CIS 415 Operating Systems

Project 1 Report Collection

Submitted to:
Prof. Allen Malony

Author:
Ethan Reinhart
ereinha3
951909814

Report

Introduction

The purpose of this project was to emulate commands executed by the shell by using system calls within a C file. The shell is the OS's command line interface as well as the interpreter for telling the kernel how to execute these commands. Shell commands are executed by taking in input from an input device (a file or keyboard input) and then telling the kernel what to do. In this project, we used system calls in C instead of going through the shell directly. System calls ask the kernel to do something in the OS and are the only way the kernel can be accessed. Making a system call switches the kernel into privileged mode. Programs can access system calls using a high-level application program interface (API) which, in our case, is using the POSIX API. Each system call is typically associated with a number and returns an integer value representing the status of the system call.

A shell includes a variety of commands that typically invoke a series of system calls. In this project, we implemented a select few commands, which each invoke the same command as if inputted in the shell. All C code invokes system calls to carry out the process described in the C code. It does this by pulling from pre-built standard libraries and are called when the code is compiled and linked. Many of these system calls that we called are from pre-built C libraries, but need a functional understanding of what the call is actually doing to implement them. (4) Many POSIX wrappers where used in system calls, which are contained in imported standard libraries, and serve as wrappers around Linux system calls. Implementing all of the above requires a high-level understanding of what each individual call is doing, what its return value should be upon execution, and what the significance of the executed call is.

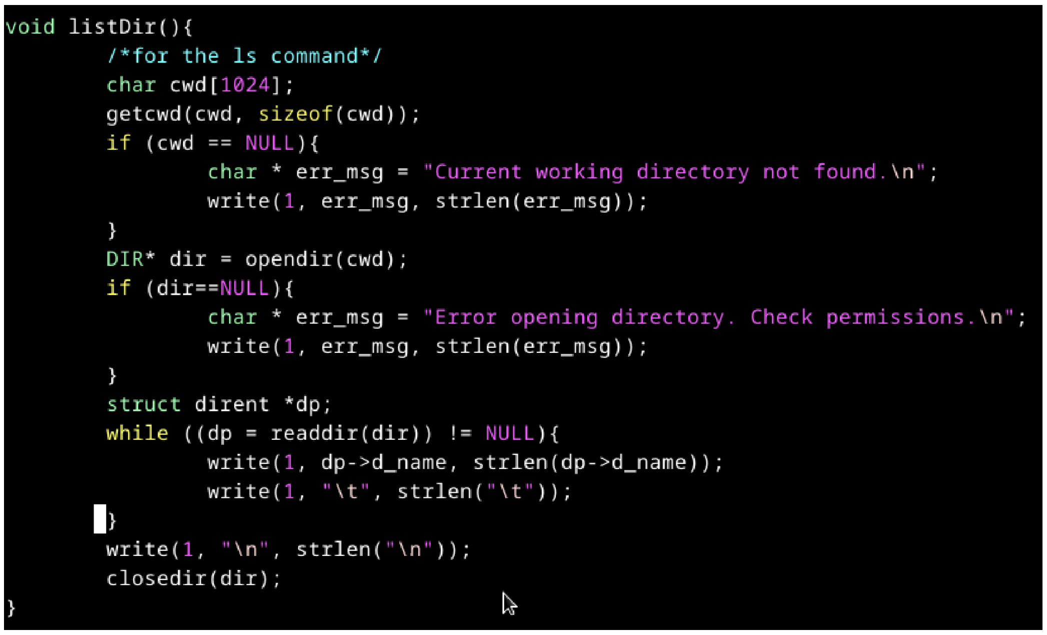
Background

A large portion of this project was preliminarily created in Lab1 and Lab2. Lab1 required the creating of "string_parser.c", which takes in an input string, parses it for delimeters, and returns a custom variable containing both the number of commands as well as a list of commands, terminated by the NULL character. This allowed for seamless tokenization of file or command-line input. Lab2 required the creation of the ls function, listing all elements of a specified directory, which allowed for the foundations of system call processes to be established. The project was executed by creating a "main.c" file, which takes in argv values and determines whether to run in user-input mode or file mode. Output in file mode must be redirected to an output file while output in user-input mode was directed to shell output. This required the implementation of both the write() and freopen() command. Write() takes in a file descriptor detailing where the output should be written to, a buffer which contains a character pointer to a string containing what is to be written, and a size_t variable which corresponds to how large the buffer is and returns a ssize_t variable which corresponds to the number of bytes written (returning -1 on failure). This required the use of freopen() to account for file-input mode. Freopen() takes in a filename of where input will be redirected, in our case this would be the output file; a mode which is indicated by a character pointer describing the files access mode, I used 'w' to write to the file; and a file stream where output is redirected from, this would be STDOUT. This means that any content written to STDOUT will now be written to our file. (3)

