Due Date: Friday, Oct 30, 11:59pm (Late days may be used.)

To give everybody a chance to test others’ plug-ins, plugins are due: Thursday, Oct 29, 11:59pm with no extensions.

This project can be done in groups of 2 students.

1 Introduction

This assignment introduces you to the principles of process management and job control in a Unix-like operating system. In addition, the assignment will give you insights into the design and use of extensible systems.

This is an open-ended assignment. Rather than pre-defining some required functionality, we encourage you to define the scope of this project yourself. We will however set minimum requirements, and, separately, provide a rough idea of what we expect most students to accomplish.

2 Base Functionality

A shell receives line-by-line input from a terminal. If the user inputs a built-in command, the shell will execute this command. Otherwise, the shell will interpret the input as the name of a program to be executed, along with arguments to be passed to it. In this case, the shell will fork a new child process and execute the program in the context of the child. Normally, the shell will wait for a command to complete before reading the next command from the user. If the user appends an ampersand ‘&’ to a command, the command is started in the background and the shell will return to the prompt immediately.

The shell provides *job control*. A user may interrupt foreground jobs, send foreground jobs into the background, and vice versa. At a given point in time, a shell may run zero or more background jobs and zero or one foreground jobs. If there is a foreground job, the shell waits for it to complete before printing another prompt and reading the next command. In addition, the shell informs the user about status changes of the jobs it manages. For instance, jobs may exit, or terminate due to a signal, or be stopped for several reasons.

At a minimum, we expect that your shell has the ability to start foreground and background jobs and implements the built-in commands ‘jobs,’ ‘fg,’ ‘bg,’ ‘kill,’ and ‘stop.’ The semantics of these commands should match the semantics of the same-named commands in bash or tcsh. The ability to correctly respond to `^C` (SIGINT) and `^Z` (SIGTSTP) is expected, as are informative messages about the status of the children managed. Like bash or tcsh, you should use consecutively numbered small integers to enumerate your jobs.

For the minimum functionality, the shell need not support pipes (`|`), I/O redirection (< > >>), nor the ability to run programs that require exclusive access to the terminal (e.g., vim).

We expect most students to implement pipes, I/O redirection, and managing the controlling terminal to ensure that jobs that require exclusive access to the terminal obtain such access. Beyond that, esh's extensibility, described in Section 6 should allow for plenty of creative freedom.

3 Strategy

You will need to use `fork()`, a variant of `exec*()`, and the `waitpid()` system calls.

3.1 Signal Handling

You will need to catch SIGCHLD to learn about when the shell's child processes change status. Since child processes execute concurrently with respect to the parent shell, it is impossible to predict when a child will exit (or terminate with a signal), and thus it is impossible to predict when this signal will arrive. In the worst case, a child may have terminated by the time the parent() returns from fork()!

You will need to block the signal in those sections of your code where you access data structures that are also needed by the handler that is executed when this signal arrives. For example, consider the data structure used to maintain the current set of jobs. A new job is added after a child process has been forked; a job may be removed when SIGCHLD is received. To avoid a situation where the job has not yet been added when SIGCHLD arrives, or - worse - a situation in which SIGCHLD arrives *while* the shell is adding the job, the parent should block SIGCHLD until after it completed adding the job to the list. If the SIGCHLD is delivered to the shell while the shell blocks this signal, it is marked pending and will be received as soon as the shell unblocks this signal.

Use sigprocmask(2) to block and unblock signals. To set up signal handlers, use the sigaction(2) system call. Set sa_flags to SA_RESTART. The mask of blocked signals is inherited when fork() is called. Consequently, the child will need to unblock any signals the parent blocked before calling fork().

3.2 Process Groups

Each process in Unix is part of a group. Each process group has a leader. To create a new group with itself as the leader, a process simply calls setpgid(0, 0). The id of a process group is the process id of the leader. Child processes inherit the process group of their parent process initially. They can then form their own group if desired, or their parent process can place them into a different process group via setpgid().

Process groups are treated as an ensemble for the purpose of signal delivery and when waiting for processes. Specifically, the kill(2), killpg(2), and waitpid(2) system calls support the naming of process groups[1]. In addition, process groups are used to manage access to the terminal, as described next.

