Lecture 15 - fork() (last updated: 2023-10-29)

In UNIX, each instance of a running program is called a process, uniquely identified by a positive number called the process ID (PID). C programs can obtain their own PID at runtime using the getpid() function.

Processes may spawn more processes, which we call "child" processes. The process that spawned the child is known as the child's "parent." A process can obtain the PID of its parent (PPID) using the getppid() function. All processes "descend" from the "init" process, whose PID is 1 and is the first process to run when a UNIX system boots.

Spawning processes with fork()

A process spawns a child process using the fork() system call:

```
pid_t fork(void);
```

It creates the child process's memory address space by duplicating the parent's address space; after returning from the fork() system call, both processes will resume execution from the same place in the code.

The parent and the child will be able to distinguish themselves from each other using the return value of fork(), which will return 0 to the child process, and the PID of the child process to the parent process:

```
pid_t pid = fork();

// Both the parent and child will resume execution here.

if (pid == 0) {
    // Child process
    printf("This is the child, my PID is %d\n", getpid());
} else {
    // Parent process
    printf("This is the parent, my child's PID is %d\n", pid);
}
// Both the parent and child will print out "Hello"
printf("Hello\n");
```

Here is the output of the program:

```
$ ./hello
This is the parent, my child's PID is 830029
Hello
This is the child, my PID is 830029
Hello
```

Since BOTH the parent and the child will resume execution from the point where fork() returns, "Hello" is printed twice!

The parent and child will execute concurrently after returning from fork(), meaning there is no guarantee of their execution order relative to one another. In other words, the pace and progress of their execution is independent from each other. In general, all processes execute concurrently with one another unless they explicitly coordinate with one another. We will learn one way to do it using waitpid().

When a parent spawns a child process by calling fork(), the entire address space of the parent process gets duplicated to create an address space for the child process. It is important to realize that, while the child process's address space starts out from a replica of the parent's address space, the two address spaces are completely separate. They share nothing; each process has their own stack, their own heap, and their own static variables. They may call different functions to grow their stacks in different ways, allocate different amounts of memory on their heaps, and write different values into their copies of static variables.

A single parent process may spawn multiple children by calling fork() multiple times. Similarly, child processes can spawn children of their own by calling fork(). The parent-child relationships form a process family tree rooted at the init process, the ancestor of all processes on a UNIX system.

Exit status codes and waitpid()

When a process terminates, it does so with an integer known as the exit status code. By convention, an exit status code of 0 indicates success, while a non-zero code indicates some kind of error (whose specific meaning is determined by each application).

When a process returns from its program's main() function, the return value of main() is used as the exit status. A process may also immediately terminate at any point during its execution by calling the exit() function, passing the status code as an argument to exit().

```
void foo(void) {
    // ...
    if (err) {
        exit(2);   // Terminate the program with exit status code of 2
    }
    // ...
}

int main(void) {
    foo();
    if (bar() == -1) {
        return 1;  // Terminate the program with exit status code of 1
    }
    return 0;      // Terminate the program with exit status code of 0
}
```

A parent process can wait until its child process terminates, and retrieve the child process's exit status code. In UNIX parlance, this is called "reaping" the child. Parent processes reap their children using the waitpid() system call:

```
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

The first parameter of waitpid() specifies which process to wait for; the parent may pass the PID of a specific child process to only wait for that process, or pass the number -1 to wait for any of its child processes. When waitpid() returns, it will return the PID of the child process it reaped, or -1 on error.

The second parameter, wstatus, tells waitpid() where to write the terminated child's exit status code. For example:

```
pid_t pid = fork();

