Homework #3: Execution Integrity

Issued: Thursday, September 21
Due: Thursday, October 12

Purpose

This assignment asks you to learn about and practice multiprocess programming, including the design and implementation of a simple Unix shell. It will support sequences, pipelines, and I/O redirection. We will omit job control (e.g., `fg` and `bg`).

Resources

Our textbook discusses much of this material. In addition, the `info` documentation for the following topics will be very useful: `bash`, `libc`, `readline`, and `history`. It is also available elsewhere. Indeed, the `libc` documentation, especially the section on "Implementing a Shell," contains code fragments for a simple shell. However, its modularity isn't very good. You should read this material, but don't just cut and paste from it. Your grade will depend on good modularity.

Grammar

A shell is an interpreter for sentences in a language. A language is best specified by a grammar. Your shell's language is specified by this grammar:

```
sequence ::=
    pipeline
    pipeline &
    pipeline ;
    pipeline & sequence
    pipeline ; sequence

pipeline ::=
    command
    command | pipeline

command ::=
    words redir
    ( sequence ) redir      # CS 552 only
    { sequence } redir      # CS 552 only

words ::=
    word
    words word

redir ::=
    ^                       # empty
    < word
    > word
    < word > word
```

Compare our grammar with the part of the `man` page for `bash` that describes legal inputs. This will convince you that we need a formal specification method for languages, and English isn't it.

Skeletal Code

A rudimentary shell is provided; it compiles, links, and runs. However, it lacks an implementation of many of the features required by this assignment. Start with this:

```
pub/hw3
```

The modules in this directory include:

- The complete *scanner* is the part of an interpreter that recognizes character sequences called "tokens" (e.g., `cd`). Tokens are delimited by whitespace. The scanner employs the `readline` and `ncurses` libraries for interactive command editing.
- A mostly complete *parser* builds a *parse tree* from the scanned tokens, according to the grammar. This is a simple recursive-descent parser. Each line read should be a complete sentence. Each kind of tree node has a `typedef` and a function to create it.
- A tree-walking *interpreter* calls very skeletal functions to execute each sentence. Since a sentence could be executed in the background, its *job* is added to the set of executing *jobs*, so other sentences can be processed. Eventually, each job terminates. A couple of builtin commands are implemented.
- A shared-library binary-only implementation of the double-ended queue, from HW#1, allows the skeletal shell to be linked into an executable, named `try`, which can be executed, but doesn't do much. Replace this library with your own implementation.
- A trivial test suite and regression tester is ready for you to add your own tests. The tester is named `run`. You might need to change the name of the program being tested. Use of this regression tester is *required*.
- A set of "suppressions" prevent the `readline` library from cluttering your `valgrind` output. See the `vg` script.

Builtin Commands

Your shell should provide several builtin commands. A builtin command should be performed in the shell's process, not a child process, unless backgrounded or as part of a pipeline or subshell. I/O redirection should work as usual.

- `exit`: Exit from the shell, after waiting for any background jobs.
- `pwd`: Print the current working directory.
- `cd <dir>`: Change the current working directory. An argument of "`-`" changes to the previous working directory. The skeleton works only for an absolute path to a directory.
- `history`: Print the previous commands. Hint: Use the function named `history_list`, which is part of the GNU History library.

Other Requirements

This material is discussed, and demonstrated, in Chapter 5 of our textbook.

- Your shell executes as a parent process. A builtin command is performed in the shell's process.
- A non-builtin command (e.g., `cat`) is executed in a child process, via `fork`/`exec`. The parent (i.e., shell) process waits for the child process to exit, before reading the next line of input.
- A command followed by an ampersand (e.g., `cat &`) is executed in the same way, but the parent reads and interprets the next line of input without blocking to wait for the child to exit. Eventually, the child will exit, and the parent will wait for it, to prevent it from becoming a "zombie."
- Commands separated by vertical bars (e.g., `cat | wc` form a pipeline. Each such command is also executed in a child process, with their `stdin` and `stdout` file descriptors initialized appropriately. Test your pipelines carefully. Consider these inputs:

```
cd /etc ; pwd
cd /etc | pwd
pwd | cat ; cd /etc | pwd
pwd & cd /etc & pwd
pwd ; cd /etc & pwd
pwd & cd /etc ; pwd
```

- A command followed by one or two redirection operators (e.g., `cat < inp > out`) is executed with its `stdin` and/or `stdout` file descriptors initialized appropriately.
- Your submission will be evaluated on `onyx`.

Development Steps

I recommend you develop your shell's features in this order:

1. simple foreground commands
2. sequences of commands
3. I/O redirection
4. background commands
5. pipelines
6. remaining builtin commands

Use the `run` regression tester to test your shell. Use `gdb` to debug your errors. Use `valgrind` to fix your memory errors.