CS 270 Systems Programming	Spring 2021

Project 2: Playing With Processes

***Update**: Mon 8 March 2021* — ***Due**: Friday 19 March 2021*

This is an **individual project**. This time, you will be writing C code, using the information contained in Chapter 8 of your textbook *Computer Systems: A Programmer's Perspective*, 3rd edition. You must do the assignment on your CS Department VM.

The objective of this assignment is to get you familiar with the programming interfaces for creating, terminating, and controlling processes, specifically `fork`, `wait`, `kill`, `signal`, and their various ancillary functions and macros (e.g., `sigaction` and `sigsuspend`).

You will turn in a `tar` file containing these files:

`./forker.c`	Solution for Part 1
`./sigs.c`	Solution for Part 2
`./ipc.c`	Solution for Part 3
`./myScript.txt`	See below

The last file should be a script produced by running your code and recording the outputs. Note that (i) your tarball should **NOT** include a directory, and (ii) your tarball **MUST NOT** include a `Makefile`.

To get started, download the tarball from the Canvas page for this assignment onto your VM, and unpack it into a directory where you will work on this project. The tarball includes the following:

- A file `dieWithError.c` containing a function that prints an error message and then calls exit; this is a simple way to deal with errors from system and library calls. You can modify this if you want or replace it with something else (e.g., the similar function from the textbook) if you like.
- A `Makefile` to build the three executable files `forker`, `sigs`, and `ipc` from the files that you will eventually turn in.
- Skeleton `forker.c`, `sigs.c`, and `ipc.c` files, which have a very basic outline of the code you will fill in. These have the `#include` statements that are sufficient to create a working solution, and in some cases a declaration or two. (If you need to add other `#include`s, you may do so.)

Your code ***must*** compile correctly with the given `Makefile`. You will receive zero points for any of the three programs that fails to compile. On each part, 3/4 of the points are for correctness, and 1/4 for efficiency, style and readability.

> **Note Well:** This assignment requires dealing with a significant amount of detail. **Remember:** `man` is your friend. It will tell you the syntax to use and what header (`.h`) files you will need to `include` in your program. You are also welcome to use the "wrapper" functions provided at the CS:APP website (see pointer on the canvas page). (To use them, simply copy them into your source files.)

1 `fork`ing and `wait`ing

For this part, which is worth 12 points, you will write a simple program (`forker.c`) that forks multiple times, until there are 4 processes in total. We'll call the original process A; the others are B, C, and D. The relationships among them are shown in the process graph in Figure 1.

Each process announces itself by calling the function `greet`, shown below, as soon as it starts running:

```
void greet(char myName) {
  printf("Process %c, PID %d greets you.\n",myName,getpid());
  fflush(stdout);
}
```

As you can see, `greet` outputs the process letter and process ID. Once it has called `greet`, each *child* process prints "Process X exits normally", where X is its letter name, and then calls `exit`. Each *parent* process, after forking, calls `wait` (or `waitpid`) to reap a child. When `wait` returns, it prints a message of the form:

Process X here: Process Y exited with status ##

where X and Y are parent and child process's names (i.e., letters, not PIDs) respectively, and "##" is the exit status.

Once a process has reaped all of its children, it prints the same "Process X exits normally" message and then calls `exit`.

Each vertical line in Figure 1 represents the logical flow of one process; horizontal lines represent forking or termination. You *must* put all the code in one file called `forking.c`. Suggestions:

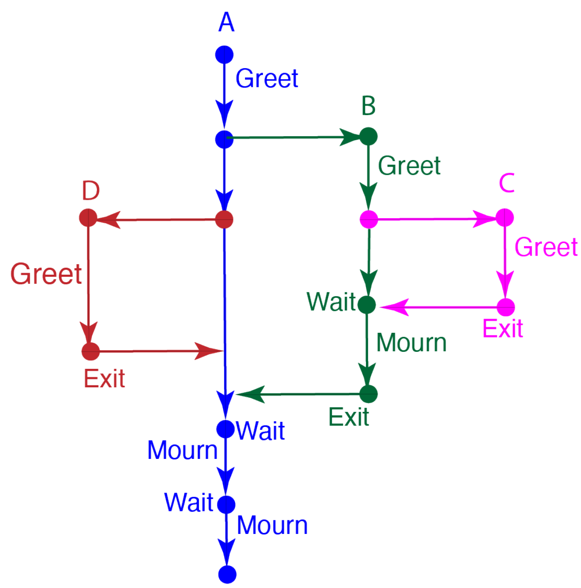

Figure 1: Process graph for Parts 1 and 3

- All the processes will eventually be printing the same thing, so write another function called `goaway`, which prints the "exits normally" message and then calls `exit`. (You can do the same thing for the message the parent prints showing the exit status, which is labeled "Mourn" in the process graph.)
- You can use `wait` or `waitpid` to reap terminated processes. Although `wait` (or `waitpid`) returns a process ID (type `pid_t`), the parent must print the letter name of the process. So write a function that keeps track of child process IDs and prints the correct name based on the PID.
- Like the examples in the slides and the textbook, include comments in your code that label branches for the results of the `fork` calls. Use the process name, i.e., the branch of the first call that returns 0 would be labeled "B". This will help you keep track of what's what.

