Project 1 Design Document

Craig Collins Mat Fukano Ryan Gliever
Leland Miller

April 2014

Abstract

This document describes our teams design for a simple MINIX shell written in C. Our design was worked on together, but each member coded a separate shell.

Overview

At the highest level our shell is a loop that takes user input, then runs commands that the user typed in. For each command in the loop our shell first scans through the arguments to find special feature symbols such as `&` for running a command in the background or `>` for redirection, and then copies the arguments besides the special symbols into a new array. This new array is passed into `execvp()` for execution in the child process.

This loop can be contained in a function separate from the code that executes the commands. It should be able to take an array of arguments, which is provided to us in the template, and call the function that executes commands. The function that executes the commands should take the command, a file to redirect output into, a file to redirect input into, and a whether or not the process should be run in the background or not. This leaves us with two major functions:

```
void strparse (char **args);
void executecmd (char **args, char* output, char* input, bool bg);
```

If redirection is not desired, output or input should be set to null. `strparse()` will also look for special cases such as exit and deal with them accordingly. In other words, `executecmd()` will only be responsible for forking and executing system commands.

Features

Exit

When the “exit” command is executed, the shell closes. This is done by using `strcmp()` to compare the first argument to string “exit” and using system call `exit()` to quit the program. This checking will be done in the `strparse()` function.

Commands (w/o arguments)

When a command with no arguments is read, we will `fork()` into a new child process. In the new process we will run `execvp()` with the command. `execvp()` transfers control of the new process to the new program while the main process is in a loop `wait()`ing for the child to finish.

Commands (w/ possible arguments)

Commands with arguments will be handled similarly to commands without arguments. We will `fork()` into a new child process, and run `execvp()` with the command, and `execvp()` handles the command arguments and passes them to the new program to be executed. The parent process still `wait()`s for the child process to finish.

Redirection

Output:

Output of a command that is to be redirected to a specified file will be done with `freopen()`. We will pass in the specified file name, mode “w”, and `stdout`. Then, we will run `execvp()` on the command, and the output will be redirected to the specified file.

Input:

If input to a command is redirected from a file, we would use `freopen()`. We will use the “r” mode which will read in the contents of the file specified. `stdin` will be the stream

Background Processes:

The only real difference in handling background processes is that the main thread should not wait for the child to finish, and instead the parent process (the shell) should just continue accepting input. The only special consideration is the killing of zombie processes.

We decided that dealing with the zombie processes will be done while we wait for the foreground processes to finish. This seemed more efficient and cleaner than using signal handlers. This way of handling zombie processes allows us to decide when our program will “clean” up the zombie processes (by waiting for them), instead of the processes interrupting our shell program. In order to implement this, when a foreground process is started we will have a loop that waits for any process, it will only break this loop is the process id returned is that of our current foreground process.

Summary

All in all this project has a relatively simple structure. The following is a general overview of our project in pseudo-code (note that `args` and `new_args` are arrays of parsed strings):

```
executecmd (new_args, output, input, bg)
    fork process
    if parent
        if not bg
            loop until wait returns foreground pid
    if child
        set redirection (using freopen)
        execvp

strparse (args)
    new_args, output, input, bg
    for arg in args
        if arg is a shell command
            set needed flag
        else
            push arg to new_args
    executecmd new_arg output input bg

main()
    while true
        get input
        args = parse input
        strparse (args)
```

Error handling will have to be done after the `fork()` and the `execvp()` calls. Other information regarding specific projects and how to build them will be included with the individual projects.