15-213/18-243, Fall 2010
Lab Assignment L5: ProcLab
Assigned: Tuesday Oct. 12

Due: Tuesday, Oct. 19, 11:59PM

Last Possible Time to Turn in: Friday, Oct. 22, 11:59PM

Logistics

This is an individual project. All handins are electronic.

Please contact the 15-213 staff list (`15-213-staff@cs.cmu.edu`) for any questions.

Overview

This lab tests your understanding of process control and signal handling. You will be asked to use most of the system calls in the following list to complete this lab. Depending on your implementation, some of them might not be used, but nonetheless, you should learn how to use all of them, as they might appear in exam questions. Examining the man pages is a good way to start.

- Creating and running processes: `fork`, `exec` variants, `exit`
- Synchronizing with children: `wait`, `waitpid`
- Macros for inspecting exit status: WIFEXITED, WEXITSTATUS, WIFSTOPPED, etc
- Signal handling: `signal`, `sigaction`, `sigsuspend`, `sigprocmask`

Downloading the assignment

Your lab materials are contained in a Unix tar file called `proclab-handout.tar.gz`, which you can download from Autolab. After logging in to Autolab at

```
http://autolab.cs.cmu.edu
```

you can retrieve the `proclab-handout.tar.gz` file by selecting "Proclab - Download Lab Materials" and then hitting the "Save File" button.

Start by copying `proclab-handout.tar.gz` to a protected directory in which you plan to do your work. Then run the command "`tar zxvf proclab-handout.tar.gz`". This will create a directory called `proclab-handout` that contains the following files (some of them are generated by `make`):

- `puzzles.c`: File in which you should write your code.
- `driver.a`: Archive of driver object files that will be linked with `puzzles.o` to produce the driver binary
- `Makefile`: Used by `make` to generate the driver binary.
- `driver`: A program you can use to evaluate your work. Type `make` to build this binary.
- `decipher`: A helper program for the `decipher` puzzle.
- `README`: A help file that describes the contents of the handout tarball.

To compile your code and build the driver:

```
linux> make clean
linux> make
```

To run the driver program to evaluate your work:

```
linux> ./driver
```

The driver has an optional commandline argument that can run specific tests. For example, if you specify

```
linux> ./driver -t 03456
```

then the driver will run tests only for puzzles 0, 3, 4, 5 and 6. Note that there must **not** be any spaces between the numbers. If you omit the -t option, then the driver will run all of the tests by default.

The driver can submit an unofficial score to the class scoreboard if and only if you run all the tests. The latest score sent to the scoreboard will overwrite any previous ones. Submission of unofficial scores will be done automatically by the driver and there is no option to disable this behavior.

IMPORTANT: To submit your work to get an **official** score, please upload `puzzles.c` to Autolab. As you know, the driver program can only submit an **unofficial** score to the class scoreboard, and an unofficial score will **not** be counted towards your grade.

WARNING: Do not let the Windows WinZip program open up your `.tar.gz` file (many Web browsers are set to do this automatically). Instead, save the file to your AFS directory and use the Linux `tar` program to extract the files. In general, for this class you should NEVER use any platform other than Linux to modify your files, doing so can cause loss of data (and important work!).

Puzzle 0: timer

Congratulations! You're one of the promising candidates whom the 213 staff think could stop Dr. Evil from releasing his newly developed virus that is so potent that it could bring down the entire Internet. Before the staff can assign you this honorable task, they have decided to give you one last challenge to help you prove your mettle. If you can impress them, they will let you move on to your first mission.

Your Task

Your job for this puzzle is to complete the function `tgets` in `puzzles.c`. The `tgets` function is a version of the C `fgets` function that times out if the user doesn't enter an input line quickly enough.

```
ssize_t tgets(char* buf, int buf_size, int timeout_secs)
```

This function reads in at most one less than `buf_size` characters from stdin and stores them into the buffer pointed to by `buf`. Reading stops after an EOF or a newline (`\n`). If a newline is read, it is stored into the buffer. A NULL character (`\0`) is stored after the last character in the buffer. If the user fails to type an input line within `timeout_secs` seconds, then `tgets` times out, returning to the caller with a value of 0 to indicate that the timeout occurred. Otherwise it returns the length of the input string, excluding the terminating NULL character.

