CSC 357 Programming Assignment 6
`vssh` -- A Very Simple Shell

ISSUED:	Friday, 25 May 2007
DUE:	On or before 11:59:59PM Wednesday 13 June, via `handin` on falcon/hornet
POINTS POSSIBLE:	100
WEIGHT:	10% of total class grade
READING:	Lecture Notes Weeks 8 and 9; Stevens Chapters 9,10,15; cited `man` pages

Specification

In this assignment you are implementing a very simple shell program named "`vssh`". The `vssh` shell provides the following functionality:

- execute a program, given its name and command-line arguments
- execute a `vssh` script, provided as a command-line argument
- redirect input using '<'
- redirect output using '>'
- pipe two or more programs using '`|`'
- execute a program in the background using '`&`'
- interrupt a running program using `Control-C`
- suspend a running program using `Control-Z`
- resume a suspended program using '`%`'
- re-execute a previous command using '`!`'
- execute the following built-in commands:
  - *o* `cd` *dir*
  - *o* `exit`
  - *o* `jobs`
  - *o* `bg`
  - *o* `fg`
  - *o* `%`*job*
  - *o* `kill` *job*
  - *o* `history`
  - *o* `!` *cmd*

Basic Vssh Behavior

Like other UNIX shells, `vssh` is an interactive program that allows the user to execute other programs. When `vssh` is executed with no argument, it outputs the prompt "`%- `" and awaits user commands. Details of the commands follow.

`Vssh` can accept a single command-line argument for executing a file in batch mode. This form of execution is described further below.

`Vssh` can itself be invoked with redirected input and/or output, as in

`vssh <` *infile* > *outfile*

where it accepts input commands from *infile* and sends its standard output to *outfile*. When `vssh` accepts redirected input, it processes the commands in the given file, and then exits without prompting for any interactive commands. When `vsh` has its output redirected, it enters a command loop, but does not print a prompt. Hence, `vssh` prints a prompt only if both `stdin` and `stdout` are a terminal device.

Program Execution

`Vssh` executes programs in the same way that other UNIX shells do. The program name can be an absolute path, relative path, or single program name. If the latter, `vssh` searches the `PATH` environment variable for the program. Note that `vssh` does not recognize the tilde character as anything special in a path, i.e., '`~`' at the beginning of a path is NOT recognized as the user's home directory, just as regular character in the path name.

The program must exist and have its *execute* permission set for the current user. If the program does not exist on the current user's path, `vssh` outputs the error

> *prog*`: Command not found`

where "*prog*" is the name of the program. If the program exists but is not executable, `vssh` outputs the error

> *prog*`: Permission denied.`

If any other error occurs when the program is invoked with `exec`, `vssh` outputs what comes from `perror` for the failed call to `exec`.

As with other shells, `vssh` executes binary executables as such, or non-binary executables using the interpreter specified in the first line "`#!`" declaration, or using `vssh` if no "`#!`" appears in the first line of the program.

Vssh Command-Line Argument

`Vssh` accepts zero or one command-line argument. Without an argument, `vssh` enters a command-input loop, with the prompt character "`%-  `". With a command-line argument, `vssh` interprets the argument as a file containing `vssh` commands. After interpreting the file, `vssh exits`, without entering its command loop.

If the command-line argument does not exist as a file, `vssh` outputs the error

> *arg:* `Command not found.`

where "*arg*" is the command-line argument. `Vssh` ignores any arguments beyond the first, without printing any error message regarding the excess arguments.

Redirection

`Vssh` performs input/output redirection in the same manner as other shells. If a redirected input file does not exist, `vssh` outputs the error

> *file:* `No  such file or directory.`

where "*file*" is the name of the redirected file. If a redirected output file exists, it is overwritten by the redirected output. If a redirected output file does not exist, it is created and written. If a redirected output file is not writable, `vssh` outputs the error:

> *file:* `Permission denied.`

Piping

`Vssh` handles pipes in the same manner as other shells. If a program in a pipeline does not exist or is not executable, `vssh` outputs the appropriate error message, and the entire pipeline is terminated immediately.

Background Execution

Normally, `vssh` waits for an executing program to terminate before outputting its next prompt. If a `vssh` command line ends in '`&`', `vssh` executes the command in the background, and outputs its next prompt without waiting for the program to finish.

The command preceding the '`&`' can have input/output redirection, but cannot be a pipeline. If it is a pipeline, `vssh` outputs the error:

```
Pipelines cannot be backgrounded.
```

and does not execute anything.

Syntactically, the '`&`' must be preceded by whitespace, otherwise it is considered to be part of the command name. If any non-whitespace characters follow the '`&`', `vssh` outputs the error

```
Junk after '&'.
```

and does not execute anything.

Backgrounded jobs that remain executing concurrently with `vssh` can be listed with the `jobs` command described below.

