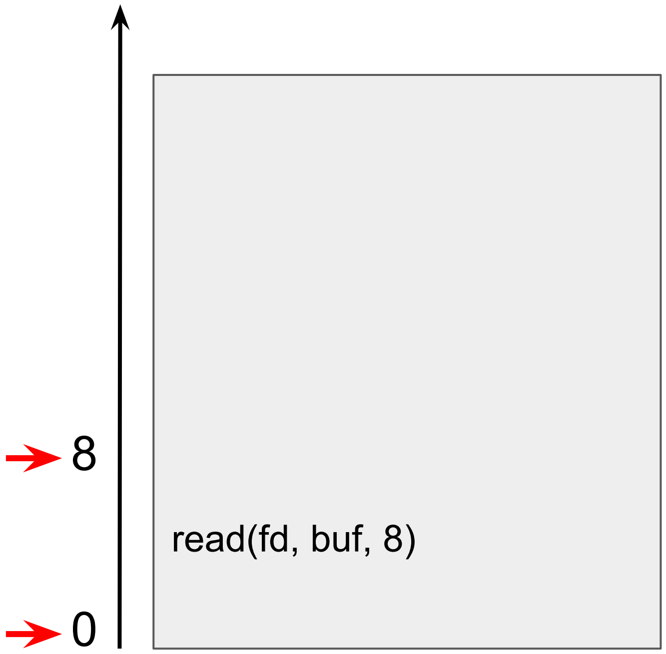


Interacting with a file



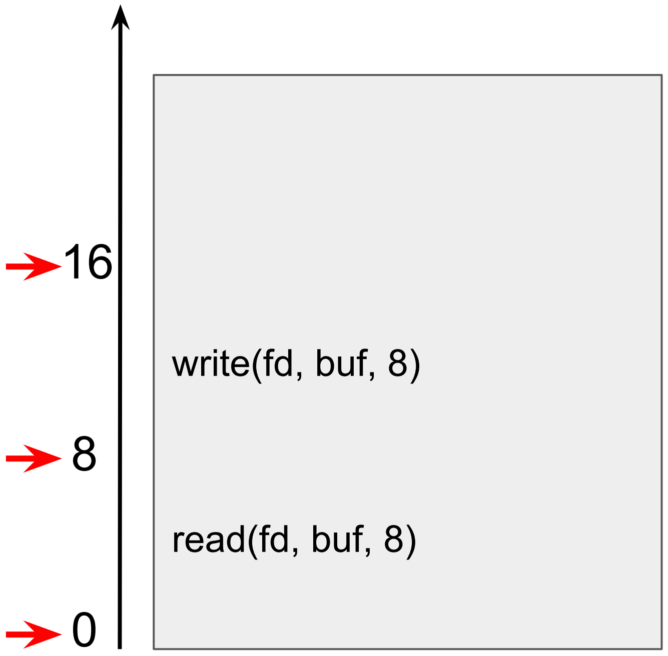
- FDs access file as an array of bytes, very similar to an address space
- Each FD has a “cursor” to the file
- An empty file cursor starts at byte 0

Interacting with a file



- FDs access file as an array of bytes, very similar to an address space
- Each FD has a “cursor” to the file
- read() advances the cursor

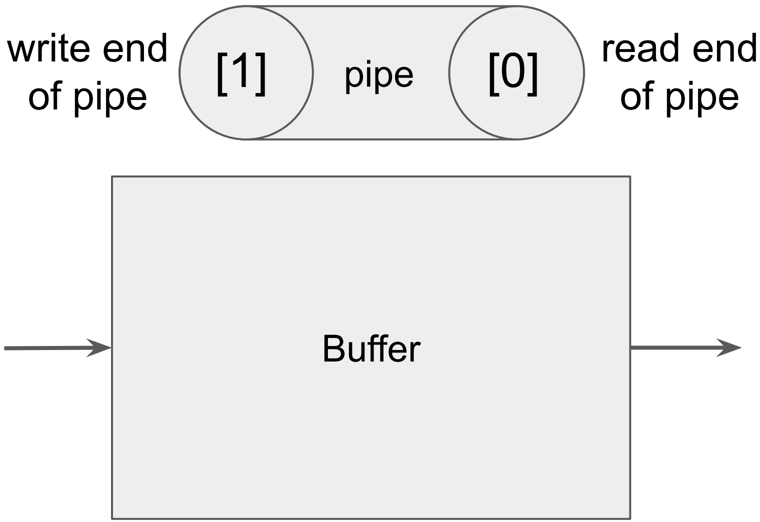
Interacting with a file



- FDs access file as an array of bytes, very similar to an address space
- Each FD has a “cursor” to the file
- read() advances the cursor
- write() advances the cursor

