ezsh Technical Manual

0. Content

1. Introduction

> ezsh is an introductory, user-friendly, and interactive shell for new and inexperienced users to the general Linux environment. The primary objective of ezsh is to help reduce the steep learning curve that is present in the most shells (for example, Bash), all the while allowing the users to interact with a new environment.

1.1 Project Overview

> This project aims to provide users with an interactive and intuitive shell by introducing a fully functional and familiar feeling GUI. The user is provided with this familiarity through an Explorer, which is like a terminal-based “My Computer” or “Finder”, and a Prompt system in the prompt akin to bookmarks, which allows users to freely jump around the file system with ease. We believe such additions to the shell environment will help the user better understand how the shell can be used. A GUI as such has been achieved by making use of the ncurses library and tmux, which allows separate programs to co-exist in a single user space.

1.2 Glossary

> tmux: a terminal multiplexer that enables a number of terminals to be created, accessed, and controlled from a single screen.

> ncurses: a programming library providing an application programming interface (API) that allows the programmer to write text-based user interfaces in a terminal-independent manner.

> readline: a software library that provides line-editing and history capabilities for interactive

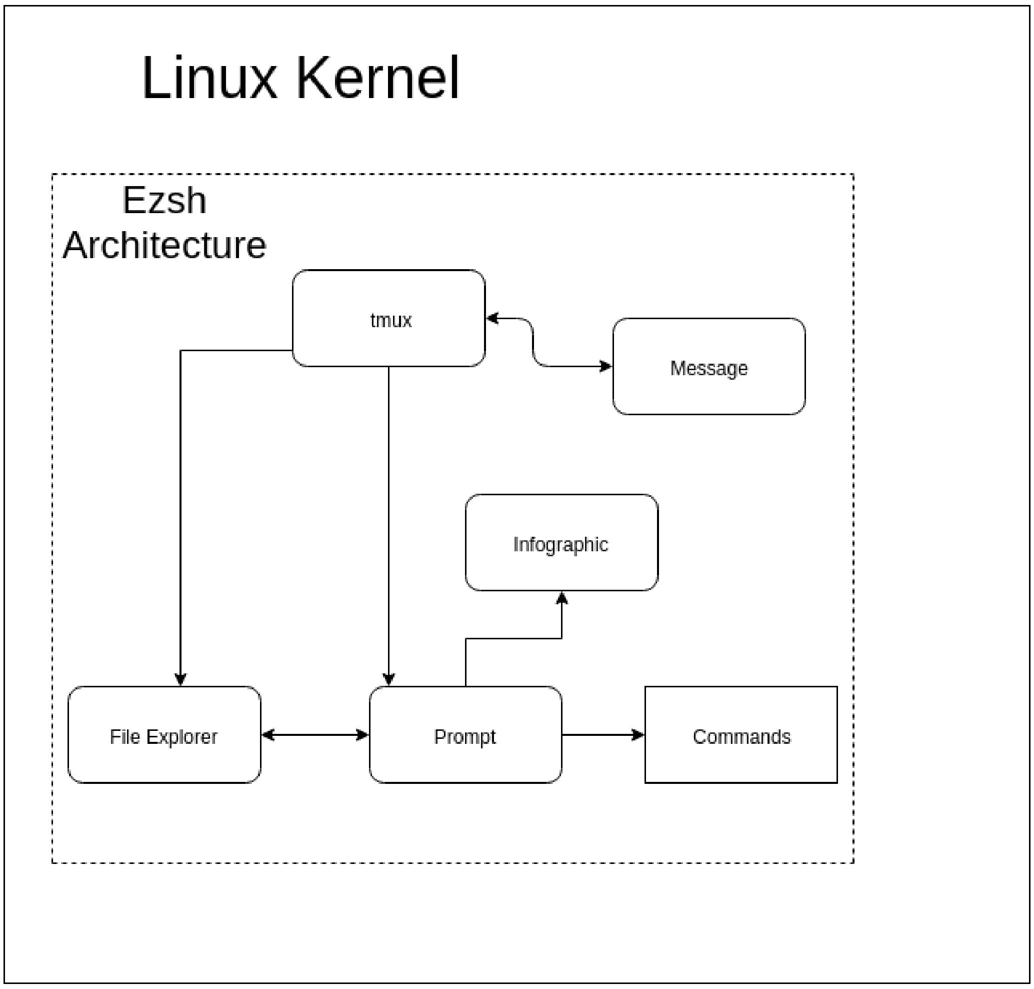
programs

2. System Architecture

The shell is made up of an `Explorer`, an `Messenger` and a `Prompt`. Each of these programs run in a separate Pane in the same Window. This window is split into the three panes respectively, and these components are said to make up the GUI of the shell.

