

# CS 137: File Systems

## Dealing With the Block Interface

# The Kernel “API”

Request is described by `struct bio`:

**bi\_sector** Starting sector, 512-byte unit

**bi\_bdev** (Block) device to do I/O on

**bi\_rw** Read or write

**bi\_size** Size in bytes (not sectors!)

**bi\_iovec** Complex description of where data is in memory

Despite size in bytes, writes must be integral number of hardware blocks

In practice, reads are always integral blocks, too

# The Unix API

FUSE clients use Unix I/O:

**read(fd, buf, n)** Read **n** bytes from current position of file descriptor **fd** into memory at address **buf**

**write(fd, buf, n)** Write **n** bytes to current position of file descriptor **fd** out of memory at **buf**

**lseek(fd, pos, SEEK\_SET)** Set current position of file descriptor **fd** to **pos** (bytes)

**Important:** **fd** can be connected to a Unix file or an actual device. We will normally use files to represent devices.

## Simulating the Kernel

Best practice is to wrap the Unix API:

`read_block(block, buf` Performs `lseek` to correct byte offset, then `reads`

`write_block(block, buf` Performs `lseek` to correct byte offset, then `writes`

Code should never assume anything about current position in file.

Up to you whether wrappers accept a “device” identifier (the `fd`) or it's hard-wired.

# Overview

FUSE client code typically has three parts:

1. Declarations of constants and data structures
  - ▶ Latter are critically important!
2. Code to implement operations
3. `main` function and initializer to get stuff started

Biggest problem for novices is dealing with block I/O interface

# The Stupid Filesystem

- ▶ Serves as example of how things are done
- ▶ Limit of 100 files & directories
- ▶ Small limit on file size
- ▶ Inflexible on-disk layout
- ▶ No reuse of deleted space!

# Stupid Filesystem Layout



- ▶ Superblock contains meta information
- ▶ Fixed-size files immediately follow superblock
- ▶ Can find everything by indexing

# Constants, Types, and Globals

```
#define STUPID_MAGIC_BIG_ENDIAN 0x7374757069642121L
#define STUPID_MAGIC_LITTLE_ENDIAN 0x2121646970757473L

#define BLOCK_SIZE      4096
#define BLOCKS_PER_FILE 100      /* Maximum file size, in blocks */
#define MAX_FILES       100      /* Maximum number of files supported */
#define DISK_SIZE       ((1 + BLOCKS_PER_FILE * MAX_FILES) * BLOCK_SIZE)

typedef size_t          block_t; /* Block-address type */

static int              backing_file_fd;
                        /* Fd for all access to backing file */
```



# Superblock

```
struct sblock {
    unsigned long    magic;           /* Magic # identifying filesystem */
    size_t          total_blocks;     /* Total blocks (disk size) */
    size_t          block_size;      /* Size of each block */
    size_t          blocks_per_file; /* How big each file is */
    block_t        files_start;      /* First block of first file */
    size_t          next_file_no;    /* Next file number to use */
};

static union {
    struct sblock    s;
    char             pad[BLOCK_SIZE];
}

superblock;
```

# Directory Entry

```
#define DIRENT_LENGTH      64
#define NAME_LENGTH       (DIRENT_LENGTH - 1 - 1 - 2 * sizeof (size_t))

typedef struct {
    size_t      file_no;      /* File's # in the system */
    size_t      size;        /* Size of the file */
    unsigned char type;      /* Entry type (see below) */
    unsigned char namelen;   /* Length of name */
    char        name[NAME_LENGTH]; /* File name */
}

                                stupid_dirent;

#define DIR_SIZE           (BLOCKS_PER_FILE * BLOCK_SIZE \
                            / sizeof (stupid_dirent))
                                /* Max entries in a directory */
```

# Useful Macros

```
#define BLOCKS_TO_BYTES(x)          ((x) * superblock.s.block_size)
#define BYTES_TO_BLOCKS(x)          (((x) + superblock.s.block_size - 1) \
                                     / superblock.s.block_size)
#define FILE_NO_TO_BLOCK(x)         (((x) - 1)*superblock.s.blocks_per_file \
                                     + superblock.s.files_start)
#define LAST_BLOCK(x)               ((x) + superblock.s.blocks_per_file)
#define OFFSET_TO_BLOCK(dirent, x) \
                                     (FILE_NO_TO_BLOCK(dirent->file_no) + (x) \
                                     / superblock.s.block_size)
#define OFFSET_IN_BLOCK(x)          ((x) % superblock.s.block_size)
```

# Global Variables

**superblock.s** Superblock contents

**backing\_file\_fd** File descriptor connected to backing file (or device)

**dirbuf** 1-block buffer with directory entries

**dirblock** Block number of current block held in **dirblock**

# Implementing Block I/O

```
static void read_block(block_t block, void *buf)
{
    assert(lseek(backing_file_fd, BLOCKS_TO_BYTES(block), SEEK_SET) \
        != -1);
    assert(read(backing_file_fd, buf, superblock.s.block_size)
        == superblock.s.block_size);
}

static void write_block(block_t block, const void *buf)
{
    assert(lseek(backing_file_fd, BLOCKS_TO_BYTES(block), SEEK_SET) \
        != -1);
    assert(write(backing_file_fd, buf, superblock.s.block_size)
        == superblock.s.block_size);
}
```