Pipe Files are a bounded buffer

```
int fds[2];
pipe(fds);
```



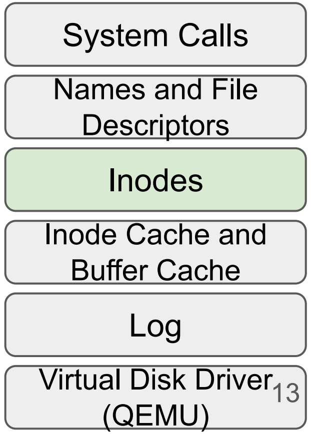

```
write(fds[1],"Hello", 5);
```

```
unsigned char buf[5];
read(fds[0],buf, 5);
```

- read blocks when the buffer is empty
- write blocks when the buffer is full
- Multiple processes can read/write from/to a pipe

Inode Layer

- **Inode:** Records the details of a file
  - Tracks the size of the file and where on the disk the data is stored
  - Has a link count (and open FD count) to figure out when to free
  - Deallocation deferred until link + open count is zero
- **I-number:** Refers to an inode, similar to an FD
  - Uniquely identifies a position on disk where Inode is located



Where is Data Stored

- On a persistent storage medium
- Persistent - data doesn't go away under loss of power
- Common storage mediums
  - HDDs: High capacity, slow, inexpensive
  - SSDs: Lower capacity, faster, more expensive
- Disks accessed in fixed-sized units (like pages)
  - Called sectors, historically 512 bytes
  - Newer drives use 4K (4096) byte sectors

Performance Characteristics

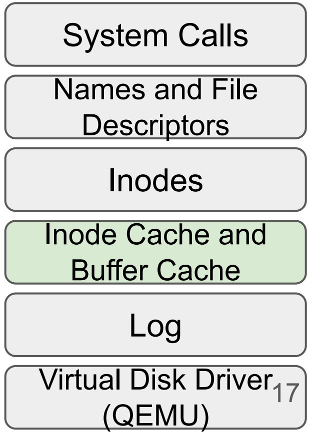
- Applies to both HDDs and SSDs, but more so the HDDs
- Sequential access much faster than random
- Big reads/writes much faster than small ones
- Both facts influence FS design

Disk Blocks

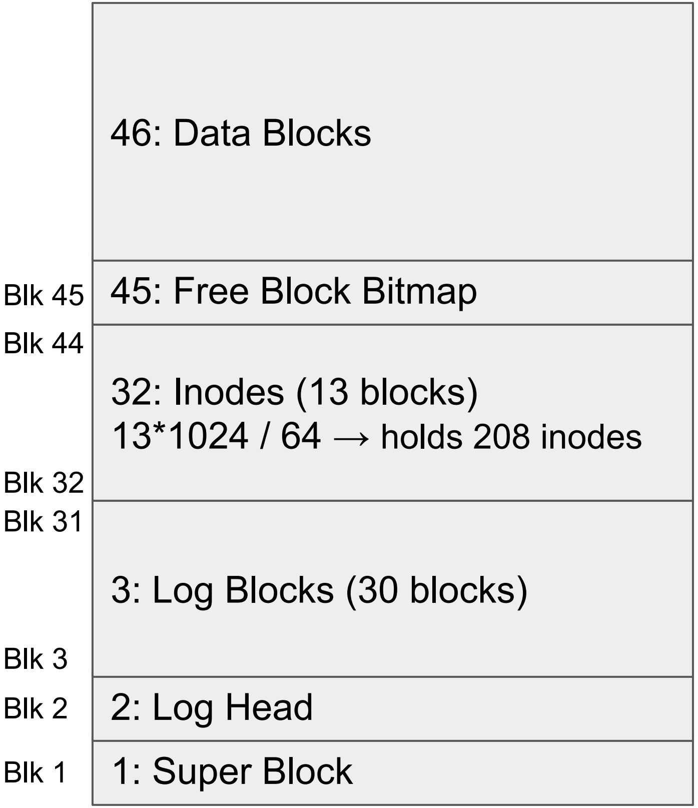
- Disk blocks are allocated to files
- Typically, multiple sectors are combined to form blocks
- e.g., a 4KB block is 8 512-byte sectors
- Combining sectors helps reduce book-keeping and seek overhead
- Xv6 uses two 512-byte sector blocks
- Every block has a block number
- Block number is like an address that identifies the location on the disk

Inode Cache and Buffer Cache Layer

- Disk accesses are slow and random
- Store copies of inodes and blocks in RAM
- Works well because the same data is often accessed many times
- e.g., the same inodes and blocks are accessed each time a file is read
- No need to access the disk if a copy is available!



