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 “segfault” you shall cause a segmentation fault (on purpose) by doing the following:
      ```
      int *bomb = NULL;
      *bomb = 42;
      ```
6. If the `exec` fails in the child process, then the **child** process must display a meaningful error message and then exit (or weird things will happen), but the parent shall **not** terminate.
7. You can choose to simplify your implementation by only handling a single command with no options, 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 any command with up to 10 items in argv, such as the following commands:

- `ls -l`
- `cat /etc/passwd`
- `cal 2018`

If a user enters more than 10 items, then your code should not overflow any buffers by blindly taking more than can be handled, but instead should ignore anything more than 10.

8. In all cases, your program must be able to successfully execute commands even if spaces are entered before and/or after the command.
9. All error messages shall be sent to `stderr`.
10. After 30 seconds of inactivity, your shell shall be programmed to receive the SIGALRM signal. (See the `alarm` function). 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`**, and then exit the program with a value of 3.

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

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`**, and then exit the program with a value of 2.

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

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`**, and then continue execution where it was interrupted by the signal.

```
Good try...I don’t die that easily.
Enter 'exit' at the prompt to terminate this shell.
```

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 create a child process.
3. You are not allowed to use the obsoleted `signal` function; you must use the `sigaction` function to register your signal handlers.

3 Submission

Post proj7.tar on Sakai by the due date, which shall include shell.c and a properly configured Makefile.

4 Grading

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

1. Compiles without errors (20).
2. Compiles without warnings (10).
3. Properly configured Makefile (5)
4. Proper memory handling when `ls` and `exit` are executed, such as `valgrind` reporting no errors, and no unintended segmentation faults (10).
5. 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. (2)
   c. In general, the program works as specified in Section 2. (25)
6. A code review shows the following:
   a. No deviations from the style guide. **Note that all deviations count as two points per violation**. (10)
   b. You used `fork`/`exec` and `sigaction` to implement the specified functionality. (10)
7. 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 a`" 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. If you are **not** handling an `exec` **error** properly, then you will see the following:

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

7. It is OK to fail on those commands that do not have an executable file, such as the "cd" command, and all other commands that are handled internally by a shell.