CprE 308 Laboratory 2: Introduction to Unix Processes

Department of Electrical and Computer Engineering
Iowa State University

1 Submission

For this lab you only need to fill the given report template. You will turn it in on Canvas. You can increase/decrease the answer boxes according to your answers.

Here is some information about the items you need to fill:

- 30 pts - A cohesive summary of what you learned through the various experiments performed in this lab. This should be no more than two paragraphs. Try to get at the main idea of the exercises, and include any particular details you found interesting or any problems you encountered.
- 60 pts - A write-up of each experiment in the lab. Each experiment has a list of items you need to include. For output, copy/paste or take a screenshot of the results from your terminal, and summarize when necessary. For explanations, keep your answers concise, but include all relevant details.

2 Resources

The following Unix calls are used in this lab: `getpid`, `getppid`, `sleep`, `fork`, `exec`, `wait`, and `kill`. For help, read the comments below. Other resources include the manual pages (e.g. `man getpid`), web searches, or any Unix reference books you may have at hand.

2.1 Comments on Learning about Unix Processes

1. When a system is booted, the first userspace process is `/sbin/init` (or `systemd` on newer systems), and has a PID of 1. This process will in turn launch startup scripts and eventually login prompts. If you do a `ps -el`, you should see that process 1 is init or systemd. Init will be the ancestor of all other processes on the system. With systemd, you may see that a second process (kthreadd) is the root of the process tree instead.
2. When you login or start a terminal, a process for the shell is started. The shell will then launch other processes, which will be children of the shell. If the parent process dies (for example, you exit the shell), init will adopt the orphaned processes (on Linux and many other Unix variants).
3. The *status* of a Unix process is shown as the second column of the process table (viewed by executing the `ps -el` command). Some of the states are R: running, S: sleeping, Z: zombie.
4. The call to the `wait()` function results in a number of actions:
   - If the calling process has no children, wait() returns -1.
   - If the calling process has a child that has terminated (a zombie), that child's PID is returned and it is removed from the process table

- Otherwise, the call blocks (suspends the process) until a child terminates. The `waitpid()` function allows the `WNOHANG` flag to be specified to return immediately in all cases.

5. Read the unix man pages (review from Lab 1). The `man` command displays on-line manual pages with information on the command, function, or file specified by name. Unix manual pages are gathered into several sections, each section is related to a given topic. Two different sections could contain two man pages with the same name. Ex. `man 1 chmod` displays section 1 of the chmod man pages. Section 1 of the man pages deals with user commands. `man 2 chmod` displays section 2 of the chmod man pages. Section 2 of the man pages deals with system calls, which will be relevant for this lab.

3 Experiments

Execute the C programs given in the following experiments. Observe and interpret the results. You will learn about child and parent processes, and much more about Unix processes in general, by performing the suggested experiments. We recommend that you ***read through an entire experiment before starting it***.

3.1 Process Table

Run the following program twice, both times as a background process (i.e. suffix it with an ampersand "`&`"). Once both processes are running as background processes, view the process table (use `ps -el` to view all processes on the system, or `ps -l` to view only the processes running on your terminal). If you see the message "I am awake", the process has finished and will no longer show up in the process table.

```
int main() {
    printf("Process ID is: %d\n", getpid());
    printf("Parent process ID is: %d\n", getppid());
    sleep(120); /* sleeps for 2 minutes */
    printf("I am awake.\n");
    return 0;
}
```

If there's too much output, you can scroll back in the terminal window using Konsole, you can pipe the output to less, or you can redirect the output to a file.

```
$ ps -l | less
```

will pipe the output to the program `less`, and

```
$ ps -l > out
```

will redirect the output to a file named `out` (overwriting it if it existed before).

The first line from a "ps -l" is duplicated here:

```
F S     UID     PID  PPID   C PRI    NI ADDR     SZ WCHAN     TTY               TIME CMD
```

You should be able to find descriptions of these fields in the ps man page (`man ps`). This time we'll help you decode the column headers: S is the state, PID and PPID are the process ID and parent process ID respectively, and CMD is the name of the program.

6 pts In the report, include the relevant lines from "ps -l" for your programs, and point out the following items:

- process name
- process state (decode the letter!)
- process ID (PID)
- parent process ID (PPID)

2 pts Repeat this experiment and observe what changes and doesn't change.

2 pts Find out the name of the process that started your programs. What is it, and what does it do?

Note: Now that you know how to find a PID for a running program, you can use `kill` to terminate an unwanted process. Just type `kill` followed by the process ID. If a process refuses to terminate, `kill -9` followed by the process ID will terminate any process you have permission to kill. The kill command is capable of sending many different *signals* to processes. Uses `kill -l` to list out the possible signals. There's also `killall`, which will kill all processes with the same name (`man killall`).

