Overview

You are to write and test a C/C++ program that implements a simple shell whose executable is called `xssh`. This document specifies a basic `xssh`. The follow-on to this project will add a few more features and lift some simplifications assumed here. The command line looks like:

`xssh [Options] [File [Arg] ... ]`

where the square brackets ([ ]) indicate an optional word(s), and "..." indicates 0 or more words. `xssh` normally reads commands from stdin. The options are:

- `-x`: The command line after variable substitution should be displayed before the command is evaluated.
- `-d DebugLevel`: Output debug messages. DebugLevel=0 means don't output any debug messages. DebugLevel=1 should output enough information so that correct operation can be verified. Larger integers indicate more debug messages or more details.
- `File [Arg] ...`: Input is from a file instead of stdin; i.e., `File` is a shell script. If there are arguments, the strings are assigned to the shell variables $1, $2, etc. The location of `File` follows the rules described in `Search Path` below.

Search Path

The shell should search for non-built-in commands and shell scripts using these rules:

- If the command is an absolute or relative pathname P, the file P should be an executable binary (e.g., `/usr/bin/zip`) or executable script (`/usr/bin/spell`).
- Otherwise, search the directories listed in the PATH environment variable.

Non-Builtin Commands

All commands that are not listed as builtin commands below should be treated as if they were non-built-in commands.

Built-in Commands

`xssh` has the builtin (internal) commands listed below. CAVEAT: There are some differences between these commands and those in Homework 4. In the descriptions below, W1, W2, ... stands for Word1, Word2, ....

- `echo W ...`: Display the word(s) followed by a newline. Multiple spaces/tabs should be reduced to a single space.
- `quit`: Quit the shell with the exit status of the last command executed.
• **quit \#:** Quit the shell with an exit status of \$W.

• **export**: Display the exported names and their values in the same format as the bash `export` command.

• **export \# ...**: Mark the name(s) \$W ... for automatic export to the environment of subsequent commands.

• **set**: Display the contents of the variable table but exclude any environment variables.

• **set \#**: Set the value of the variable \$W to the empty string.

• **set \#1 \#2 ...**: Set value of the variable \$W1 to the concatenation of the words \$W2 ....

• **wait \#:** The shell should wait for child process \$Pid to complete. If \$Pid is -1, the shell should wait for ANY child process to complete.

• **chdir**: Change the current directory to the path given by the environment variable HOME.

• **chdir \#:** Change the current directory to \$W where \$W must be either an absolute or relative Unix pathname. The current directory should be maintained in an environment variable called PWD.

### Variable Substitution

Single-level variable substitution is always done BEFORE command evaluation. The dollar character ($) as the first character of a word signifies variable substitution; e.g., $$XY$$Z means "the value of the variable XYZ". Note that unlike real shells, an embedded dollar sign has no special meaning. *Single-level variable substitution* means that there is no recursive substitution; e.g., if the value of $$X$$ is stored as "$Y", no further substitution is done and the value is the string "$Y".

Furthermore, all undefined variables have a value of the null string. And $x$$ssh$ understands the three special shell variables $$, $$, and $!$. These variables have the same meaning as the same *Special Parameters* in sh and bash:

- $$ PID of this shell
- $$ Decimal value returned by last foreground command
- $$! PID of last background command

### Stdin/Stdout Redirection

Stdin and/or stdout can be redirected from/to a file using the following syntactic construct:

\[ C < F1 > F2 \]

which means to redirect command C's stdin from the file F1 and stdout to the file F2. Both redirections are optional.
Background Commands

A command that should be run in the background is followed by the ampersand character (&):

```
C &
```

After each foreground command has terminated, the user should be notified of any background processes that have recently terminated by displaying the PID and the command (after variable substitution). This feature can be combined with redirection (e.g., "C < F1 > F2 ").

Other Features

Here are the other features of `xssh`:

a) Each word (token) is separated by white space.

b) Multiple spaces/tabs are reduced to a single space during the substitution and line scanning phase.

c) The command line prompt should be the three character sequence `'' ' (i.e., >, >, space).

d) The # character signifies the beginning of a comment. All other characters following and including the # should be ignored during interpretation.

e) Blank lines should be ignored.

Note that there is simple variable substitution, but there is no filename substitution nor command substitution. In addition to fork(2), waitpid(2), execvp(2), sh(1), exit(3), and fgets(3), see getenv(3), putenv(3), chdir(2), dup2(2), open(2), read(2), write(2), and close(2).

Implementation Notes

In this implementation, you can assume that the system will be small and therefore simple data structures are appropriate (i.e., there is no need at this time for sophisticated data structures). It is up to you to determine what you will need, but remember that simplicity will be a virtue in this assignment. Furthermore, for this part of Project A, you can assume that an error should terminate your shell with an error message if that will simplify your code.

The main processing loop of the `xssh` interpreter might look something like this:

```c
Initialize;
Process command-line args;
while ( (line=getcmd()) != EOF ) {  // prompt&get cmd until EOF
   (nwords, word[]) = parse(line);  // word[i] points to ith word
   xword[] = do_subst(word[]);      // do var substitution
   quit = eval_builtin(nwords,xword[]) if (is_builtin(xword[0]));
   status = eval_nonbuiltin(nwords,xword[]) if (!is_builtin(xword[0]));
} until (quit);
Cleanup;
```
Additional Guidelines

- Pay close attention to the implementation of "-d 1" so that the output is informative but not overly verbose.

- Code readability is of the utmost importance. The Web page will contain a summary of coding guidelines that you should follow in spirit. I am not rigid about these guidelines, but unreadable code will be penalized.

