CIS 5480 Project 0
penn-shredder: Turtles in Time

"Tonight, I dine on turtle soup!"
shredder, *Teenage Mutant Ninja Turtles the Animated Series*

CIS5480 Staff

DUE: Wednesday, January 25th @ 10pm

Directions
This is an *individual* assignment. You may not work with others. Please regularly check *EdStem* throughout this course for project specification clarifications. You are required to version control your code, but please only use the GitHub repository created for you by the 5480 staff. **Do not work in public GitHub repositories! Please avoid publishing this project at any time, even post-submission, to observe course integrity policies.**

Overview
In this assignment you will implement a basic shell named *penn-shredder* that will restrict the runtime of executed processes. Your shell will read input from the user and execute it as a new process, but if the process exceeds a timeout, it will be terminated. You will complete this assignment using only a specified set of Linux system calls and standard C library functions.

1 Specification
At its core, a shell is a simple loop. Upon each iteration, the shell prompts the user for a program to run, executes the program as a separate process, and waits for the process to finish. The shell exits the loop when the user presses `Ctrl-D` at the prompt. Your shell, *penn-shredder*, will do all of that, but with a slight twist.

penn-shredder takes a command line argument that specifies a timeout for the user-entered program. If any program runs for longer than the specified timeout, *penn-shredder* will terminate the program and report to the user his snide catch-phrase, "Bwahaha ... Tonight, I dine on turtle soup!". The following is an example of input that should invoke the catch-phrase:

```
$ penn-shredder 2
penn-shredder# /bin/sleep 3
Bwahaha ... Tonight, I dine on turtle soup!
penn-shredder# /bin/sleep 1
penn-shredder#
```

Here, *penn-shredder* was executed with a timeout argument of 2 seconds, and any program that runs longer than 2 seconds will be terminated (or shredded). The `sleep` program with an argument of 3 seconds was terminated after exceeding the timeout. Conversely, `sleep 1` completes before the timeout, without upsetting Shredder and his henchmen.

We will give you the Makefile used by the autograder. In your C source code, you should use the two macros `PROMPT` and `CATCHPHRASE` defined in the Makefile instead of defining your own macros. See the skeleton code below.

1.1 `read`, `fork`, `exec`, `wait`, and repeat!

As described previously, a shell is just a loop performing the same procedure over and over again. Essentially, that procedure can be completed with these system calls:

- `write(2)` : write the prompt to `stderr`
- `read(2)` : read input from `stdin` into a buffer
- `fork(2)` : create a new process that is a copy of the current (parent) process
- `execve(2)` : replace the current (child) process with another
- `wait(2)` : wait for a child process to finish before proceeding

Your program, in skeleton code, should look like this:

```
while (1) {
  write(STDERR_FILENO, PROMPT, ...);
  read(STDIN_FILENO, ...);
  pid = fork();
  if (pid == 0) {
    execve(...);
    perror(...);
    exit(EXIT_FAILURE);
  }
  if (pid > 0)
    wait(...);
}
```

You should spend some time carefully reading the man pages for all of these systems calls. To do so, in a docker terminal enter:

```
$ man 2 read
```

where `2` specifies the manual section. If you do not specify the manual section, you may get information for a different `read` command.

1.2 Timing is Everything

To time a running program, your shell will use the `alarm(2)` system call, which simply tells the operating system to raise a `SIGALRM` signal after a specified time. Your shell may need to catch the signal to avoid being terminated by the signal.

To handle a signal, a signal handing function must be registered with the operating system via the `signal(2)` system call. When the signal is delivered, the operating system will interrupt your shell's current operations (e.g., waiting for the program to finish).

1.3 Hit the `kill` switch!

The `kill(2)` system call send a signal to a process. Despite its morbid name, it will only *kill* (or terminate) a program if the right signal is delivered. One such signal which will *always* do just that is `SIGKILL`. This signal has the special property that it cannot be caught or ignored, so no matter what program your shell executes, it must heed the signal.

1.4 Prompting and I/O

Your shell must prompt the user for input. See Section 1.1 above. Following the prompt, your shell will read input from the user. The system call `read(2)` will return when the user presses Enter or `Ctrl-D`. The maximum line length is 4095 bytes (plus the terminating newline character if the user pressed Enter). It's recommended that you call `read(2)` as follows:

```
const int maxLineLength = 4096;
char cmd[maxLineLength];
int numBytes = read(STDIN_FILENO, cmd, maxLineLength);
```

If the user presses `Ctrl-D` after typing something, you should print a newline before executing the program or re-prompting as if the Enter key were pressed.

1.5 Argument Delimiters

The user-entered program may have arguments which are whitespace (space and tab) delimited. There may be leading and trailing whitespaces from the user input. For example:

```
penn-shredder#      /bin/sleep     10
```

1.6 Executing a Program

Your shell may only use the `execve(2)` system call to execute a program. This system call instructs the operating system to replace the current running program – that would be the child of your shell – with the specified program. Please refer to the manual for more details.

