Project I: What the Shell, Jim?

Writeup by: Christopher Short, Dmytro Shmagin, Sabrina Sammel

TO COMPILE:

1. cd into src
2. Run “make”
3. Output files will be stored in ../bin, where there you can run ./wanna_be_bash to access the shell
4. (Optional) make clean removes the ../bin directory and all of its contents

Wanna_be_bash: Shell Implementation

The wanna_be_bash program is our implementation of the Linux shell. It first takes a string using fgets from the user, then splits it up using a string tokenizer. The tokens are then structured in an array (to be passed in as the argument vector) and passed as an argument to execv, which executes the program in argv[0] (with some additional string formatting to define the location of the executables). The program also handles errors from the user in both syntax and permissions by comparing against errno and printing the corresponding output.

In addition to parsing and executing the processes detailed in this project, our implementation of the shell allows for setting environment variables to pass to its children. It does so with the help of the export command and the generated output file (labelled env_vars). This command is dangerous if used improperly - for example, if the current PATH is changed, then none of the commands (processes such as ls and mkdir) will run unless their executables are located in the new PATH directory/directories. This includes export, so if the PATH variable is changed improperly you will not be able to export the PATH variable back because the export command is now unreachable. For this reason, our shell comes with an embedded command, “reset”, which sets the PATH variable back to the default to where all of the commands are reachable. It is used by simply invoking “reset”. The file env_vars is destroyed upon exiting the shell, so any environment variables that are set are strictly for the current shell. To exit the shell, invoke “exit”. Any other command (we have implemented mkdir, rmdir, ls, whoami, and the aforementioned export) is run by invoking its name and then its corresponding parameters, if they are needed. These parameters and the command itself are stored in a parameter array and then passed to their children processes as arguments. In addition, because we are using execve, the PATH and PWD environment variables will always be set unless the user defines otherwise - this way functions like ls and rmdir can function by invoking getenv on the environment variables they need.

Export: Environment Variable Definition Function

Export()
The export function exists to allow the user to input their own environment variables for both the current running shell and its children processes. As noted before, this command is dangerous as it can potentially lead to the loss of functionality of all shell commands except for “exit” and “reset”. To use this function, an assignment must be specified in the form of key=value as the second parameter, e.g. “export PWD=/some/directories/”. These will then be stored into a temporary file named env_vars which is destroyed at the end of the shell’s lifespan.

Whoami: User Reference Function

The whoami() function works by returning the current user of a system. In our program we were able to use the built in functionality of environment variable to quickly and easily compute this. Our main function’s purpose is to primarily call the whoami function. One interesting thing we found during testing is that docker containers do not set the ‘USER’ environment variable. For easy testing and rather than having having to run a VM in virtualbox which is very slow and chews through battery life. In this case we were using the official GCC docker image which is built off of Debian. This prompted us to have to find another way to map the UID to a username. ‘getpwuid()’ in pwd.h was the function we were learning for which allowed us to pass in a UID and get the username of that UID in return.

Whoami()
We definitely found that whoami() was the easiest function to implement. Without any parameters needed, once the whoami function is called, the process calls the getuid() function to return the user’s id number. After that, it prints the user’s name after having been translated by the getpwuid() function using the user’s id as its calling variable.

Mkdir: Make Directory Function

The mkdir function was less difficult than we had anticipated, with amusing results. We were able to test the use of this function as we tested other functions like rmdir and ls. The make directory function works by passing the two arguments: *int argc* and *char** argv* into the main function. We set an initial command flag that will help us determine what options we are using in our command. From there we call the getenv() function to dynamically determine our file’s location. We open our current directory and set a pointer that will be used to help us parse through the directory we are currently in. We are then ready to read in the arguments passed through to us from the *wannabe bash function*. In this program we look to use the *verbose* and *parents* options when building our directories. We have included the null option in which a user calls the mkdir function and then neglects to provide us with an option or an argument. In this case, we print an error message to inform them that they are missing an operand. We then read

in the option input and map our longform option input to the short form so that either is a viable option when making directories. Should the user input an incorrect option, or one that we have not implemented, will print and error message that they have entered an unrecognizable option. Depending on the options they input (or did not input), we set the values located within our command flag and determine which function to call thusly.

