Bill Daws

CIS 3207

Professor Eugene Kwatny

Project 2 Report

In this report I will discuss the design and testing processes of my project.

Project Description:

I wrote myshell in C using emacs, gcc, bash, and Ubuntu as my tools.

I was given a skeleton and some files describing what a shell is and how it works.

I was tasked with, well, making a shell that works. To expand on this, making a shell

that is capable of the following:

\* execution of built-in commands and external programs (like emacs),

where the commands and programs are determined by the user's input.

\* background process execution

\* redirection including piping

\* accepting batchfiles as input

It was suggested that I refer to Dawei's slides from lab, the advice of Dawei,

Graeme, and Professor Kwatny, and the man pages for system calls like dup2() and

the exec() family.

Generally, it was recommended to complete steps in order of complexity such that

simpler parts, like supporting commands like echo and dir, were completed first and

complicated parts like piping were completed after.

Design: Solutions, Approaches, and Methods

After asking Professor Kwatny about this report in class, I will try not to

spend too much time on discussing the design since the pseudocode and readme

should take up the majority of that.

However, since the pseudocode may not accurately reflect my project as it is

now I will briefly discuss the structure of myshell.c.

First, in main, we initiate the user-reading loop and include the necessary code

if the user invokes the shell with a batchfile. We use the user-reading loop to

invoke myshell with batchfiles.

Either loop will use reading functions, parse(), and redirection commands to

allow execute() to properly execute built-in functions and programs.

Essentially, a graphical representation of my code would look like this:

batchfile loop <----------------------|

\ |

\ |

v |

read => parse ==> execute() --------------------

^ (redirect file descriptors |

/ if needed) |

/ |

main ----> user loop <-------------------------------------------------|

Where execute will fork and execute built-in functions and programs as the user

requests. execute() can lead to either main loop, it just depends on what command(s)

it is passed.

Every function that isn't mentioned in the above document is either a built-in

or is an assistant function. (e.g. background(char \*\* command) is a simple program

that just helps execute() do what it needs to; redirect\_in(char \*\* command) does the

same; my\_cd(char \*\* command) is the implementation of cd necessary for execute

to support changing directory; etc.)

The comments provided in my source code elaborate on the methods my functions employ

and the reasoning behind such methods.

Tests and Results:

Regarding testing, the simplest way to describe my methods was to just type commands

into myshell and see if the actual results matched the expected results.

For a more rigorously defined process, I began by testing the simplest built-in

function: my\_quit.

To do this, I ran myshell, and then if typing "quit" into the shell worked, then I

would be returned to the bash shell, and that would be a good sign.

I then applied this process to every other built-in function. I would know if the

command worked properly if the results of myshell running the command mirrored the

results of the bash shell running the command or a command that is analogous.

(For example, "help" in myshell will not be the same as "help" in bash, but "help"

in myshell is equivalent to "more readme" in bash.)

To test if myshell could run external processes, I did the same for internal processes.

For example, if myshell could start emacs and then make and save changes to files, I knew

it was working properly. Additionally, myshell could start gcc and compile its own source

code.

This general process also held true for background execution and arguments to externals,

so I knew it worked.

To test batchfiles, I wrote a batchfile (that I might have even accidentally included in

my original submission--sorry) and tried to see if the execution of such a batchfile would

result in something that resembles if I were to execute each command manually.

I encountered a few problematic bugs during the development of myshell, and the most

important of them would easily be the dreaded segfault. Myshell segfaulted for several

reasons throughout development, but some of the most important segfaults were caused by

inappropriate redirection and inappropriate memory allocation. I found that the segfaults

mainly came from piping, where I was closing the pipe too early in the function for the child

processes to properly inherit the correct descriptors; and again from my batchfile handling,

where I gave no adequate way to parse the lines of the batchfile if said batchfile did not

end in a newline character.

I fixed the piping issue by simply closing the pipe later in the function.

I fixed the batchfile issue by specifying that the user should end the batchfile in a newline

character. While this isn't the best way to fix a bug in the code, I figured it was better

than the solution I originally envisioned, which was to save the last syntactical command

and replace it (which would require opening the batchfile for writing) with an implanted

null- or newline character. It's obviously not desirable to replace contents of the file

if the user did not explicitly state a desire for this. Additionally, since EOF can occur at

any moment, and myshell can run executables, it is impossible to programmatically recognize

a syntactical command before trying to execute it.

Another critical bug I encountered was (embarrassingly) forkbombing myself. This happened

during my batchfile handling implementation because I didn't properly give the loop a way to

terminate. So eventually I wound up creating background processes until my computer crashed.

The most important lesson here is to be careful when dealing with fork(). (And to save my work

in open programs before trying out code that deals with fork().)

Conclusions and Thoughts:

This is easily the most challenging assignment I've ever had to do. If it wasn't for the week

extension, I would be submitting a wildly different project. I have learned to more closely

monitor my due dates and to start as early as possible. Additionally, I learned a lot about

general code development to create a user-friendly experience, multiprocessing and I/O, which

are obviously the purposes of this project. I feel like I am walking away from this project

as a completely different (more experienced and mature) programmer.