FIT2100 Tutorial #4 Process Management and Job Control in Unix/Linux Week 7 Semester 2 2020

September 11, 2020

Revision Status:

`$Id: FIT2100-Tutorial-03.tex, Version 2.0 2018/08/14 17:30 Jojo $`
`Updated by Muhammed Tawfiqul Islam, September 2020.`

Acknowledgements

2016-2018 versions written by Jojo Wong. Content presented in this tutorial was adapted from the courseware of FIT3042 prepared by Robyn McNamara. Some parts of the content were adapted from the following texts:

- David A. Curry (1996). *UNIX Systems Programming for SVR4*, O'Reilly.
- W. Richard Stevens and Stephen A. Rago (2013). *Advanced Programming in the Unix Environment (Third Edition)*, Addison-Wesley.

Contents

1 Background

In this tutorial, we will learn about how processes can be created and managed in Unix/Linux using system calls in C. We will also discuss how processes (or jobs) are managed within the command-line shell itself.

You should complete the following reading (Section 2) before attending the tutorial. You should also prepare the solutions for the practice tasks given in Section 3, including the assessed task.

2 Pre-tutorial Reading

2.1 Working with Processes

A process can be regarded as an instance of a program that is currently loaded into the memory and being executed. It is possible for multiple instances of the same program to be running at the same time within different processes.

In the Unix/Linux environment, the execution state of each process, which a program is running within it, and a lot of other state information are all managed in one of the OS data structures called the *process table*.

Each process has a unique process identifier (*process ID* or simply *PID*) which is a non-negative integer, usually ranging from 0 to about 32,000. The process ID is actually used as an *index* on the process table to obtain the context information associated with a particular process.

The process ID of a process can be obtained through a system call with the `getpid()` function defined in the `unistd.h` (Unix standard) library.

The special return data type for this function (`pid_t`), is actually just an integer, defined to hold a suitable range of values for process IDs.

```
1 #include <unistd.h>
2 #include <sys/types.h>  /* pid_t is defined in sys/types.h */
3
4 pid_t getpid(void);
```

Process Table Some of the information that the OS stores about each process in the process table includes:

Variable	Description
PID	a unique process ID
UID	the user ID of the owner of the process
PPID	the PID of the parent process
TTY	the terminal ID associated with the process (if it is running in a console mode)
STAT	the status of the process

The Unix command `ps` (process status) presents the current state of the process table. To view a list containing all the state information about all the processes currently running on your computer system, use the Unix command `ps` with the options `-e` and `-f`:

```
$ ps -e -f
```

2.2 Creating Processes in C

The system call routine — `fork()` — provided in the `unistd.h` library can be used to create (or *spawn*) a new process from an existing process in Unix/Linux. The new process created as the result of the invocation of `fork()`, starts out as a copy of the parent process.

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

pid_t fork(void);   // Does not take any arguments
```

After executing a `fork()`, *how can we distinguish the parent process from the child process?*

Interestingly, `fork()` is invoked once but *returns twice* — once in the parent process, and once in the newly-created child process. In the parent process, `fork()` returns the process ID (`PID`) of the child process. In the child process, however, `fork()` returns `0`. By checking the return value of `fork()`, we can then determine whether execution is continued in the parent process or the child process.

The reason of returning the child's process ID to the parent is that a parent process can indeed spawn many child processes, and there is no function available to retrieve the process IDs of its children. The reason for returning 0 from `fork()` to the child is that each process can only

have one parent, and the child process can always obtain its parent's process ID by invoking the `getppid()` function.

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

pid_t getppid(void);
```

(**Note:** If `fork()` returns the value of `-1` to the parent process, this indicates that the child process could not be created.)

Parent and Child Processes

After a `fork()`, the parent and child processes have a nearly identical state:

- the child's environment variables are inherited copies of the parent's;
- the child's program code is exactly the same as the parent's;
- the child has its own copy of the same file descriptors (`stdin`, `stdio`, `stderr`) as the parent; however, sharing the same file offset for each descriptor.

There are some differences between the child process and its parent, where:

- the parent and child processes do not share the same memory space (they have their own copies of global variables);
- the child process is issued with its own process ID (`PID`);
- the child process's `PPID` should be the same as the parent's `PID`.

The parent process and the child process are managed separately as far as the OS kernel is concerned.

2.3 Loading and Executing New Programs

Having two copies of the same program in main memory is not doing anything useful.

Often times, we would want the newly created process (i.e. the child process) to load up and execute a different program. This can be accomplished by using one of the several functions from the `exec()` family.

```c
#include <unistd.h>

int execl(const char *path, const char *arg0, ..., const char *argn,
          /* (char*) NULL */);
int execv(const char *path, const char *argv[]);
int execle(const char *path, const char *arg0, ..., const char *argn,
          /* (char*) NULL */, const char *envp[]);
int execve(const char *path, char *const argv[], const char *envp[]);
int execlp(const char *file, const char *arg0, ..., const char *argn,
          /* (char*) NULL */);
int execvp(const char *file, const char *argv[]);
```

The main difference in all these different forms of `exec()` function, is that the first four functions take a *pathname* argument, and the last two take a *filename* argument. If the *filename* argument specified with a *slash*, it is considered as a pathname for the executable file (or program).

