prosh: The Productivity Shell

A custom shell with pomodoro functionality

Petr Sabovčik
Pascal Wohlwender

Project Report

Operating Systems Course
Spring 2023

Contents

1 Introduction

Currently there are a lot of discussions going on about the effects of social media on our attention span. We and (we suspect) many other students are faced with a myriad of distractions from work daily. We find ourselves procrastinating far too often. It impedes our studies and makes us feel bad for indulging in these distractions. Thus, we found it a relevant issue enough to base our project on combating it.

The question that escapes many is how does one prevent procrastination? And in addition to that, what is the level at which procrastination becomes a problem? Of course, no one can maintain focus permanently, nor would anyone find this desirable. To mitigate the problem caused by procrastination we also had to ask in what ways we procrastinate.

The goal of this project was to create a custom shell which can prohibit the user getting distracted by social media sites, games, etc. Our approach was to (in a brute-force fashion) simply disallow the user from connecting to sites and starting processes we consider not pertinent to work on their computer.

2 Background

A shell is practically an interface for an interface. User inputs to system calls to OS. The main purpose of a shell is to parse user inputs and execute corresponding system calls. As such, the relevant technologies to the project are functions that can take user input and parse it and system calls for purposes like changing directory or executing files.

3 Implementation

Firstly, a brief overview of the shell. The core of the shell is a while loop, which takes user input and returns the appropriate command ID. A switch statement is then used to decide what is the appropriate action to be taken. We classified commands into 5 categories: changing directory, listing files in a directory, file execution, a help command, and productivity mode related commands. There are a further 6 categories dedicated to handling productivity mode commands: start and end productivity mode, show status of the productivity mode, as well as adding, removing, and listing processes and domains to be blocked during productivity mode. Domains are blocked using the `/etc/hosts` file and processes are blocked using `pkill`.

Secondly, handling user input. We used the GNU readline library to handle inputs. It provides a function which takes inputs similarly to the one in Python and also is able to keep track of a history of commands. Then `strcmp` is used to figure out what the user wishes to do and action is taken accordingly as described above. Arguments are parsed using the `strtok` function. It splits a string into several substrings using a user supplied delimiter (here the delimiter is a space). By calling this function until it returns `NULL`, we can extract all the arguments and also determine whether any arguments were supplied at all. We do not however exhaustively check all supplied arguments for functions defined by us (e.g., "*cd ./some_folder invalid_argument*" would still be a valid input). Since `strtok` alters the string supplied to it, we must store a copy of the command that the user inputs. This way, we can retain the original command even after checking which category it falls into. Executing files would not be possible otherwise because some arguments would be lost.

Our implementations of the *change directory* (`CD`) and *list files in directory* (`LS`) functionalities are quite simple. If no arguments are supplied, `CD` simply returns and `LS` lists the files in the current working directory. Otherwise, the corresponding system calls (`chdir` and `scandir`) are used and their result codes are used to determine the success.

The last general shell function is the execution of files. If a user inputs something not defined by us, we attempt to execute their input using the `execvp` system call. Arguments are passed by constructing an array of arguments with `strtok` as described above. Since we wish to retain control even after the file has finished executing, we must execute this file in another task. We then wait for this task and after it has finished we continue with the main loop.

Lastly, a short description of the productivity specific commands. An array keeps track of the domains and processes to be blocked. Domain and process names can be added to this 'blacklist' as described in **Productivity Mode Configuration**. Once productivity mode is started, each time a new window is opened the shell attempts to kill every process stored in the blocked process array. Domains to be blocked are added to the `/etc/hosts` file once productivity mode is started and are removed from the file (using a stored copy of the original hosts file) once productivity mode ends. The list command simply iterates over the array of either blocked processes or domains and prints their names. When the user requests the status of productivity mode a simple message is printed informing them if it is active.

4 Result

We developed two executables: *prosh* and *proshdom*. While *prosh* contains the basic shell as well as most of the commands, *proshdom* executes functions which modify the `/etc/hosts` file.

4.1 Shell Setup

Both *prosh* and *proshdom* are needed to run the shell and they have to be placed inside the same directory. Afterwards, prosh can be started using the terminal.

```
./prosh
```

To get yourself started, list all available commands.

```
help
```

To start the productivity mode, use the following command.

```
prod start
```

Be sure to end it before you exit the shell to reset your `/etc/hosts` file.

```
prod end
```

4.2 Productivity Mode Configuration

prosh provides a default blacklist for domains and processes that are blocked while the productivity mode is running but it is possible to add and remove entries.

