

6.828 Fall 2011 Lab 5: File System

Handed out Wednesday, November 2, 2011

Due Friday, November 17, 2011

Introduction

In this lab, you will implement a simple disk-based file system. The file system itself will be implemented in micro-kernel fashion, outside the kernel but within its own user-space environment. Other environments access the file system by making IPC requests to this special file system environment.

Getting Started

Use Git to commit your Lab 4 source (if you haven't already), fetch the latest version of the course repository, and then create a local branch called lab5 based on our lab5 branch, origin/lab5:

```
athena% cd ~/6.828/lab
athena% add git
athena% git commit -am 'my solution to lab4'
nothing to commit (working directory clean)
athena% git pull
Already up-to-date.
athena% git checkout -b lab5 origin/lab5
Branch lab5 set up to track remote branch refs/remotes/origin/lab5.
Switched to a new branch "lab5"
athena% git merge lab4
Merge made by recursive.
 kern/env.c | 42 +++++
 1 files changed, 42 insertions(+), 0 deletions(-)
athena%
```

The main new component for this part of the lab is the file system environment, located in the new fs directory. Scan through all the files in this directory to get a feel for what all is new. Also, there are some new file system-related source files in the user and lib directories, particularly lib/file.c, lib/fd.c, lib/spawn.c, and new global header files inc/fs.h and inc/fd.h. Be sure to scan through all of these files.

You should run the pingpong, primes, and forktree test cases from lab 4 again after merging in the new lab 5 code. You will need to comment out the ENV_CREATE (fs_fs) line in kern/init.c because fs/fs.c tries to do some I/O, which JOS does allow yet. Similarly, temporarily comment out the call to close_all() in lib/exit.c; this function calls subroutines that you will implement later in the lab, and therefore will panic if called. If your lab 4 code doesn't contain any bugs, the test cases should run fine. Don't proceed until they work. Don't forget to uncomment these lines when you start Exercise 1.

If they don't work, use `git diff lab4 | more` to review all the changes, making sure there isn't any code you wrote for lab4 (or before) missing from lab 5. Make sure that lab 4 still works.

Lab Requirements

As before, you will need to do all of the regular exercises described in the lab and *at least one* challenge problem. Additionally, you will need to write up brief answers to the questions posed in the lab and a short (e.g., one or two paragraph) description of what you did to solve your chosen challenge problem. If you implement more than one challenge problem, you only need to describe one of them in the write-up, though of course you are welcome to do more. Place the write-up in a file called `answers-lab5.txt` in the top level of your `lab5` directory before handing in your work.

File system preliminaries

The file system you will work with is much simpler than most “real” file systems including that of xv6 UNIX, but it is powerful enough to provide the basic features: creating, reading, writing, and deleting files organized in a hierarchical directory structure.

We are (for the moment anyway) developing only a single-user operating system, which provides protection sufficient to catch bugs but not to protect multiple mutually suspicious users from each other. Our file system therefore does not support the UNIX notions of file ownership or permissions. Our file system also currently does not support hard links, symbolic links, time stamps, or special device files like most UNIX file systems do.

On-Disk File System Structure

Most UNIX file systems divide available disk space into two main types of regions: *inode* regions and *data* regions. UNIX file systems assign one *inode* to each file in the file system; a file’s inode holds critical meta-data about the file such as its stat attributes and pointers to its data blocks. The data regions are divided into much larger (typically 8KB or more) *data blocks*, within which the file system stores file data and directory meta-data. Directory entries contain file names and pointers to inodes; a file is said to be *hard-linked* if multiple directory entries in the file system refer to that file’s inode. Since our file system will not support hard links, we do not need this level of indirection and therefore can make a convenient simplification: our file system will not use inodes at all and instead will simply store all of a file’s (or sub-directory’s) meta-data within the (one and only) directory entry describing that file.

Both files and directories logically consist of a series of data blocks, which may be scattered throughout the disk much like the pages of an environment’s virtual address space can be scattered throughout physical memory. The file system environment hides the details of block layout, presenting interfaces for reading and writing sequences of bytes at arbitrary offsets within files. The file system environment handles all modifications to directories internally as a part of performing actions such as file creation and deletion. Our file system does allow user environments to *read* directory meta-data directly (e.g., with `read`), which means that user environments can perform directory scanning operations themselves (e.g., to implement the `ls` program) rather than having to

rely on additional special calls to the file system. The disadvantage of this approach to directory scanning, and the reason most modern UNIX variants discourage it, is that it makes application programs dependent on the format of directory meta-data, making it difficult to change the file system's internal layout without changing or at least recompiling application programs as well.