Orphan()
The orphan function is called when we have called to make a directory with no preceding options. While, simple we strove to make this function as identical to a bash shell command as possible. We start the function by passing in the *int argc* and *char** argv* values as parameters. We first check to make sure that we have no errors in the arguments we received before we enter the wheels of the function. Using a while loop, we first test whether or not our directory to build exists solely as a forward slash. Since this is an invalid character and already exists in linux, we print an error message to let the user know that it is an invalid entry. If not, we then make the file and grant it read, write, and execute permissions. We can see that this while loop exists for as many files as we input in the command line argument. So long as they are separated by a space and contain no forward slashes, it will be made into a directory.

Loud()
The Loud function runs nearly identically to the Orphan() function and it called by using only the verbose option. It's simplest job is to make directories while also announcing that the directory has been made. Once the directory has been made, it announces that we have created a directory.

Parents()
The Parents() function is called by using the -p or --parents option and is one of the more complex functions and requires the additional argument (alongside argc and argv) the character pointer "dirname" which points to the path of our current directory. We have several variables to instantiate and buffers to build. Our first objective is to copy over the entire connected argument to its own string which we will then parse. We learned we could then call the c function strtok() to split our array on the delimiter of our choice. Since we are emulating a bash shell, we will use the forward slash as our token and populate a new array with the values that we separated from our old array. After we have prepared our arguments, we then establish our variables that will construct another path for the directory we about to add. We first check to make sure that we have a directory to add, and that it is not equal to a forward slash, at which point we would print an error message. We then make the directory, and append its name to the file path. We can then place another directory within that folder as we update our current directory. We increment through until we have no more directories to add.

Duo()
The Duo() function runs nearly identically to the Parents() function and it called by using both the verbose and parent options. It's job is to make directories within directories, while also

announcing that the directory has been made. Once the directory has been made, it announces that we have created a directory for every directory we make.

The most difficult part of implementing this command line function, surprisingly came from building its error handling. While the functionality of the code it above, we felt that it was important to mention the function of the many, many if loops and how much error handling went into building this function. In short: we have accounted for an enormous amount of user errors including the misuse of forward slashes, invalid input directory names, directory duplicates, and invalid option parameters. We built in error handling that ensured only the correct input would derive the correct output. Error messages are nearly identical to that of the original bash shell. This was by far the most difficult, seg fault generating part of developing this function and remove directory, but the most useful. We learned the internal functions of strtok() and argv and were able manipulate them to create safe use.

Rmdir: Remove Directory Function

Definitely the harder of the two directory function, remove directory has a lot of directory movement to make it both intriguing and very challenging. The remove directory function works by passing the two arguments: *int argc* and *char** argv* into the main function. We set an initial command flag that will help us determine what options we are using in our command. From there we call the getenv() function to dynamically determine our file's location. We open our current directory and set a pointer that will be used to help us parse through the directory we are currently in. We are then ready to read in the arguments passed through to us from the *wannabe bash function*. In this program we look to use only the *parents* option when building our directories. We have included the null option in which a user calls the rmdir function and then neglects to provide us with an option or an argument. In this case, we print an error message to inform them that they are missing an operand. We then read in the option input and map our longform option input to the short form so that either is a viable option when making directories. Should the user input an incorrect option, or one that we have not implemented, will print and error message that they have entered an unrecognizable option. Depending on the options they input (or did not input), we set the values located within our command flag and determine which function to call thusly.

Orphan()
The Orphan function is called when we have called to remove a directory with no preceding options. We started the function by passing in the *int argc* and *char** *argv* values as well as a char* dirname as parameters. We initialize several of our directory variables at the beginning of this function due to its complex maneuvering throughout different directories. We first check to make sure that we have no errors in the arguments we received before we begin the function. We first build our current, entire file path using the arguments we have been given. If the directory name we have been given does not exist or is invalid, we alert the user by printing an error message to tell them about their invalid entry. If the directory does exist, then we start to parse it for any files or directories that would reside in such. If the directory is not empty, then it

will not be deleted and message will be printed out for the user stating that the directory is not empty. If the directory is completely empty, then we remove the directory. Using the strdup() we can set the new file path back to the original to reset where are files are being removed from. We then iterate over the rest of the file arguments to remove and more files we wish.

