Lab 6: Processes

CSC 357, Fall 2023

Demonstration and Submission Deadline: 13 Nov, 2023

10 Points

Motivation

It is likely that to this point in our use of computers we have only initiated the creation of additional processes by specifying the program to run as a command to the shell, by double-clicking in a graphical user interface (a graphical shell, if you will), or by tapping an icon on a phone or tablet. Each of these methods for creating processes is useful and important as a user of a device, but for this lab we will begin our exploration of process creation from within our own programs.

Course Learning Objectives

The lab addresses the following course learning objectives.

Direct

- Distinguish language features from operating system features.
- Consider basic resource management in the development of programs.
- Gain experience with low-level programming in the UNIX environment.
- Discuss the architecture of the UNIX operating system from a system-programmer's perspective and be able to write programs that use operating system services (system calls) directly.

Setup

This lab does not require any provided files, so there is no repository to clone.

Lab Tasks

There are 4 separate tasks listed below, but first a word of caution.

Limiting Processes

In this lab you will work with the fork system call so there is a real risk that a simple bug in your program will result in many, many processes being spawned (often referred to as a “fork bomb”). Here are two techniques that you need to use to limit the impact of such an honest mistake.

- Limit in the shell: Use ulimit -Su 300 to limit the number of processes that you can spawn (you can choose a number other than 300, but choose something relatively small). You must type this each time you start a new shell.
- Limit in your program: Since you may forget the above when you open a new shell, you might prefer to add the following function to your program and call it as the first action in main.

```
void limit_fork(rlim_t max_procs)
{
   struct rlimit rl;
   if (getrlimit(RLIMIT_NPROC, &rl))
   {
      perror("getrlimit");
      exit(-1);
   }
   rl.rlim_cur = max_procs;
   if (setrlimit(RLIMIT_NPROC, &rl))
   {
      perror("setrlimit");
      exit(-1);
   }
}

int main(void)
```

```
{
   limit_fork(300);
   /* continue with program logic here */
   return 0;
}
```

Task 1: fork

Write a program named f_test. This program must take a single integer, N, as a command-line argument. This program must fork a child process. The child must print the odd numbers from 1 to N (inclusive) and then exit(), while the parent prints the even numbers from 1 to N (inclusive). The parent process should properly wait for the child process to terminate.

For the odd numbers, use “%d\n” as the format string for printf. For the even numbers, use “\t%d\n” as the format string for printf.

Run the program with a large enough value for N to observe an interleaving in the output.

Task 2: odds and evens

This task is done in prepration for the next; *this task does not use fork*.

Write two C programs: odds and evens. Each of these programs must take a single integer, N, as a command-line argument. odds prints the odd numbers from 1 to N (inclusive). evens prints the even numbers from 1 to N (inclusive); formatting for evens and odds should be as in task 1.

Task 3: exec

Write a program named fe_test. This program will behave similarly to the program from the first part, but will use one of the "exec" system calls to execute the programs written in the Task 2. The parent should fork two child processes. One child process will "exec" the odds program. The other will "exec" the evens program. The parent process should properly wait for both child processes to terminate but allow them to execute concurrently (i.e., wait after both children have been created).

Task 4: Redirection

Write a program named to file. This program will take two command-line arguments. The first is the name of another program and the second is the name of a file (which need not yet exist).

Your program is going to use "exec" to run the specified program. Before doing so, however, your program will need to open the specified file and take the appropriate steps to redirect standard output to the specified file. This setup is done so that the exec'd program will write its output to the file.

Recall discussions of file descriptors (including standard uses), open, and dup.