A vital component of the shell is the tmux multiplexer, which is to aggregate these three core programs. The tty's of each pane created by tmux are logged to aid message passing of commands. Though the programs are indeed separate, inter-process communication is carried out through the use of named pipes to ensure the separate programs stay in sync after directory changes.

The shell operates in only Linux environments. This means it will be able to operate on multiple Linux distributions. These distributions include Ubuntu, Fedora, OpenSUSE and many others.



2.1 Dependencies

The dependencies in order to use ezsh are as follows:

- gcc version 7.3.0
- GNU make version 4.1
- GNU readline version 7.0
- tmux version 2.6
- ncurses version 6.1
- TOIlet version 0.3

2.2 Language Choice

> C was chosen as the primary language of choice for developing ezsh shell.
>
> Since we were developing a shell we deemed it necessary to use C due to its ability to work very close to the Linux kernel, which is also written in C. This meant we were able to easily tap into many system commands and even create our own by using system calls.
>
> Another factor which led to us choosing C over a language like Python was performance. In all, the speed of language like C is far greater than in comparison to the speed of Python.
>
> C was created in 1972 and is documented heavily. We knew that documentation for anything that we planned or needed to do would be relatively easy to find in comparison to a newer systems language such as Golang or Rust.

3 High Level Design

3.1 Initial Design

> Initially the shell was made up of an `Explorer`, an `Infograph` and a `Prompt`. Each of these programs would run in a separate pane in the same Window. This window would split into the three panes, the `Explorer`, the `Prompt` and the `Infograph`. These components were said to make up the GUI of the shell.

> In the software design stages since this idea was put forward, we have decided to introduce a `Messenger Pane`, which relays to the user the commands that get executed on either side. The `Infograph` still exists, but only as an optional plugin. We made this decision as we believe it would be more beneficial if users could view the commands they execute.

> This Multiplexer (tmux) could be seen as a way to manage the three programs independently of each other. The transmission of data from one to the other would be facilitated by the `ezsh server`. This waits for changes/actions to be taken in either the `Explorer` pane or the `Prompt` pane and transmits data to the other panes where appropriate.

> In lieu of a server, we decided to instead save the overhead of running a socket server on a port on the user’s system and implement the transmission of data from one pane to the other by tracking their tty’s and using named pipes. These tty’s can then be used to facilitate the echoing of messages, relaying changes/actions in both panes where appropriate.

3.2 Current Design

> The shell, as previously stated in our initial design, is primarily made up of three main programs. First off is the `Explorer` which provides the user an interactive way of navigating through there computers file system using the ncurses library, the ability to create, delete files and directories and even edit files.

> The second main program is that of the `Prompt`. The prompt reads user input similar to Bash and executes the command. However, it also has additional commands that Bash does not such as ‘history’ and ‘star’ which provides the user the ability to view both past commands and execute them again with ease and the star allows the user to save the path to a directory and jump to it with ease.

> The `Messenger` pane is the one place that is far less interactive than the other two and is simply there so that the user can see or refer back to commands they executed from both the prompt or file explorer. Unlike the previous two programs no named pipes were required. Instead the program executing the command would simply use `echo` and send the command out to that pane.

> As stated above in our initial design, the relaying of messages is achieved through the medley of stored tty numbers and two named pipes, one for each direction (Prompt to Explorer and vie-versa).

4. Problems and Resolution

> During the development of ezsh a few issues arose some being more drastic than others in regards to time, effort and overall complexity of the solution or workaround in very few cases. Some of the key issues that blocked further development were as follows.

---

1. File Explorer displaying random characters.

Issue:
This issue had taken some time before it was initially realized and then even longer until it was fixed. As the file explorer was developed independently from the tmux enviroment it was only after merging the two that we began to realize there was an issue. Initially we believed the issue to be poor memory management as it would occur arbitrarily however as many potential fixes were pushed and then later fail it turned out that the ncurses window when left with blank space would render in random characters.

Solution:
The fix for this was to dynamically generate the number of lines for the window with the max amount being fifteen as to not overload the screen with text and the minimum being the number of files and directories at that current point in the system.

---

2. ncurses allowing output to non-existent co-ordinates.

