

Improvement to File System Navigation in the UNIX Terminal
Group 92

Prof. Michael Barbeau
COMP 3000 Final Project
April 13th 2018
Carleton University

Ted Kachulis / Sam Wong / Nicholas Lewanowicz

1.0 Introduction

1.1 - Context

While using a terminal console in the UNIX system, there are three main activities the average user will spend their time doing - edit documents, compile/run software, and navigate their file system. Our project addresses the latter of the three, the topic of navigating the file system in UNIX. More specifically, we will be adding functionality to the `'ls'` command, short for *list*, from the *GNU Coreutils* package.

Though `'ls'` has modifiers with arguments allowing for different display models of information, our team finds that they are lacking in the elegance required to properly model the file system in a sufficiently aesthetic fashion. You can read about the Linux command line and `'ls'` documentation in *'The Linux Command Line'* by William Shotts.

1.2 - Problem Statement

As previously mentioned, our team observes an insufficient implementation of the `ls` command. While many users use `'ls'` as a main tool for navigating and searching through directories, we find that there is not enough focus on an oversight of the file system, leaving the file structure surrounding your current working directory (CWD) unknown.

The current solution attempting to provide an oversight of directories is the addition of the argument `-R`, meaning the `'ls'` command is `'ls -R'` which calls `'ls'`

on the CWD, as well as recursively on any subdirectories it may have. We find this insufficient for two main reasons. One, there is no input parameter for the depth of subdirectories which we would like it to display. Secondly, '`ls -R`' lists directories one at a time rather than in a clearly readable format, such as a tree layout.

Our first goal is to add an input parameter to limit the depth of this tree according to the user's requirement. We hope this will improve both ease of use as well as limit the amount of system memory and work that is to be allocated to this job - why should the system recurse through every folder if the user's goal is only to search through subdirectories one step down?

Our second goal is to ensure that when displayed, the file structure contents are not shown directory by directory, but instead in a tree format resembling an expanded cascading list. We hope to explore the color tools the command line offers in hopes of adding to the improved aesthetic of the command.

1.3 - Result

Our implementation of the extended list flag for the core utility '`ls`' has accomplished its goal of displaying a clearer and more aesthetically pleasing tree directory printout, this being done all through modification of the 5000 line '`ls`' coreutil file itself and not just a aliased command. Through this we have added a depth flag which will determine the number of subdirectories to print out. We have also been able to colorize the file by type and organize the files by name as well as not display any directory errors the '`ls -R`' command usually prints out.

Our implementation has also become very modular as we have been able to create a install script which will work on any Ubuntu 16.04 OS alsong as it has the right dependencies. The project is also PATH based in which the installation

will only affect terminals with their environment PATH variable set too "\$HOME/coreutils/bin:\$PATH", this insures your system can be saved incase of a fatal error.

We believe we have solved the problem of having an unreadable recursive list by creating a whole new flag which displays our directories in an aesthetically pleasing tree format. This way when you want to find a file in a subdirectory it becomes instantly clearer which path the file lies in, instead of having to parse through the lines of text printed from the original 'ls -R' format.

We achieved this by first studying and understanding the 'ls' function and how it works internally, we also interpreted how 'ls' handles flags and uses them to modify the output of the command.

After understanding the 'ls' core utility we then proceeded to modify and create our own unique flag and source incorporate it into the 'coreutils' library itself.

1.4 - Outline

The rest of this report will detail the background, process, and results of our efforts. The structure will be as follows; section 2.0 presents background information relevant to 'ls' and 'ls -R'. Section 3 debriefs the reader on the results obtained, and section 4 continues to describe barriers to the development of the improvement, and new findings since our original project scope. We will conclude in section 5 with our closing statements on the project, and section 6 with supporting materials used in the report.

The structure, sections one through six, is as follows on the next page.

Structure:

1.0 - Introduction

1.1 - Context

1.2 - Problem Statement

1.3 - Result

1.4 - Outline

2.0 - Background Information

2.1 - The 'ls' Command

2.2 - The 'ls -R' Command

3.0 - Results: Project

3.1 - Walkthrough & Installation

3.2 - Functionality & Implementation

4.0 - Results: Evaluation Debrief

4.1 - Developer Review

4.2 - Issues & Findings

4.3 - Ease of Use Study

5.0 - Conclusion

5.1 - Goal Review

5.2 - Closing Statements

6.0 - References & Appendices

6.1 - References

6.2 - Appendices

2.0 Background Information

2.1 - The `'ls'` Command

The `'ls'` command, abbreviated from the word list, is specified by POSIX and is one of the most standard commands in the GNU Core Utilities package. When invoked the command lists all information about the files in the specified directory.