Otherwise, the executable file (or program) is searched for in the directories listed in the `PATH` environment variable; and the first such file encountered is then executed.

Note: *Environment variables* are not C variables. They are special string variables maintained by the OS itself. They normally exist before a new process starts, and are inherited from the parent process. If you are interested to see to the current value of `PATH` defined in your system, use the following Unix command.

```
$ echo $PATH
```

Arguments: One other key difference among these functions is about the passing of the argument list. The functions — `execl`, `execle`, and `execlp` — require each of the command-line arguments to the new program to be passed on as separate arguments.

Note that the end of the argument list is indicated by a `NULL` pointer. If this `NULL` pointer is specified by a constant value of `0`, it should be cast into a `char` pointer (`char *`). For the other three `exec()` functions, they accept the command-line arguments as an array of `char` pointers (`char *argv[]`).

By convention, the argument list (`argv`) should have at least one element — it is essentially the name of the process as displayed by the `ps` command. The `argv[0]` is usually given as the pathname of the executable file (or program) itself, or the last component of the pathname.

Environment Strings: Another difference that should be noted is the functions — `execle` and `execve` — which allow us to pass on an array of pointers to the environment strings as an argument, which are to be assigned to the environment variables for the new process.

For other `exec()` functions that do not accept this additional argument, the new process will inherit the parent process's environment strings.

(**Note:** If your system supports functions such as `setenv` and `putenv`, the environment variables associated with a process can be changed but with some restriction.)

Here is an example of how the `exec()` family of functions can be used. Suppose that we want the new process to run the Unix command "`ls -alR`", this can be accomplished by writting the following C code:

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

int main() {
    pid_t pid;
    char *params[] = {"/bin/ls", "-alR", 0};

    if ((pid = fork()) < 0) {     /* create a new process */
        perror("fork error");
        exit(1);
    } else if (pid == 0) {          /*  in the child process */
        execv("/bin/ls", params);
    } else {                        /* in the parent process */
        sleep(10);
    }
    exit(0);
}
```

How, actually, does a command shell work? By now, you should be able to figure out how the command shells (such as `bash`) work. So long as the user does not exit or logout, the command shell does the following:

1. Display the prompt (i.e. "$")
2. Read the command line from the keyboard
3. Establish that the first "word" in the command line represents an executable program
4. If `fork()` returns `0`, run "execv(program, arguments)"
5. Otherwise, wait for the child to finish

2.4 Waiting for Process Termination

The last step in the command shell algorithm is indeed to wait for the child process to terminate and to collect its termination status.

In the above example, we were just assuming that by putting the parent to sleep for 10 seconds, this would be sufficient for the child process to complete its execution. A more robust way to do this to *suspend* the operation of the parent process until the child process terminates.

The `wait()` and `waitpid()` functions from the `<sys/wait.h>` library are two of the system calls in C that can be invoked to achieve this. With these functions, we can determine the process ID of the child process that has terminated and its status information can be retrieved. Further, we are also able to display useful error messages if a child process did not terminate cleanly.

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

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Some differences between these two functions are that:

- `wait()` can block the calling process (the parent process) until a child process terminates; however `waitpid()` has an option that prevents it from being blocked.
- `waitpid()` does not wait for the child that terminates first; it has a number of options that control which child process it waits for.

Note that `waitpid()` provides greater control over waiting for processes, in which the `pid` argument is assigned with different control values:

Value	Meaning
`pid == -1`	Wait for any child process
`pid > 0`	Wait for the child whose process ID is equal to `pid`
`pid == 0`	Wait for any child process in the same process group
`pid < -1`	Wait for any child process whose process group ID is equal to the absolute value of `pid`

Termination Status The termination status of a child process is stored in the memory location pointed by the `status` argument. If the calling process (i.e the parent process) does not care about the child's termination status and is only interested in waiting for the child to terminate, the `status` argument can be given as a `NULL` pointer.

There is a set of macros (symbolic constants) defined in `sys/wait.h` that can be used to decode the various types of termination status returned by the `wait()` and `waitpid()` functions. Some of the macros are summarised below:

Macro	Description
`WIFEXITED`	Evaluate to `True` if `status` was returned for a child that terminated normally.
`WIFSIGNALED`	Evaluate to `True` if `status` was returned for a child that terminated abnormally due to an uncaught signal.
`WIFSTOPPED`	Evaluate to `True` if `status` was returned for a child that is currently stopped.
`WIFCONTINUED`	Evaluate to `True` if `status` was returned for a child that has been continued from a stopped state.

Zombie Processes After a process has terminated, the operating system keeps track of its status information (in the process table) in case the parent process issues the `wait()` call. If the parent process never does a `wait()`, the information of the child process is never cleared out. The child process is regarded as a "zombie" — i.e. hanging around and occupying an entry in the process table.

Remark: The use of `fork()` and the exec() functions under Unix/Linux is not the same as *multithreading*. To use threads, check out the `pthreads` library.

2.5 Job Controlling in Unix/Linux

Now, let's look at how processes (or jobs) are managed in the Unix's command shell itself.

By default, a Unix shell (e.g. the `bash` shell) will spawn a child process to run your command and will then wait for that child to terminate before returning to input (i.e. to read in the next command). You can in fact tell the shell not to wait for the child to terminate. To do this, simply putting an ampersand ("`&`") at the end of the command line (as shown below).