Issue:
Initially, during informal testing this had not been a problem as we tested the functionality of rendering all the directory contents and selecting different options on a small scale; it was when we decided to try to render a directory with a mix of over twenty files and directories that we saw that not every entry was being rendered or even existed. ncurses allows the user to output to anywhere on and off the screen.

Solution:
At this point our knowledge of ncurses was still in its infancy therefore instead of opting for pads we decided to parse the array of contents into a decent size which rendered on the screen nicely and when the user reached the bottom of the rendered list and attempted to go down again it would make use of labels to re-render the list but this time rendering the next section of the array causing a jump to next page effect.

---

3. Interprocess communication

Issue:
Once both the prompt and the explorer had been completed, the next task at hand was to get both programs communicating. Initially, we said this would be achieved using sockets however upon further research named pipes became a more straight-forward and optimal solution and if things did not go to plan we could at least fail fast and try another approach unlike with the sockets approach where a lot of time would be needed to understand it in its entirety. Once simple message passing was achieved with the named pipes an issue arose as to how will the program know that a pipe has been sent a message if both processes must wait for user input.

Solution:
To ensure no messages were missed we needed to place the code that checks the pipe, we will call this the writer in a infinite loop; both the prompt and the explorer use infinite loops also so at its current state the program would either be always reading indefinitely or always waiting for user input. So we opted to use threads and made the reader into a daemon that would constantly check if a message had been sent to the file, the message being the name of the directory either the explorer or prompt changed into. If a message had been sent and read successfully then for the prompt side the daemon would update the prompt alerting the user that the directory has been changed. In regards to the explorer, it would set a flag so that the next time the user attempts to enter any input it would refresh the entire screen so that it would correspond to the prompts current working directory.

---

4. Implementing History and Stars

Issue:
When tokenizing the input of a line like `!21` or `*11`, to execute line 21 of history and line 11 of stars, in would instead execute line 2 of history and line one of stars. Similarily `!38` would execute line 3 of history, and so on. So the issue was actually parsing the array of characters.

This is where Connor jumped in, he encouraged my to reverse the string and multiply each (after ascii conversion - 48) by a power of ten, which would increase based on the number of digits. While it does seem a simple fix, I would have probably never though of it that way, so it's due to Connor these features exist in the prompt.

Solution:
We decided to reverse the array of characters being passed at the prompt, skipping the first character which was either `!` or `*`. Each digit was then multiplied (after ascii conversion - 48) by a power of ten, which would increase based on the how many digits had been previously passed.
This functionality for parsing the input was applied to both feature sets.

---

5. Splitting The Terminal

Issue:
One of the most important parts of the shell is the ability to manage multiple panes in a single shell session to allow different programs to run concurrently. The issue was, how do we get the user's terminal session to start up in the same fashion everytime, and how do we get each pane to call a specific program.

Solution:
The easiest solution to the problem turned out to be a medley of a shell script, `ezsh`, and a configuration file, `.ezsh.conf`. The `./ezsh.conf` file by default contains three lines. Each line represents a pane and has a size and a program to run. This information is read into the aforementioned shell script, line by line, and creates a pane to each lines specification.
Upon completion, the `ezsh` script boots tmux to the specified standards outlined in the configuration file.

---

6. Catching Ctrl-C Correctly

Issue:
After implementing a handler which would catch SIGINT's (Crtl-C), there was still one outstanding issue. If the user had partially entered a command `vim`, then sent a SIGINT with Ctrl-C, and entered the command `ls` on the new line, ezsh interpreted this as the command `vim ls`, and opened a file for editing in vim called ls.

Solution:
After the SIGINT is caught, the handler needed to replace the text readline() was storing with an empty character. Once readline forces a redisplay, the prompt is reset using the ezshPrompt function. Since this code has been changed, we have yet to see this issue rear its head again.

---

5. Installation Guide

5.1 Installing Compilers and Build Tools

> Both the compiler `gcc` and build tool `make` usually come preinstalled with most Linux distributions.
> In the case they are not you can run:

- `sudo apt install gcc` to install the gcc compiler.
- `sudo apt install make` to install the make build tool.

5.2 Installing Shell Utilities

> The shell utilities, that is, the dependencies which the shell requires to run, can be installed using the `requirements` shell script `/code`. The dependencies installed by using this script are:

- tmux
- GNU readline
- ncurses
- TOIlet

> To run the shell script you need to give it the correct permissions to allow execution. Using:

`chmod +x requirements`

> allows this action, and this script can be executed by running

`./requirements`.

5.3 Run ezsh

After you have installed both the build tools and shell utilities you may run `./ezsh` to run ezsh.