To solve this puzzle you will need to modify the `tgets` function and write your own SIGALRM handler. To help you, we have provided you with two functions in `puzzles.c` called `mygets` and `set_alarm_handler`.

The `mygets` function behaves like `fgets`, but if it receives any signal while running, it will return -1 and set `errno` to EINTR. In this case, you must make sure that the received signal is really SIGALRM and not any other signal, because it is the SIGALRM signal that indicates that the timeout has occurred.

In addition to modifying the `tgets` function, you will need to write a SIGALRM handler. The `set_alarm_handler` function installs your SIGALRM signal handler and returns the address (a function pointer) of the previous handler, which you can use later to restore the original handler.

Notes

- You should write exactly two functions: the SIGALRM handler and the `tgets` function.
- You must use `mygets` to read a line from stdin.
- Before `tgets` returns, you must restore the original alarm signal handler that you have overridden.
- In Linux, slow system calls are automatically restarted after they are interrupted by signals. However, we do not want this behavior to happen for `tgets`, otherwise the timeout feature will not work. Fortunately, the staff has written `set_alarm_handler` such that system calls will **not** be automatically restarted when SIGALRM is received. The details can be found in `set_alarm_handler`'s code.
- You must use the SIGALRM signal to implement the timeout feature. You can use a global variable to help you determine whether the signal that woke `mygets` up is really SIGALRM.

- The exact specification of `tgets` can be found in `puzzles.c`. You may assume that the caller will **not** pass in invalid parameters to `tgets`.

Puzzle 1: racer

We have been intercepting a stream of characters from two different processes that Dr. Evil has sent, but we aren't sure what they mean. Your job is to synchronize the processes such that both strings can be extracted. We know that one string is printed (one character at a time) from the parent process, and each character in the other string is printed when a child exits. However, only one of these gets printed, based on which process acted first.

In other words, each time Dr. Evil's program calls `fork`, if the parent runs first, then a character from string 2 will be printed. But if the child exits first, then a character from string 1 will be printed instead. If you are still confused, try running Puzzle 1 a few times and look at the output.

In both races, the "child" letter is actually printed from the sigchld_handler in the parent, so it only gets printed if the parent receives a SIGCHLD before running.

Note: The driver will not detect anything that your code prints to stdout. Any call to `printf` or some variant you make while testing should be deleted before submission.

Race 1

For the first race, you must complete the function `racer1` (which is invoked in the parent just after the call to `fork`) such that the child always runs before the parent prints a letter. This prints all of string 1 and none of string 2.

More explicitly, the code for race 1 looks something like this:

```
for(i=0; i<STRLEN; i++){
  pid = fork();
  if (pid == 0)
    exit(0);
  else racer1(pid);
}
```

Race 2

For the second race, you are given two template functions, `setup_race2` (which is called before `fork`), and `racer2` (which is called after `fork`, in the same place as `racer1`). You should edit these to block the SIGCHLD from being received until after the parent's letter has been printed. This prints all of string 2 and none of string 1.

More explicitly, the code for race 2 looks something like this:

```
for(i=0; i<STRLEN; i++){
  setup_race2();
  pid = fork();
  if (pid == 0)
```

```
      exit(0);
    else racer2(pid);
  }
```

Notes

- No global variables allowed.

Puzzle 2: decipher

Now that you have gained access to Dr. Evil's system, you decided to peek at the contents of some of his files. None of his files are encrypted except for one particular file that contains several lines of encrypted words. Fortunately, your hacker's toolkit comes with a handy helper `decipher` program that can break any encryption.

Your Task

Your job here is to complete the `decipher` function in `puzzles.c`. The `decipher` function takes in an array of pointers, where each pointer refers to a string that is an encrypted word. The last element in the array is a `NULL` pointer that marks the end of the array. The goal is to get the `decipher` program (don't confuse it with the function) to decrypt the array of words passed to the `decipher` function. The `decipher` program takes an encrypted word as a command line argument and prints out the decrypted word on stdout. For example:

```
linux> ./decipher ruyTbPN
```

You should be able to see the decrypted word. Of course, if you give the program a non-encrypted word, it will output garbage. Due to some technical constraints, the `decipher` program can only accept one encrypted word at a time, so you have to run it **once** for each encrypted word. Your job is to perform this decrypting process by using the `fork` and `exec` functions to load and run the `decipher` program, once for each encrypted word in the input array.