Parents()

The Parent() function is called when we wish to remove a directory as well as all the directories preceding it. We initialize the function by passing in the *int argc* and *char** argv* values as well as a char* dirname as parameters. We initialize several of our counter, directory, and character variables at the beginning of this function due to its even more so complex maneuvering throughout different directories. Similar to the parents function in mkdir(), we call the c function strtok() to split our array on the forward slash delimiter (since we are emulating a bash shell). The forward slash is our token to split our given array. After which, we iterate over the split items and populate a new array. We then build our pathname down the entire list of arguments we received, since we will be starting the the bottom directory. We concatenate our original pathname with a forward slash and the file name and then update our directory location to build the path. As before, we then parse around our current directory to determine if there are any files or directories within. If the directory contains any files or other directories, we about the removal of the entire pathway starting from that specific directory (it does not affect any directory below it, which was really interesting to learn and practice). We alert the user that the directory is not empty and end the program. If the directory is empty, then we remove the directory and update the file path. We update the file path using a copy of the very original file path we used in this program. We can copy it to our variable “dirname” which then resets. After which we update our counters so that we repeat the process but instead of going down to x number of files, we aim for x-1. This way we can start at the bottom of our directory stack and work our way back up again.

Loud()

As a bonus, I also implemented an additional (unlisted) option for this program segment. This code enlists the use of the -v or verbose option. It’s simplest job is to remove directories while also announcing that the directory has been removed. Once the directory has been removed, it announces that we have removed a directory. Despite not being asked for directly, we felt that it was an easy and necessary addition to our shell to balance with the mkdir function.

Duo()

As another bonus function, I added The Duo() function. It runs nearly identically to the Parents() function and it called by using both the verbose and parent options. It’s job is to remove directories within directories, while also announcing that the directory has been removed. Once the directory has been removed, it announces that we have removed a directory for every directory we parse through. Again, we felt that it was an easy and necessary addition to our shell to balance with the mkdir function.

Just as with the make directory function, the most difficult part of building remove directory came from building its error handling. In this error handling, we have accounted for an enormous amount of user errors including:
Error messages are nearly identical to that of the original bash shell. The attempt to fix the seg fault that came from attempting to delete a non existent directory almost remained fruitless since it was the hardest segment of code for some of us, but it is fixed, and we learned a lot from debugging this function.

LS: List Program

The 'ls' program was by far the most complicated program to write. Ls took the longest and was the last one we finished. After writing whoami which was our first program that we wrote, we switched our attention on getting argument parsing implemented so that we could start working on ls. We first found a function called getopt() which seemed perfect until we realized that it did not accept the long form of options like '--group-directories-first' or '--all'. We found its sibling function, getopt_long, which did take accept long form option but at a cost: difficulty to implement. Getting it to work properly took time as it almost seems it is intended to only be set up in one way. After finally getting the ordering down pat we found that we needed a way to store the flags that have been set. We decided on an array of size 8 that initialized to all zeros and once a flag was set then that corresponding zero in the array was set to 1. We need needed to know grab the directory/directories that followed the options, if any, so the if-else statement at the bottom of main handles this. If a directory was given then it is grabbed and 'opened' with the opendir() function which creates a dir object that contains info on all of the files in that directory. If a directory is not supplied then the present working directory is used as per the normal functionality of the ls program. The option flags, current_dir as a dir, and a string of the inputted directory which is either the PWD or the inputted directory. The stat function is used to test the inputted directory to see if that file exists. If the stat function returns a -1 then that file does not exist and errno is set and a perror is displayed.

