

The xssh Shell

Overview

Project A will ask you to implement `xssh`, an extremely simple shell. It is a *toy shell* since we will assume that the program only operates correctly when given correct input; the behavior of `xssh` when given incorrect input is undefined. It is an *incomplete shell* since it is missing many features included in other shells. Some of the missing features are briefly described at the end of this document. It is a *pedagogic shell* since it produces output that expose the workings of the shell and contains many of the fundamental features in common shells. 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 is displayed before the command is evaluated.
- **-d DebugLevel**: Output debug messages. DebugLevel=0 means don't output any debug messages. DebugLevel=1 outputs 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.

External Versus Internal Commands

An *internal* or *built-in* command is one that the shell executes itself instead of running another program (via `fork/exec`). All commands that are not listed as internal commands below are treated as if they were *external* or *non-built-in* commands.

Search Path

The shell searches for external commands and shell scripts using these rules:

- If the command name 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.

Internal (Built-In) Commands

`xssh` has the built-in (internal) commands listed below. **CAVEAT**: There are some differences between these commands and those in Homework 3. In the descriptions below, W1, W2, ... stands for Word1, Word2,

- **echo W ...**: Display the word(s) followed by a newline. Multiple spaces/tabs are reduced to a single space.
- **quit**: Quit the shell with the exit status of the last command executed.

- `quit W`: Quit the shell with an exit status of *W*.
- `sleep W`: Sleep for *W* seconds.
- `setenv`: Display the environment variable names and their values in the same format as the bash `export` command.
- `setenv W`: Export the name *W* to the environment and set its value to the empty string.
- `unsetenv W`: Remove *W* from the environment.
- `setenv W1 W2 ...`: Set value of the environment variable *W1* to the concatenation of the words *W2 ...*
- `set`: Display all local variables and their values.
- `set W`: Set the value of the local variable *W* to the empty string.
- `set W1 W2 ...`: Set value of the local variable *W1* to the concatenation of the words *W2 ...*
- `wait W`: The shell waits for child process *W*, a *Pid*, to complete. If *W* is -1, the shell waits for ANY child process to complete.
- `chdir`: Change the current directory to the path given by the environment variable `HOME`.
- `chdir W`: Change the current directory to *W* where *W* must be either an absolute or relative Unix pathname. The current directory is maintained in an environment variable called `PWD`.

Variable Substitution

Shell variables include environment variables and local variables. 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., `XYZ` means "the value of the shell variable `XY$Z`". Note that unlike real shells, an embedded dollar sign has no special meaning. *Single-level variable substitution* means that there is no recursive substitution; e.g., if the value of `$X` is stored as "`$Y`", no further substitution is done and the value is the string "`$Y`".

When looking up the value of a shell variable, the local variable table is searched first. If the variable is not found in the local variable table, the environment variables are searched. If the shell variable is not found, the shell variable is undefined and has a value of the null string.

And `xssh` 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 is 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 &").

Pipelines

A foreground pipeline has the same syntax as in other shells:

```
C1 | C2 | ... | Cn
```

In a sense, a single command C is a degenerate or 1-process pipeline. Note that an n-process pipeline is just a 1-process pipeline followed by the character | and then a pipeline with n-1 processes. A background pipeline is a foreground pipeline followed by the ampersand character &:

```
C1 | C2 | ... | Cn &
```

Note that the commands above can have arguments and stdin/stdout redirection in the usual way. For example,

```
C1 < Infile | C2 | ... | Cn > Outfile &
```

Terminal-Generated Signals (ctrl-c and ctrl-z)

When the user presses certain keys, the terminal driver will alert the user by generating signals for the foreground process group. `xxsh` handles two signals related to job control. Job control refers to the ability of an interactive user to move processes back and forth between the foreground and background.

The terminal interrupt character (normally ctrl-c) generates the interrupt signal (SIGINT). `xxsh` terminates the foreground process(es) and returns to the command line prompt. However, it should not terminate background processes or `xxsh`.

The terminal stop character (normally ctrl-z) generates the stop signal (SIGTSTP). `xxsh` stops the foreground process(es) (puts them to sleep) and return to the command line prompt. However, it does not stop background processes or `xxsh`. Every pipeline is considered a job and is assigned a unique non-negative job number. The stopped processes can be continued by entering the `bg` command which assigns the pipeline a unique job number and puts the processes in the background where they continue running. The processes in a pipeline can be brought back to the foreground through the `fg %n` command where `n` is the job number.

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 is the three character sequence '>>' (i.e., >, >, space).
- d) The `#` character signifies the beginning of a comment. All other characters following and including the `#` is ignored during interpretation.
- e) Blank lines are ignored.

Note that there is simple variable substitution, but there is no filename substitution nor command substitution.

Missing Features

Examples of missing features include:

- **Control Structures:** There are no branching structures (e.g., `if`, `while`, subroutines) that are useful in a scripting language. A primitive form could be added because the external command `test` already allows the user to set the exit status according to the results of a condition.
- **Interactive Features:** Interactive shells have features that allow the user to be more productive. These features include a history mechanism and command editing. However, these are straightforward to add to `xssh`.
- **Filename Expansion (Substitution):** Other shells recognize metacharacters that have special meaning. Examples are tilde (`~`) which expands to `$HOME`, asterisk (`*`) which matches any string of characters, question mark (`?`) which matches any single character, square brackets (`[...]`) which matches any character listed, and negated square brackets (`[! ...]`) which matches any character *not* listed. The existence of metacharacters also implies the need for a quoting mechanism to remove the special meaning of these symbols when they should not have their special meaning.
- **Command Substitution:** Other shells allow the user to capture the `stdout` output of command execution.