Chapter 9

Exec

Chapter 2 stopped with the `initproc` invoking the kernel's `exec` system call. As a result, we took detours into interrupts, multiprocessing, device drivers, and a file system. With these taken care of, we can finally look at the implementation of `exec`. As we saw in Chapter 0, `exec` replaces the memory and registers of the current process with a new program, but it leaves the file descriptors, process id, and parent process the same. `Exec` is thus little more than a binary loader, just like the one in the boot sector from Chapter 1. The additional complexity comes from setting up the stack. The memory image of an executing process looks like:

```
[XXX better picture: not ASCII art,
show individual argv pointers, show argc,
show argument strings, show fake return address.]

+--------------------------------------+
| text | data | stack | args | heap ... |
+--------------------------------------+
                       ^
                       |
                  initial sp
```

In xv6, the stack is a single page—4096 bytes—long. The command-line arguments follow the stack immediately in memory, so that the program can start at `main` as if the function call `main(argc, argv)` had just started. The heap comes last so that expanding it does not require moving any of the other sections.

Code

When the system call arrives, `syscall` invokes `sys_exec` via the `syscalls` table (2879). `Sys_exec` (4951) parses the system call arguments, as we saw in Chapter 3, and invokes `exec` (4972).

`Exec` (5009) opens the named binary `path` using `namei` (5021) and then reads the ELF header. Like the boot sector, it uses `elf.magic` to decide whether the binary is an ELF binary (5025-5029). Then it makes two passes through the program segment and argument lists. The first computes the total amount of memory needed, and the second creates the memory image. The total memory size includes the program segments (5032-5041), the argument strings (5043-5048), the argument vector pointed at by `argv` (5049), the `argv` and `argc` arguments to `main` (5049-5051), and the stack (5053-5054). `Exec` then allocates and zeros the required amount of memory (5056-5061) and copies the data into the new memory image: the program segments (5063-5077), the argument strings

and pointers (5079-5090), and the stack frame for `main` (5092-5099).

Notice that when `exec` copies the program segments, it makes sure that the data being loaded into memory fits in the declared size `ph.memsz` (5069-5070). Without this check, a malformed ELF binary could cause `exec` to write past the end of the allocated memory image, causing memory corruption and making the operating system unstable. The boot sector neglected this check both to reduce code size and because not checking doesn't change the failure mode: either way the machine doesn't boot if given a bad ELF image. In contrast, in `exec` this check is the difference between making one process fail and making the entire system fail.

During the preparation of the new memory image, if `exec` detected an error like an invalid program segment, it jumps to the label `bad`, frees the new image, and returns –1. `Exec` must wait to free the old image until it is sure that the system call will succeed: if the old image is gone, the system call cannot return –1 to it. The only error cases in `exec` happen during the creation of the image. Once the image is complete, `exec` can free the old image and install the new one (5107-5111). After changing the image, `exec` must update the user segment registers to refer to the new image, just as `sbrk` did (5112). Finally, `exec` returns 0. Success!

Now the `initcode` (6700) is done. `Exec` has replaced it with the real `/init` binary, loaded out of the file system. `Init` (6810) creates a new console device file if needed and then opens it as file descriptors 0, 1, and 2. Then it loops, starting a console shell, handles orphaned zombies until the shell exits, and repeats. The system is up.

Real world

`Exec` is the most complicated code in xv6 in and in most operating systems. It involves pointer translation (in `sys_exec` too), many error cases, and must replace one running process with another. Real world operationg systems have even more complicated `exec` implementations. They handle shell scripts (see exercise below), more complicated ELF binaries, and even multiple binary formats.

Exercises

1. Unix implementations of `exec` traditionally include special handling for shell scripts. If the file to execute begins with the text `#!`, then the first line is taken to be a program to run to interpret the file. For example, if `exec` is called to run `myprog arg1` and `myprog`'s first line is `#!/interp`, then `exec` runs `/interp` with command line `/interp myprog arg1`. Implement support for this convention in xv6.