In this assignment, you will implement a basic shell that restricts the runtime of processes executed from within it. Your shell will read input from the users and execute it as a new process, but if that program runs longer than it should, your shell will kill the process and report this to the user. To complete this task, you may only use system calls and not any functions from the standard C library. This means that, for example, printf(3), fgets(3), system(3), etc, are verboten for use.

1 Specification

At its core, a shell is a simple lop. Upon each iteration, it prompts the user for a command, attempts to execute that command as a new process, waits for that process to finish, and re-prompts the user until EOF. Your shell will do all of this with a slight twist.

The user of your shell has the option to specify a program timeout on the command line. If any process runs for longer than the specified timeout, then your shell will kill the current running program and make some snarky remark. If, however, the process ends before the specified timeout, a remark indicating the shell’s frustration at being unable to kill the process will be displayed instead. Below we consider the shredder shell in homage to the infamous nemesis of the Teenage Mutant Ninja Turtles, but feel free to make a shell in tribute to your favorite evil henchman (e.g., fudd-sh (“I’m hunting wabbits”), claw-sh (“I’ll get you next time, Gadget”), etc.). Particularly witty or humorous rejoinders may score bonus marks.

Let’s see shredder in action:

```
bash# ./shredder 10
shredder# /bin/cat
Bwahaha ... tonight I dine on turtle soup
shredder# /bin/pwd
/home/butler
Argg ... Bee-Bop, how could you let them get away?!?
```

Here, shredder was executed with the argument “10” and any program that runs longer than 10 seconds will be killed. cat with no arguments will run indefinitely; thus, it was killed with glee. Conversely, pwd returns quickly and was not killed, to the displeasure of shredder and his henchman.

1.1 read, fork, exec, wait, and repeat

As described above, a shell is just a loop performing the same procedure repeatedly. 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

---

1Many thanks to Adam Aviv, Perry Metzger, Sandy Clark, Micah Sherr, Stefan Miltchev, and Eric Cronin for the design and refinement of this assignment.
• wait(2): wait for a child process to finish before proceeding

Your program, in pseudocode, will most likely look like this:

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

While this may appear to be simple, there are many, many things that can go wrong. You should spend some time reading the entire man page for all four of these system calls. As a reminder, `read(2)` means that it is in section 2 of the manual, so to read its man page in a terminal, type:

```
bash# man 2 read
```

If you don’t specify “2” explicitly, you may get information for a different `read` command.

### 1.2 Timing and Signals

To time the current running program, you will use the `alarm(2)` system call, which tells the operating system to deliver a `SIGALRM` signal after some specified time. If `SIGALRM` is unhandled, the shell will exit. Signal handling is done by registering a signal handling function with the operating system. When the signal is delivered, the current execution will be pre-empted and the signal handling function will execute. This paradigm is demonstrated in the examples below.

In this following code, no signal handler is registered:

```c
#include <signal.h>
#include <unistd.h>

int main (int argc, char **argv) {
    alarm(1);
    while (1);
}
```

When executed, the message Alarm Clock will be printed to stderr because the program exited due to an unhandled alarm signal.

However, this program will exit cleanly because the `SIGALRM` is handled, resulting in a call to `exit(2)`:

```
What happens if there is no `exit(2)` call?
```
```c
#include <signal.h>
#include <unistd.h>

void handler (int signum) {
    exit(0);
}

int main (int argc, char **argv) {
    signal(SIGALRM, handler);
    alarm(1);
    while (1);
}
```

By using the `alarm(2)` and `signal(2)` system calls, your shell can time the current running program. Handling asynchronous signaling is far more nuanced than described here: you should spend some time reading the entire man pages for these system calls and referencing online and printed resources (such as the books suggested on the course web page) to gain a better understanding of signals and signal handling.

### 1.3 The `kill`-switch

To terminate a running process, you will use the `kill(2)` system call. Despite its morbid name, the purpose of `kill(2)` is to deliver a signal to another process (not necessarily killing it). It will only kill the process if it delivers the right kind of signal. There is one such signal that you will use to do just that, `SIGKILL`. `SIGKILL` has the special property that it cannot be caught or ignored, so no matter the program your shell just executed, it must heed the signal.³

### 1.4 Prompting and I/O

As previously stated, you must use only system calls, and nothing from the C standard library. This includes input and output functions such as `printf(3)` and `fgets(3)`. Instead, you will use the `read(2)` and `write(2)` system calls. Your shell must prompt the user for input in a natural way, such as:

```
shredder#
```

Following the prompt, your program must read input from the user. You do not need to read input of arbitrary length; instead, you can truncate input to 1024 bytes, but you must gracefully handle input longer than that. Your shell should not crash in such situations.

### 1.5 Argument Restrictions

We do not require you to parse program arguments in your shell; that is, a user can provide any number of arguments but you do not actually have to parse them. However, your shell must still execute the program as intended. For example:

```
shredder# /bin/sleep 100
sleep: missing operand
Try ‘sleep --help’ for more information.
```

Even though the user provided the argument “100”, you are not required to pass this argument along to `execve`. Due to this restriction, you may find it useful to write separate testing programs that exit after a specified time (perhaps based on the `SIGALRM` example above).