Xv6 Disk Layout



- Xv6 provides mkfs program
- Generates this layout for a new (empty) FS
- The layout is static for the lifetime of the FS
- mkfs macros
  - #define NINODES 200
  - #define BSIZE 1024
  - #define FSSIZE 2000 ← blocks on disk
  - #define IPB (BSIZE / sizeof(struct dinode))
  - int nbitmap = FSSIZE/(BSIZE*8) + 1;
  - int ninodeblocks = NINODES / IPB + 1;
  - int nlog = LOGSIZE;
- File System Metadata
  - Everything other than file content
  - Super block, inodes, bitmap, directory content

Inodes

Disk Inode

```
struct dinode { // disk inode
  short type;   // File type: dir, file
  short major;  //
  short minor;  //
  short nlink;  // # links to inode
  uint size;    // file sz (bytes)
  uint addrs[NDIRECT+1]; // Data blocks
}

NDIRECT is 12
sizeof(struct dinode) is 64 bytes
2 + 2 + 2 + 2 + 4 + (13 * 4)
16 dinodes per disk block
```

Memory Inode

```
struct inode { // mem copy of inode
  uint dev;     // device number
  uint inum;    // inode number
  uint ref;     // refs to inode
  struct sleeplock lock // prot all below
  int valid;    // inode read from disk
  short type;   // File type: dir, file
  short major; //
  short minor; //
  short nlink; // # links to inode
  uint size;    // file sz (bytes)
  uint addrs[NDIRECT+1]; // Data block
}
```

- What are some file attributes missing from these inodes?

System Calls

Names and File Descriptors

Inodes

Inode Cache and Buffer Cache

Log

Virtual Disk Driver (QEMU)

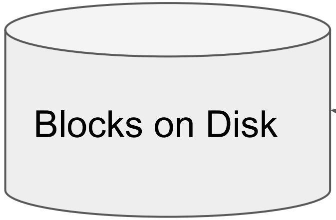
```
int fd = open("file.txt", , );
```

```
struct proc {
  struct spinlock lock;
  // p->lock held for these
  enum procstate state;        // Process state
  void *chan;                  // sleep channel
  int killed;                  // !=0, killed
  int xstate;        // exit status returned to parent
  int pid;                     // Process ID
  // wait_lock held for this
  struct proc *parent;         // Parent proc
  // p->lock not needed for these private
  uint64 kstack;               // kernel stack for proc
  uint64 sz;                   // Size proc mem (bytes)
  pagetable_t pagetable;       // user pagetable
  struct trapframe *trapframe; // data for trampoline
  struct context *context;     // swtch()
  struct file *ofiles[NFILE]; // Open files
  struct inode *cwd;           // Current dir
  char name[16];               // Process name
};
```

```
struct file {
  enum {FD_NONE, FD_PIPE,
        FD_INODE, FD_DEVICE } type;
  int ref;              // ref count
  char readable;
  char writable;
  struct pipe *pipe; // FD_PIPE

  struct inode *ip;  // FD_INODE
  uint off;    // rd/wr spot
  short major // FD_DEVICE
};
```

```
struct inode { // mem copy of inode
  uint dev;     // device number
  uint inum;    // inode number
  uint ref;     // refs to inode
  struct sleeplock lock // prot all below
  int valid;    // inode read from disk
  short type;   // File type: dir, file
  short major; //
  short minor; //
  short nlink; // # links to inode
  uint size;    // file sz (bytes)
  uint addrs[NDIRECT+1]; // Data blocks
}
```

```
struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```



On Disk Inode Layout

- type: Free, file, directory, or device
- nlink: number of links
- size: the size of the file in bytes
- addrs: addresses of data blocks (array)
- Example: Find file’s byte at 4000
  - 4000 / BSIZE (=1024) = 3;
  - addrs[3] contains the block number (659) with data byte at 4000