You will also need to ensure that the decrypted words are printed out in the **same order** as the encrypted words are stored in the array.

Important: You do not have to extract the output printed out by the `decipher` program. The driver will have a way to capture this output. All you have to do is to run the `decipher` program several times to decrypt the array of encrypted words.

Notes

- You can use any variant of the `exec` function, such as `execle`.
- You must make use of `fork` and either `wait` or `waitpid` in your solution.
- Write all of your code in the `decipher` function.
- No global variables allowed.

Puzzle 3: counter

After rummaging through his files, you couldn't find the virus that you're tasked to destroy. Instead, you came across an image file that illustrates the hierarchy of processes running in Dr. Evil's computer system. It looks so interesting that you suddenly forget all about your mission and decide to spend some time investigating this tree of processes.

Here is Dr. Evil's digital organization structure, a.k.a, Tree of Evil:

```
Level 0:            Dr. Root of Evil
                     /       |      \
                    /        |       \
                   /         |        \
Level 1:        Minion-1     Minion-2  \...other minions...
                 /   \          \___
                /     \            \
Level 2:  Assassin-1  Assassin-2     \...other assassins...
              /           |     \
Level ...    ...         ...      ...
             |            |        |  \__
             |            |        |      \
Level n:  Worker-1    Worker-2  Worker-3    \...other workers...
```

As you can see, Dr. Evil has even named a process after himself. That must be some command and control program that he uses to carry out his evil plans. Some other properties you observe:

- At level 0, "Dr. Root of Evil" is the root of this whole hierarchy and it has a number of different child processes running under it.
- The lowest level n consists of worker processes.
- Every Minion can have a different number of Assassins or none at all. Likewise, every Assassin can have a different number of subordinate processes or none at all.

Hmmm, you wonder how many processes there are in total in this digital organization. Since the organization is so huge, you decide to use a computer to count it for you, rather than to count it manually.

Your Task

The goal here is to fill in the `counter` function in `puzzles.c` so that it counts the total number of children a process has in its subtree (including the process itself). After a parent process has spawned **all** of its child processes, it will call `counter` with a `num_direct_children` parameter that specifies the number of *direct children* the process has. Processes that don't spawn any children will call `counter(0)`. When it has completed, your `counter` function should exit the process by calling `exit(n)`, where `n` is the total number of children the current process has in its subtree, including itself. Processes with no children should should call `exit(1)`.

For example, suppose that the Dr. Evil root process spawns 5 Minion processes. After spawning these processes, the Dr. Evil process will call `counter(5)`. Eventually, the Dr. Evil root process should exit with its `exit` code set to the total number of processes in the entire tree.

Notes

- You should write **all** of your code in the `counter` function.
- The total number of processes in the entire tree is always less than 255.
- No global variables are allowed.

Puzzle 4: reaper

We've made much progress tracking down Dr. Evil's mainframe, and have narrowed it down to a few locations. We sent some spies (aka 213-ninjas) to each of these locations, but Dr. Evil has such well-laid traps that very few of them come back. This time, each spy has a process attached to them that reports back to us how it died. This will allow us to discover what kind of traps Dr. Evil is setting so we can finally catch him. Your job is to write a handler that will catch the signals from each spy and print to stdout how it died.

Your Task

Your task here is to fill in the `reaper` function in `puzzles.c`. A signal handler in the parent process (in this case our driver program) calls your `reaper` function each time the parent receives a SIGCHLD signal. There are three different cases when this could happen:

1. The child called `exit`.
2. Receipt of a signal caused the child to terminate.
3. Receipt of a signal caused the child to stop (temporarily halt execution).

Each time your `reaper` function is invoked, it should use `waitpid` to reap all available zombie children, extract their termination or stopped status, and print the result to stdout. You will need to make use of the `options` argument to `waitpid`. Since this is in a signal handler, you should use `safe_printf` instead of `printf` (Discussed in Lecture 14 on signals and available online from the course schedule's web page).

In particular, if a child exited normally, you should call:

```
safe_printf("Job (%d) exited with status %d\n", pid, status);
```

If the child terminated as a result of receiving a signal, you should call

```
safe_printf("Job (%d) terminated by signal %d\n", pid, sig);
```

If the child stopped as a result of receiving a signal, you should call

```
safe_printf("Job (%d) stopped by signal %d\n", pid, sig);
```

In these cases, `pid` is the pid of the child that was reaped, `status` is its exit status, and `sig` is the number of the signal that killed/stopped it.