³Extra credit: What other signal has this property? Which signals will terminate the program if left unhandled? Which man page has this information? Answer these questions in the README file that you submit.
Extra credit (5 pts): For extra credit, write a simple parser that will handle arbitrary arguments and pass them to `execve` appropriately. Note that you cannot use any function from the C standard library. This includes everything from `string.h`, so no use thinking about using `strdup(3)` or `strtok(3)` or their variants.

1.6 Shell arguments

Despite the fact that there is no requirement to handle arguments within your shell, the shell program itself (`shredder`) must take an optional argument, the execution timeout. If no argument is provided, then `shredder` (or whatever you’re calling your shell) imposes no time restriction on executing processes.

You may be asking yourself, “Self, how am I going to convert a command line argument into an integer if I’m not allowed to use anything from the C standard library?” For this purpose alone, you may make a single call to `atoi(3)` or `strtol(3)` to convert command line input into an integer. You should check for errors in this conversion, e.g., when a user provides a character instead of a number.

Extra credit (5 pts): For extra credit, implement your own version of `atoi(3)` that can handle arbitrary numbers within the 32-bit width of an integer.

2 Acceptable Library Functions

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

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

You may also use the following non-system calls:

- `malloc(3)` or `calloc(3)`
- `free(3)`
- `perror(3)` for error reporting
- `atoi(3)` or `strtol(3)`, but only once

Using any library function other than those specified above will adversely affect your mark on this assignment. In particular, if you use the `system(3)` library function, you will receive a ZERO.
3 Error Handling

All system call functions that you use will report errors via the return value. As a general rule, if the return value is less than zero, then an error has occurred and errno is set accordingly. You must check your error conditions and report errors. To expedite the error checking process, we will allow you to use the perror(3) library function. Although you are allowed to use perror, it does not mean that you should report errors with voluminous verbosity. Report fully but concisely.

4 Memory Errors

You are required to check your code for memory errors. This is a non-trivial task, but a very important one. Code that contains memory leaks and memory violations will have marks deducted. Fortunately, the valgrind tool can help you clean these issues up. It can be installed inside your Linux VM (or native distro, if that’s how you’re running it). Remember that valgrind, while quite useful, is only a tool and not a solution. You must still find and fix any bugs that are located by valgrind, but there are no guarantees that it will find every memory error in your code: especially those that rely on user input.

5 Developing Your Code

The best way to develop your code is in Linux running on a virtual machine. This way, if you crash the system, it is straightforward to restart. This also gives you the benefit of taking snapshots of system state right before you do something potentially risky or hazardous, so that if something goes horribly awry you can easily roll back to a safe state. You may use the room 100 machines to run Linux within a Virtualbox environment, or run natively or within a VM on your own personal machine. Distribution doesn’t matter too much (we’ll be testing on Ubuntu 10), but it must be Linux. Importantly, do not use ix for this assignment. If you develop on Solaris or a BSD variant such as Mac OS X, you must ensure that the code will run on a Linux machine.

6 Submission Instructions

You must turn in the following:

1. A README file. The README will contain the following:
   - Your name
   - A list of submitted source files
   - Extra credit answers (if any)
   - Compilation instructions
   - Overview of work accomplished
   - Description of code and code layout
   - General comments and anything that can help us grade your code

2. A Makefile. We should be able to compile your code by simply typing make. If you do not know what a Makefile is or how to write one, ask Kaveh for help or consult one of the many online tutorials.
3. Your code. This is not a joke: people have been known to forget to include it.

To submit the code, email cis415-projects@cs.uoregon.edu with your solution attached as a gzipped tar file (i.e., it should have a .tar.gz or .tgz file extension). Place all relevant code and documents into a subdirectory named lastname-proj1. Since my last name (incidentally my username as well) is butler, I’d create a directory named butler-proj1 that contained the files I was turning in. The command to make the gzipped tar file from the parent directory would then be

```bash
    tar cvzf lastname-proj1.tar.gz lastname-proj1
```

where lastname is your last name, of course.

7 Grading Guidelines

This programming project will be graded as follows:

- 10% Documentation
- 20% Proper use of system calls and general code design
- 70% Functionality and correct operation of your shell

Note that general deductions may occur for a variety of programming errors including memory violations, lack of error checking, poor code organization, etc. Also, do not take the documentation lightly, as it provides those evaluating your project with a general roadmap of your code; without good documentation, it can be quite difficult to grade the implementation. Once again, these will be graded on machines running the Ubuntu 10 Linux distribution.

8 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 that you will be challenged with this assignment, and that you will learn about systems programming most effectively by doing the work yourself.

In order to avoid issues of plagiarism and cheating in this project, you must attribute any outside sources that you use. This includes both resources used to help you understand the concepts, and resources containing sample code that you may have borrowed as a template for your shell. The course textbook only needs to be cited in this latter case; all other sources must be attributed. You should use external code sparingly. For example, using most of an example from the man page for pipe(2) would be reasonable. Using a function such as ParseInputandCreateJobandHandleSignals() from a Write Your Own Shell in Four Easy Steps site would not be OK, either using text verbatim or closely following the structure for your own shell design.