CSSE 332 Exam 1: September 28, 2021

Introduction

In this exam, we will extend our simple shell to support more useful and tricky commands. I have provided you with the base code that runs the commands. All of your coding will be in the file `functions.c` where you will implement each and every command.

Before you get started, please make sure you install all the dependencies for this exam by running the following command from your virtual machine terminal:

```
sudo apt install -y libreadline-dev
```

Note: In this exam, you are not allowed to make use of the `system` system call. All you have in your toolbox are `fork`, the `exec` family, and the `pipe` system calls.

Part 0: Makefile (15 points)

I have already provided you with a makefile that includes an `all` rule that builds everything and a `clean` rule that removes all binaries and `.o` files.

In this part, you will need to complete the `Makefile` to generate all the required binaries, which include `shell`, `ping`, and `hibernate`. Your `Makefile` must satisfy the following requirements:

- Compile and generate the binaries correctly
- All of the `.c` files should be compiled individually to generate object files using the `-c` flag in `gcc`.
- You will need to correctly link the object files (not the source files) to produce the corresponding executables.
- You should correctly specify dependencies, such that no unneeded files are ever rebuilt.
- Your `Makefile` should trigger the correct operations when a `.h` file is changed.

Note: Linking the `shell` executable requires that you pass the `-lreadline` flag to `gcc`. So make sure you do so in your final recipe.

> If you fail to complete this part, you may use the following command to build the exam
>
> ```
> gcc shell.c functions.c -o shell -lreadline
> gcc hibernate.c -o hibernate
> gcc ping.c -o ping
> ```

Interlude: The status command

To help you succeed in this project, I have provided you with a `status` command that is built-in to your exam shell. This command will return the list of processes that are currently running within your terminal window. Use this command freely to debug your implementation and double check that you have no zombie or orphan processes.

Example

Here's an example of using the `status` command:

```
 $ ./shell
CSSE332 >> status
    PID TTY          TIME CMD
  93088 pts/0    00:00:00 shell
  93089 pts/0    00:00:00 sh
  93090 pts/0    00:00:00 ps
CSSE332 >>
```

and here's another example showing some zombie processes

```
CSSE332 >> status
  PID TTY          TIME CMD
93088 pts/0    00:00:00 shell
93091 pts/0    00:00:00 shell <defunct>
93093 pts/0    00:00:00 shell <defunct>
93095 pts/0    00:00:00 shell <defunct>
93097 pts/0    00:00:00 sh
93098 pts/0    00:00:00 ps
```

Part 1: Hibernate (10 points)

In this part, you will implement the `hibernate` command in the `functions.c` file. The `hibernate` process, implemented in `hibernate.c` will only sleep for 5 seconds and then exit, nothing too sophisticated here.

Your implementation must satisfy the following conditions:

- Your command must launch the `hibernate` binary.
- Your command must run in the foreground. In other words, the shell should be occupied and not return until the hibernation is complete.
- Your implementation should produce no zombie or orphan processes.
- Your implementation should return to the shell prompt after hibernation is finished.

Here is a sample output of running the hibernate command

```
$ ./shell
CSSE332 >> hibernate
Hello from process hibernate...
CSSE332 >>
```

Part 2: Sleep (10 points)

In this part, you will implement the `sleep` command, implemented as `csse332_sleep` in the `functions.c` file. `csse332_sleep` is very similar to hibernate, except that it runs in the background.

Your implementation must satisfy the following conditions:

- Your command must launch the `hibernate` binary (no this is not a typo).
- Your command must run in the background, i.e., the shell prompt should show back up and you should be able to input new commands regardless of the status of the `sleep` command.
- It is okay if your implementation generates zombie processes, but no orphan processes.
- Your implementation must print the phrase `Sleep process done in the background...` every time the sleep process has completed execution.

Example output

Running a single instance

```
CSSE332 >> sleep
CSSE332 >> Hello from process hibernate... <-- shell returned here, sleep is in the backgrou
Sleep process done in the background...
```

Running multiple instances

```
 $ ./shell
CSSE332 >> sleep
CSSE332 >> Hello from process hibernate...
sleep
CSSE332 >> Hello from process hibernate...
sleep
CSSE332 >> Hello from process hibernate...
sleep
CSSE332 >> Hello from process hibernate...
Sleep process done in the background...
Sleep process done in the background...

CSSE332 >> Sleep process done in the background...
Sleep process done in the background...
```

```
CSSE332 >>
```

Part 3: Cleanup (20 points)

In this part, you will implement the `cleanup` command in the `functions.c` file. `cleanup` is a command that tracks zombie processes and removes them from the system (the details of figuring that out are up to you).

Your implementation must satisfy the following conditions:

- Your command must launch in the **foreground**.
- Your command must find and neutralize all zombie processes.
- If no zombie processes exist, your command should not crash, it should simply do nothing.
- Consecutive calls to cleanup should not crash.

NOTE: The `exit` command is configured to always call the `cleanup` command before exiting the shell. Your implementation must not break the `exit` command.