Sectors and Blocks

Most disks cannot perform reads and writes at byte granularity and instead perform reads and writes in units of *sectors*, which today are almost universally 512 bytes each. File systems actually allocate and use disk storage in units of *blocks*. Be wary of the distinction between the two terms: *sector size* is a property of the disk hardware, whereas *block size* is an aspect of the operating system using the disk. A file system's block size must be a multiple of the sector size of the underlying disk.

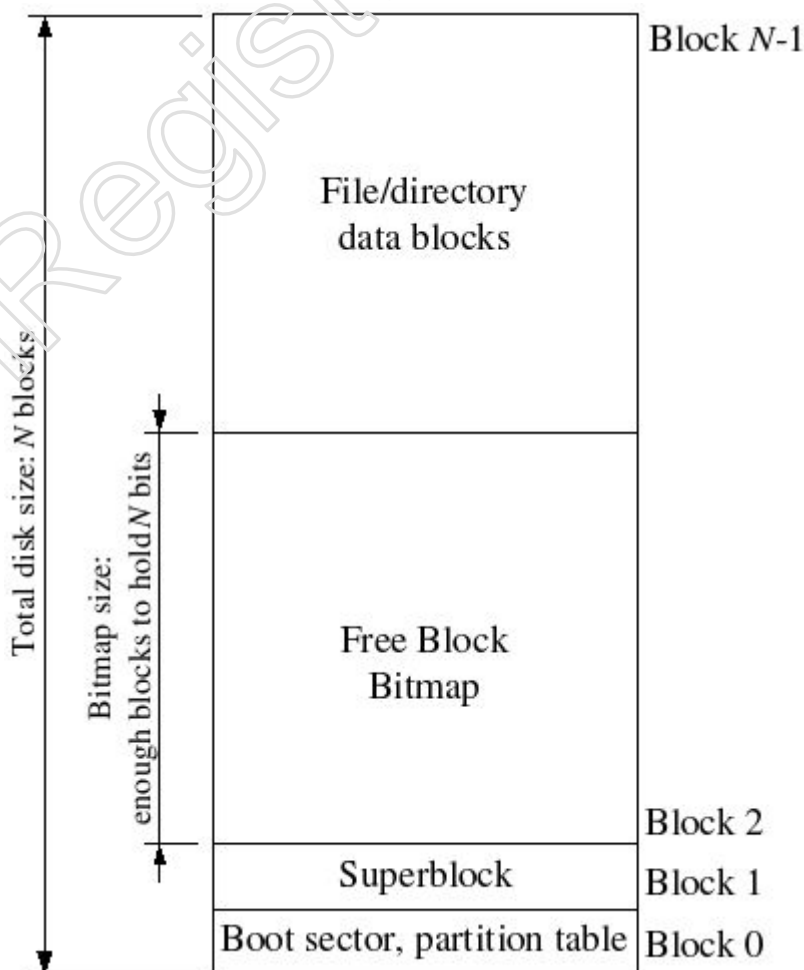
The UNIX xv6 file system uses a block size of 512 bytes, the same as the sector size of the underlying disk. Most modern file systems use a larger block size, however, because storage space has gotten much cheaper and it is more efficient to manage storage at larger granularities. Our file system will use a block size of 4096 bytes, conveniently matching the processor's page size.

Superblocks

File systems typically reserve certain disk blocks at "easy-to-find" locations on the disk (such as the very start or the very end) to hold meta-data describing properties of the file system as a whole, such as the block size, disk size, any meta-data required to find the root directory, the time the file system was last mounted, the time the file system was last checked for errors, and so on. These special blocks are called *superblocks*.

Our file system will have exactly one superblock, which will always be at block 1 on the disk. Its layout is defined by struct `Super` in `inc/fs.h`. Block 0 is typically reserved to hold boot loaders and partition

tables, so file systems generally do not use the very first disk block. Many "real" file systems maintain multiple superblocks, replicated throughout several



widely-spaced regions of the disk, so that if one of them is corrupted or the disk develops a media error in that region, the other superblocks can still be found and used to access the file system.