Note: If you print something that does not match the above format, the driver will not give you credit.

Puzzle 5: shower

So the last spy who survived has somehow managed to capture Dr. Evil's mainframe and destroy the virus, but Dr. Evil is still on the loose. A few days later, the 213 staff intercepted some message signals and after much investigation they found out that he was plotting some new nefarious deed. When he realized that his signals were being intercepted, Dr. Evil decided to send them constantly in a seemingly random order, like a *signal shower*, so that it would be much more difficult to figure out the message hidden amongst those signals. As your final task, you are required to intercept those signals and make sense out of them.

Your Task

Your process will be bombarded with numerous instances of the following 5 signals: SIGALRM, SIGUSR1, SIGUSR2, SIGCONT, and SIGCHLD. They will be sent randomly, and each time a signal is received, a unique word associated with the signal will be printed out to the screen. The goal is to get the words printed out in the **correct** order so that you can see the message.

There are two functions in `puzzles.c` that you will need to complete: `shower_setup` and `shower_run`.

In `shower_setup`, you should install a signal handler to catch all 5 of the signals. As soon as `shower_setup` returns, the signals will start to be received, so you will want to block **all** signals before returning. Each time your signal handler catches a signal, your handler should call the `signal_received` function with the number of the signal that it received. Calling `signal_received(k)` will cause the word associated with signal k to be printed to stdout.

In `shower_run`, you should block the appropriate signals in such a way that the 5-word message from Dr. Evil is printed to stdout. The idea is to block all signals except for the signal associated with the first word of the message, wait for your handler to catch it, then block all signals except for the signal associated with the second word of the message, wait for your handler to catch that signal, and so on.

Notes

- **Do not simply call `signal_received` five times to get the message printed out.** You can only call it in your signal handlers, not in `shower_setup` nor `shower_run` function. Also, you cannot call your signal handler directly. You have to let your signal handlers be triggered by the actual receival of signals.
- You are allowed to write other helper functions.
- Global variables are allowed.
- You might need to call `sigprocmask` multiple times to get the correct message printed out.
- You will need to figure out the word associated with each signal.

Style Requirements

- Make sure you check for **all** possible system calls failure. You do not have to check for `alarm` failing since it does not report any error condition. So you can assume `alarm` will always succeed. As a reminder, do not assume `exec` will always succeed.
- When an error is detected, print an error message and handle the error appropriately. You can choose to exit or simply return. **Do not silently ignore errors**. Exiting will cause the driver to terminate and returning will cause the current test to fail. But since errors are rare (unless you are using the system functions incorrectly), you do not have to worry about the autograding process being affected. If you have an error in your official submission, just try submitting again.
- In `tgets`, however, you must return -1 when an error occurs.
- We are not providing `csapp.h,c`, so please do not call `Fork, Exec` and etc. (those that start with a capital letter)
- In puzzle shower, if you find that your code is very repetitive, please shorten your code by writing other helper functions.
- Write a brief comment in the function headers for the functions you have to complete. In general, please comment your code.
- Indent all your code consistently except for `set_alarm_handler`
- It is a good practice to put global variables(if any) and any prototypes you are declaring at the top of the `puzzles.c` file. For those using the old version of the `puzzles.c` file, please move the following prototypes to the top of the file: `void printALetter(), void safe_printf(), void signal_received()` The updated version of the `puzzles.c` file already have those prototypes at the top of the file.
- The style guideline on the class website , http://www.cs.cmu.edu/~213/codeStyle.html , applies as well.

Evaluation

This section describes how your work is evaluated. The full score for this lab is 100 points:

```
Puzzle 0 - timer:    15 Points
Puzzle 1 - racer:    10 Points
Puzzle 2 - decipher: 10 Points
Puzzle 3 - counter:  15 Points
Puzzle 4 - reaper:   20 Points
Puzzle 5 - shower:   20 Points
Style:               10 Points
```

There is no partial credit for a puzzle, that is, you either get full credit or no credit at all for a puzzle.

If one or more rules is violated for a puzzle, we reserve the right to take away the credit for that puzzle.

The style points are given based on the style requirements as described above as well as the quality of your code.