I won't go into detail regarding all commands implemented, but a particularly difficult one was the cp command, which takes in a source file and a destination file argument and then copies the contents of the source file to the destination file. This was especially difficult because you must account for two cases: the destination is a file or the destination is a directory. It doesn't matter if the file already exists because my implementation forced a creation, which would override the current file and create a new blank file. In the case that the *destination is a directory, I first called upon the chdir() command, which takes in a char* pointer string which describes the directories name. If this command fails, chdir() returns a value of -1. Upon success, the current directory is changed to the specified directory. I then called upon the str_filler() command, implemented in Lab1, to parse the destination directory into only what is after the last "/", which would be the file name. If no "/" are present, this would simply return the file name. The next thing I did is the same for both file-input and user input, I forced a file open with the specified file name and used read() to write the contents of the input file to the output file. (5) I also closed both of these files to avoid memory leaks. The reason this does not ovevrride the initial file with the same name is because the directory was change, so the new file name is implicitly newDir/filename. All of these built-in POSIX wrappers to LINUX system calls are pulled from built-in libraries and syntax and usage where implemented via guidelines from the LINUX system calls website. (6)*

Implementation

Below is an implementation of my listDir() function which is performing the ls command via system calls. I opted to use the same command as used in the pwd command: getcwd(). The getcwd() command works by by taking in a char argument that serves as the buffer, where the directories name is stored, as well as a size_t argument that gives the buffers size. I then used opendir() to open the directory which requires the use of a DIR* pointer to the directory. Opendir() will point to the NULL pointer if the directory fails to open and will otherwise serve as a pointer to the first element in the directory. Struct dirent is a built in structure that is used to return information about directory entries. The dirent structure contains information about file types and offsets to next entry as well as the file name, which was the only important part for this project. (2) This is accessed by pointer dereferenceing using the -> operator. The directory can then be iterated over by using readdir() on the directory, which takes in the DIR pointer and points to the ith element of the directory, depending on how many times readdir() has been called; meaning the first time it is called, it will point to the first element. The name can then be accessed by dereferencing the pointer and is written to STDOUT. The significance of the "1" in all write commands is that this number corresponds to STDOUT in write() (ie STOUT = 1). Depending on the mode we are in (file-input or user-input) signifies where STDOUT will be written, as in file-input mode, STDOUT has already been redirected to our output file. (7) The DIR pointer must then be*



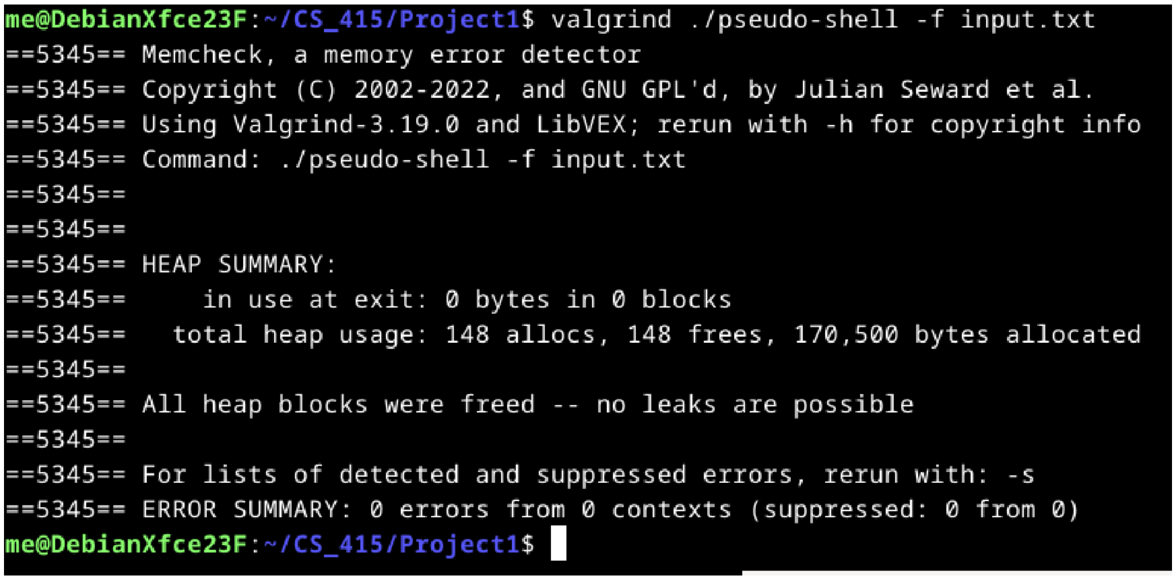
```c
void listDir(){
        /*for the ls command*/
        char cwd[1024];
        getcwd(cwd, sizeof(cwd));
        if (cwd == NULL){
                char * err_msg = "Current working directory not found.\n";
                write(1, err_msg, strlen(err_msg));
        }
        DIR* dir = opendir(cwd);
        if (dir==NULL){
                char * err_msg = "Error opening directory. Check permissions.\n";
                write(1, err_msg, strlen(err_msg));
        }
        struct dirent *dp;
        while ((dp = readdir(dir)) != NULL){
                write(1, dp->d_name, strlen(dp->d_name));
                write(1, "\t", strlen("\t"));
        }
        write(1, "\n", strlen("\n"));
        closedir(dir);
}
```