The Block Bitmap: Managing Free Disk Blocks

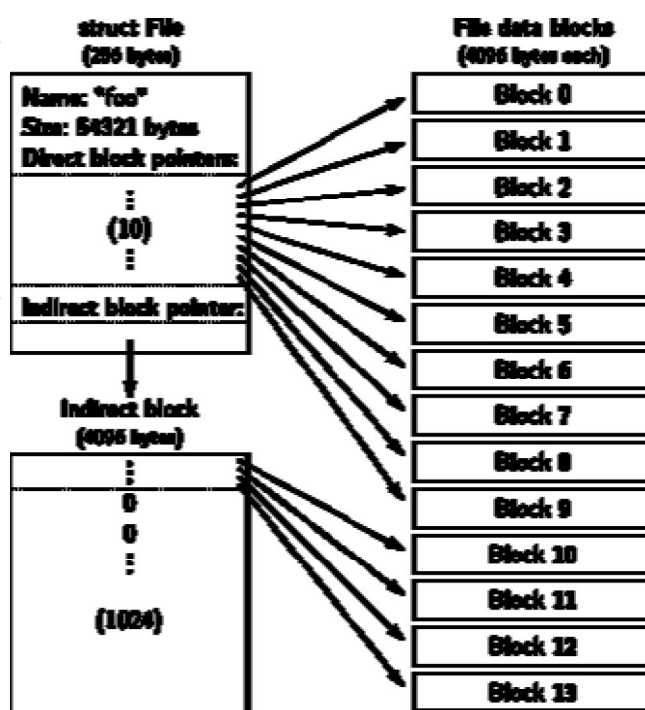
In the same way that the kernel must manage the system's physical memory to ensure that a given physical page is used for only one purpose at a time, a file system must manage the blocks of storage on a disk to ensure that a given disk block is used for only one purpose at a time. In `pmap.c` you keep the `Page` structures for all free physical pages on a linked list, `page_free_list`, to keep track of the free physical pages. In file systems it is more common to keep track of free disk blocks using a *bitmap* rather than a linked list, because a bitmap is more storage-efficient than a linked list and easier to keep consistent. Searching for a free block in a bitmap can take more CPU time than simply removing the first element of a linked list, but for file systems this isn't a problem because the I/O cost of actually accessing the free block after we find it dominates for performance purposes.

To set up a free block bitmap, we reserve a contiguous region of space on the disk large enough to hold one bit for each disk block. For example, since our file system uses 4096-byte blocks, each bitmap block contains $4096 \times 8 = 32768$ bits, or enough bits to describe 32768 disk blocks. In other words, for every 32768 disk blocks the file system uses, we must reserve one disk block for the block bitmap. A given bit in the bitmap is set if the corresponding block is free, and clear if the corresponding block is in use. The block bitmap in our file system always starts at disk block 2, immediately after the superblock. For simplicity we will reserve enough bitmap blocks to hold one bit for each block in the entire disk, including the blocks containing the superblock and the bitmap itself. We will simply make sure that the bitmap bits corresponding to these special, "reserved" areas of the disk are always clear (marked in-use).

File Meta-data

The layout of the meta-data describing a file in our file system is described by `struct File` in `inc/fs.h`. This meta-data includes the file's name, size, type (regular file or directory), and pointers to the blocks comprising the file. As mentioned above, we do not have inodes, so this meta-data is stored in a directory entry on disk. Unlike in most "real" file systems, for simplicity we will use this one `File` structure to represent file meta-data as it appears *both on disk and in memory*.

The `f_direct` array in `struct File` contains space to store the block numbers of the first 10 (`NDIRECT`) blocks of the file, which we call the file's *direct* blocks. For small files up to $10 \times 4096 = 40\text{KB}$ in



size, this means that the block numbers of all of the file's blocks will fit directly within the File structure itself. For larger files, however, we need a place to hold the rest of the file's block numbers. For any file greater than 40KB in size, therefore, we allocate an additional disk block, called the file's *indirect block*, to hold up to $4096/4 = 1024$ additional block numbers. Our file system therefore allows files to be up to 1034 blocks, or just over four megabytes, in size. To support larger files, "real" file systems typically support *double-* and *triple-indirect blocks* as well.

