**Chapter 5: Interlude: Process API**

Discuss the process creation in UNIX systems. UNIX presents one of the most intriguing ways to create a new process with a pair of system calls: **folk()** and **exec().** A third routine, **wait()**, can be used by a process wishing to wait for a process it has created to complete.

5.1. The **folk()** System Call:

The folk() system call is used to create a new process. However, it is certainly the strangest routine you will ever call. For example,

Text, letter

Description automatically generated

The output is

Text

Description automatically generated

First, as we run the program, it prints out the process identifier (29146). As we create a folk, the created process is an (almost) exact copy of the calling process. That means that to the OS, it now looks like there are two copies of the program p1 running, and both are about to return from the fork() system call. The newly-created process (the child) does not start running at main(). Rather, it comes to life as if it had called fold() itself.

The child is not an exact copy. Although it has its own copy of the address space, its own register, its own PC, etc., the value it returns to the caller of folk() is different. Specifically, while the parent receives the PID of the newly-created child, the child receives a return code of zero. This differentiation is useful, because it is simple then to write the code that handles the two different cases.

The output is not deterministic because we now have two active processes and on a single CPU, either of the child or parent might run. In the above image, the parent ran first. In other cases, it might be

Text

Description automatically generated

This is the result of CPU **scheduler** as it determines which process runs at a given moment in time. This **non-determinism**, as it turns out, leads to some interesting problems, particularly in **multi-threaded programs**.

5.2 The **wait()** System Call

Sometimes, it is quite useful for a parent to wait for a child process to finish what it has been doing. This can be accomplished by the wait() system call.

Text, letter

Description automatically generated

In this example, the parent process calls wait() to delay its execution until the child finishes executing. When the child is done, the wait() returns to the parent.

This makes the output deterministic because the child will always print first, and the parent must wait for the child to be finished. Even if the parent runs first, the wait() system call won’t return until the child is executed.

5.3 The **exec()** System Call

This system call is useful when you want to run a program that is different from the calling program. For example, the folk in the example is 5.2 is only useful if you want to keep running copies of the same program. However, if we want to run a different program, exec() will do it.

Text, letter

Description automatically generated

In this example, the child process calls execvp() in order to run the program wc on the source file p3.c. exec() is not normal either. What it does is that given the name of the executable (wc), and some arguments (p3.c), it loads code (and static data) from that executable and overwrites its current code segment (and current stat data) with it. The heap and stack and other parts of the memory space of the program are re-initialized. Then the OS simply runs that program, passing in any arguments as the argv of that process. Thus, it does not create a new process; rather, it transforms the currently running program (formerly p3) into a different running program (wc). After the exec() in the child, it is almost as if p3.c never ran; a successful call to exec() never returns.

**5.4 Why? Motivating The API.**

The separation of fork() and exec() is essential in building a UNIX shell, because it lets the shell run code after the call to fork() but before the call to exec(); this code can alter the environment of the about-to-be-run program, and thus enables a variety of interesting features to be readily built.

The shell is just a user program. It shows you a prompt and then waits for you to type something into it. When we type in a command, in most cases, the shell then figures out where in the file system the executable resides calls folk() to create a new child process to run the command, calls some variant of exec() to run the command, and then waits for command to complete by calling wait(). When the child completes, the shell returns from wait() and prints out a prompt again, ready for the next command.

The separation of fork() and exec() allows the shell to do a whole bunch of useful things rather easily.



In the above example, the output of the program wc is **redirected** into the newfile.txt. The way shell accomplishes this is when the child is created, before calling exec(), the shell closes standard output and opens the file newfile.txt. Any output of the soon-to-be-running program wc are sent to the file.

Example of such **redirection**:

Text

Description automatically generated

The reason this redirection works is due to an assumption about how the operating system manages file descriptors. Specifically, UNIX systems start looking for free file descriptors at zero. In this case, STDOUT FILENO will be the first available one and thus get assigned when open() is called. Subsequent writes by the child process to the standard output file descriptor, for example by routines such as printf(), will then be routed transparently to the newly-opened file instead of the screen.

Text

Description automatically generated with medium confidence

We cannot see the output when we execute the program because it has been redirected into p4.output, although the program did call folk() to create a new child, and the run the wc program via a call to execvp(). The cat command is to peek the content of p4.output to the screen.

UNIX pipes are implemented in a similar way, but with the pipe() system call. In this case, the output of one process is connected to an in-kernel pipe (i.e., queue), and the input of another process is connected to that same pipe; thus, the output of one process seamlessly is used as input to the next, and long and useful chains of commands can be strung together. As a simple example, consider looking for a word in a file, and then counting how many times said word occurs; with pipes and the utilities grep and wc, it is easy; just type *grep -o foo file | wc -l* into the command prompt and marvel at the result.

**5.5 Process Control and Users**

kill() system call is used to send signals to a process, including directives to pause, die, and other useful imperatives. In most UNIX shells, a certain keystroke combinations are configured to deliver a specific signal to a running process. For example, ctrl+C sends a SIGINT signal to interrupt the process (normally to terminate it) and ctrl+Z sends a SIGTSTP (stop) signal thus pausing the process in mid-execution.

Who can send a signal to a process and who cannot? Generally, the systems we use can have multiple people using them at the same time; if one of these people can arbitrarily send signals such as SIGINT (to interrupt a process, likely terminating it), the usability and security of the system will be compromised. Thus, modern systems include a strong conception of the notion of a user. The user, after entering a password to establish credentials, logs in to access to system resources. The user can then launch one or many processes and exercise full control over them. Users generally can only control their own processes; it is the job of the operating system to parcel out resources (such as CPU, memory, and disk) to each user (and their processes) to meet overall system goals.

**5.6 Useful Tools:**ps: see which processes are running (**man pages** will provide useful flags to pass to ps).

top: displays the processes of the system and how much CPU and other resources they are eating up.

kill: send signals to processes, as can the slightly more user friendly killall.

**Superuser** (sometimes called **root**) can administrate the system, kill an arbitrary process that is not started by this user, etc.

MenuMeters can see how much CPU is being utilized at any moment in time.