Interrupting and Suspending Programs

When `vssh` is waiting for a program to complete execution, the program can be interrupted (and thereby terminated) by typing the `Control-C` character on the terminal. When the program is terminated, `vssh` prints its next prompt.

An executing program can be suspended by typing the `Control-Z` character. A suspended program can be resumed with the `fg` or `%` commands described below. Suspended programs can be listed using the `jobs` command, described below.

Re-Executing Previous Commands

The `history` command described below lists previously-executed `vssh` commands. Any one of the listed commands can be re-executed using the command "`!` *job*", where *job* is the number of the command listed in the history.

When `vssh` re-executes a command, it first echos the command, then performs it, then outputs the next prompt.

If the *job* number does not appear in the history list, or is not a number, `vssh` outputs the error:

job: `Event not found.`

where *job* is the value following the `!`.

Syntactically, the *job* number can follow the `!` with zero or more intervening whitespace characters.

Built-In Commands

The built-in `vssh` commands are not executable programs, but specific commands executed internally by `vssh`. The reason these commands cannot be executed as external programs is because they require some form of special processing. For example, `cd` cannot be executed as a program, because it would change the working directory of the child process, but not `vssh` itself.

None of the built-in commands can involve redirection, be part of a pipeline, or be backgrounded. That is, a built-in command must be the first and only command on the line. If any user inputs follows a syntactically legal built-in command, `vssh` outputs the error

```
Junk after built-in command.
```

and does not execute the command. If a built-in command appears anywhere in the middle of a pipeline, or as the argument of a redirection, `vssh` assumes the command is the name of an executable program or file, as appropriate. E.g., the `vssh` command

```
ls cd > bg
```

runs the `ls` program on a file or directory named "`cd`", sending the output to a file named "`bg`".

Details of each built-in command follow.

cd

The `cd` command changes the current working directory of `vssh`. All subsequent commands are relative to the changed-to path.

exit

If there are no suspended jobs, the `exit` command echos the "`exit`" command name then exits `vssh` . If there are suspended jobs, `vssh` outputs the message:

```
There are suspended jobs.
```

without exiting. If the user executes immediately executes another `exit` command, without any other intervening commands, then `vssh` terminates all suspended jobs and exits.

jobs

The `jobs` command lists all backgrounded and suspended jobs, in the format:

[1] $command_1$
[2] $command_2$
...
[n] $command_n$

where the $command_i$ are the commands that invoked the backgrounded or suspended programs.

The order of the jobs list is based on the chronological order of its "job time". For a backgrounded job initially invoked with '`&`', its job time is its invocation time. For a job initially invoked without '`&`', but subsequently suspended with `control-Z`, its job time is the time the `control-Z` was issued.

The next section on the `kill` commands describes the effect of killing on the order of the jobs list.

bg, fg, %, and kill

The `bg` command backgrounds the most recently suspended command. `fg` foregrounds the most recently backgrounded command. Foreground means having the command executing such that `vssh` is waiting for it, as if the command had been started without being backgrounded.

The `%`*n* command foregrounds the backgrounded job *n*. If *n* is not one of the job numbers listed by `jobs` or is not a number, `vssh` outputs the error

n`: No such job.`

The `kill` command takes a job number in the form "`%`*n*", or a plain integer *n*. In the "`%`*n*" case, `vssh` kills the specified job, or outputs the "`No such job.`" message if *n* is not one of the job numbers listed by `jobs` or is not an integer. When the argument to `kill` is a plain integer, `vssh` invokes "`/bin/kill` *n*" to kill the job. Any error subsequent messages are up `bin/kill`.

When a job is killed, there is a "hole" in the jobs list, if the killed job is not the last in the list. E.g., suppose the jobs list looks like this:

```
[1] progA
[2] progB
[3] protC
[4] progD
```

and then a "`kill %3`" command is issued. The next `jobs` command output looks like this:

```
[1] progA
[2] progB
[4] progD
```

If a new job is invoked in the background or suspended, any holes in the jobs list are filled. E.g., if the user invokes `progE` in the immediately preceding jobs state, the next `jobs` list looks like this:

```
[1] progA
[2] progB
[3] progE
[4] progD
```

history and !

The history command outputs a list of up to 100 most recently executed commands, in the format:

> *i* $command_i$
> *i+1* $command_{i+1}$
> ...
> *n* `history`

where *i* = 1 if *n* <= 100, or *i* = *n* - *100* otherwise.

The command `!`*k* re-executes the *kth* command from the history. If *k* is not listed in the history or is not a number, `vssh` outputs the error

```
k: Event not found.
```

Sample Execution

There will be an executable version of `vssh` on hornet in

```
~gfisher/classes/357/programs/6/testing/vssh
```

Run this program to see how it performs the commands specified above.

(The program will be available by Friday, 26 May.)

Implementation Details

The work you've done in the most recent program and lab has involved the use of `fork` and `exec` that you'll be using in the implementation of `vssh`. The examples discussed in the week 8 and 9 lecture notes show the use of other system calls you'll be using for `vssh`.