3.3 Managing Access To The Terminal

Running multiple processes on the same terminal creates a sharing issue: if multiple processes attempt to read from the terminal, which process should receive the input? Similarly, some programs - such as vi - output to the terminal in a way that does not allow them to share the terminal with others.

To solve this problem, Unix introduced the concept of a foreground process group. Each terminal maintains such a group. If a process in a process group that is not the foreground process group attempts to perform an operation that would require exclusive access, it is sent a signal: SIGTTOU or SIGTTIN, depending on whether the use was for output or input. The default action is to suspend the process. In this case, the parent can learn about this status change by calling waitpid(). WIFSTOPPED(status) will be true in this case. To allow this process to continue, its process group must be made the foreground process group of the controlling terminal via tcsetpgrp(), and then the process must be sent a SIGCONT signal.

Signals that are sent as a result of user input, such as SIGINT or SIGTSTP, are also sent to a terminal's foreground process group.

Reduced functionality: If you do not implement access to the terminal for programs that require such exclusive access, you may use the following simplified technique: create a new process group for each job, but keep the shell's process group as the foreground process group as far as the terminal is concerned. (This would happen by default if you do not call tcsetpgrp() at all.) Signals such as SIGINT or SIGTSTP are then delived to the shell. You can catch them and forward them (via 'killpg(2)') to the current foreground job. If you do implement sharing of the terminal, do not use this technique.

3.4 Pipes and I/O Redirection

To implement pipes, use the pipe(2) system call. A pipe must be set up by the parent shell process before a child is forked. Forking a child will inherit the file descriptors that are part of the pipe(). The child must then redirect its stdout/stdin file descriptor to the pipe's input or output end as needed using the dup2(2) system call.

Note that all processes that are part of a pipeline are children of the shell, e.g., if a user runs `a | b` then the process executing `b` is *not* a child process of the process executing

[1]Note the idiosynchracies of the API: kill(-pid, sig) does the same as killpg(pid, sig). Make sure to use the correct call.

the program `a`.

Generally, a pipeline of commands is considered one job. All processes that form part of a pipeline should thus be part of the same process group.

Although the parent shell process creates the pipe, it will not actually write to it or read from it. Make sure that the parent shell process closes the file descriptors referring to the pipe after the child was forked in order to avoid leaking file descriptors. Closing a file descriptor affects only the current process's access to the underlying object. When the parent shell closes the file descriptor referring to the pipe it created, the child processes will still be able to access the pipe's ends. This is true for file descriptors in general. Each file descriptor represents a reference to an underlying object. The actual object (such as a pipe or file) is closed only when the last process who has a file descriptor referring to the object closes it.

Additional information can be found in the GNU C library manual, available at `http://www.gnu.org/s/libc/manual/html_node/index.html`. Read, in particular, the sections on Signal Handling and Job Control.

4 Provided Code

The provided code is in the directory ˜cs3214/esh-20091010. If updates or bug fixes are required, they will be announced on the forum.

The code contains a command line parser that implements the following grammar:

```
cmd_line : cmd_list

cmd_list :
         | pipeline
         | cmd_list ';'
         | cmd_list '&'
         | cmd_list ';' pipeline
         | cmd_list '&' pipeline

pipeline : command
         | pipeline '|' command

command : WORD
        | input
        | output
        | command WORD
        | command input
        | command output

input : '<' WORD

output : '>' WORD
```

```
        | '>>' WORD
```

Look at the provided esh.c main function to see how to invoke the parser. If a command line is semantically correct, the parser code will create a `esh_command_line` data structure, which refers to a list of `esh_pipeline` structures. Each `esh_pipeline` corresponds to a job. It may consist of one or more individual commands that form a pipeline. Each command is represented as a `esh_command` structure. Study the definitions of these structures.

By default, the provided code will read a line, parse it, and dump the parsed command line to stdout.

5 Testing

Since we do not describe what functionality to implement, it is up to you to develop a testing strategy.

We will provide a driver that can help you automate your testing scenarios, as well as some sample client programs.

