6.828 Fall 2012 Lab 5: Spawn and Shell

Handed out Lecture 15
Due one day after Lecture 17

Introduction

In this lab, you will implement `spawn`, a library call that loads and runs on-disk executables. You will then flesh out your kernel and library operating system enough to run a shell on the console.

Getting Started

Use Git to fetch the latest version of the course repository, and then create a local branch called `lab5`.

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,

`fs/fs.c`	Code that mainipulates the file system's on-disk structure.
`fs/bc.c`	A simple block cache built on top of our user-level page fault handling facility.
`fs/ide.c`	Minimal PIO-based (non-interrupt-driven) IDE driver code.
`fs/serv.c`	The file system server that interacts with client environments using file system IPCs.
`lib/fd.c`	Code that implements the general UNIX-like file descriptor interface.
`lib/file.c`	The driver for on-disk file type, implemented as a file system IPC client.
`lib/console.c`	The driver for console input/output file type.
`lib/spawn.c`	Code skeleton of the `spawn` library call.

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 not 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 un-comment these lines when you start Exercise 1.

If they don't work, use `git diff lab4` 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

We have provided you with a simple, read-only, disk-based file system. You will need to slightly change your existing code in order to port the file system for your JOS, so that `spawn` can access on-disk executables using path names. Although you do not have to understand every detail of the file system, such as its on-disk structure. It is very important that you familiarize yourself with the design principles and its various interfaces.

The file system itself is 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.

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 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.

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`.

Question

1. 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? Why?

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 our file system should only ever touch disk 1; disk 0 is used only to boot the kernel.

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.

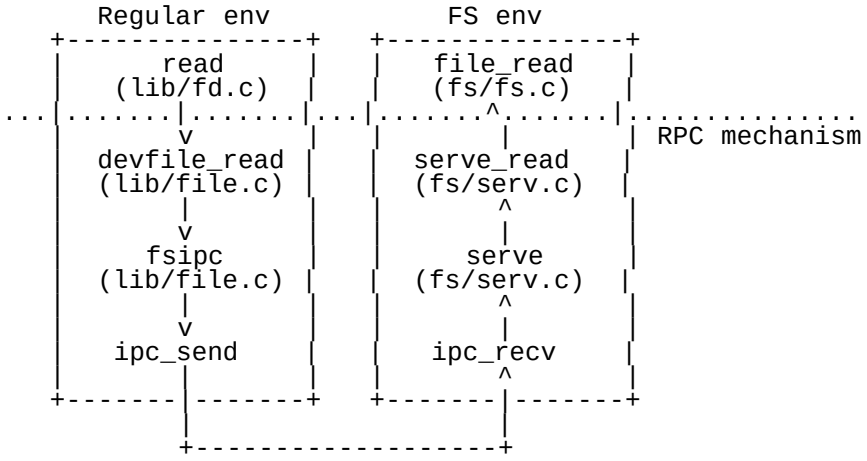
Exercise 2. Implement the `bc_pgfault` functions in `fs/bc.c`. `bc_pgfault` is a page fault handler, just like the one your wrote in the previous lab for copy-on-write fork, except that its job is to load pages in from the disk in response to a page fault. When writing this, keep in mind that (1) `addr`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

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

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` global variable. After this point, we can simply read from the `super` structure as if they were in memory and our page fault handler will read them from disk as necessary.

The file system interface

Now that we have 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 can have more device types, like pipes). `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` (`0x0ffff000`).

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 the file descriptor page shortly.

Challenge! Extend the file system to support write access. Here are a few points you need to consider:

1. Use the block bitmap starting at block 2 to keep track of which disk blocks are free and which are in use. Look at `fs/fsformat.c` to see how the bitmap is initialized.
2. Make use of the `alloc` argument in `file_block_walk`. In `file_get_block`, allocate new disk blocks as necessary.
3. In your block cache, use the VM hardware (the `PTE_D` "dirty" bit in the `uvpt` entry) to keep track of whether a cached disk block has been modified, and thus needs to be written back to the disk.
4. Handle `O_CREAT` and `O_TRUNC` open modes in `serve_open`.
5. Handle more file system IPC requests, such as `FSREQ_SET_SIZE`, `FSREQ_WRITE`, `FSREQ_FLUSH`, `FSREQ_REMOVE` and `FSREQ_SYNC`, in `fs/serv.c`. We have defined the argument for these calls for you in `inc/fs.h`. Also, write the corresponding service routines in `fs/fs.c` and hook them to client stubs in `lib/file.c`.
6. For more information about the file system's on-disk structure, read `inc/fs.h` and `fs/fsformat.c`. You may also refer to last year's lab 5 text.