The utility first appeared in the original AT&T version of Unix and was derived from similar commands named `'list'` from the Multics operating systems. Even back in 1969, during the creation of Multics, `ls` was rooted as a core command, the Multics OS was one of the first to provide hierarchical file systems with symbolic links and removable devices and thus users needed efficient means to traverse data.

Nowadays the `'ls'` command is used by most if not all terminal users running Unix based Operating systems. By default, without any flags, the `ls` command will print out the files from your CWD or current working directory, if you have just opened your terminal the PATH would be under your `/home/$USER` directory.

The command has around 60 flags that can be used to change your output, this can range from changing the colour of directories printed, hyperlinking certain file names, or displaying block-sizes of files.

2.2 - The `'ls -R'` Command

The `-R` flag in particular is used to recursively print out subdirectories. If you have ever wanted to be able to view not just the toplevel of your directory but also

what is inside the child directories of your path then you would use this flag.

When this flag is initialized the ls command creates a data structure that detects and directory cycles and uses recursion to traverse a directory structure, that is why you will most likely view the output as messy and unorganized. Most use cases for this flag are in when you are in lower level directories with fewer files or when this command is piped into another for better readability.

3.0 Result

3.1 - Walkthrough & Installation


Brief

Within the timeframe of this project we were able to implement a feature flag in the coreutils module for 'ls' which accomplishes the stated goal functionality. The majority of our time was spent analyzing the understandably large ls.c file which can be broken down into several sections including the variable and enumerate values, generic functions that are run when flags are present, follow up functions to execute code for particular flags, and finally help documentation for the end user describing the flag functionality.

Through understanding the above we were able to see where and how we can write our fairly straight forward code within this seemingly complex utility. It's important to mention that due to the modularity of the core utils we were able to add all of our changes by only touching the /src/ls.c file ultimately reducing difficulty debugging.

Installation

We wanted this functionality to be easy and straightforward to install and test, so to accomplish this we reduced the already minimal 3 command install with a source script. This allows a user to simply clone our repo and execute:

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. It displays three lines of terminal commands in a light green monospace font.

```
> git clone https://github.com/NickLewanowicz/extended-ls.git  
> cd extended-ls  
> source ./install
```


The existing install instructions for coreutils requires you to manually rebuild the utilities, modify a configuration script, set your path variable, and execute the installation script. Our simplified installation will reduce the overhead associated with using our utility and increase the likelihood we get feedback on the modification.

3.2 - Functionality & Implementation Details

As discussed in section 2 the ls command natively will output the files and folders within the PWD of the terminal, though there are cases where you are looking for a file that's nested in a directory within PWD.

The screenshots and notes in the pages to come outline a typical use case, showcasing the functional details of our flag in comparison to ls -R.

All images used are created and used under our own ownership.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The prompt is 'James@COMP3000VM:~/Documents/Primary_dir\$' and the command 'pwd' has been entered. The output is '/home/James/Documents/Primary_dir'.

Fig.1 Above: "pwd command will list our current directory as ~/Primary_dir"

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The prompt is 'James@COMP3000VM:~/Documents/Primary_dir\$' and the command 'ls' has been entered. The output lists 'Secondary_dir_1', 'Secondary_dir_2', 'Secondary_dir_3', 'Secondary_file_1.txt', 'Secondary_file_2.txt', and 'Secondary_file_3.txt'. The prompt is repeated at the bottom.

Fig.2 Above: "if we are looking for some hidden/nested file default ls isn't helpful"

```
James@COMP3000VM:~/Documents/Primary_dir$ ls -e --reach=1
.:
|>Secondary_dir_1
|>Secondary_dir_2
|>Secondary_dir_3
| Secondary_file_1.txt
| Secondary_file_2.txt
| Secondary_file_3.txt
>
```

Fig.3 Above: “using `ls -e --reach=n` we are able to specify a depth and recurse our listing”

```
James@COMP3000VM:~/Documents/Primary_dir$ ls -e --reach=2
.:
|>Secondary_dir_1
|>Secondary_dir_2
|>Secondary_dir_3
| Secondary_file_1.txt
| Secondary_file_2.txt
| Secondary_file_3.txt

./Secondary_dir_2:
| Tertiary_file_2
|>Tertiary_dir_2

./Secondary_dir_3:
| Tertiary_file_3
|>Tertiary_dir_3

./Secondary_dir_1:
| Tertiary_file_1
|>Tertiary_dir_1

James@COMP3000VM:~/Documents/Primary_dir$
```