```
1  $ ps -ef &
```

(**Note:** By ending the command line with "`&`", you should expect to see the shell prompt is displayed before the `ps` output — because `bash` does not wait for `ps` to complete its execution.)

Stopping Programs You can attempt to stop the running programs by doing the following:

- Press `Ctrl-Z` to make a process stop *temporarily*
- Press `Ctrl-C` to terminate a process *permanently*

Backgrounded and Foregrounded Processes If you stopped a program with `Ctrl-Z`, you can make it run in the background by using the `bg` command. You can then check this by running the `top` command, which shows the processes in the system that are currently taking up the most system resources.

(**Note:** You can also move the backgrounded processes back to the foreground with the `fg` command.)

How do we find out the processes running at the background? By running the `jobs` command will display a list of all the backgrounded jobs that the current `bash` session is managing. Note that `jobs` is not a program but is built into the `bash`.

Terminating Processes As we may have seen, processes can react to certain keystrokes (such as `Ctrl-C` or `Ctrl-Z`). However, sometimes we might want to stop or terminate a process without foregrounding it or pressing `Ctrl-C`. The solution is to use the `kill` command.

To terminate a process, we first have to find out the process ID (`PID`) or the job ID of the process you want to kill. This can be done by using the `ps` or `jobs` command. Then, use the `kill` command with either the process ID or the job ID.

```
1 $ kill -level PID
2 $ kill -level %jobID
```

The `level` option indicates the signal you want to send to the target process that you are attempting to kill. Below is the summary of the standard Unix signals.

Signal	Name
1	`SIGHUP` (hang up)
2	`SIGINT` (interrupt, Ctrl-C)
3	`SIGQUIT` (quit)
4	`SIGILL` (illegal instruction)
5	`SIGTRAP` (trace trap, used in `gdb`)
8	`SIGFPE` (floating point exception, e.g. divide by zero)
9	`SIGKILL` (signal cannot be caught or ignored)
11	`SIGSEGV` (segmentation fault)
18	`SIGCONT` (continue if stopped, e.g. using `fg`)
20	`SIGSTP` (terminal stop, Ctrl-Z)

Note that the OS kernel sometimes sends signals to processes. This is indeed how various different kinds of program crashes are implemented — such as segmentation faults and divide-by-zero errors. In fact, pressing `Ctrl-C` sends an interrupt signal to the target process; while pressing `Ctrl-Z` sends a terminal stop signal.

3 Tasks

3.1 Task 1

Look up the `man` pages for the following Unix commands:

```
ps
top
jobs
uptime
pgrep
pkill
```

3.2 Task 2 (2 marks)

When a new program is executed, a new process is created with the next available process ID. However, there are a few special processes that always have the same process ID, which are usually given the ID value less than 5 — these are called *system processes*. Can you identify which of the two system processes have the process ID of 0 and 1 respectively?

3.3 Task 3 (2 marks)

Trace the following C code carefully **without typing or running it**. What are the possible outputs for this program?

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    pid_t pid;

    printf("X\n");
    pid = fork();
    printf("Y\n");

    if (pid > 0) {
        wait(NULL);
    }
    printf("Z\n");
    return 0;
}
```

3.4 Task 4 (2 marks)

Trace the following C code carefully **without typing or running it**. Find out how many times the program prints "FIT2100". (You may want to draw a simple tree-like diagram to understand the parent-child hierarchy of the processes spawned.)

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

int main() {
    int i;

    printf("FIT2100\n");

    for (i=0; i < 2; i++) {
        fork();
    }

    printf("FIT2100\n");
    return 0;
}
```

3.5 Task 5 (4 marks)

Write a C program that creates (or spawns) a child process to execute the Unix's command "`echo`" with a string as the command-line argument; while the program itself (the parent process) will execute another Unix command "`less`" with a filename as the command-line argument.

If you aren't sure how the `echo` and `less` commands are meant to behave, experiment with them by running them directly from the command line first.