As an example, here is one possible output sequence from the program:

```
Process A, PID 6553 greets you
Process B, PID 6554 greets you
Process C, PID 6555 greets you
Process D, PID 6556 greets you
Process C exits normally
Process D exits normally
Process A here: Process D exited with status 0x4300
Process B here: Process C exited with status 0x44
Process B exits normally
Process A here: Process B exited with status 0x4200
Process A exits normally
```

Note that if process A calls `wait` it may reap either B or D first. If it calls `waitpid`, it can specify its children to be reaped in a specific order. Here is an example of an ordering that is *not* feasible:

```
Process A, PID 7912 greets you
Process B, PID 7913 greets you
Process C, PID 7914 greets you
Process A here: Process D exited with status 0x4300
Process C exits normally
Process D, PID 7915 greets you.
Process D exits normally
Process A here: Process B exited with status 0x4200
Process B here: Process C exited with status 0x44
Process B exits normally
Process A exits normally
```

This cannot happen because D (B) must exit before A can say that it exited. (This assumes that writes to the terminal are output in the order they are issued, which you may assume is the case.)

2 Handling Signals

The part is worth 12 points. It is intended to expose you to some of the subtleties of using signals, including why counting events with signals doesn't work.

The signal SIGALARM is used to arrange *timeouts*—that is, a process can arrange to be sent a signal when a certain amount of time has passed. For this assignment we will be using the `alarm(n)` library function, which causes SIGALARM to be delivered to the calling process after n seconds. (Note: there are other, more flexible ways of causing SIGALARM to be delivered; see `setitimer()` for example. But `alarm` is sufficient for this assignment.) The default action for SIGALARM is termination, so it is usually used with a signal handler.

For this part you will write two signal handlers, one for SIGALARM and one for SIGINT. (The skeleton file `sigs.c` has placeholders for them; it also defines the constants `LIMIT` and `PERIOD` and declares the global variables mentioned below.)

The SIGALARM handler does the following: (i) block all signals; (ii) increment the global counter `int sigalarm_count`; (iii) call `alarm(PERIOD)` to reset the alarm; (iv) restore the original set of blocked signals.

The SIGINT handler (i) blocks all signals; (ii) increments the global counter `int sigint_count`; (iii) restores the original set of blocked signals.

The first thing the `main` function of sigs.c does is fork. From then on, the behavior of the two processes is as follows:

- The child process calls `getppid` to get the process ID of its parent, then reads one character from the standard input (use `getchar()` for this). This will block until you type a character. After `getchar()` returns (the return value may be ignored), the child process enters a loop in which it sends SIGINT to the parent process `LIMIT` times. Finally it prints the message "Child: finished sending SIGINT LIMIT times." and calls `exit`.

- The parent sets up the two signal handlers, one for SIGALARM and another for SIGINT. It then calls `alarm(PERIOD)`, which will cause a SIGALARM to be delivered after 5 seconds. Then it enters a `while` loop that will iterate until either `sigint_count` or `sigalarm_count` exceeds the defined `LIMIT`. The test of the loop *must* take place with SIGALARM and SIGINT blocked. The body of the loop prints a single dot ('.') to standard output, and then calls `sigsuspend` to unblock all signals and wait for something to happen. (So the parent just waits until either SIGALARM or SIGINT has been delivered more than `LIMIT` times.) After the loop ends, it prints out the value of both counters and calls `exit`.

The reason the child process reads from the terminal is so that you can control when it starts sending SIGINT; the parent must have time to install the signal handlers. To be safe, you can hit "enter" on the keyboard after the parent prints the first dot.

Your code should use `sigaction` to set the signal handlers.

3 Synchronizing with Signals

This part is worth 20 points. In it, you will combine everything you have learned from the first two parts to impose some additional ordering on the events shown in Figure 1. (You will not be using timeouts or SIGALARM, however.)

The two signals SIGUSR1 and SIGUSR2 are not sent by the OS; they are available for application programs to use for inter-process communication. In this part, you will first copy `forker.c` to `ipc.c` to use as a starting point. The observable behavior of `ipc.c` will be the same as that of `forker.c`, but you will modify the code to use SIGUSR1 and SIGUSR2 to co-ordinate the actions of processes A, B, C, and D. The process graph will look exactly the same as before, but you must add signal handlers and calls to the `kill` function to impose the following *additional* ordering constraints on the events:

- C must call `greet` before D calls `greet`.

- D must call `greet` before C exits.

- D must exit *after* C exits.

To make this work, C needs to inform D when it has called `greet`, and D needs to wait for that signal before it calls `greet`. Similarly, C needs to wait to hear that D has called `greet` before C exits, and then D needs to hear that C has exited before it exits. The processes will convey this information using the two signals. (**Note:** As noted in class, signals don't carry any information other than their number and the fact that they were sent. But that is all you need to implement the constraints.)

Your processes may only send signals to other processes whose IDs they learn either from a call to `fork` or `getppid` (i.e., to their child or parent). In particular, they may not use anything like `ps` to learn other process IDs.

Hints:

- Because signals can only be sent between parents and children, the parent processes (i.e., A and B) will need to relay the signals back and forth between C and D. The suggestion is to use one of the signals (say, SIGUSR1) for signals traveling toward D (right to left in Figure 1), and the other for signals traveling in the other direction.

- To keep your handlers simple, have all the calls to `kill` made from the main program. You can use simple (global) flag variables (of type `sig_atomic_t`—see L17) to let the handler indicate to the main program that the signal has arrived. The handler can set the value to 1 to indicate signal arrival; after reading it, the main program resets it to 0.

- As in the previous part of this assignment, you will need to pause the programs at certain points to wait for signals to arrive. However, because the communication is so simple, you can just use `pause()` rather than `sigsuspend()`.

- The additional constraints described above amount to adding additional "happens-before" arrows between events in C and D in Figure 1. Drawing those arrows on the figure may help you keep track of what each process needs to do in terms of waiting for and relaying signals.