The purpose of this project is provide experience with process manipulation and signal handlers in Unix.

**Problem Statement**
This project consists of completing a program to serve as a shell interface. That is, the program accepts user commands and then executes each command in a separate process. Unless otherwise specified, the parent process waits for the child to exit before continuing as shown in Figure 3.11 in the textbook. As is typical, the shell also will allow the child process to run concurrently as a background process by specifying the ampersand (&) at the end of the command. And finally, the shell will support a history mechanism that allows users to repeat commands previously entered.

**Project Details**
This project is organized into two parts:

1. Create a child process that executes the given command that the parent may or may not wait for using the `fork()`, `exec()`, and `wait()/-waitpid()` system calls (described in Section 3.3 of the textbook)
2. Implement a history mechanism using a signal handler (notes at the end of this assignment and covered in lecture on 1/11/2013)

Base shell code and the signal handler demonstration code from lecture written in C is available in the directory `/home/hwang/cs470/project1`. However, this project may be written in any language as long as it supports process creation and signals, and runs on csserver.

**Simple Shell**
The program in `shell.c` provides the basic operations of a command line shell. The program consists of two functions: `main()` and `setup()`. The `setup` function reads in the user's next command (up to 80 characters), and then parses it into separate tokens that are used to fill the argument vector for the command to be executed. Note that the `setup` function creates the `args` array by manipulating `inputBuffer`. It replaces all whitespace and any `&` in `inputBuffer` by the null terminator. The elements of `args` are pointers to the first letter of each word in `inputBuffer`. If an `&` is encountered, the `background` parameter is set to inform the main program. The program is terminated when the user enters `Ctrl-d` which causes the `setup` function to invoke the `exit()` system call.

The `main` function presents the user prompt and then invokes the `setup` function, which waits for the user to enter a command. When the `setup` function returns, the `args` array is suitable for passing on to

---

1 This project is based on the UNIX Shell and History Feature project published in previous editions of the course textbook: Sibershatz, Galvin, and Gagne, *Operating Systems Concepts, 7e*, John Wiley & Sons, Inc., 2005.

Revised: 01/11/2013, fixed typos in example run
the `execvp()` system call. For example, if the user enters "ls -l" at the prompt, `args[0]` is the C-style string "ls" and `args[1]` is the C-style string "-l".

Creating a Child Process
The first part of the project is to modify the `main` function so that upon the return of the `setup` function, a child process is forked and executes the command specified by the user using the `execvp()` system call, which has the following interface:

```
int execvp(char *command, char *params[]);
```

where `command` is the command to be performed and `params` stores the command line. Thus for this project, the call should be `execvp(args[0], args)`. The `background` variable determines whether the parent process is to wait for the child to exit or not.

Some things to note:
- An exec call only returns if there was an error trying to start the new program, e.g. if the command does not exist. However, if an exec call fails, its process must be terminated, otherwise it will continue to run the parent (shell) code. In particular, it will print (extra) prompts to the screen.
- Background processes still need to be waited upon in order for them to terminate properly. Otherwise, when they try to terminate, they will become zombies for the duration of the new shell execution. The best way to handle this is to take advantage the fact that when a process that is orphaned (i.e., its parent has terminated) it is adopted by the init process that automatically issues a wait for it. By having the child process do a second fork and have the grandchild process do the exec call while the child process terminates immediately. This is called the `double-fork` technique and often is used for long-lived server-type processes (e.g., web server) that fork a separate process to handle an individual client request.

Creating a History Mechanism
The second part of the project is to add a history mechanism that allows the user to access up to the 10 most recently entered commands. The user will be able to list these commands, with the least recent command listed first and the most recent command listed last, by pressing `Ctrl-\` (control-backslash), which generates the `SIGQUIT` signal.

From the history list, the user can run any of the previous 10 commands (numbered 0-9 from oldest to newest) by entering on the command line "!n" where 'n' is the number of the command. Also, the user should be able to run the most recent command again by entering "!!". Any command executed from the history list should be echoed on the screen and placed in the history list as the next command. Note that this is the actual command executed (e.g. "ls -l", not "!3" or "!!"). Here is a short example:

```
$ ./shell
COMMAND-> ps
<ps output>
COMMAND-> ls -l
<ls -l output>
COMMAND-> pwd
<pwd output>
```
COMMAND-> !!
pwd
<pwd output>
COMMAND-> !2
ls -l
<ls -l output>
COMMAND->^\$
0 ps
1 ls -l
2 pwd
3 pwd
4 ls -l
COMMAND-> !x
Error: bad history attempt
COMMAND->^\$
0 ps
1 ls -l
2 pwd
3 pwd
4 ls -l
COMMAND->^d
$

It may be assumed that history commands will be well-formed consisting of '!' followed immediately by a single character followed by a newline. I.e., you may ignore any extra characters that appear after the second character in commands starting with '!'. If the user attempts to use the history mechanism incorrectly (e.g., '!x' where x is not a digit), an error message should be displayed and the command not entered into the history list, and of course, execvp should not be called. Note this is only for history usage errors and will not prevent erroneous commands that appear to be valid from being added to the history list and attempted to be executed; handling this is beyond the scope of this project.