s_perm()
We could not find a system function that outputs the file permissions in the format that 'ls' uses with its '-l' option. We found a reference from a function that Solaris used called 'sperm()' that fills an array with the permissions in a human readable form. 's_perm()' is a function we wrote to work with linux. It takes a mode_t object as a parameter which stores the permissions of a file. It then bitwise ands the mode with the permission constant for a single permission and does this with each of the 9 file permissions. If bitwise and produces the permission constant for that permission then that file has that permission and in the appropriate index in the array, that permission is store as either a 'r' for read, 'w' for write, or 'x' for execute. If that file does not have the permission then a '-' is stored in the array at that index.

reverse_array()
This function takes the array of dirent structs named entries and simply reverses the order in of the structs in the array. This function is function is called with the '-r', the '-S', and the '-t' options.

insertionSortSize() & insertionSortTime()
Finding a way to sort the array of dirent structs was difficult at first since the struct did not contain any statistics on the files themselves. We found that stat.h had a function called stat what when given a stat buffer for a struct and a directory it would fill the struct buffer with statistics on that given directory or file. This seemed to be the only good way of getting this information so they next step was to find how to do this. We starting looking into sorting algorithms and after deciding that insertion sort, even being one of the slowest, required the least amount of complexity to code. Quicksort required that the array be partitioned and then re-combined, same with merge sort so they were off the time. Insertion sort could be written in a single function and it easily allowed for the comparisons to be made with the stat statistics and the swapping took place with the entries themselves. Both insertionSortSize() and insertionSortTime() do the same thing aside from comparing the st_size integers and the st_mtime integers, respectively.

sortAlpha()
Originally we used the same insertion sort algorithm for alphabetically sorting the files in the entries array. We only found out that this needed to be done when it was discovered that when ran on MacOS without sorting that files were already sorted by file name but when we tested it on linux this was not the case. We found, however, that the insertion sort method we first devised only looked at the first character's ACSII value so we needed another way. We found the strcmp function which made this much easier. It just looks at the strings in their entirety and if the returned value is greater than 1 then the two adjacent values need to be swapped. This comparison is done with all filenames in the array.

groupDirectoriesFirst()
This function is called if the --group-directories-first flag is set. It looks at all of the files in a given directory and adds files to a new a array and directories to another array. The new directories array and then added to the the beginning of a new array of dirent structs and then the new objects in the new array of files are added to this new array of dirent structs. Lastly, this new array of dirent structs is used to overwrite the order of the structs in the original entries array so that all directories come before all files in the array in the same respective order than they were when the original array was passed in.

listLong()
This function is called if the -l flag is set. This was created in a refactoring of the code base to help move code out of main so that the recursive function could be made. Before its creation, the code base supported either the -l flag or the -R flag but not both without copying and pasting the code from -l to the -R area of the code. This function prints out stats on the files in the current directory and this stat information comes from the stat function in stat.h

ls()

This function contains the bulk of the code and is also the byproduct of the refactor that took place to implement the -R option. This function checks what option flags were set and then carries out the corresponding actions necessary. For the the '-d' option section of the code, if no directory is supplied then '.' is printed, if a directory is supplied then that supplied directory is printed. The '-a' option section of the code reads through all of the files in the given directory and adds them to the dirent struct array named entries. The -S and -t option sections of the code sends the entries array, the counter that has the number of file structs in the entries array, and supplied directory to the insertionSort algorithms. Since these algorithms order the entries from least to greatest you then must call the reverse_array function with the entries array which reverses the order of the structs in the array. The -r option section of the code just simple sends the entries array into the reverse_array function to reverse the order of the structs in the array. The --g section of the code sends the entries array to the groupDirsFirst function which just sorts the structs in the array so that directories come become files. The -l section of the code sends the entries array to the listLong function as well as the counter that contains the number of structs in the array and the inputted the directory so that the output can be formatted to long option specifications. The last section of code in the ls() function is the -R option section. This function first prints out all of the files in the entries array in order then iterates back through the array and finds the directories where it then prints out that directory's name and then sends the array that contains the option flags, the dir object of the new current directory which is the old current directory with the directory's name appended to it, and the string representation of the new directory into the ls() function to do that same thing until there are no more directories and files to print out.