closed. Our struct dirent pointer need not be freed as it is dynamically closed when we close our directory using closedir(). (1)

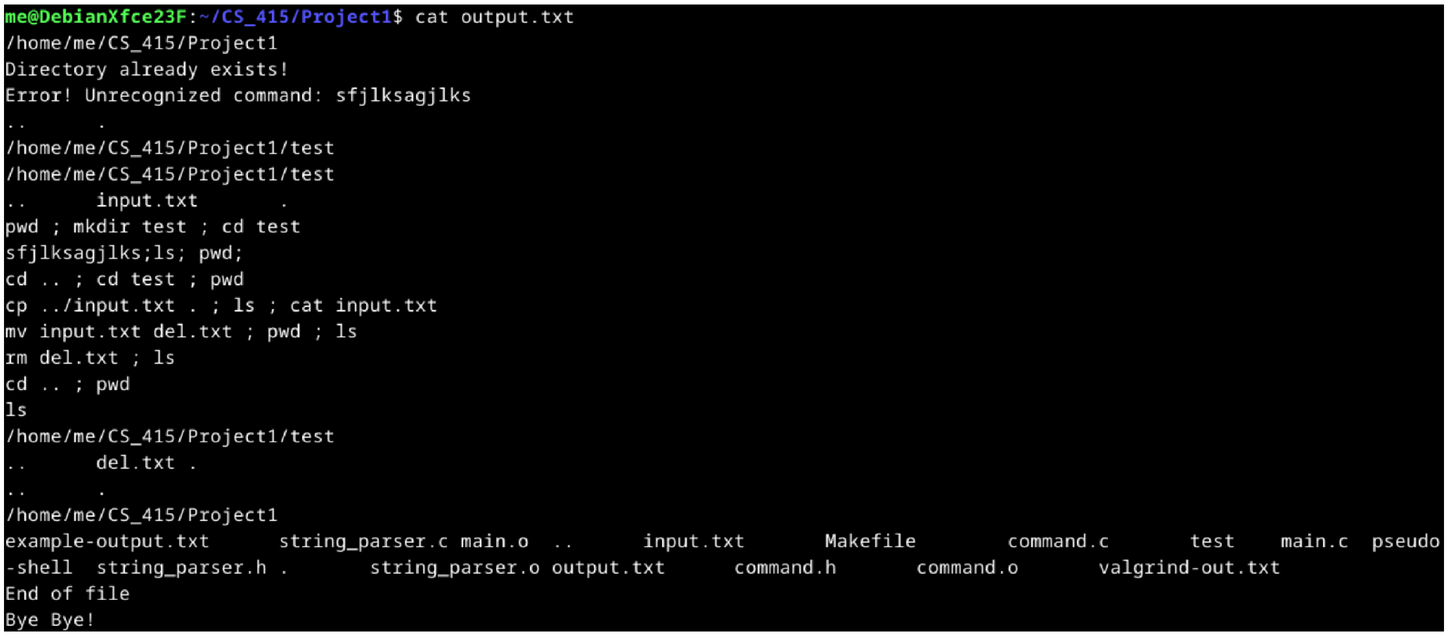
Performance Results and Discussion

My program appears to give the correct output for all given commands in both file-mode and user-input mode. I tried comparing my output file to the provided output file, and there were a few smaller differences. However, I am confident that my output file is correct, and there are a few things in the provided output file that I disagree with. Walking through the commands in the input file, I see that my output file executes them in consecutive order, while the provided output file skips over a few. It took a while to trouble-shoot all small errors caused by memory leaks as well as a multitude of segmentation faults caused by the commands implemented in "command.c". Many of the commands were trouble-shooted by extensive use of Office Hours, but one that was a specific difficulty was the above described cp command. I would consistently get segfaults when the file would try to be opened but it would recognize it as a directory instead. It would also execute the code after which would force create a file with the same name as the input, and wipe the input file clean. This resulted in having to redownload the input file and create multiple copies to test this.