Some things to note:
- As noted below, if the list is displayed from within the handler, the write system call must be used, since printf is not signal safe.
- After a signal handler is called, control goes back to the location of the interruption. For this project, this likely will happen someplace in the middle of the read system call, which causes the system call to fail and return a negative value. The shell program should ignore such attempts to read, and loop back to reissue the prompt and the read call in this case so that a user can ask for the history as many times as they wish.

Assignment
(20 points) Implementation of the shell program described above. This project is to be done individually. While there are no explicit design guidelines for this course, programs should exhibit a modular or object-oriented design. Projects exhibiting particularly poor design will not earn full credit. This project may be written in any language as long as the language supports process creation and signals, and runs on cserver. Please note that the instructor will only be able to provide assistance for projects written in C/C++/Java. Provide a makefile that will make your project if it needs to be compiled.

Revised: 01/11/2013, fixed typos in example run
(10 points) Provide a high-level discussion of the program describing the functionality of the major components of the program and how they interact with each other, and a more detailed discussion for the data structures and algorithms used in the history feature portion of the program. If the program does not meet all of the project requirements, describe which parts are missing and what you think should be done to implement them.

(10 points) Provide a test plan for your project. I.e., give a list of commands that will demonstrate the features of your shell. Annotate this list with notes regarding what result is expected. The grade for this portion of the project will depend on how thorough the test plan is. Note that the test plan should cover all of the project requirements whether or not the program actually implements them.

In addition, answer the following questions:

1. What aspect of process manipulation or signal handling did you find most difficult to understand?
2. What aspect of process manipulation or signal handling did you find least difficult to understand?
3. What, if anything, would you change in your current design?
4. What, if anything, did you find interesting or surprising about process manipulation or signal handling that you did not know before doing this project?

What to Submit
Create a tarfile or zipfile containing the following items:

- The well-documented code source code for your shell. If the project is not written in C/C++/Java, put instructions on how to run the program in a comment at the beginning of the main program file.
- A makefile to make your project, if needed
- The discussion of the functional design of your project, preferably in PDF format.
- The test plan for your project and the answers to the questions above, preferably in PDF format.

Submit this tarfile or zipfile electronically by emailing it as an attachment to the instructor no later that 4:30pm on the due date.

Basic Notes About Signals
Unix uses signals to notify a process that a particular event has occurred. Signals may be either synchronous or asynchronous, and generated by hardware, software, or user input. Once a signal is generated, it is delivered to a process where it must be handled. A process receiving a signal may handle it in one of three ways:

- Ignore the signal
- Use the default signal handler
- Provide a separate signal-handling function

Each signal has a default action associated with it (e.g. for SIGQUIT, it is to terminate the process), and a few signals have permanent actions that cannot be changed (e.g. SIGKILL always terminates a process).
A demonstration program is available in file `signal.c`. Signals are defined in the header file `signal.h` (More detailed information on signals may be found by running "`man 7 signal`" on a Linux box. A signal handler simply is a function with the following interface:

```c
void handler (int signum);
```

where `x` is the signal number being handled. When a signal is received, the current instruction is interrupted and control transfers to the handler function. When the handler returns, the interrupted instruction is continued. A handler may access any global variables and execute most any code. The main restriction is that I/O must be done using the `read` and `write` system calls directly as higher-level I/O routines such as `scanf` and `printf` in C or `<<` and `>>` in C++ are not signal-safe.

To attach a custom handler to a signal, a `sigaction` structure is created with the `sa_handler` field set to the handler function. The rest of the structure must be initialized, but is otherwise not used for this project. The structure is passed to the `sigaction()` system call, which has the interface:

```c
int sigaction (int signum, const struct sigaction *act, 
               struct sigaction *oldact);
```

where `signum` is the number of the signal (e.g. `SIGQUIT`), `act` is the the structure with the new handler, and `oldact` is filled with the previous structure information if is it not `NULL`. 