Project 1
penn-shredder: Turtles in Time

"Tonight I dine on turtle soup!"

shredder, *Teenage Mutant Ninja Turtles the Animated Series*

Directions

This is an *individual* assignment. You may not work with others. Please regularly check the coding style guidelines provided. You are encouraged to version control your code, but **do not work in public GitHub repositories. Please avoid publishing this project at anytime, 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 run time of executed processes. Your shell will read input from the users and execute it as a new process, but if the process exceeds a timeout, it will be killed. You will complete this assignment using only a specified set of *system calls* and without the use of standard C library functions (*e.g.*, `printf(3)`, `fgets(3)`, `system(3)`, etc.).

You may freely use any functions in `string.h`. Please be aware of the accuracy of your output contents, especially because the autograder will take into consideration whitespace for correctness. **Even if the functionality is correct, the autograder will mark your submission incorrectly if the output contents are not accurate.**

You are provided with starter code for the most of the functionality listed in the Background section. Read through the section carefully to understand what the starter code is accomplishing.

This project is divided into three 3 parts:

1. Replace "/bin/ls" command with a program that takes in user input via standard input (`STDIN`) system call, run the user command to execute the command in a child process.
2. Modify the code to set a timer to 'kill' your child process if it runs past X seconds.
3. Peer-grade a random partner in class. We will assign your peer-grading partner after part 2 is completed.

Advice

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*.

Even though the project is quite challenging, it is also gratifying based on past experience as long as you pass the learning curve. To be an efficient learner, please make good use of the C - Refresher, GDB tutorial (week 3), TA hours, Open Office Hours, and Recitations. They are excellent supplements to the lectures and have many practical system programming tips to improve your project. If you feel that the current TA hour set-up does not fit your schedule well, please let us know.

Moreover, you are highly encouraged to utilize Ed Discussion to discuss the project with classmates and TAs. Before asking the question, please search in Ed Discussion to see if there are any similar questions asked before. If you plan to ask about your code/implementation, please consider filling in the following template to make the communication efficient. We are here to help you and make sure you get the most out of the course!

- GitHub URL:
- Description of Question:
- Expected Behavior:
- Observed Behavior:

1 Background

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; waits for the process to finish; and then re-prompts the user completing the loop. The shell exits the loop when the user provides `EOF` (end of file) (i.e., `Ctrl-D`). Your shell, *penn-shredder*, will do all of that, but with a slight twist.

penn-shredder takes a command line option for specifying a program timeout. If any process runs for longer than the specified timeout, *penn-shredder* will kill the program and report to the user his snide catch-phrase, "Bwahaha ... tonight I dine on turtle soup". The exact whitespace usage in the catch-phrase is necessary for the autograder to correctly grade your assignment.

The following is an example of input that should invoke the catch-phrase:

```
bash# ./penn-shredder 10
penn-shredder# /bin/cat
Bwahaha ... tonight I dine on turtle soup
penn-shredder# /bin/pwd /home/yourusername
penn-shredder#
```

Here, *penn-shredder* was executed with a timeout argument of 10 seconds, and any program that runs longer than 10 seconds will be killed (or shredded). The `cat` program executed without arguments runs indefinitely, and thus was killed after exceeding the timeout. Conversely, `pwd` returns quickly and was not killed, upsetting *shredder* and his henchman.

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 four system calls:

- `read(2)` : read input from `stdin` into a buffer

- `fork(2)` : create a new process that is an exact copy of the current running program.
- `execve(2)` : replace the current running program with another
- `wait(2)` : wait for a child process to finish before proceeding

The program, in pseudo-code, looks roughly like this:

```
while(1){
  read(cmd, ...);
  pid = fork();
  if(!pid){
    execve(cmd,...);
  }else{
    wait();
  }
}
```

This may seem simple, but there are many, many things that can go wrong. You should spend some time carefully reading the **entire** man page for all four of these systems calls. To do so, in a terminal type:

```
bash# 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 Executing a Program

Your shell uses 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. Explicitly, you may not use `execl(3)`, `execv(3)`), or any other function listed in the `exec(3)` manual page. As a result, the user of your shell must specify the entire path to a program to execute it. To learn where a program *lives* (i.e., its path), use the `which` program in your non-penn-shredder shell.

1.3 Ctrl-C behavior

`Ctrl-C` (`SIGINT`) is a very helpful signal often used from the shell to stop the current running program. 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 invoked without a child process running).