```
struct dinode { // disk inode
  short type;   // File type: dir, file
  short major; //
  short minor; //
  short nlink; // # links to inode
  uint size;   // file sz (bytes)
  uint addrs[NDIRECT+1]; // Data blocks
}

NDIRECT is 12
sizeof(struct dinode) is 64 bytes
2 + 2 + 2 + 2 + 4 + (13 * 4)
16 dinodes per disk block
```

	addrs[0]	addrs[1]	addrs[2]	addrs[3]
addrs	Block 421	Block 52	Block 7732	Block 659

Inode Fixed Size - File Sizes

- How can we fit large files into addrs?
- Use indirect block: a full block of more addrs
- Xv6 has one indirect block
- You can let addrs be all indirect blocks

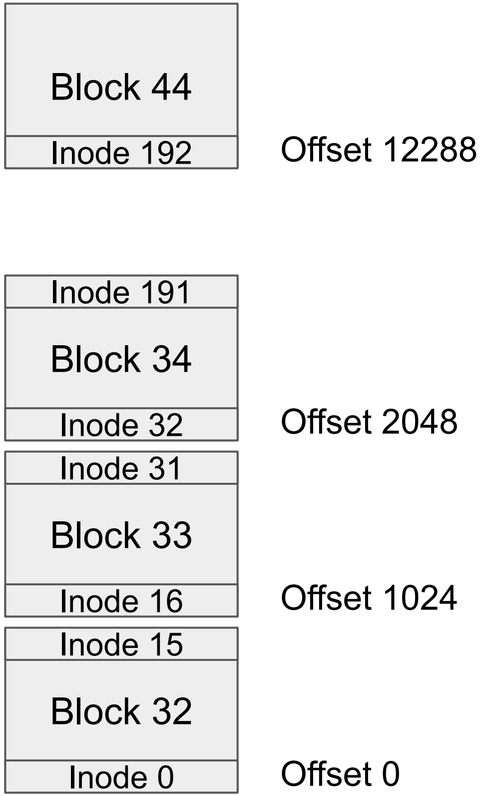
```
struct dinode { // disk inode
  short type;   // File type: dir, file
  short major; //
  short minor; //
  short nlink; // # links to inode
  uint size;   // file sz (bytes)
  uint addrs[NDIRECT+1]; // Data blocks
}

NDIRECT is 12
sizeof(struct dinode) is 64 bytes
2 + 2 + 2 + 2 + 4 + (13 * 4)
16 dinodes per disk block
```

	NDIRECT[0]	NDIRECT[1]	NDIRECT[2]		
Indirect Block: 73	Block 221	Block 572	Block 765	...	256 additional blocks

	addrs[0]	addrs[1]	addrs[2]		addrs[NDIRECT]
addrs	Block 421	Block 52	Block 7732	...	Block 73

Converting an Inum to an Inode

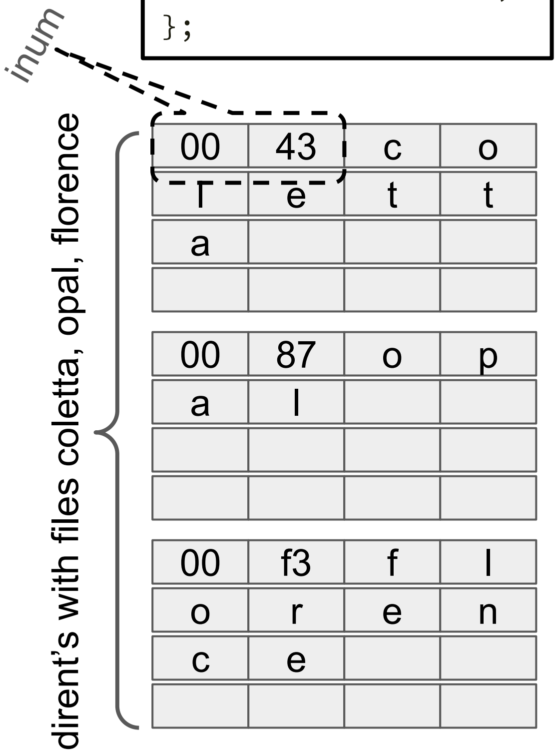
- Two-hundred Dinodes on disk blocks 32 through 44
- i-number is used to select a Dinode
- Each Dinode is 64 bytes long
- If you read the Dinode blocks into memory
- Byte Offset of Inode(i-number) = 64*i-number
- Directories contain data struct for files
- Directory data struc has i-numbers
- Use i-num to get Dinode
- Use Dinode to get file
- We use both Dinodes and Inodes



Directories