Directories versus Regular Files

A File structure in our file system can represent either a *regular* file or a directory; these two types of "files" are distinguished by the type field in the File structure. The file system manages regular files and directory-files in exactly the same way, except that it does not interpret the contents of the data blocks associated with regular files at all, whereas the file system interprets the contents of a directory-file as a series of File structures describing the files and subdirectories within the directory.

The superblock in our file system contains a File structure (the root field in struct Super) that holds the meta-data for the file system's root directory. The contents of this directory-file is a sequence of File structures describing the files and directories located within the root directory of the file system. Any subdirectories in the root directory may in turn contain more File structures representing sub-subdirectories, and so on.

The File System

The goal for this lab is not to have you implement the entire file system, but for you to implement only certain key components. In particular, you will be responsible for reading blocks into the block cache and flushing them back to disk; allocating disk blocks; mapping file offsets to disk blocks; and implementing read, write, and open in the IPC interface. Because you will not be implementing all of the file system yourself, it is very important that you familiarize yourself with the provided code and the various file system interfaces.

Much of the code for the file system has been rewritten for this year, so please let us know if you come across any bugs.

Disk Access

The file system environment in our operating system needs to be able to access the disk, but we have not yet implemented any disk access functionality in our kernel. Instead of taking the conventional "monolithic" operating system strategy of adding an IDE disk driver to the kernel along with the necessary system calls to allow the file system to access it, we will instead implement the IDE disk driver as part of the user-level file system environment. We will still need to modify the kernel slightly, in order to set things up so that the file system environment has the privileges it needs to implement disk access itself.

It is easy to implement disk access in user space this way as long as we rely on polling, "programmed I/O" (PIO)-based disk access and do not use disk interrupts. It is possible to implement interrupt-driven device drivers in user mode as well (the L3 and L4 kernels do this, for example), but it is more difficult since the kernel must field device interrupts and dispatch them to the correct user-mode environment.

The x86 processor uses the IOPL bits in the EFLAGS register to determine whether protected-mode code is allowed to perform special device I/O instructions such as the IN and OUT instructions. Since all of the IDE disk registers we need to access are located in the x86's I/O space rather than being memory-mapped, giving "I/O privilege" to the file system environment is the only thing we need to do in order to allow the file system to access these registers. In effect, the IOPL bits in the EFLAGS register provides the kernel with a simple "all-or-nothing" method of controlling whether user-mode code can access I/O space. In our case, we want the file system environment to be able to access I/O space, but we do not want any other environments to be able to access I/O space at all.

In the tests that follow, if you fail a test, `obj/fs/fs.img` is likely to be left inconsistent. Be sure to remove it before running `make grade` or `make qemu`.

Exercise 1. `i386_init` identifies the file system environment by passing the type `ENV_TYPE_FS` to your environment creation function, `env_create`. Modify `env_create` in `env.c`, so that it gives the file system environment I/O privilege, but never gives that privilege to any other environment.

Make sure you can start the file environment without causing a General Protection fault. You should pass the "fs i/o" test in `make grade`.

Do you have to do anything else to ensure that this I/O privilege setting is saved and restored properly when you subsequently switch from one environment to another? Make sure you understand how this environment state is handled.

Read through the files in the new `fs` directory in the source tree. The file `fs/ide.c` implements our minimal PIO-based disk driver. The file `fs/serv.c` contains the `umain` function for the file system environment.

Note that the `GNUmakefile` file in this lab sets up QEMU to use the file `obj/kern/kernel.img` as the image for disk 0 (typically "Drive C" under DOS/Windows) as before, and to use the (new) file `obj/fs/fs.img` as the image for disk 1 ("Drive D"). In this lab your file system should only ever touch disk 1; disk 0 is used only to boot the kernel. If you manage to corrupt either disk image in some way, you can reset both of them to their original, "pristine" versions simply by typing:

```
$ rm obj/kern/kernel.img obj/fs/fs.img
$ make
```

or by doing:

```
$ make clean
```

\$ **make**

Challenge! Implement interrupt-driven IDE disk access, with or without DMA. You can decide whether to move the device driver into the kernel, keep it in user space along with the file system, or even (if you really want to get into the micro-kernel spirit) move it into a separate environment of its own.

