ITF22519: Introduction to Operating Systems

Fall Semester, 2021

Lab5: Processes in Linux

Submission Deadline: September 30th, 2021 23:59

In this lab, you will learn how to create a new process in Linux Operating System and do some practices with several system calls such as `fork`, `sleep`, `exec`, `wait`, and `kill`. Before you start, pull the new lab:

```
$ cd Introduction2OS/labs
$ git pull upstream main
$ cd lab5
```

1 About Unix Processes

- When a system is booted, the first user space process is `systemd` or `/sbin/init` depending on your Linux distribution. This process has proccess id (PID) of 1. It will launch startup scripts and eventually login prompts. If you do a `ps -el`, you should see that process 1 is `systemd`. It is the ancestor of all other user processes on the system.
- 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, `systemd` will adopt the orphaned processes.
- The ***`status`*** of a Unix process is shown as the second column of the process table (viewed by executing the `ps` command). Some of the states are `R`: running, `W`: waiting, `S`: sleeping, `Z`: zombie.

Whenever a program or a command executes, there is a process. A process can be run in foreground or background.

- Foreground: Every process when started runs in foreground by default. It receives input from the keyboard and then sends output to the screen. When a command/process is running in the foreground, it takes a lot of time and no other processes can be run because the prompt would not be available until the program finishes processing and comes out.
- Background: A process runs in the background without keyboard input and waits till keyboard input is required. Thus, other processes can be done in parallel with the process running in the background since they do not have to wait for the previous process to be completed. Adding suffix with an ampersand `&` starts the process as a background process.

1.1 Process Table

The program *print_pid.c* prints out its process id and its parent process id, and then sleeps for 2 minutes. Run the program *print_pid.c* twice, both times as a background process. In the meantime, press `Ctrl+C` to terminate the process. When you see the message "I am awake", the proccess is finished and no longer show up on the screen.

```
$ ./print_pid &
Ctrl+C
$ ./print_pid &
Ctrl+C
```

Then,

- Do `ps -el` to view all processes in the sysem
- Do `ps -l` to view only processes running on your terminal
- Do `ps -l | less` to pipe output to the less
- Do `ps -l > output` to direct output the the file named `output`
- What is the process that started your *print_pid.c* programs. What does it do?

The first line from a the command `ps -l` is:

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

A short description of those fields can be found in this link.

1.2 Killing a process

You have known how to find the PID of a process. Now, you can terminate an unwanted process by typing the command `kill` followed by the process ID. If a process refuses to be killed, type `kill -9` followed by the process ID to terminate any process you have permission to kill. You might also try `killall` to kill all processes. For example:

```
$ killall print_pid
```

2 System call

In a computer OS, the user space is what most users interact with when using the computer. It is a portion of memory with restricted permission and access. Under the user space is the kernel space - the actual operating system. Kernel space has complete access to everything on the system.

Because of privileged rights, it is not wise to give any user the access to the kernel space. Therefore, there is a restricted and well-defined interface between user space and kernel space: the system calls. When a system call is called, the program execution stops and execution is switched to the operating system address space. The parameters are passed to the OS through a pre-determined register passing scheme. The operating system will then check if the program has the permissions to perform the requested operations. If yes, the task is completed.

2.1 The fork() system call: Creating a new process

The `fork()` system call is used to create a new process which becomes the child process of the calling process. If the `fork()` does not return a negative value, the creation of the child process is successful. The child process will have PID of 0 and the parent process has a possitive PID. The returned process ID is of type `pid_t` defined in `sys/types.h`. Normally, the process ID is an integer. To create a new process in a C program, the follwing piece of code is needed:

```
#include <sys/types.h>
#include <unistd.h>
// ...
pid_t child;
child = fork();
// ...
```

The child process is the copy of the parent process. After spawning a child process, all of the statements after `fork()` system call are executed by both the parent and child processes. However, both processes are now separate and changes in one do not affect the other. The code fragment below calls fork() once, which results in the creation of one child process. Then, each process prints out its id and the parent's ID.

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

One of the output of this program can be:

```
Process 14427's parent process ID is 6891
Process 14428's parent process ID is 14427
```

Here, the ID of the main function is 14427, the child ID is 14428. The the ID of bash shell is 6891. The process tree of this program is showed below.

