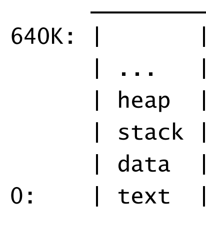
DRAFT as of November 19, 2010: Copyright 2009 Cox, Kaashoek, Morris

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 user memory image of an executing process looks like:



The heap is above the stack so that it can expand (with `sbrk`). The stack is a single page—4096 bytes—long. Strings containing the command-line arguments, as well as an array of pointers to them, are at the very top of the stack. Just under that the kernel places values that allow a program to start at `main` as if the function call `main(argc, argv)` had just started. Here are the values that `exec` places at the top of the stack:

```
"argumentN"                        -- nul-terminated string
 ...
"argument0"
0                                  -- argv[argc]
address of argumentN
 ...
address of argument0               -- argv[0]
address of address of argument0    -- argv argument to main()
argc                               -- argc argument to main()
0xffffffff                         -- return PC for main() call
```

Code

When the system call arrives, `syscall` invokes `sys_exec` via the `syscalls` table (3250). `Sys_exec` (5351) parses the system call arguments, as we saw in Chapter 3, and invokes `exec` (5373).

Exec (5409) opens the named binary path using namei (5422) and then reads the ELF header. Like the boot sector, it uses elf.magic to decide whether the binary is an ELF binary (5426-5430). Then it allocates a new page table with no user mappings with setupkvm (5432), allocates memory for each ELF segment with allocuvm (5443), and loads each segment into memory with loaduvm (5445). allocuvm checks that the virtual addresses requested are within the 640 kilobytes that user processes are allowed to use. loaduvm (2679) uses walkpgdir to find the physical address of the allocated memory at which to write each page of the ELF segment, and readi to read from the file. The ELF file may contain data segments that contain global variables that should start out zero, represented with a memsz that is greater than the segment's filesz; the result is that allocuvm allocates zeroed physical memory, but loaduvm does not copy anything from the file.

Now exec allocates and initializes the user stack. It assumes that one page of stack is enough. It is going to put the arguments passed to the system call at the top of the stack, so it first calculates how much space they will need (5459) and at what user address they will start (5462). It places a null pointer at the end of what will be the argv list passed to main, and then copies each argument string to its place in the stack (5469), and places a pointer to each argument string to the right place in the argv array (5465). Finally exec pushes argv, argc, and a fake return program counter onto the stack.

During the preparation of the new memory image, if exec detects 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 install the new image (5495) and free the old one (5497). Finally, exec returns 0. Success!

Now the initcode (7200) is done. Exec has replaced it with the real /init binary, loaded out of the file system. Init (7310) 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.