The Block Cache

In our file system, we will implement a simple “buffer cache” (really just a block cache) with the help of the processor’s virtual memory system. The code for the block cache is in `fs/bc.c`.

Our file system will be limited to handling disks of size 3GB or less. We reserve a large, fixed 3GB region of the file system environment’s address space, from `0x10000000` (`DISKMAP`) up to `0xD0000000` (`DISKMAP+DISKMAX`), as a “memory mapped” version of the disk. For example, disk block 0 is mapped at virtual address `0x10000000`, disk block 1 is mapped at virtual address `0x10001000`, and so on. The `diskaddr` function in `fs/bc.c` implements this translation from disk block numbers to virtual addresses (along with some sanity checking).

Since our file system environment has its own virtual address space independent of the virtual address spaces of all other environments in the system, and the only thing the file system environment needs to do is to implement file access, it is reasonable to reserve most of the file system environment’s address space in this way. It would be awkward for a real file system implementation on a 32-bit machine to do this since modern disks are larger than 3GB. Such a buffer cache management approach may still be reasonable on a machine with a 64-bit address space.

Of course, it would be unreasonable to read the entire disk into memory, so instead we’ll implement a form of *demand paging*, wherein we only allocate pages in the disk map region and read the corresponding block from the disk in response to a page fault in this region. This way, we can pretend that the entire disk is in memory.

Exercise 2. Implement the `bc_pgfault` and `flush_block` functions in `fs/bc.c`. `bc_pgfault` is a page fault handler, just like the one you wrote in the previous lab for copy-on-write fork, except that its job is to load pages in from the disk in response to a page fault. When writing this, keep in mind that (1) `addr` may not be aligned to a block boundary and (2) `ide_read` operates in sectors, not blocks.

The `flush_block` function should write a block out to disk *if necessary*. `flush_block` shouldn’t do anything if the block isn’t even in the block cache (that is, the page isn’t mapped) or if it’s not dirty. We will use the VM hardware to keep track of whether a disk block has been modified since it was last read from or written to disk. To see whether a

block needs writing, we can just look to see if the PTE_D "dirty" bit is set in the vpt entry. (The PTE_D bit is set by the processor in response to a write to that page; see 5.2.4.3 in [chapter 5](#) of the 386 reference manual.) After writing the block to disk, `flush_block` should clear the PTE_D bit using `sys_page_map`.

Use `make grade` to test your code. Your code should pass "check_bc", "check_super", and "check_bitmap".

The `fs_init` function in `fs/fs.c` is a prime example of how to use the block cache. After initializing the block cache, it simply stores pointers into the disk map region in the super and bitmap global variables. After this point, we can simply read from these structures as if they were in memory and our page fault handler will read them from disk as necessary.

Challenge! The block cache has no eviction policy. Once a block gets faulted in to it, it never gets removed and will remain in memory forevermore. Add eviction to the buffer cache. Using the PTE_A "accessed" bits in the page tables, which the hardware sets on any access to a page, you can track approximate usage of disk blocks without the need to modify every place in the code that accesses the disk map region. Be careful with dirty blocks.

The Block Bitmap

After `fs_init` sets the bitmap pointer, we can treat bitmap as a packed array of bits, one for each block on the disk. See, for example, `block_is_free`, which simply checks whether a given block is marked free in the bitmap.

Exercise 3. Use `free_block` as a model to implement `alloc_block`, which should find a free disk block in the bitmap, mark it used, and return the number of that block. When you allocate a block, you should immediately flush the changed bitmap block to disk with `flush_block`, to help file system consistency.

Use `make grade` to test your code. Your code should now pass "alloc_block".

File Operations

We have provided a variety of functions in `fs/fs.c` to implement the basic facilities you will need to interpret and manage File structures, scan and manage the entries of directory-files, and walk the file system from the root to resolve an absolute pathname. Read through *all* of the code in `fs/fs.c` and make sure you understand what each function does before proceeding.

Exercise 4. Implement `file_block_walk` and `file_get_block`. `file_block_walk` maps from a block offset within a file to the

pointer for that block in the struct `File` or the indirect block, very much like what `pgdir_walk` did for page tables. `file_get_block` goes one step further and maps to the actual disk block, allocating a new one if necessary.