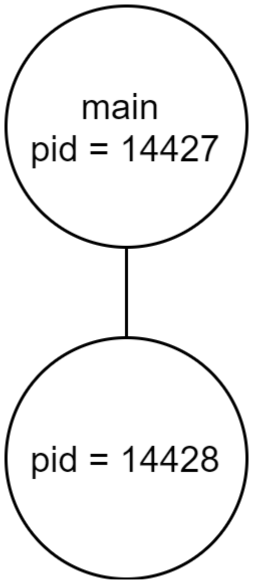

Figure 1: A process tree

Task 1: The following program prints different messages in the child and parent processes. Complete the condition of the if statements in this program and save the code in a *fork_ex2.c* file.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
       pid_t pid;
       pid = fork();
       if ( ) { // ADD YOUR CODE HERE

           printf("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 if ( ){  // ADD YOUR CODE HERE

           printf("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());
      }
     else {
          perror("fork"); // fork() fails; handle error here
          exit(-1);
     }
     sleep(2);
     return 0;
}
```

2.2 The wait() system call: Waiting on a child process

The child process and its parent process are at their own places. Therefore, they execute independently and one can finish before the other. In some situations, this might not be the desired behavior. There are two system calls which guarantee that the parent process waits for its child process to complete. These system calls are `wait()` and `waitpid()`.

`wait()`: This function blocks parent process until the child process (whose PID stored in `child`) exits or a signal is received. After child process terminates, parent process continues its execution after `wait()` system call instruction. The snippet of code for `wait()` is as follows:

```
#include <sys/types.h>
#include <sys/wait.h>
...

int status = 0;
....

pid_t child = fork();
if ( 0 == child){
      // do something here
} else if ( 0 < child ) {
      wait(&status);
      printf("child process is done, status is: %d\n", status);
      // do something else here
     return 0;
} else {
     perror("fork");
     exit(-1);
}
}
```

`waitpid()`: If the parent process has multiple children, the knowedge of a specific child process' termination is of importance. In this case, `waitpid()` system call should be used instead of `wait()`. The `waitpid()` specifies which child process the parent process is waiting for. The snippet of code for `wait()` is as follows:

```
#include <sys/types.h>
#include <sys/wait.h>
...
int status = 0;
...
child = fork();
if(child == 0){
       //does something here
}
else if(child > 0){
        waitpid(child, &status, 0);
        printf("child process is done, status is: %d\n", status);
        // do something else here
        return 0;
}
else {
        perror("fork");
        exit(-1);
}
```

For more information about `wait()` and `waitpid()`, use `man page` or google it.

2.3 The execve() system call family: Making a process run another program

The `fork()` system call is useful for splitting a process into two, but it is not very useful to execute new programs for some reasons. To start a new program, a different system call which is `execve()` family of system call must be used. The following snippet is needed:

```
#include <unistd.h>
.
.
.
execve(programExecutable, argArray, envArray);
.
.
.
```

The process should run the programprogramExecutable with the arguments argArray and environments envArray. For more information about the exec family of function calls, use `man page`:

```
$ man 3 exec
$ man 2 execve
```

You can also go through this tutorial exec for more information about the usage of the `exec` family.

Task 2: Run the following program and explain the result.

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

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

2.4 The kill() system call: Sending a signal to another process

It is not that the `kill()` system call is used to terminate another process. It is used to send all kinds of signal to a process. Snippet for using `kill()` system call is as the following:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

In the `kill()` function, the first argument is the PID of the process you want to send a signal to. The second argument is the signal you want to send. If the call to `kill()` is successful, it returns 0. Otherwise, it returns a negative value.

3 Exercises

3.1 Exercise 1 (30 points)

Compile and execute the program *`fork_ex1.c`*.

- Include the output of the program in your lab5 report.
- Draw the process tree and label each process with its PID.
- Explain how the tree was built.
- Remove `sleep` statement in the code. Run the code and explain what happens.

3.2 Exercise 2 (30 points)

Compile and execute the programs *`no_wait.c`*, *`wait.c`*, *`waitpid.c`*.

- Redirect the output of each program to a file.
- Concatenate above three output files and include the result into your report.
- Explain the difference of the three programs.

3.3 Exercise 3 (40 points)

- The program *`kill.c`* has `sleep` statement for the parent process and `usleep` statement for the child process. Use `man page` to know more about about `sleep` and `usleep`. What do you expect the child process to count just before it finishes? Why?
- Run the program and add the output in your report.
- The program has an infinite loop `while (1)`. Why does it stop even with the infinite loop?
- If the child does not reach your expected count in the first question, explain the reason.

3.4 Exercise 4 (20 points)

- What is a zombie process?
- Write a C program that creates a zombie process and describe how to verify that the zombie process has been created.

4 What To Submit

Complete the exercises in this lab. Then, put all of files into the **lab5** directory of your repository. Make a report for each exercise. After that, run `git add .` and `git status` to ensure the file has been added and commit the changes by running `git commit -m "Commit Message"`. Finally, submit your files to GitHub by running `git push`. Check the GitHub website to make sure all files have been submitted.