```
bash# ./penn-shredder
penn-shredder# /bin/cat
^Cpenn-shredder# ^C/bin/pwd
<Your working directory>
penn-shredder#
```

Note: The above command `^C/bin/pwd` does not exactly match a real shell program but is the expected output for `penn-shredder`. If you're curious what that does in a real shell, try it out!

In the starter code, we provide an example of `SIGINT` signal handler so that the parent process (penn-shredder) will kill the child process when it receives `Ctrl-C`. However, when the child process executes a command, the `execve(2)` system call will reset the behavior of a handled signal to default as described in `signal(7)` manual page:

during an `execve(2)`, the dispositions of handled signals are reset to the default; the dispositions of ignored signals are left unchanged.

By default, `SIGINT` will terminate the process so the child process will also get killed by `SIGINT` in the child process. A command like "/bin/cat" followed by `Ctrl-C` could lead to a race condition that both the parent process and the child process try to kill the child process. In the worst case, it may cause an error if the parent process tries to kill a child process which has already been terminated. One way to avoid the race condition is to ignore `SIGINT` in the child process since the dispositions of ignored signals won't be changed. You can ignore a signal by setting the disposition of the signal to `SIG_IGN` via the `signal(2)` system call.

We will not test your signal handling in part A to simplify the learning objectives but we will test it in part B.

1.4 Ctrl-D behavior

`Ctrl-D` (End Of File: EOF character) is a very helpful symbol mostly used to exit from the shell if it not currently running any other process. There are two situations about `Ctrl-D`:

1. `Ctrl-D` without text
2. `Ctrl-D` with text (valid command plus `Ctrl-D`).

For `Ctrl-D` without text, it is straightforward it should exit the shell if it is not currently running any other process.

```
bash# ./penn-shredder
penn-shredder# ^D
<Should exit the shell>
bash#
```

For `Ctrl-D` with text (valid command plus `Ctrl-D`), i.e. /bin/ls `Ctrl-D` it should not exit shell without completing current running process and it should go back to "`penn-shredder#` "prompt. The auto-grader will give you a the points as long as your Shredder dose not exit or crash after `/bin/ls Ctrl-D`. But it would be even better if your Shredder returns the result of `/bin/ls`.

```
bash# ./penn-shredder
penn-shredder# /bin/ls^D
<list of files in the directory>
penn-shredder#
```

Note that `Ctrl-D` is not a signal; while we will only test signal handling in part B, this is not strictly a signal and we will test for `Ctrl-D` handling in part A.

2 Project Milestones and Submission

2.1 Part 1A: Do more than 'ls'!

Your first task is to replace the hard-coded "/bin/ls"call in `getCommandFromInput()` such that the program can take in user input and run that command instead of "/bin/ls". Your program must be able to handle user input with spaces before and after the command (spaces are ignored).

2.1.1 Prompting and I/O

In programming your shell, *you will only use system calls* and nothing from the C standard library (except string.h). This includes input and output functions like `printf(3)` and `fgets(3)`. Instead you will use the `read(2)` and `write(2)` system calls. Consult the manual pages for these functions' specification.

Your shell must prompt the user for input as follows:

```
penn-shredder#
```

Note that the prompt has a white space after the octothorpe, so if a user begins typing, they would see:

```
penn-shredder# somestringhere
```

This part is already in the starter code.

Following the prompt, your program will read input from the user. You may assume that the user input will be at most 1024 bytes (including the null-termination byte).

2.1.2 Submission

Develop your code in the Git repository we assigned to you in project 0. However, create a brand new directory called **project1a**, and put all your code there. When you are done, check in and push your code to the repository. You should only check in source files but not binary or object files. Please also include an (optional but encouraged) README file that describes your experiences working on this assignment. This can include time spent and your most difficult bug and how you resolved it.

Separately, create a tar-ball of your submission. This can be done by going to the directory above project 1, and then typing in the Linux command: `tar -zcvf project1a.tar.gz project1a`. Upload this tar-ball onto the Canvas submission site for project 1. The submission of this tar ball will trigger execution of an auto-grader. You may submit as often as you would like. We encourage you to start the project early so that you can validate your output against our auto-grader.

2.2 Part 1B: Hit the `kill`-switch!

The `kill(2)` system call delivers 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 will *always* do just that, `SIGKILL`. The `SIGKILL` signal has the special property that it cannot be handled or ignored, so no matter the program your shell executed it must heed the signal.

2.2.1 Timing is Everything

To time a running program your shell will employ the `alarm(2)` system call. The `alarm(2)` system call simply tells the operating system to deliver a `SIGALRM` signal after a specified time. The `SIGALRM` signal must be handled by your shell; otherwise, your shell will exit.

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 preempt your shells current operations (e.g., waiting for the program to finish) and execute the signal handler.

To best understand signal handling, here are some questions you could try and answer:

- What happens if you remove the call to `signal(2)`?
- What happens if you provide different arguments to `alarm(2)`?
- What happens if you use the `sleep(3)` function instead of the busy wait?
- Will the signal handler be inherited by the child after calling `fork(2)`?
- What happens to the signal handler after calling `execve(2)`?

You should also spend time carefully reading the **entire** man page for these systems calls and references in *APUE* regarding signals and signal handling.

2.3 Interrupt Signalling

You should ensure that your implementation of `Ctrl-C` (`SIGINT`) handling matches the specifications above. Do note that the starter code has a complete `SIGINT` signal handler implementation; you do not need to (and should not attempt to) modify the provided signal handler. The obvious hint is that other parts of your program may need to be modified to meet the specifications.

2.3.1 Submission

Develop your code in the Git repository we assigned to you in project 0. However, create a brand new directory called **project1b**, and put all your code there. When you are done, check in and push your code to the repository. You should only check in source files but not binary or object files. Please also include an (optional but encouraged) README file that describes your experiences working on this assignment. This can include time spent and your most difficult bug and how you resolved it.

Separately, create a tar-ball of your submission. This can be done by going to the directory above project 1, and then typing in the Linux command `tar -zcvf project1b.tar.gz project1b`. Upload this tar-ball onto the Canvas submission site for project 1. As before, the submission of this tar ball will trigger execution of an auto-grader.

2.4 Part 1C: Peer review

Project 1c is a two-part assignment. The first part requires you to submit your Project 1 code. The second part requires you to review your peers' Project 1 codes. This assignment is worth 2% of your final grade for this class. The purpose of this exercise is to help you to write readable and easily maintainable code by evaluating the work of someone else. You can use the *Coding Style Guidelines* in the Canvas Project Resources Module to guide your review of your peer's code.

Submit Your Code

You need to submit your Project 1b code in a `tar.gz` file to Canvas before reviewing others' work. Optionally, you can beautify your code before submitting for the peer review. This is a chance for you to focus on the code layout after your functionality is working. Note that this is optional – you can also submit your code as it is in Project 1b for peer review.

Peer Review

Some of the peer review questions appear to be asking questions about our starter-code. You may be wondering why. The reason is that, even though you did not write that code, we want to draw your attention to code you did not write, and we want you to be thinking about why we wrote that code in a certain way.

You will need to run a memory leak detection tool to look for memory errors in your peers' code. We have a short introduction about `Valgrind` in the next section. Also, we prepared a demo video, called *Introducing Errors*, in Week 1: Lesson 5: C-Refresher, to demonstrate how to detect memory errors with `Valgrind`.

Logistics

1. By the due date given in the Ed Discussion announcement about 1c, submit your 1b code to the 1c assignment. Extensions are not permitted for this part of the assignment. You will need to wait to review your peers' code until after this deadline has passed and the Course Manager has assigned your peer reviews.

2. By the assignment deadline in Canvas, review your peers' code that was assigned to you. To do so, download their tar file, unzip it, read their code, and run their code with Valgrind.

3. To put your review in Canvas, click on "Show Rubric". This will open the rubric where you assign points for each rubric item and add comments per rubric item. You can also leave global feedback in that comment box at the end of the rubric. When you are sure your review is complete, click "Save Rubric". This will mark your review as final, and your peer will immediately be able to see your review.

4. You will be manually graded, based on the below criterion, after all peer reviews have been completed.

Grading

We will give full credit to students who submit 3 reviews, regardless of how their code has been reviewed by other students. This project is meant to have students learn and exchange ideas with each other without grading pressures.

3 Guidelines

3.1 Error Handling

All system call functions you use will report errors via the return value. As a general rule, if the return value is less than 0, an error occurred and `errno` is set appropriately. **You must check your error conditions and report errors with a reasonably phrased error message that contains the keyword 'invalid'**. To

expedite the error checking process, we allow you to use `perror(3)` library function. Although you are allowed to use `perror`, it does not imply you should report errors at an extreme verbosity. Instead, try and strike a balance between sparse and verbose reporting.

3.2 Code Organization

Code organization is critical for all software. You 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.

3.3 Memory Leaks

You are strongly encouraged to check your code for memory leaks. This is a nontrivial task, but an extremely important one. Memory leaks will cause your computer to slowly run out of memory and may cause programs to crash. Writing code without memory leaks is a critical part of developing programs in C.

In this class, we introduce you `Valgrind`, a very nice tool to help you identify leaks. However, there is no guarantee it will find *all* memory leaks in your code, **especially those that rely on user input!** Also, you still need to find and fix any bugs that these tools locate by yourself.

To clarify, having memory leaks will not cause you to lose points, as long as your shell otherwise functions correctly. Our autograder will catch many memory leaks and give you feedback (but will not take points off). In the peer review, you will check each other's shells for memory leaks, but again you will not be penalized. Still, we strongly encourage you to fix your memory leaks and post on Ed Discussion if you need help to ensure that your shell does not have any bugs and for the best learning experience.

3.4 Memory Errors

In constrast to memory leaks, memory management errors are serious and must be handled appropriately. Our autograder will be checking for the most common types of memory errors: for example, buffer overflow and the usage of uninitialised variables. These types of memory management errors can lead to difficult to trace bugs because they may only occur with certain inputs or sequences of events.

Based on our experience, most students will encounter at least one of two common memory management errors in this course:

1. *Use of uninitialised variables*

   C does not initialise variables to a default state. You must have the value of any variable set before attempting to use it.

   ```
   // allocates space on the stack by moving the stack pointer
   int i;