Another major problem was memory leaks. I wanted to create perfect code that resulted in memory leaks and continuously, found small amounts of bytes leaking from my code. I tested this in multiple ways. First, I reran Lab 1 in valgrind to see if my str_filler function was causing memory leaks. It wasn't, so I then tested user-input with only the exit command. This was creating memory leaks. I read over my file dozens of times to see where it could be as well as ran valgrind with as many flags as possible but from my perspective, I was always freeing everything dynamically allocated. Then, I realized, user input of exit was a special case that forced the function to end upon execution of the command. What I didn't have was all my free statements in that section of code as well. I added them all there as well and immediately got memory-leak free code, as shown below. "output.txt" is also shown below which I believe to be correct, other than a little funky syntax regarding the ls command.



```
me@DebianXfce23F:~/CS_415/Project1$ valgrind ./pseudo-shell -f input.txt
==5345== Memcheck, a memory error detector
==5345== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==5345== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==5345== Command: ./pseudo-shell -f input.txt
==5345==
==5345==
==5345== HEAP SUMMARY:
==5345==     in use at exit: 0 bytes in 0 blocks
==5345==   total heap usage: 148 allocs, 148 frees, 170,500 bytes allocated
==5345==
==5345== All heap blocks were freed -- no leaks are possible
==5345==
==5345== For lists of detected and suppressed errors, rerun with: -s
==5345== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
me@DebianXfce23F:~/CS_415/Project1$
```



```
me@DebianXfce23F:~/CS_415/Project1$ cat output.txt
/home/me/CS_415/Project1
Directory already exists!
Error! Unrecognized command: sfjlksagjlks
..      .
/home/me/CS_415/Project1/test
/home/me/CS_415/Project1/test
..      input.txt       .
pwd ; mkdir test ; cd test
sfjlksagjlks;ls; pwd;
cd .. ; cd test ; pwd
cp ../input.txt . ; ls ; cat input.txt
mv input.txt del.txt ; pwd ; ls
rm del.txt ; ls
cd .. ; pwd
ls
/home/me/CS_415/Project1/test
..      del.txt .
..      .
/home/me/CS_415/Project1
example-output.txt      string_parser.c main.o  ..      input.txt       Makefile        command.c       test    main.c  pseudo
-shell  string_parser.h .       string_parser.o output.txt      command.h       command.o       valgrind-out.txt
End of file
Bye Bye!
```

Conclusion

I learned a great deal from this project. It required me to implement a lot of the ideas we have been learning in class into practice. Writing this report, the lecture slides as well as notes from class helped a lot to explain topics. I learned a lot about the syntax of system calls as well as their significance with regard to the API and the kernel. I also learned what POSIX actually does: how it serves as a wrapper for LINUX system calls and allows you to execute thoes system calls within a C file. It also served as a nice refresher for debugging in C as there were a multitude of bugs I experience throughout this project. I think I greatly benefitted from this project and gained a much deeper understanding behind what happens when you run a command in the terminal or how C files actually compile!

Bibliography:

Not all websites used are shown. To understand syntax and functionality, I probably referenced 50+ different websites. Those significant to the report are given below.

1. *Closedir*, pubs.opengroup.org/onlinepubs/009604599/functions/closedir.html. Accessed 17 Oct. 2023.
2. *Directory Entries (the GNU C Library)*, www.gnu.org/software/libc/manual/html_node/Directory-Entries.html. Accessed 17 Oct. 2023.
3. "Freopen." *Cplusplus.Com*, cplusplus.com/reference/cstdio/freopen/. Accessed 17 Oct. 2023.
4. "Introduction of System Call." *GeeksforGeeks*, GeeksforGeeks, 25 Sept. 2023, www.geeksforgeeks.org/introduction-of-system-call/.
5. *Read(2) - Linux Manual Page*, man7.org/linux/man-pages/man2/read.2.html. Accessed 17 Oct. 2023.
6. *Syscalls(2) - Linux Manual Page*, man7.org/linux/man-pages/man2/syscalls.2.html. Accessed 17 Oct. 2023.
7. *Write(2) - Linux Manual Page*, man7.org/linux/man-pages/man2/write.2.html. Accessed 17 Oct. 2023.