Use the following command to add *wikipedia.org* to the blacklist.

```
prod add domain wikipedia.org
```

To remove it, execute the following command.

```
prod remove domain wikipedia.org
```

Replace `domain` with `process` for processes.

5 Discussion

In April, when we wrote the project proposal about our shell, we had one executable in mind. Now, at the end of our project, the productivity mode only works with two combined: *prosh* and *proshdom*. The reason for this inconvenience is the editing of the `/etc/hosts` file that needs root permissions. Those can either be obtained by executing the `sudo` command or by using `setuid` flags. Sadly, we did not manage to use the latter, so had to call `sudo`. However, we figured that starting our whole shell with `sudo` is not a good idea because it enables users to execute any command with root permissions. We therefore had to exclude all functions which modified `/etc/hosts` but we still wanted it to feel like a part of the shell. Hence, we created a command that calls the `sudo ./proshdom` in the background.

Technically speaking, the blocking of domains is therefore not part of our shell because we had to move it into a separate executable. We therefore failed to create a shell that blocks domains and instead created a shell that calls a program which handles it.

There is another problem with our approach of blocking domains. Subdomains are not automatically included but have to be added separately to the blacklist. For example, if *wikipedia.org* is added, *de.wikipedia.org* can still be opened. This has to do with how the `/etc/hosts` file works. It does not include subdomains.

6 Conclusion

As stated in the introduction, our approach of preventing users from getting distracted was to block domains and processes.

While we have failed to create a reliable way of blocking domains and including that functionality in prosh itself, we succeeded in creating a working productivity mode that blocks processes. As long as the blacklist contains the right process name, the user cannot open it.

If we had found a better way of blocking domains, we could have made a more reliable productivity shell but we learned a lot in the process.

7 Lessons learned

7.1 Petr Sabovčik

Personally, I mostly applied previously learned knowledge rather than acquiring new knowledge throughout working on this project. Although it did take me quite a bit of courage to start working on the project, once I did I felt quite sure in my decisions and the biggest difficulties were wrestling with C strings and system call documentation. As a result, I have come away from this project with more self-confidence in my programming capabilities. Unlike during previous projects I was able to identify and remedy problems whenever they arose (no rewriting the whole project in a state of panic). Other than that, I also was able to compare a previous project to this one. A project I took part in last semester was devoid of communication and the division of labor was also lacking. This time my project partner encouraged a very healthy communicative environment with an emphasis on making progress in time for deadlines, which paid off.

In short, I learned the values of project planning and communication.

7.2 Pascal Wohlwender

At the time this project started, my experience with C was limited to the few exercises we had solved in parallel with the OS course. I was an absolute beginner who had only written a few functions and still had problems with pointers. While working on this project with Petr, I had to learn some basic concepts like writing and using header files. Additionally, I was able to get more practice with pointers and deepen my knowledge about threads and file access. While also struggling with strings, I spent most of my time and energy reading about acquiring root permissions. Now, at the end of the project, I am still a beginner but I have lost my fear of programming in C. Furthermore, I was once again remembered that having and keeping to a solid project schedule is an important requirement to prevent stress and bugs.

8 References

Apart from the standard C header files we have used the following libraries.

1. **GNU Readline**: This library handles command line inputs.[1]
2. **Xlib**: This library provides functions and events related to the X window system.[2]

[1]https://tiswww.case.edu/php/chet/readline/rltop.html
[2]https://www.x.org/releases/X11R7.5/doc/libX11/libX11.html

9 Appendix

9.1 Separation of Tasks

While we reviewed, discussed and improved the code together, we had a clear separation of responsibility. Petr Sabovčik worked on the files *main.c* and *blacklist_manager.c*. He created the shell with command line input processing and process execution. Pascal Wohlwender on the other hand worked on *productivity_mode.c*, the header files and the proshdom executable. He created the productivity mode with process and domain blocking.

Concerning this project report, Petr Sabovčik wrote the sections *Introduction*, *Background*, and *Implementation*, while Pascal Wohlwender wrote *Result*, *Discussion*, and *Conclusion.*

9.2 Declaration of Independent Authorship

We attest with our individual signatures that we have written this report independently and without outside help. We also attest that the information concerning the sources used in this work is true and complete in every respect. All sources that have been quoted or paraphrased have been marked accordingly. Additionally, we affirm that any text passages written with the help of AI-supported technology are marked as such, including a reference to the AI-supported program used. This report may be checked for plagiarism and use of AI-supported technology using the appropriate software. We understand that unethical conduct may lead to a grade of 1 or "fail" or expulsion from the study program.[3]

Petr Sabovčik and Pascal Wohlwender, June 2023

[3]Legal Services, University of Basel, April 2023