Use `make grade` to test your code. Your code should pass "file_open", "file_get_block", and "file_flush/file_truncated/file_rewrite".

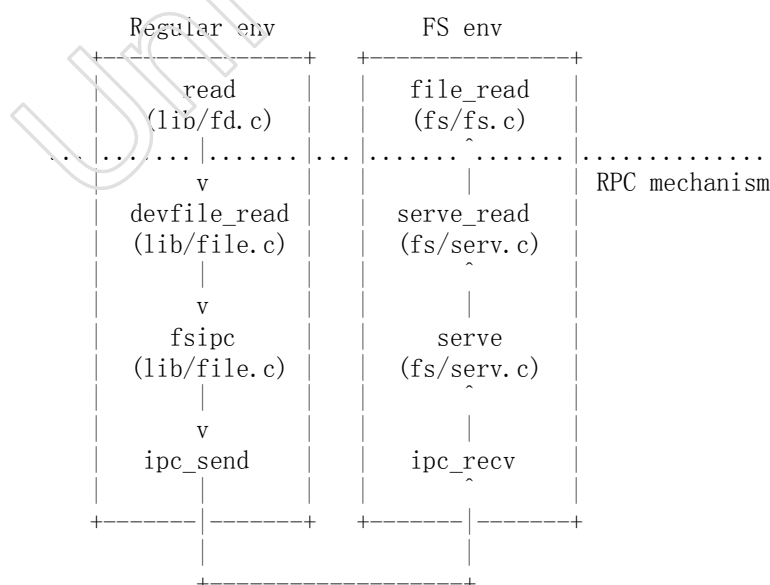
`file_block_walk` and `file_get_block` are the workhorses of the file system. For example, `file_read` and `file_write` are little more than the bookkeeping atop `file_get_block` necessary to copy bytes between scattered blocks and a sequential buffer.

Challenge! The file system is likely to be corrupted if it gets interrupted in the middle of an operation (for example, by a crash or a reboot). Implement soft updates or journailling to make the file system crash-resilient and demonstrate some situation where the old file system would get corrupted, but yours doesn't.

Client/Server File System Access

Now that we have implemented the necessary functionality within the file system environment itself, we must make it accessible to other environments that wish to use the file system. Since other environments can't directly call functions in the file system environment, we'll expose access to the file system environment via a *remote procedure call*, or RPC, abstraction, built atop JOS's IPC mechanism.

Graphically, here's what a call to the file system server (say, read) looks like



Everything below the dotted line is simply the mechanics of getting a read request from the regular environment to the file system environment. Starting at the beginning, `read` (which we provide) works on any file descriptor and simply

dispatches to the appropriate device read function, in this case `devfile_read` (we'll have more device types in future labs, like pipes and network sockets). `devfile_read` implements `read` specifically for on-disk files. This and the other `devfile_*` functions in `lib/file.c` implement the client side of the FS operations and all work in roughly the same way, bundling up arguments in a request structure, calling `fsipc` to send the IPC request, and unpacking and returning the results. The `fsipc` function simply handles the common details of sending a request to the server and receiving the reply.

The file system server code can be found in `fs/serv.c`. It loops in the `serve` function, endlessly receiving a request over IPC, dispatching that request to the appropriate handler function, and sending the result back via IPC. In the `read` example, `serve` will dispatch to `serve_read`, which will take care of the IPC details specific to `read` requests such as unpacking the request structure and finally call `file_read` to actually perform the file read.

Recall that JOS's IPC mechanism lets an environment send a single 32-bit number and, optionally, share a page. To send a request from the client to the server, we use the 32-bit number for the request type (the file system server RPCs are numbered, just like how syscalls were numbered) and store the arguments to the request in a union `Fsipc` on the page shared via the IPC. On the client side, we always share the page at `fsipcbuf`; on the server side, we map the incoming request page at `fsreq` (`0x0fff000`).

The server also sends the response back via IPC. We use the 32-bit number for the function's return code. For most RPCs, this is all they return. `FSREQ_READ` and `FSREQ_STAT` also return data, which they simply write to the page that the client sent its request on. There's no need to send this page in the response IPC, since the client shared it with the file system server in the first place. Also, in its response, `FSREQ_OPEN` shares with the client a new "Fd page". We'll return to this RPC shortly.