The `execve(2)` system call is the base of a larger collection of functions; however, you may not use those other functions. Specifically, you may not use `execl(3)`, `execv(3)`), or any other function listed in the `exec(3)` man page. As a result, the user of your shell must specify the path to a program to execute it. To learn where a program *lives* (i.e., its path), use the `which` program in bash.

Since the number of arguments varies from program to program, you are not allowed to declare a fixed-size array for the second argument of `execve()`. Instead, you must count the number of arguments for each user-entered program and use `malloc(3)` to allocate just enough memory for the array.

You should not invoke a system call if it's simple to determine from the internal state that the system call would do nothing, and you should monitor the status of any forked child process. (For example, your shell should ignore empty or all-whitespace lines instead of always calling `fork(2)`.)

1.7 Ctrl-C behavior

`Ctrl-C` (`SIGINT`) is a very helpful signal often used from the shell to terminate the current running process. Child processes started from *penn-shredder* should respond to `Ctrl-C` by following their normal behavior on `SIGINT`, but *penn-shredder* itself should not exit (even when `Ctrl-C` is typed without a child process running). Instead, your shell must catch `SIGINT` and re-prompt (after printing a newline character if appropriate):

```
$ penn-shredder
penn-shredder# /bin/cat
^C
penn-shredder# ^C
penn-shredder# sleep^C
penn-shredder# /bin/sleep 1
penn-shredder#
```

1.8 Ctrl-D behavior

If the user presses `Ctrl-D` at the beginning of the input line, your shell must exit. If the user types `Ctrl-D` when the input line is not empty, your shell should not exit; instead, the input is sent to `read(2)` and you should print a newline character.

1.9 Arguments to *penn-shredder*

The *penn-shredder* program must take an optional argument: the execution timeout. If the value of the argument is 0, then *penn-shredder* imposes no time restriction on child processes. For example, an optional argument of 10 results in a timeout of 10 seconds:

```
$ penn-shredder 10
```

Omitting the optional argument results in no timeouts. You must error out if the value of the argument is negative or more than one argument is provided. In either case, *penn-shredder* should return a nonzero (`EXIT_FAILURE`) status.

2 Error Handling

Most system calls and library functions report errors via the return value. You must check for errors for those with an asterisk in Section 5 below.

3 Code Organization

Sane code organization is critical for all software. Your code should not be all in one giant penn-shredder.c file. Rather, you should create reasonable modules and expose the relevant interfaces and constants in .h files. Make sure that your source code (.c and .h files) is in the top-level directory and can be built by the autograder's Makefile using the following command:

```
$ make -B
```

Your code should adhere to DRY (Don't Repeat Yourself). If you are writing code that is used in more than one place, you should write a function or a macro.

4 Memory Errors

You are required to check your code for memory errors. This is a nontrivial task, but an extremely important one. **Code with memory leaks and memory violations will incur deductions**. Fortunately, there is a very nice tool `valgrind` that is available to help you. You must find and fix any bugs that `valgrind` locates, but there is no guarantee it will find *all* memory errors in your code.

Below is a sample run of *penn-shredder* in valgrind:

```
$ valgrind ./penn-shredder 2
==151614== Memcheck, a memory error detector
==151614== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==151614== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==151614== Command: ./penn-shredder 2
==151614==
penn-shredder#          sleep
sleep: No such file or directory
==151627==
==151627== HEAP SUMMARY:
==151627==     in use at exit: 0 bytes in 0 blocks
==151627==   total heap usage: 3 allocs, 3 frees, 1,512 bytes allocated
==151627==
==151627== All heap blocks were freed -- no leaks are possible
==151627==
==151627== For lists of detected and suppressed errors, rerun with: -s
==151627== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
penn-shredder#  /bin/sleep
/bin/sleep: missing operand
Try '/bin/sleep --help' for more information.
penn-shredder# /bin/sleep       3
Bwahaha ... Tonight, I dine on turtle soup!
penn-shredder# /bin/sleep 1
penn-shredder#  cat     ^C
penn-shredder# /bin/cat
Bwahaha ... Tonight, I dine on turtle soup!
penn-shredder# ^C
penn-shredder# /bin/cat
^C
penn-shredder#
==151614==
==151614== HEAP SUMMARY:
==151614==     in use at exit: 0 bytes in 0 blocks
==151614==   total heap usage: 6 allocs, 6 frees, 112 bytes allocated
==151614==
==151614== All heap blocks were freed -- no leaks are possible
==151614==
==151614== For lists of detected and suppressed errors, rerun with: -s
==151614== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

5 Acceptable System Calls and Library Functions

In this assignment you may use only the following system calls and library functions. **You must check errors for those with an asterisk.** For example:

```
pid_t pid = fork();
if (pid == -1) {
  perror("fork");
  exit(EXIT_FAILURE);
}
```

5.1 System calls (Section 2)

- `alarm`
- `execve*`
- `_exit`
- `fork*`
- `kill*`
- `read*`
- `signal*`
- `wait*`
- `write*`

5.2 Library functions (Section 3)

- `atoi`
- `exit`
- `free`
- `malloc*`
- `perror`
- `strlen`
- `strtok`

6 Mandatory Additional Credit

For 2% of your final grade implement the following mandatory additional credit: don't use `kill(2)`. This is in addition to, not in place of, the regular credits. You must wrap your additional credit code with the C macro `EC_NOKILL` as follows:

```
#ifdef EC_NOKILL
    // mandatory additional credit code
#else
    // regular credit code
#endif

#ifdef EC_NOKILL
    // mandatory additional credit code
#endif

#ifndef EC_NOKILL
    // regular credit code
#endif
```

Don't do `#define EC_NOKILL` anywhere in your code. The autograder will define it via `make` as follows:

```
$ make -B CPPFLAGS=-DEC_NOKILL
```

7 Developing Your Code

In general, there are two environments for you to develop projects in this course: (1) **Docker Container** or (2) *SpecLab* servers for remote developing. We recommend the first environment.

For the first choice, we provide a Docker Image for your convenience to create a course-standardized container. You can find the set-up instructions on Ed. We highly recommend you to use it as your developing environment because all grading will be done on it. Please do not develop on macOS directly as there are significant differences between macOS and Unix variants such as Ubuntu Linux. If you decide to develop on a different environment anyway, you must compile and test your *penn-shredder* on the course-standardized environment to ensure your *penn-shredder* runs as you expect during grading.

For the second choice, *SpecLab* will be available to you as CIS 5480 students, however, we **discourage its usage** unless your personal machine has super, super weak performance specifications such as a single core machine with 2GB of RAM or less. *SpecLab* is a collection of older desktops that run the same Linux variant as *eniac*, and most importantly, you can crash them as much as you want without causing too much chaos. You access *SpecLab* remotely over *ssh*. There are roughly 10 *SpecLab* machines up at any time. When you are ready to develop, choose a random number between 01 and 50, or so, and then issue this command:

```
$ ssh specDD.seas.upenn.edu
```

where `DD` is replaced with the number you chose. If that machine is not responding, then add one to that number and try again. Not all *SpecLab* machines are currently up, but most are. **You must be on SEASnet to directly *ssh* into *SpecLab***. The RESnet (your dorms) is not on SEASnet, but the machines in the Moore

computer lab are. If you are not on SEASnet, you may still remotely access *SpecLab* by first *ssh*-ing into eniac and then *ssh*-ing into *SpecLab*.

Do not develop your code on *eniac*. CIS 5480 students are notorious for crashing *eniac* in creative ways, normally via a "fork-bomb." This always seems to happen about two hours before a homework deadline, and the result is chaos, not just for our class, but for everyone else using *eniac*.

Students who are caught running *penn-shredder* directly on *eniac* will be penalized.

8 What to turn in

You must provide each of the following for your submission, *no matter what*. Failure to do so may reflect in your grade.

1. `README` file. In the `README` you will provide
    - Your name and PennKey
    - A list of submitted source files
    - Overview of work accomplished
    - Description of code and code layout
    - General comments and anything that can help us grade your code
2. Your code. You may think this is a joke, but at least once a year, someone forgets to include it.

9 Submission

Submission should be done through Canvas. Please compress your files via the following:

```
tar cvaf yourpennkey-project0.tar.gz *.c *.h README
```

As explained above, please remember to include your README and any header files you have written, along with your main *penn-shredder* source code. Canvas will accept multiple submissions, but we will grade only the most recent submission submitted before the submission deadline.

10 Grading Guidelines

- 1% GitHub Username
- 4% Documentation
- 90% Functionality
- 5% Peer Review (will be released later)

Please note that general deductions may occur for a variety of programming errors including memory violations, lack of error checking, poor code organization, etc.

You may use any comment style you wish, but we highly suggest using DOxygen style comments as they promote fully documenting function inputs, outputs, and error cases. A useful guide to this comment style can be found here: `https://www.doxygen.nl/manual/docblocks.html`

After the submission, you will evaluate another student's code based on a given rubric using a profiling tool such as `valgrind` and sanitizers. Keep a lookout on *edStem* for more details about this after your submission.

Your programs will be graded on the course-standardized environments and must execute as specified there. Although you may develop and test on your local machine, you should always test that your program functions properly there.

11 Attribution

This is a large and complex assignment, using arcane and compactly documented APIs. We do not expect you to be able to complete it without relying on some outside references. That said, we do expect you to struggle a lot, which is how you will learn about systems programming, by *doing things yourself*. Please feel free to refer to work you have done in previous Penn courses (such as CIS 240); it can be a helpful reminder for things like Makefiles and general C syntax.

The primary rule to keep you safe from plagiarism/cheating in this project is to attribute in your documentation any outside sources you use. This includes both resources used to help you understand the concepts, and resources containing sample code you incorporated in your shell. The course text and APUE need only be cited in the latter case. You should also use external code sparingly. Using most of an example from the `pipe(2)` man page is ok; using the `ParseInputAndCreateJobAndHandleSignals()` function from *Foo's Handy Write Your Own Shell Tutorial* is not (both verbatim, and as a closely followed template on structuring your shell).