```
struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

- A directory is a file.
- Directory has a name and blocks with data
- Users can not directly write contents
- Content is an array of dirents. Each direct:
  - i-number (of the file in the directory)
  - 14-byte file name (Xv6 filenames are old-school)
  - dirent is free (unused) if inum == 0
  - sizeof(struct dirent) is 16 bytes
  - A 1024 byte block holds 64 struct dirent



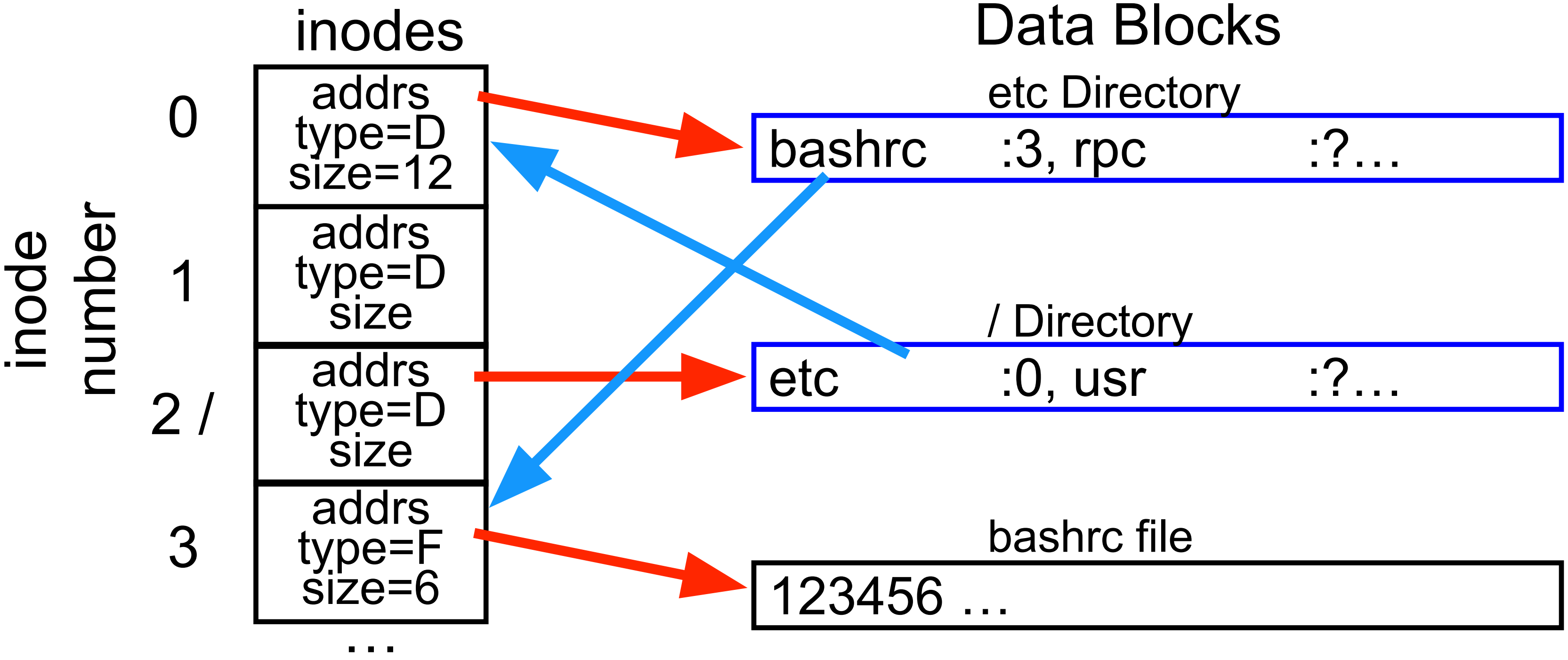
On Disk Structure is a Tree

- Layer 1: Directory
- Layer 2: Inodes
- Layer 3: Disk Blocks
- Layer 4: Disk Sectors

Pools of information to Allocate

- Inodes
- Blocks

read **/etc/bashrc**



Reads for traversing /etc/bashrc

read root dir (inode and data)
read etc dir (inode and data)
read bashrc file (inode and data)

```
struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```