Process Creation | Coursera

coursera.org/learn/interacting-system-managing-memory/supplement/ywo87/process-creation

Process Creation

Process Creation

While we do not want to delve into too many details of process creation, we do want to briefly mention a few points to assure you that there is no magic involved in making new programs, or getting the command line arguments to **main**. When the command shell (or any other program—the command shell is just a normal program) wants to run another program, it makes a couple of system calls. First, it makes a call to create a new process (**fork**). This new process (which is an identical copy of the original, distinguishable only by the return value of the **fork** system call) then makes another system call (**execve**) to replace its running program with the requested program. The **execve**system call takes an argument specifying the file with the binary (or script) to run, a second argument specifying the values to pass the new program as **argv** (which must end with a NULL), and a third argument specifying the values to pass for **envp** (even if main ignores **envp**, these are still passed to the new program so they can be accessed by the various environment manipulation functions mentioned in the previous subsection.

When the OS executes the **execve** system call, it destroys the currently running program (the system call only returns on an error), and loads the specified executable binary into memory. It writes the values of **argv** and **envp** into memory in a pre-agreed upon format (part of the *ABI*—application binary interface: the contract between the OS and programs about how things work). The kernel then sets the execution arrow to a starting location specified in the executable binary (On Linux with gcc*,* the entry point is a symbol called *_start*—but the details are platform specific).

This startup code (which resides in an object file that is linked with any C program you compile—unless you request explicitly for it not to be) then calls various functions which initialize the C library. This startup code also counts the elements of**argv** to compute the value of **argc** and eventually calls **main**. Regardless of how **main** is declared, it always passes in **argc**, **argv**, and **envp**—if **main** is declared with fewer arguments, it still receives them but simply ignores them.

When **main**returns, it—like all other functions—returns to the function that called it. In the case of main, the caller is this startup code. This code then performs any cleanup required by the C library, and calls **exit** (which quits the program), passing in the return value of **main**as the argument of **exit**—which specifies the exit status of the program.

The shell (or other program that ran the program in question) can make a system call, which waits for its "child" process(es) to exit, and then collects their return values.

✓

Completed