Exercise 5. Implement `serve_read` in `fs/serv.c` and `devfile_read` in `lib/file.c`.

`serve_read`'s heavy lifting will be done by the already-implemented `file_read` in `fs/fs.c` (which, in turn, is just a bunch of calls to `file_get_block`). `serve_read` just has to provide the RPC interface for file reading. Look at the comments and code in `serve_set_size` to get a general idea of how the server functions should be structured.

Likewise, `devfile_read` should pack its arguments into `fsipcbuf` for `serve_read`, call `fsipc`, and handle the result.

Use `make grade` to test your code. Your code should pass "`lib/file.c`" and "`file_read`".

Exercise 6. Implement `serve_write` in `fs/serv.c` and `devfile_write` in `lib/file.c`.

Use `make grade` to test your code. Your code should pass "`file_write`" and "`file_read after file_write`".

Client-Side File Operations

The functions in `lib/file.c` are specific to on-disk files, but UNIX file descriptors are a more general notion that also encompasses pipes, console I/O, etc. In JOS, each of these device types has (or will have in later labs) a corresponding `struct Dev`, with pointers to the functions that implement read/write/etc. for that device type. `lib/fd.c` implements the general UNIX-like file descriptor interface on top of this. Each `struct Fd` indicates its device type, and most of the functions in `lib/fd.c` simply dispatch operations to functions in the appropriate `struct Dev`.

`lib/fd.c` also maintains the *file descriptor table* region in each application environment's address space, starting at `FSTABLE`. This area reserves a page's worth (4KB) of address space for each of the up to `MAXFD` (currently 32) file descriptors the application can have open at once. At any given time, a particular file descriptor table page is mapped if and only if the corresponding file descriptor is in use. Each file descriptor also has an optional "data page" in the region starting at `FILEDATA`, though we won't use that until later labs.

For nearly all interactions with files, user code should go through the functions in `lib/fd.c`. The one "public" function in `lib/file.c` is `open`, which constructs a new file descriptor by opening a named on-disk file.

Exercise 7. Implement `open`. The `open` function must find an unused file descriptor using the `fd_alloc()` function we have provided, make an IPC request to the file system environment to open the file, and return the number of the allocated file descriptor. Be sure your code fails gracefully if the maximum number of files are already open, or if any of the IPC requests to the file system environment fail.

Use `make grade` to test your code. Your code should pass "open", "large file", "motd display", and "motd change".

Challenge! Change the file system to keep most file meta-data in Unix-style inodes rather than in directory entries, and add support for hard links.

Spawning Processes

We have given you the code for `spawn` which creates a new environment, loads a program image from the file system into it, and then starts the child environment running this program. The parent process then continues running independently of the child. The `spawn` function effectively acts like a `fork` in UNIX followed by an immediate `exec` in the child process.

We implemented `spawn` rather than a UNIX-style `exec` because `spawn` is easier to implement from user space in "exokernel fashion", without special help from the kernel. Think about what you would have to do in order to implement `exec` in user space, and be sure you understand why it is harder.

Exercise 8. `spawn` relies on the new syscall

`sys_env_set_trapframe` to initialize the state of the newly created environment. Implement `sys_env_set_trapframe`. Test your code by running the `user/icode` program from `kern/init.c`, which will attempt to spawn `/init` from the file system.

Use `make grade` to test your code.

Challenge! Implement Unix-style `exec`.

Challenge! Implement `mmap`-style memory-mapped files and modify `spawn` to map pages directly from the ELF image when possible.

Questions:

1. How long approximately did it take you to do this lab?
2. We redesigned the file system this year with the goal of making it more comprehensible in a week-long lab. Do you feel like you gained an understanding of how to build a file system? Feel free to suggest things we could improve.

This completes the lab. As usual, don't forget to run `make grade` and to write up your answers and a description of your challenge exercise solution. Before handing in, use `git status` and `git diff` to examine your changes and don't forget to `git add answers-lab5.txt`. When you're ready, commit your changes with `git commit -am 'my solutions to lab 5'`, then `make handin` and follow the directions.