- All system calls that should never fail (e.g., fork(2)) should be wrapped so that any fatal errors will cause an error message to be displayed followed by an exit. If you can recover from a system call error (e.g., execvp(2)), output an error message and continue. I will not examine any solutions that do not follow this rule. By convention, the wrapped system call name will be the same as the actual system call name except the first character should be capitalized (e.g., Fork is the wrapped version of fork).

What to Submit by 2400 Hours Wednesday, Oct. 11

Electronically submit the following PLAIN TEXT files by 2400 Hours Wednesday, Oct. 11 (instructions follow):

1) A listing of your most impressive working test case.

2) The output of your interpreter for the above test case.

3) The SOURCE FILES for your latest version of the xsh interpreter. NOTE: I do NOT want object code or executables. As a minimum, this code should contain the control structure for handling all built-in commands with stub code for each evaluation function. The stub code should display a message that says that the function was entered.

There will be only three grades assigned: 0, 5 or 10 points.

**Oct. 11 Electronic Submission:** The end result should be that you mail to kenw@arl.wustl.edu a single shar (shell archive) file containing your files. Do NOT submit object code or binaries. The following commands will create a shar file named A.shar containing the required files and then send mail to me:

```
shar mytest.out ... source files ... > A.shar
mail -s A.shar kenw@arl.wustl.edu < A.shar # mail is usually in /bin
```

If you prefer, send the shar file as an attachment and use whatever mailer you are comfortable with. **NOTE:** The shar file should be relatively small (try ls -l) and make sure it is not more than a few hundred thousand bytes.

What to Submit by Oct. 18 and 19

The CSE422S Web page contains a link to the documentation template. You should complete the template and submit it in both hardcopy AND electronic form. Submit the completed documentation template AND a listing of the source code. The electronic submission (described below) should include the completed documentation template, the source code, the Makefile, test scripts, and test output. The electronic copy is due by 2400 hours, Wednesday Oct. 18. The hardcopy should be submitted to my office by 1800 hours (6 PM) Thursday, Oct. 19. This submission is worth 100 points.
Words of Caution

Here are some observations from my years of experience with projects like this:

- **The documentation template is non-trivial.** Do not expect to complete it in less than an hour. Furthermore, working code but no documentation is almost useless. So, don’t forget to fill out the documentation template.
- Trivial bugs can consume tens of hours of time. Yes, I said tens of hours, not just hours. You need to start small, test the control structure, and incrementally add features on top of what seems like rock solid code.
- Keep different versions of your "rock solid code versions" so that you can rollback to and recover from a stable version. This also helps if you mistakenly delete your latest source code!!!
- Don’t ignore error messages and think they will disappear on their own. They don’t. They just come back and bite you when you least want to be bitten.
- Try to understand the origins of your bugs rather than always doing trial and error changes. (Some trial and error may be appropriate in small test cases.)
- If the approach you are taking seems like it will be a nightmare to implement, then don’t implement it. Find a better way or better understand the system calls you are trying to use.
- The more lines of code you have, the more chances for bugs to appear.
- Have a plan. Don’t try to do everything at once.

Electronic Submission

The end result should be that you mail to kenw@arl.wustl.edu a single *shar* (shell archive) file containing your files. Do **NOT** submit object code or executables. The following commands will create a shar file named A.shar containing the files xssh.c and other files and then send mail to me:

```sh
shar README Makefile xssh.c ... other files ... > A.shar
mail -s A.shar kenw@arl.wustl.edu < A.shar  # mail is usually in /bin
```

The README file is the completed documentation template. If you prefer, send the shar file as an attachment and use whatever mailer you are comfortable with. **NOTE:** The shar file should be relatively small (try `ls -l`) and make sure it is not more than a few hundred thousand bytes.

Late Policy

There is no late submission date for the first deadline. The final submission can be one week late for a 20 point penalty. But I do not recommend taking this option unless you are extremely far behind. Look at the grading form to see the potential impact of bugs before electing this option. Note that you should submit something even if the final version still has bugs.
Extra Credit (20 Points)

Below is a set of features that can be implemented and documented for extra credit.

-c Command Line Flag

The optional command line flag `-c String` means that the input is the string `String` instead of stdin. Furthermore, it overrides any other input option including the `File` option.

The "", " Connector and Multiple Connectors on One Line

Simple commands can be tied together on the same line using the following connectors ";" (semicolon) and "&" (ampersand), ',' (comma). These connectors have the following meaning (C is a command with perhaps redirection):

C ;  Sequential execution (C must terminate before starting the next command).
C &  Parallel execution (Run C in the background, and proceed to the next command.)
C ,  Sequential background execution (Run C in the background, but if the following command is a background command, do not proceed to the next command until C terminates. But the next command can proceed if it is a foreground command.)

Note: Because of the introduction of the semicolon connector, processing will be simplified if you convert the newline character on each input line to a semicolon before processing each input line.

The connectors ";", "&" and "," can be mixed together on one line and have left-to-right, equal precedence. An example is:

```
C1 ; C2 , C3 , C4 & C5 ; C6 &
```

which means:

- C2 can begin in the background after C1 terminates.
- C3 can begin in the background after C2 terminates.
- C4 can begin in the background after C3 terminates.
- C5 can begin in the foreground after C1 terminates.
- C6 can begin in the background after C5 terminates.

Note that the connectors define an ordering. Semicolon alone defines a total ordering in the foreground. Comma alone defines a total ordering in the background. Ampersand alone defines no ordering (parallel execution).

Personal Feature

Design, implement, test and document a feature of your choice.

Documentation

Full extra credit can only be obtained if the features are well tested and documented. The documentation template contains a section at the end for documenting the extra credit features.