6. Configuration

6.1 Configuration File Location

> You can change many things about the shell so that is runs to suit your needs. The file used for configuartions is found at `~/.ezsh/.ezsh.conf`.

> The following is an example config you can use to get your started.

6.2 Default ezsh Configuration

```
ezsh 0 tmux resize-pane -t 0 -x 3 && tty > .ezsh/.ezsh.tty && clear && ./exp
ezsh 1 tmux resize-pane -t 1 -y 3 && tty >> .ezsh/.ezsh.tty && clear && toilet -f big
ezsh 2 tmux resize-pane -t 2 -x 120 -y 3 && tty >> .ezsh/.ezsh.tty && clear && ./msg
```

Each line may be broken down as follows:

```
ezsh PANE_NUMBER tmux resize-pane -t TARGET_NUMBER -x WIDTH -y HEIGHT && tty > ./ezsh
```

6.3 Building your own ezsh Configuration

> Keeping the breakdown of the line above in mind, the following parts of that line can be changed in accordance with the following restrictions:

> - **PANE_NUMBER:**
>   This is the number of the pane you are on.
>   Assuming you start on line one, if you find yourself on the nth line, the PANE_NUMBER should be `n-1`.

> - **TARGET_NUMBER:**
>   This should always be the same as the current PANE_NUMBER.

> - **WIDTH:**
>   WIDTH can be an integer between 1 and 180 which corresponds to the width of that PANE_NUMBER

> - **HEIGHT:**
>   HEIGHT can be an integer between 1 and 180 which corresponds to the width of that PANE_NUMBER

> - **PROGRAM_NAME**:
>   The PROGRAM_NAME can be on of the following four plugins which ezsh can use:

- ./prompt
- ./exp
- ./msg
- ./info

6.4 Sample ezsh Configurations

- **Sample One**

```
ezsh 0 tmux resize-pane -t 0 -x 90 && tty > .ezsh/.ezsh.tty && clear && toilet -f big
ezsh 1 tmux resize-pane -t 1 -x 70 && tty >> .ezsh/.ezsh.tty && clear && ./exp
ezsh 2 tmux resize-pane -t 2 && tty >> .ezsh/.ezsh.tty && clear && ./info
ezsh 3 tmux resize-pane -t 3 -x 40 && tty >> .ezsh/.ezsh.tty && clear && ./msg
```

- **Sample Two**

```
ezsh 0 tmux resize-pane -t 0 -x 90 && tty > .ezsh/.ezsh.tty && clear && toilet -f big
ezsh 1 tmux resize-pane -t 1 -x 70 -y 25 && tty >> .ezsh/.ezsh.tty && clear && ./exp
ezsh 2 tmux resize-pane -t 2 && tty >> .ezsh/.ezsh.tty && clear && ./info
```

7. Testing

7. Functional

> For testing the team used a continuous integration model. This did not work out as we expected

> at the the start because of various difficulties setting up the pipeline. This was due to our lack of experience with Gitlab CI.

> The setup of our CI contains a before_script section which installs and builds the testing framework of our choice. There are then two separate jobs, each consisting of unit-tests. These two jobs are designated to running test suites for the `Prompt` and the `Explorer` respectively. After successfully passing, the after_script section performs clean up.

> This CI configuration is run every time either of us makes a commit to any branch of the repository.

> In general, as a comment, the code coverage of the `Prompt` is better than that of the `Explorer`. While there are extensible ncurses-based testing framework, we prioritised our feature set over setting this up.

7.2 Non Functional

> As highlighted earlier, we wanted ezsh to be user-friendly and easy to use shell utility from the get go.

> In it's current state, we hold the conviction that the shell is incredibly simple to use and imparts more knowledge on shell functionality as you use it.

> This is, however, for the most part a biased metric without user testing. Our user testing was conducted as a focus group where each individual cloned the project and installed it. Each individual used Ubuntu Linux during this session.

> The participants were given a set of tasks to complete in both ezsh and Bash. These were simple, every day prompt based tasks such as creating directories, removing directories and editing files.

Ease of Use

> When it came to usability and interactiveness on a scale of 1 - 10, ezsh averaged a score of 8.5 while Bash averaged a score of 6.5.

> This is in part due to the Explorer, which is much more interactive than any utility offered by Bash.

Speed of Use

> When it came to what shell allowed the users to perform tasks faster, tasks on ezsh averaged a time of 3 minutes and 23 seconds while Bash averaged a time of 5 minutes and 45 seconds.