6 Plug-Ins

It is often impossible to anticipate the future uses and needs of a system or application. Extensible architectures address this problem by allowing the loading of plug-ins that provide additional functionality or enhance built-in functionality.

When started with the ’-p dir’ flag, ’esh’ will dynamically load shared libraries contained in the directory ’dir.’ Multiple -p flags may be provided. Each shared library must define a strong global symbol named `esh_module`, which shall refer to an instance of `struct esh_plugin`. This struct contains information about the plug-in, including a set of function pointers to invoke the plug-in’s functionality.

Multiple plug-ins may be loaded; a plug-in may specify its rank relative to others. Your shell should invoke the plug-ins’ functions in increasing rank order. If plug-ins share the same rank, their execution order is not defined. Some functionality (e.g., built-ins) requires that invocation stop if a plug-in provides this functionality.

Here are some ideas for plug-ins:

- Change current directory (cd)
- Glob expansion (e.g., *.c)
- Setting and unsetting environment variables

- Timing commands: ”time” or time-outs.
- Aliases
- Shell variables
- pushd, popd, etc.
- Command-line history (perhaps using’s GNU History library)
- Backquote substitution
- Smart command-line completion
- Embedding applications: scripting languages, web servers, etc.

A side-note on Unix philosophy - in general, Unix implements functionality using many small programs and utilities. As such, built-in commands are often only those that must be implemented within the shell, such as cd. Although the plug-ins I suggested above are all of the kind that must be implemented within the shell, don’t feel limited by this criterion.

You will note that the functions to read from the terminal and to parse the command line are invoked indirectly as function pointers that are part of `esh_shell`. Advanced plug-ins may replace those if desired.

7 Honor Code

You will receive credit for every plug-in you write, and for every plug-in written by others which your shell can successfully load and run. You should publish plug-ins you have developed on the forum.

It is ok to sit together and debug a situation that arises if a plug-in written by one group does not run successfully in another group’s shell.

However, *you may not share any code - electronically or otherwise* - for the shell or a plug-in - across groups. To allow others access to your plug-ins, copy the .so files, and only the .so files, to `/web/people/<` *yoursloid* `>/esh-plugins` where `yoursloid` is your SLO id. This directory is accessible to all students. In addition, provide a description of the plugin.

In addition, note that the code contained in the plug-ins you load will run with the full privileges of the user executing the shell. In practice, this setup requires that you trust the provider of the plug-in. The “Acceptable Use of Information Systems” policy, published at `http://www.vt.edu/about/acceptable-use.html`, applies. If you are in doubt whether a plug-in you’ve written would violate this policy, please ask first.

8 Grading

Coding Style. Your coding style should match the style of the provided code. You should follow proper coding conventions with respect to documentation, naming, and scoping.

You must check the return values of all system calls and library functions, except for close(2), closedir(3), and malloc(3). (Production code would need to check for those as well; this is a simplification for this project.)

Submission. You should submit a design document as an ASCII document. Describe the functionality you implemented, the design you used, and describe how you tested it.

Include your test cases, and a script or scenario to run them. *The TA will assign credit only for the functionality for which test cases exist.*

You should submit a .tar.gz file of your ′src′ directory, which contains a Makefile. Please use the submit.pl script or web page and submit as ′p3′. Only one group member need submit. List both group members in your README file.

We expect to award about 30-40 points for the minimum functionality, about 80-100 points for the functionality we expect most students to implement, and between 5-25 points per plug-in you've developed, depending on its complexity. Running a plug-in others have written will give you 2 points per plug-in, with a to-be-determined maximum. To provide an incentive to help others run your plug-ins, we'll also award 2 points for each group that can run your plug-in, with a to-be-determined maximum.

9 Final Note

I believe there are 2 unique aspects about this project. First, the “specification” for this project is intentionally open. I believe this setup mirrors the situation you will commonly encounter in your career in which the requirements of a project are not set in stone at the beginning of the project, but evolve as you develop and test your work.

Second, I encourage you to interact with each other by running each other's plug-ins. This will work only if you actively publish your plug-ins and seek out and test what others have published, and do so in time before the deadline. My goal in choosing this approach is to encourage the class to work together, while at the same time ensuring that each student masters the material individually.

Good Luck!