The following C library functions may be particularly useful in your implementation. You can read about these in Stevens and the man pages.

Function	**Description**
`fork` and `exec`	you've been working with these recently
`sigaction`	you've worked with this also; note that you may want to use the extended form of signal handler that takes a `sininfo` parameter, for dealing with programs that send `SIGTSTP` to `vssh`
`kill`	send a signal to a process
`waitpid`	you've worked with this in recent assignments and labs; note utility of `WNOHANG` option for use in job control
`pipe`	you've worked with this in the most recent lab
`isatty`	indicate if a file descriptor is associated with a terminal

To parse `vssh` commands, you are being provided a pre-implemented parsing program. It is available in

```
357/programs/6/parser
```

You can use the parser as-is, for performing the syntax analysis of `vssh` command lines, including the handling of syntax error messages.

The parser has a number of `.c` and `.h` files, plus a Makefile. A listing of the key `.h` file is attached to the end of this writeup. The Makefile compiles the `.o` files that you can include with your implementation of `vssh`. It also builds a testing program named `parse-cl` that parses a command line and dumps the data structure it builds.

Here are some additional points to consider for your `vssh` implementation:

- Don't forget that stdout is buffered. Be sure to `fflush()` stdout after commands complete. There may be unwritten output still in the buffer.
- Remember that signal masks are inherited over a `fork()` and `exec()`. If you block `SIGINT` while launching your pipeline stages, remember to unblock it in each child before the `exec()`. If you forget, the children will be ever-after deaf to `SIGINT`.
- Keep a list of child processes. This is the only way to know how many to wait for and which processes to kill on a signal.
- Remember that `wait()` will get interrupted by the signal handler, so be sure to check its return value to be sure a child actually exited. If you don't, you risk losing count of your children.
- Setting up pipelines involves opening a lot of file descriptors. Be careful to remember to close these file descriptors when done. If you don't, the file descriptor table can fill up and prevent you from running any more commands. After each `fork()`, the child has a copy of all file descriptors open in the parent. All unneeded ones should be closed before the `exec()`.
- Be particularly careful to close the *parent's* copy of the write-end of a pipe. The process reading from the read-end will never get an end of file from the pipe until all open descriptors to the write end are closed. If you forget to do this, pipelines like "`ls | more`" will hang forever. (Even when the `ls` terminates, the parent still has an open descriptor to the pipe.)

Deliverables

Submit a set of `.c` and `.h` files plus a `Makefile` that compiles your program into an executable named `vssh`.

Scoring Details

The testing plan file in the 357 `program/6` directory has the precise point breakdown.

Collaboration

NO collaboration is allowed on this assignment. Everyone must do their own individual work.

How to Submit the Deliverable

Submit your deliverables using the `handin` program on hornet. For Section 1 of 357, the command is

```
handin gfisher prog6-s1 ... Makefile
```

For Section 2, it's

```
handin gfisher prog6-s2 ... Makefile
```

where "..." are your program files.

Note that `handin` does not support the hand-in of directories. If your program is structured to use the supplied parser files from a `parser` subdirectory, your Makefile must create this subdirectory and move the parser files into it before compiling. Given this, it's easier just to work with the parser files in the same directory as your own files.

Listing of pipeline.h from the Parsing Package

Follows on the next page (in the paper version of this writeup).

```
/**
 * This file defines a pipeline data structure that represents a shell command
 * line.  A pipeline is a list of command-line stages, defined by the clstage
 * structure.  A stage contains the name of a command, its redirected input,
 * redirected output, argc, and argv.
 *
 * To see what a pipeline looks like for a shell command, run the parse-cl
 * program.  It will dump the pipeline structure for each command line input
 * after the "what? " prompt.
 */

#ifndef PIPELINE
#define PIPELINE

#include <stdio.h>
#include <sys/types.h>
#include "stringlist.h"

typedef struct clstage *clstage;

struct clstage {
  char *inname;                 /* input filename (or NULL for stdin) */
  char *outname;                /* output filename (NULL for stdout)  */
  int  argc;                    /* argc and argv for the child        */
  char **argv;                  /* Array for argv                     */

  clstage next;                 /* link pointer for listing in the parser */
};

typedef struct pipeline {
  char           *cline;              /* the original command line  */
  int            length;              /* length of the pipeline     */
  struct clstage *stage;              /* descriptors for the stages */
} *pipeline;


/* prototypes for pipeline.c */
extern void     print_pipeline(FILE *where, pipeline cl);
extern void     free_pipeline(pipeline cl);
extern pipeline parse_pipeline(char *line);
extern clstage  make_stage(slist l);
extern void     free_stage(clstage s);
extern void     free_stagelist(clstage s);
extern clstage  append_stage(clstage s, clstage t);
extern pipeline make_pipeline(clstage stages);
extern int      check_pipeline(pipeline pl, int lineno);

#endif
```