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 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 1/20/2010)

The base shell code and the signal handler demonstration code described below is written in C and 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 cserver.

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 `exit()`.

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 the `execvp()` system call. For example, if the user enters "ls -l" at the prompt, `args[0]` is the C-

---

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.
style string "ls" and \texttt{args[1]} is the C-style string "-1".\\

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

\begin{verbatim}
    execvp(char *command, char *params[]);
\end{verbatim}

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

\textit{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 most recent command at the bottom of the list, by pressing \texttt{Ctrl-\backslash} (control-backslash), which generates the \texttt{SIGQUIT} signal. As noted below, if the list is displayed from within the handler, the \texttt{write} system call must be used, since \texttt{printf} is not signal safe.\\

Also note that 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 \texttt{read} system call, which causes the system call to fail and return a negative value. The shell program should loop back to reissue the prompt and the \texttt{read} call in this case so that a user can ask for the history as many times as they wish.\\

From the history list, the user can run any of the previous 10 commands by entering "! \texttt{x}" where \texttt{x} is the first letter of that command. If more than one command starts with \texttt{x}, the most recent one is executed. Also, the user should be able to run the most recent command again by entering just '!'. It may be assumed that only one space will separate '!' and the first letter, and that the letter will be followed by a newline. Likewise, '!' alone will be followed immediately by a newline if the user wants to execute the most recent command.\\

Any command executed in this manner should be echoed on the screen and placed in the history list as the next command. Note that this is the actual command executed, not "! \texttt{x}" or '!'. If the user attempts to use the history mechanism incorrectly (e.g., there is no command starting with \texttt{x}), an error message should be displayed and the command not entered into the history list. And of course, \texttt{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; doing this is beyond the scope of this project.\\

\textbf{Assignment}\\
(20 points) Implementation. This project is to be done individually. Provide a makefile that will make your project if it needs to be compiled.\\

(10 points) Provide a high-level functional analysis and design of the program describing the functionality of the major components of the program and how they interact with each other, and a more
detailed analysis and design 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
Submit items no later than 4:30pm on the due date.

Create a tarfile containing the source code (even if there is only one source file) and makefile (if needed) for the project. Submit this tarfile electronically by emailing it as an attachment to the instructor.

Submit the following items in hardcopy:

- A printout of well-documented code for your shell, preferably 2-up in landscape mode.
- The functional analysis and design of your project
- The test plan for your project and the answers to the questions above

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 \texttt{SIGQUIT}, it is to terminate the process), and a few signals have permanent actions that cannot be changed (e.g. \texttt{SIGKILL} always terminates a process).

A demonstration program is available in file \texttt{signal.c}. Signals are defined in the header file \texttt{signal.h}. A signal handler simply is a function with the following interface:

\begin{verbatim}
void handler (int x);
\end{verbatim}
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 `scanf` and `printf` 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`. 
