Ben Williams
CPRE 308
Lab 2 - Forking and Processes

The purpose of this lab was to become familiar with the creation of new processes and have them do something useful. We also learned about linux reference files (man pages) which closely detail the inputs and expected outputs of certain linux shell commands. We also were expected to learn or have previous knowledge of string manipulation, including parsing, concatenation and building new strings.

To start it all off, we started with forking. fork() as a function in C creates an identical child process and starts its execution at the same time as the parent. **fork() either return 0 if the program is in the child process, or the PID of the child, if the program is the parent. The return value of fork can then be checked to determine if we are in the child (0), and if so, to do the useful child task. If not, the parent may wait until the child is finished, or continue on functioning in parallel.**

After we have checked the return value, we must make the child process do something different. Since we have a clone of the parent, we must check the return value of fork so that the parent and child aren't doing the exact same thing. By embedding the fork call inside an if statement and checking if fork is the child, we can tell exactly whether the code is operating as the child, the parent, or some error occurred while forking.

To wait on a child process (as the parent), the two useful functions to us as programmers are wait() and waitpid(). These can be passed integers which will return the status of the child process after it has completed running. This is where I learned to use man pages (just kidding, I learned at my job last summer.) They are very useful locally stored references to Linux shell commands.

We can either use the more basic wait() and pass the address of an integer to write the status to, or use the more complex waitpid() by assigning the fork value to some integer and then using the PID returned to the parent to more specifically wait on this child.

To make a process replace it's image with another, we must use some form of command in the exec() family. I read up extensively on what exec() does and what its inputs are in the man pages, and found that there are many similar commands that all achieve the same result through differing inputs. An optional argArray and envArray are passed in as parameters for the program that you are trying to start up. I opted to use execlp() in my final file opening program because I could enter a literal array of command line arguments like I was already used to, and then simply enter a NULL at the end to terminate the array.

With all of this reference information (and my previous experience with string.h), I embarked on the task at hand: create a universal file opener for doc, odt, png, txt, pdf, and mp3 files. To begin, I parsed the input given in argv[1].

I used strlen() to find the length of the argument and created a string with length 3 **fileType** to store the type of file input. I used strncpy() to grab the last three characters of input and placed them into fileType.

I checked the correctness of the input by checking whether the 4th to last character of the input was the ascii value of an period (46.) If the file name did not include a period before the 3 character extension, it could be safely ruled as an incorrect input. I exited the program using perror in this event.

After this, I used strcmp() to compare the fileType with different string literals I wrote out representing the different file types needed. I used a simple if/else if chain to check for each of the required files. Inside each of these, I forked and checked the value for whether it was greater than or equal to 0. If it was equal, I used execlp() and all necessary command line arguments to launch the proper child program with the correct inputs. This took the most time out of the entire lab since I was not intimately familiar with terminal commands, as I usually use the GUI to operate a Linux machine.

I'm not entirely sure if I did the best practice, but for most file types, libreoffice was able to open them, even the pdf. I used libreoffice for doc, odt, txt, and pdf. For mp3 I attempted to use rhythmbox, which opened the program and added the mp3 file to the play queue. Unfortunately, admin rights are required on the computer to install a special extra decoder to play mp3s. Finally, I used eog to open all png files.

After the if/else if chain, I had one final else that caught all file types that were not supported. I exited using perror and informed the user which options were legitamite.

All in all, it was a fun lab that showed how you can use processes and the Linux shell to make life easier on yourself. I wish I had more time to complete the extra credit portion, but unfortunately I have had way too much on my plate in the past week. These basic shell commands seem to be very