1 Objectives

The objectives of this lab are:

1. Getting more practice with C programming in general.
2. Getting experience with signal handling.
3. Getting experience with process creation.

2 Requirements

2.1 Big Picture

Create a simple command-line shell.

2.2 Interface Requirements

Create a command-line shell that complies with the following requirements:

1. Once started, it displays a simple prompt, as shown:

```
$ ./shell
prompt>
```

2. As a general guide, your shell should behave like a typical command-line shell.
3. For simplicity, you may assume that a command given by the user is less than 80 characters.
4. If the user only presses the Return key at the prompt (or white space followed by a Return), it should **not** produce an error message, but instead shall produce another prompt.
5. Like the typical command-line shell, it shall attempt to `fork/exec` the program the user enters. However, there are two special cases where it will not try to `fork`/`exec`:
   - When the user enters "exit", your program shall terminate with a status of 0.
   - When the user enters "explode" you shall cause a segmentation fault (on purpose) by doing the following:

     ```
     int *bomb = NULL;
     *bomb = 42;
     ```
6. If an error occurs during a `fork`, then a meaningful message shall be displayed to the user, but your shell program shall **not** terminate.
7. If the `exec` fails in the child process, then the child process must exit (or weird things will happen).
8. The shell shall also maintain a file called "shell-history" that contains a list of all commands executed by the shell. If the shell is terminated and then started again, the new commands shall be appended at the end of the history file.
9. All error messages shall be sent to `stderr`.
10. After 30 seconds of run-time, your shell shall be programmed to receive the SIGALRM signal. (See the `alarm` function).

11. The program shall have a signal handler that can catch the SIGSEGV signal (i.e., a segmentation fault). When this signal is received, this handler shall display the information below on `stderr`:

```
A segmentation fault has been detected.
Exiting...
```

    The handler shall then flush the history file, and then exit the program with a success code of -1.
12. The program shall have a signal handler that can catch the SIGINT signal. When SIGINT is received, this handler shall display the information shown below on `stdout`:

```
The Interrupt signal has been caught.
Exiting...
```

    The handler shall then flush the history file, and then exit the program with a success code of -2.
13. The program shall have a signal handler that can catch the SIGALRM signal. When SIGALRM is received, this handler shall display the information shown below on `stdout`:

```
The session has expired.
Exiting...
```

    The handler shall then flush the history file, and then exit the program with a success code of -3.
14. Your program should be able to handle commands with up to 10 space-separated items. For example, "`ls -l`" has two space-separated items. You can choose to simplify your implementation by only handling a single item, but the maximum grade you can earn is 92 with that approach. Example one-word commands (with no options):
    - `ls`
    - `top`
    - `clear`
    - `date`
    - `pwd`
    - `cal`

    Otherwise, to qualify for up to 100 points, your shell should be able to properly process the following commands:
    - `ls -l`
    - `cat /etc/passwd`
    - `cal 2017`
    - `cat 1 2 3 4 5 6 7 8 9`

    In all cases, your program must be able to successfully execute commands even if spaces are entered before and/or after the command.

2.3 Other Requirements

1. Implement this program in a file called `shell.c`.
2. You are **not** allowed to use the `system` function; you must use the `fork`/`exec` approach to starting a child process.
3. You are not allowed to use the `signal` function; this function is being obsoleted, so you must use the `sigaction` function.

3 Submission

Post shell.c on Sakai by the due date.

4 Grading

Your grade will be based on the following **guidelines**:

1. Compiles without errors (20).
2. Compiles without warnings (10).
3. Proper memory handling when `ls` and `exit` are executed, such as `valgrind` reporting no errors, and no segmentation faults (10).
4. It performs as specified:
    a. It can handle as many as ten items on the command-line. (8)
    b. There is no debugging code still producing output.
    c. In general, the program works as specified in Section 2.
5. A code review shows the following:
    a. No deviations from the style guide. **Note that each deviation will now count as two points per violation**. (10)
    b. You used `fork`/`exec` and `sigaction` to implement the specified functionality. (10)
6. I reserve the right to deduct points as I see fit.

5 Advice and Information

1. If you are going to focus on single-word commands, then the easiest approach is to use `execlp`, because you do not need to find the full path to the executable. In other words, just take the user's input and pass it to `execlp`.
2. If you are going to support command options, then `execvp` will be the easiest to use.
3. Functions that may be useful:
    a. `strtok`
    To find the different parts of the user input, e.g., if the user input "`ls -l`", `strtok` will help you to find `ls` and `-l` as separate strings.
    b. `fgets` with `stdin`
    The `fgets` function can be used to get a command from the user. However, be warned that the returned string will include the Carriage Return (i.e., '\n') at the end of the string (before the '\0'). You will need to overwrite that Carriage Return with a '\0' before using it in an `exec` call.

4. If your program is running and you cannot get it to terminate, then from another terminal you will need to execute "`ps -ef`" to obtain the PID of the program, and then use "`kill -9 PID`" to terminate it.
5. To test your ability to catch the SIGINT signal, press control-C at the prompt.
6. A test to make sure you are properly handling an `exec` error (such as the program does not exist), is to do the following:

```
prompt> jjjj
[informative error message]
prompt> exit
```

   If you are handling the error-condition properly in the child process, then you should **not** see the "prompt>" again.