# Directory I/O

```
static void fetch_dirblock(size_t block)
{
    if (dirblock == block)
        return;                               /* Efficiency: no work needed */
    dirblock = block;
    read_block(dirblock, dirbuf);
}

static void flush_dirblock()
{
    write_block(dirblock, dirbuf);
}
```

# Reading a Superblock

```
assert(lseek(backing_file_fd, 0, SEEK_SET) != -1);
size = read(backing_file_fd, &superblock, sizeof superblock);
if (size == sizeof superblock
    && superblock.s.magic == STUPID_MAGIC_LITTLE_ENDIAN) {
    /* Do any other initialization here */
    return NULL;
}
```

# Initializing an Empty Superblock

```
memset(&superblock, 0, sizeof superblock);
superblock.s.magic = STUPID_MAGIC_LITTLE_ENDIAN;
superblock.s.total_blocks = DISK_SIZE / BLOCK_SIZE;
superblock.s.block_size = BLOCK_SIZE;
superblock.s.blocks_per_file = BLOCKS_PER_FILE;

/*
 * The root directory always starts just past the superblock,
 * and has file number 1.  So the next available file number is 2.
 */
superblock.s.files_start = \
    sizeof(superblock) / superblock.s.block_size;
superblock.s.next_file_no = 2;
/* Not written here */
```



## Initializing the Root Directory

```
dirbuf = (stupid_dirent*)calloc(superblock.s.block_size, 1);
dirend = (stupid_dirent*)((char *)dirbuf + superblock.s.block_size);

dirblock = superblock.s.files_start;
dirbuf[0].type = TYPE_DIR;
dirbuf[0].file_no = 1;
dirbuf[0].size = DIR_SIZE * DIRENT_LENGTH;
dirbuf[0].namelen = 1;
memcpy(dirbuf[0].name, ".", 1);

dirbuf[1].type = TYPE_DIR;
dirbuf[1].file_no = 1;
dirbuf[1].size = DIR_SIZE * DIRENT_LENGTH;
dirbuf[1].namelen = 2;
memcpy(dirbuf[1].name, "..", 2);
write_block(superblock.s.files_start, dirbuf);
```

## A Tricky Point: Extending the Backing File

```
ftruncate(backing_file_fd, DISK_SIZE);

/*
 * Finally, write the superblock to disk. We write it last so
 * that if we crash, the disk won't appear valid.
 */
write_block(0, &superblock);
```

## Directory Lookup (Partial Code)

```
static stupid_dirent* lookup_component(block_t block,
    const char *start, const char *end)
{
    stupid_dirent*    dirent;
    size_t            len = end - start;
    block_t           last_block;
    if (len > NAME_LENGTH)
        len = NAME_LENGTH;
    for (last_block = LAST_BLOCK(block); block < last_block; block++) {
        fetch_dirblock(block);          /* Reads into dirbuf */
        for (dirent = dirbuf; dirent < dirend; dirent++) {
            if (dirent->type != TYPE_EMPTY && len == dirent->namelen
                && memcmp(dirent->name, start, len) == 0)
                return dirent;
        }
    }
    return NULL;
}
```

# Handling Functions You Don't Want to Write

```
static int fuse_stupid_rename(const char *from, const char *to)
{
    /*
     * Getting rename right is hard; you may need to remove the
     * destination, * and it has to support cross-directory renames.
     * I'm just going to prohibit it.
     */
    return -ENOSYS;
}
```

# Opening a File

```
static int fuse_stupid_open(const char *path, struct fuse_file_info *fi)
{
    stupid_dirent*      dirent;

    dirent = find_dirent(path, 0);
    if (dirent == NULL)
        return -ENOENT;
    if (dirent->type != TYPE_FILE)
        return -EACCES;
    /*
     * Open succeeds if the file exists.
     */
    return 0;
}
```

## Reading Data (Setup)

```
static int fuse_stupid_read(const char *path, char *buf, size_t size,
    off_t offset, struct fuse_file_info *fi)
{
    block_t          block;
    char             blockbuf[BLOCK_SIZE];
    size_t           bytes_read;
    stupid_dirent*   dirent;
    size_t           read_size;

    dirent = find_dirent(path, 0);
    if (dirent == NULL)
        return -ENOENT;
    if (dirent->type != TYPE_FILE)
        return -EACCES;
    read_size = dirent->size;           /* Amount to read (max is file siz
    if (offset >= read_size)
        return 0;
```

## Reading Data (The Loop)

```
block = OFFSET_TO_BLOCK(dirent, offset);
offset = OFFSET_IN_BLOCK(offset);

for (bytes_read = 0; size > 0; block++, offset = 0) {
    read_size = superblock.s.block_size - offset;
    if (read_size > size)
        read_size = size;
    read_block(block, blockbuf);    /* Read in full-block units */
    memcpy(buf, blockbuf, read_size);
    bytes_read += read_size;
    buf += read_size;
    size -= read_size;
}

return bytes_read;
```

# What About the Rest?

- ▶ This is quite a bit of code
- ▶ But there's more to a real filesystem
- ▶ Important lesson: It's up to you to divide user requests up into single-block accesses
- ▶ On-disk data is raw bytes; you must typecast to what you want
- ▶ Also must make sure you use block-size units
- ▶ `void *` pointers are helpful