3.2 The fork() system call

Run the following program and observe the number of statements printed to the terminal. The fork() call creates a child that is a copy of the parent process. Because the child is a copy, both it and its parent process begin from just after the fork(). All of the statements after a call to fork() are executed by both the parent and child processes. However, both processes are now separate and changes in one do not affect the other.

```
int main() {
    fork();
    fork();
    usleep(1);
    printf("Process %d's parent process ID is %d\n", getpid(), getppid());
    sleep(2);
    return 0;
}
```

1 pt Include the output from the program

4 pts "Draw" the process tree (the nodes should be PIDs)

3 pts Explain how the tree was built in terms of the program code

2 pts Try the program again without `sleep(2)`. Explain what happens when the sleep statement is removed. You should see processes reporting a parent PID of 1. Redirecting output to a file may interfere with this, and you may need to run the program multiple times to see it happen.

3.3 The fork() syscall, continued

Finish and run the following program and observe the output. In Unix, fork() returns the value of the child's PID to the parent process, and it returns 0 to the child process.

```
int main() {
    int ret;
    ret = fork();
    if (/* TODO add code here to make the output correct */) {
        /* this is the child process */
        printf("The child process ID is %d\n", getpid());
        printf("The child's parent process ID is %d\n", getppid());
    } else {
        /* this is the parent process */
        printf("The parent process ID is %d\n", getpid());
        printf("The parent's parent process ID is %d\n", getppid());
    }
    sleep(2);
    return 0;
}
```

2 pts Include the (completed) program and its output

4 pts Speculate why it might be useful to have fork return different values to the parent and child. What advantage does returning the child's PID have? Keep in mind that often the parent and child processes need to know each other's PID.

3.4 Time Slicing

"Time slicing" refers to processes sharing time on the CPU according to the OS's scheduling. Run the below program to observe time slicing. On a modern, multi-core system, the parent and child could run simultaneously on separate cores—but access to the output file is still multiplexed. To ensure what you're observing is the effect of CPU-level time slicing, run the program using the following command:

```
taskset 1 [prog-path] > [output-file]
```

This forces Linux to run the program (parent and child) only on CPU core 0.

```
int main() {
    int i = 0;
    int pid = fork();
    if (pid == 0) {
        for(i = 0; i < 500000; i++) {
            printf("Child: %d\n", i);
        }
    } else {
        for(i = 0; i < 500000; i++) {
            printf("Parent: %d\n", i);
        }
    }
}
```

If you don't see interesting output in the output file, you may need to increase the number of iterations.

2 pts Include small (but relevant) sections of the output

4 pts Make some observations about time slicing. Can you find any output that appears to have been cut off? Are there any missing parts? What's going on (mention the kernel scheduler)?

3.5 Process synchronization using wait()

Run the following program, again using `taskset` (though it should do the same thing without!), and observe the result of synchronization using wait().

```
int main() {
    int i = 0;
    if(fork() == 0) {
        printf("Child starts\n");
        for(i = 0; i < 500000; i++) {
            printf("Child: %d\n", i);
        }
        printf("Child ends\n");
    } else {
        wait(NULL);
        printf("Parent starts\n");
        for(i = 0; i < 500000; i++) {
            printf("Parent: %d\n", i);
        }
        printf("Parent ends\n");
    }
    return 0;
}
```

6 pts Explain the major difference between this experiment and experiment 4. Be sure to look up what wait does (`man 2 wait`).

3.6 Signals using kill()

Examine and run the following program to observe one use of kill():

```
#include <stdio.h>
#include <signal.h>

int main() {
    int i = 0;
    int child_pid = fork();
    if(child_pid == 0) {
        while(1) {
            i++;
            printf("Child at count %d\n", i);
            usleep(10000);
        }
    } else {
        printf("Parent sleeping\n");
        sleep(10);
        kill(child_pid, SIGTERM);
        printf("Child has been killed. Waiting for it... ");
        wait(NULL);
        printf("done.\n");
    }
    return 0;
}
```

2 pts The program appears to have an infinite loop. Why does it stop?

4 pts From the definitions of sleep and usleep, what do you expect the child's count to be just before it ends?

2 pts Why doesn't the child reach this count before it terminates?

3.7 The execve() family of functions

Run the following program and explain the results.

```
#include <stdio.h>
#include <unistd.h>

int main() {
    execl("/bin/ls", "ls", (char *)NULL);
    printf("What happened?\n");
}
```

8 pts Read the man page for `execl` and relatives (`man 3 exec`). Under what conditions is the printf statement executed? Why isn't it always executed? (consider what would happen if you changed the program to execute something other than `/bin/ls`.)

3.8 The return value of main()

Run the following program a few times to observe the use of the status argument to wait():

```
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main() {
    int status;
    if (fork() == 0) {
        srand(time(NULL));
        if (rand() < RAND_MAX / 4) {
            /* kill self with a signal */
            kill(getpid(), SIGTERM);
        }
        return rand();
    } else {
        wait(&status);
        if (WIFEXITED(status)) {
            printf("Child exited with status %d\n", WEXITSTATUS(status));
        } else if (WIFSIGNALED(status)) {
            printf("Child exited with signal %d\n", WTERMSIG(status));
        }
    }
    return 0;
}
```

2 pts What is the range of possible exit status values printed for the child process?

2 pts What is the signal value the child process uses to terminate itself, as determined and printed by the parent process?

2 pts When do you think the return value of main() would be useful? Hint: look at the commands `true` and `false`.