   // allocates space on the stack and writes \0 into memory
   int j = 0;

   // allocates space on the stack and writes the return value to memory
   int k = someFunction();
   ```

2. *Writing outside array boundaries*

C does not check if you write outside the defined limits of an array. Be careful to check the length of data structures using the `sizeof` operator and use safe functions that prevent writing past a buffer's allocated space (e.g. `strncpy` vs `strcpy`).

To clarify, having memory errors will cause you to fail the test scenario, even if it appears to work from a functionality standpoint. Our autograder is using Valgrind to catch these errors and you should too!

3.4.1 Using `valgrind`

We strongly encourage you to use to check for memory leaks and we require that you handle memory errors. Please use Valgrind as follows (this is the same as what the autograder uses):

```
valgrind --leak-check=full --track-origins=yes ./penn-shredder
```

Be sure to add the additional shell timeout argument when testing part B.

After running this command, interact with your shell as you normally would. When you are finished and exit the program, you should see a report on memory usage. **This is what you should aim for:**

```
==121== HEAP SUMMARY:
==121==      in use at ext: 0 bytes in 0 blocks
==121==    total heap usage: 28 allocs, 28 fres, 1,750 bytes allocated
==121==
==121== All heal blocks were freed -- no leaks are possible
==121==
==121== For counts of detected and suppressed errors, rerun with: -v
==121== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Note that just because you get this type of summary one time does not mean that you will get it every time. You should test your shell very thoroughly, with an extensive set of inputs to cover all possible cases that could cause memory leaks. There may only be one particular set of inputs that cause memory leaks. You should identify these inputs and resolve the memory leaks in your code.

Is it okay to have "Still Reachable" memory? We do not want you to have "Still Reachable" memory. While "Still Reachable" memory is memory to which there is still a pointer to upon termination and can be freed correctly under the hood, it still means that you are not freeing memory correctly when exiting. For example, a common mistake is not freeing the memory allocated in your `getCommandFromInput()` function when exiting on `Ctrl + D` (of course, you should also be freeing the memory correctly for all other inputs).

Memory errors are usually reported as soon as Valgrind detects them, so you may see commands interleaved with Valgrind output. The value between the == is the process ID of the process with the error. Using the `--track-origins=yes` option will show a stack trace of the error where the memory was allocated.

```
pennshredder# /bin/ls
==121== Conditional jump or move depends on uninitialised value(s)
==121==    at 0x400E0A: getCommandFromInput (penn-shredder.c:194)
==121==    by 0x400A1B: executeShell (penn-shredder.c:130)
==121==    by 0x400968: main (penn-shredder.c:41)
```

Please test your work thoroughly and post on Ed Discussion if you have any difficultly solving memory issues. Good luck!

4 Acceptable Library Functions

In this assignment you may use only the following system calls:

- `execve(2)`
- `fork(2)`
- `wait(2)`
- `read(2)`
- `write(2)`
- `signal(2)`
- `alarm(2)`
- `kill(2)`
- `exit(2)`

And you may use these non-system calls:

- `malloc(3)` or `calloc(3)`
- `free(3)`
- `perror(3)` for reporting errors
- `atoi(3)` or `strtol(3)` but just once!
- `string(3)` for string utility functions

5 Developing Your Code

We highly recommend you use the course-standardized virtual machine given by course staff because all grading will be done on the course virtual machine. Please do not develop on macOS as there are significant enough differences between macOS and Unix variants such as Ubuntu Linux. If you decide to develop on a different machine anyway, you must compile and test your penn-shredder on the course-standardized virtual machine to ensure your penn-shredder runs as you expect.

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*. Having said that, the starter code is designed to aid you in aiding, and you will not need to add new `.c` or `.h` files.

Your programs will be graded on the course-standardized virtual machines, 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.