Example output

Here's an example output of running the cleanup command after `sleep` generated three zombie processes.

```
CSSE332 >> status
    PID TTY          TIME CMD
  93178 pts/0    00:00:00 shell
  93179 pts/0    00:00:00 shell <defunct>
  93181 pts/0    00:00:00 shell <defunct>
  93183 pts/0    00:00:00 shell <defunct>
  93193 pts/0    00:00:00 sh
  93194 pts/0    00:00:00 ps
CSSE332 >> cleanup
CSSE332 >> status
    PID TTY          TIME CMD
  93178 pts/0    00:00:00 shell
  93195 pts/0    00:00:00 sh
  93196 pts/0    00:00:00 ps
CSSE332 >> exit
```

Part 4: Pong (25 points)

In this part, you will implement the game `pong` between two processes, (1) the shell process (i.e., the main process) and (2) another process running the `ping`

executable. Your implementation must satisfy the following conditions:

- Your code should send the character `a` back and forth between the two processes using pipes.
- In your implementation, the shell process begins by **receiving** the character from the `ping` process, and then it sends it back and forth.
- Conversely, the `ping` process starts by sending the `a` character and then awaits for the shell to send it back. And so on goes the game
- Your implementation must send the `a` character **five** times between the two processes.
- Your implementation must not generate any zombie or orphan processes.
- Your implementation must print the phrase `<pid>: Read a from the child` every time the parent receives an `a` character from the child, where `<pid>` is the process ID of the parent process.

The ping executable

Take a look at `ping.c` to understand how the ping process operates. The executable accepts two arguments from the command line: the first represents a read descriptor (an integer) and the second represents a write descriptor (another integer). The program will start by sending the `a` character on the writing end and then listening for the `a` character on the reading end.

Example output

Here's a sample output of running the `pong` command in the shell

```
CSSE332 >> pong
93178: Read a from child
93308: Read a from parent
93178: Read a from child
93308: Read a from parent
93178: Read a from child
93308: Read a from parent
93178: Read a from child
93308: Read a from parent
93178: Read a from child
93308: Read a from parent
CSSE332 >>
```

Part 5: Fpoint (20 points)

In this part, you will implement the command `fpoint` that launches a floating point computation **in the foreground**. The `fpoint` function will run a stupid

floating point computation an infinite number of times. Your job is to force the `fpoint` process to exit when either of the following two conditions are met:

1. When the `fpoint` program generates a floating point exception, or
2. 5 seconds after the `fpoint` program has started, whichever comes sooner. The floating point computation is contained in the `do_fp_work`.

In other words, although the `fpoint` function runs forever, your code must ensure that the program runs for at most 5 seconds or until a floating point exception happens, whichever is sooner.

Your implementation must satisfy the following conditions:

- The `fpoint` command must run the `do_fp_work` function in the foreground.
- There are two ways in which the `fpoint` process can exit, either with a timeout after 5 seconds or with a floating point error. Your job is to detect these two conditions and then make sure that you exit with the corresponding status code.
- The shell must capture how the `fpoint` program has exited (either after 5 seconds or through a floating point exception).
- The shell must print `Exit success` if the `fpoint` program terminates after 5 second with no floating point exceptions.
- The shell must print `Exit failure` if the `fpoint` program terminates with a floating point exception.

Example output

Here's an example output of running the `fpoint` command where the outcome was a floating point exception:

```
CSSE332 >> fpoint
PANIC: FLOATING POINT EXCEPTION
Exit failure
```

You don't have to print the `PANIC` line, I only added it here for clarification.

In case of a successful run, here's the expected output

```
CSSE332 >> fpoint
5 seconds have passed, exit with success
Exit success
```

Again, you don't have to print the `5 seconds...` line, I only added that one for clarification.

Bonus: Prime sieve (30 points)

In this bonus problem, you are to implement the primes sieve, an idea due to Doug McIlroy, the inventor of Unix pipes.

Your solution should run in the **foreground** and should use `pipe` and `fork` to set up a pipeline and print the prime numbers that are less than 35. The first process (i.e., the shell in this case) generates the numbers 2 through 35 and feeds the numbers that are not multiples of 2 into the pipeline. The second process eliminates the multiples of 3, the third process eliminates the multiples of 5, etc. For each prime number, you will create a process that reads from its left neighbor and writes to its right neighbors, as shown in the figure below

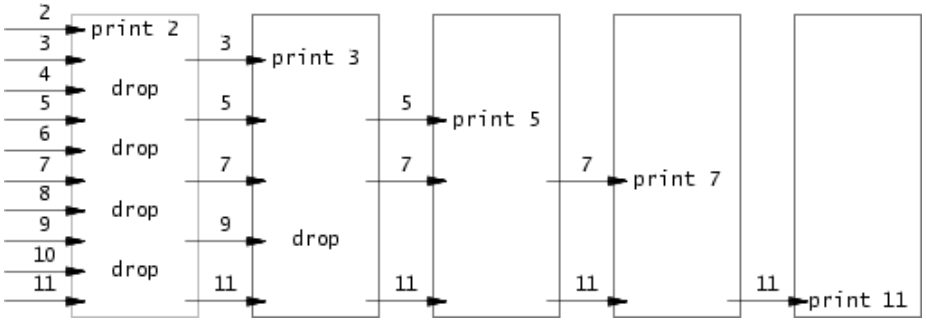

Figure 1: sieve

In other words, every process will print exactly one prime number, call it `p`, which it will read from its left neighbor, and will then feed its right neighbor all the numbers greater than `p` that are not multiples of `p`.

Your implementation should satisfy the following requirements:

- The `primes` command must run in the foreground.
- Once the shell reaches the number 35, it should **wait** until the entire pipeline terminates, including all children and grandchildren, etc.
- Processes should only be created on a per-need basis, i.e., you should not preallocate all processes and then run the sieve.
- The command should only return when all the pipeline has been destroyed, i.e., there shouldn't be any zombie or orphan processes.
- Each prime number must be printed by a separate process.

Pseudocode

Here's a sample pseudocode that illustrates what each process in the pipeline must do

```
p = get a number from left neighbor
print p
loop:
    n = get a number from left neighbor
    if (p does not divide n)
        send n to right neighbor
```

Example output

```
$ ./shell
CSSE332 >> primes
prime 2
prime 3
prime 5
prime 7
prime 11
prime 13
prime 17
prime 19
prime 23
prime 29
prime 31
CSSE332 >>
```