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 3. `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/spawnhello` program from `kern/init.c`, which will attempt to spawn `/hello` 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.

Sharing library state across fork and spawn

The UNIX file descriptors are a general notion that also encompasses pipes, console I/O, etc. In JOS, each of these device types has 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`, which devices can use if they choose.

We would like to share file descriptor state across `fork` and `spawn`, but file descriptor state is kept in user-space memory. Right now, on `fork`, the memory will be marked copy-on-write, so the state will be duplicated rather than shared. (This means environments won't be able to seek in files they didn't open themselves and that pipes won't work across a fork.) On `spawn`, the memory will be left behind, not copied at all. (Effectively, the spawned environment starts with no open file descriptors.)

We will change `fork` to know that certain regions of memory are used by the "library operating system" and should always be shared. Rather than hard-code a list of regions somewhere, we will set an otherwise-unused bit in the page table entries (just like we did with the `PTE_COW` bit in `fork`).

We have defined a new `PTE_SHARE` bit in `inc/lib.h`. This bit is one of the three PTE bits that are marked "available for software use" in the Intel and AMD manuals. We will establish the convention that if a page table entry has this bit set, the PTE should be copied directly from parent to child in both `fork` and `spawn`. Note that this is different from marking it copy-on-write: as described in the first paragraph, we want to make sure to *share* updates to the page.

Exercise 4. Change `duppage` in `lib/fork.c` to follow the new convention. If the page table entry has the `PTE_SHARE` bit set, just copy the mapping directly. (You should use `PTE_SYSCALL`, not `0xfff`, to mask out the relevant bits from the page table entry. `0xfff` picks up the accessed and dirty bits as well.)

Likewise, implement `copy_shared_pages` in `lib/spawn.c`. It should loop through all page table entries in the current process (just like `fork` did), copying any page mappings that have the `PTE_SHARE` bit set into the child process.

Use `make run-testpteshare` to check that your code is behaving properly. You should see lines that say "`fork handles PTE_SHARE right`" and "`spawn handles PTE_SHARE right`".

Use `make run-testfdsharing` to check that file descriptors are shared properly. You should see lines that say "`read in child succeeded`" and "`read in parent succeeded`".

The keyboard interface

For the shell to work, we need a way to type at it. QEMU has been displaying output we write to the CGA display and the serial port, but so far we've only taken input while in the kernel monitor. In QEMU, input typed in the graphical window appear as input from the keyboard to JOS, while input typed to the console appear as characters on the serial port. `kern/console.c` already contains the keyboard and serial drivers that have been used by the kernel monitor since lab 1, but now you need to attach these to the rest of the system.

Exercise 5. In your `kern/trap.c`, call `kbd_intr` to handle trap `IRQ_OFFSET+IRQ_KBD` and `serial_intr` to handle trap `IRQ_OFFSET+IRQ_SERIAL`.

We implemented the console input/output file type for you, in `lib/console.c`.

Test your code by running `make run-testkbd` and type a few lines. The system should echo your lines back to you as you finish them. Try typing in both the console and the graphical window, if you have both available.

The Shell

Run `make run-icode` or `make run-icode-nox`. This will run your kernel and start `user/icode`. `icode` execs `init`, which will set up the console as file descriptors 0 and 1 (standard input and standard output). It will then spawn `sh`, the shell. You should be able to run the following commands:

```
echo hello world | cat
cat lorem |cat
cat lorem |num
cat lorem |num |num |num |num |num
lsfd
cat script
sh <script
```

Note that the user library routine `cprintf` prints straight to the console, without using the file descriptor code. This is great for debugging but not great for piping into other programs. To print output to a particular file descriptor (for example, 1, standard output), use `fprintf(1, "...", ...)`. `printf("...", ...)` is a short-cut for printing to FD 1. See `user/lsfd.c` for examples.

Run `make run-testshell` to test your shell. `testshell` simply feeds the above commands (also found in `fs/testshell.sh`) into the shell and then checks that the output matches `fs/testshell.key`.

Your code should pass all tests at this point. As usual, you can grade your submission with `make grade` and hand it in with `make handin`.

Questions

2. How long approximately did it take you to do this lab?
3. We simplified the file system this year with the goal of making more time for the final project. Do you feel like you gained a basic understanding of the file I/O in JOS? 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.

6.828 Operating System Engineering

Fall 2012