if (pid == 0) {
    // Child process
    // ...
} else {
    // Parent process
    int wstatus;
    waitpid(pid, &wstatus, 0);
    printf("Child (PID = %d) exited with: %d\n", pid, WEXITSTATUS(wstatus));
}
```

Note that we use the WEXITSTATUS() macro to read the status code from wstatus. This is because the waitpid() system call packs other information -- like why and how the child terminated -- into the bits of that integer. If we want to have the parent process simply wait for the termination of the child process, and are not interested in retrieving the exit status code, we can pass NULL as the second argument.

The third parameter, options, lets us specify options to change waitpid()'s behavior. By default -- when options is 0 -- waitpid() will return the PID of the child process if the specified child process has already terminated and is waiting to be reaped, but if the child process is still running, waitpid() will block until the child terminates.

We can change this behavior by passing the WNOHANG option. With WNOHANG option, waitpid() will never block. If the specified child process has already terminated (or -1 was passed for the pid parameter and there was a terminated child process waiting to be reaped), then the PID of the terminated child process is returned. If the specified child is still running (or -1 was passed and every child is still running), then 0 is returned. On error, -1 is returned.

A child process that has terminated but has not yet been reaped is called a "zombie process." The OS needs to keep around some metadata about the zombie process to report to the parent if and when it is reaped, such as the PID and the exit status code. Though this metadata consumes a minimal amount of the system's resources, it can add up as the system maintains metadata for a large number of zombies, so parent processes mustn't forget to reap their children!

However, when a parent process terminates before its children do, the child process is said to have become an "orphan process." All orphan processes are adopted by the init process and automatically reaped when they terminate. Unlike zombie processes, orphan processes are not inherently bad, and are actually quite common in UNIX systems. In particular, long-running processes, called "daemons," often run as orphan processes, acting as servers or performing background tasks (e.g., taking periodic file system snapshots).

Executing programs with exec()

In UNIX, fork() is the only way to create a new process. But when a child process is created by fork(), the child process runs the same exact program as its parent process (although it can take different paths in the program code). How can a UNIX system run a variety of different programs then? The answer is through the exec() family of system calls. Among the six variants of the exec() family -- execl, execlp, execle, execv, execvp, execvpe -- we will describe the two basic ones: execl() and execv().

The execl() function replaces the program that the current process is running with another program. In other words, when a process calls execl(), the process continues -- i.e., it retains its process ID and who its parent process is -- but its memory address space gets completely gutted. The program code is replaced by a new program; the stack, heap, and static data section get reset for the execution of the new program; and the new program starts to run from its main() function. Here is an example:

```
// hello.c

int main()
{
    printf("I'm about to become 'echo hello'...\n");

    // execl(const char *program_to_run,
    //       const char *argv0,
    //       const char *argv1,
    //       ...
    //       (char *)NULL

    execl("/bin/echo", "/bin/echo", "hello", (char *)NULL);

    printf("If all goes well, this line should NOT be printed.\n");
}
```

And here is the output:

```
$ ./hello
I'm about to become 'echo hello'...
hello
```

When the hello.c program called execl(), the process morphed into the /bin/echo program, printed "hello" to the terminal, and terminated. The last printf() statement never gets executed because the whole hello program got completely erased when it called execl() to morph itself into a different program, /bin/echo.

execl() is a variadic function whose parameters are the name of the program to run, and a list of strings to form the argv array to pass to the new main() function, ending with (char *)NULL to terminate the argv array.

The execv() variant takes the argv array directly:

```
int execv(const char *program_to_run, char **argv);
```

We could replace the execl() call above with the following execv() call:

```
char *argv[] = { "/bin/echo", "hello", NULL };
execv(argv[0], argv);
```

A common pattern is to combine fork() and exec(), where the parent process waits for its child to execute another program:

```
    pid_t pid = fork();

    if (pid == 0) { // Child process

        char *argv[] = { "/bin/echo", "hello", NULL };
        execv(argv[0], argv);

        die("exec");

    } else { // Parent process

        waitpid(pid, NULL, 0);
    }
```

In fact, this is basically what shell programs like Bash and Zsh do! They read in your command from stdin, fork() and exec() a child process to run your command, and waitpid() for that child to terminate before prompting you for another command. Here is a rudimentary implementation of a shell program:

```
    int main()
    {
        char    buf[100];
        pid_t   pid;
        int             status;

        printf("PROMPT> ");
        while (fgets(buf, sizeof(buf), stdin) != NULL) {

            if (buf[strlen(buf) - 1] == '\n') {
                buf[strlen(buf) - 1] = 0; // replace newline with '\0'
            }

            pid = fork();

            if (pid < 0) {
                die("fork error");

            } else if (pid == 0) { // child process

                char *argv[] = { buf, NULL };
                execv(argv[0], argv);

                // or alternatively:
                // execl(buf, buf, (char *)0);

                die("execl failed");

            } else { // parent process

                if (waitpid(pid, &status, 0) != pid) {
                    die("waitpid failed");
                }
            }
            printf("PROMPT> ");
        }
    }
```