Fig.4 Above: “using `ls -e --reach=2` we can look further still not finding the file we want”

```
James@COMP3000VM:~/Documents/Primary_dir$ ls -e --reach=3
.:
|>Secondary_dir_1
|>Secondary_dir_2
|>Secondary_dir_3
| Secondary_file_1.txt
| Secondary_file_2.txt
| Secondary_file_3.txt

./Secondary_dir_2:
| Tertiary_file_2
|>Tertiary_dir_2

./Secondary_dir_2/Tertiary_dir_2:

./Secondary_dir_3:
| Tertiary_file_3
|>Tertiary_dir_3

./Secondary_dir_3/Tertiary_dir_3:
| Hidden_file.txt

./Secondary_dir_3/Tertiary_dir_3:
| Hidden_file.txt

./Secondary_dir_1:
| Tertiary_file_1
|>Tertiary_dir_1

James@COMP3000VM:~/Documents/Primary_dir$
```

Fig.5 Above: “using `ls -e --reach=3` we finally found our `Hidden_file.txt`”

```
James@COMP3000VM:~/Documents/Primary_dir$ ls -R
.:
Secondary_dir_1 Secondary_dir_2 Secondary_dir_3 Secondary_file_1.txt
Secondary_file_2.txt Secondary_file_3.txt

./Secondary_dir_2:
Tertiary_file_2 Tertiary_dir_2

./Secondary_dir_2/Tertiary_dir_2:

./Secondary_dir_3:
Tertiary_file_3 Tertiary_dir_3

./Secondary_dir_3/Tertiary_dir_3:
Hidden_file.txt
ls: cannot open directory './Secondary_dir_3/Tertiary_dir_3/Hidden_file.txt': Not a
directory

./Secondary_dir_3/Tertiary_dir_3:
Hidden_file.txt
ls: cannot open directory './Secondary_dir_3/Tertiary_dir_3/Hidden_file.txt': Not a
directory

./Secondary_dir_1:
Tertiary_file_1 Tertiary_dir_1

./Secondary_dir_1/Tertiary_dir_1:

./Secondary_dir_1/Tertiary_dir_1:
student@COMPBase:~/Documents/Primary_dir$
```

Fig.6 Above: “In comparison using `ls -R` we get a much different output which is both uniform in color, lines that overflow, and even returns error statements”

Implementation Details

The feature flag introduced accepts flag `-e` and `--reach=n` where `n` is some value between 0 and 4 exclusively. This was accomplished by creating bool value mapped against the presence of the `-e` flag and a enum mapping against the value of `n` for the `--reach` flag.

Using the `-e` bool and input mapping from `--reach` there is a select case within `ls.c` that is used by all the existing flags to call specific functions related to the operation of the particular case, for instance we use this as an opportunity to enable our custom formatting by setting `"e_format"` to true which will be checked in another select case and is responsible for the custom characters that appear adjacent to the files and directories. The most difficult part of this was understanding and modifying the `"cwd_n_used"` which is responsible for collating all the files and directories within a given file and outputting the array of objects.

Ultimately our code shared many similarities with the existing `-R` implementation but with the key difference that we allow a custom break clause when there is a reach value specified.

Finally, no feature flag would be complete without a `-help` entry, we were able to enter this along with all the existing help dialogues fairly easily, this will be essential to improve the usability of our flag.

4.0 Results: Evaluation Debrief

4.1 - Developer Review

Overall this experience from a development point of view was enlightening and valuable. The complexity of simple commands can be surprising and taken for granted, even a trivial feature such as we implemented here with limited novel use took a fair amount of ramp up time and polish to get to the state it is now. This is with full understanding that we are not yet complete with our flag and concessions were made with respect to the breadth of input options, in our case we limited reach to 3 in lieu of a flag to limit the number of items that would appear in each sub directory. That being said the work done is still a novel improvement on the existing ls -R which is a great accomplishment nonetheless.

4.2 - Issues & Findings

After concluding our modifications we looked back in retrospect on what were the top issues to overcome. One of the still outstanding issues was the inline tree format, we ended up using a sequential tree format but having another flag for inline would have been great to allow the user to see directly with a directory what files are within, unfortunately this was problematic and due to the less than clear documentation for coreUtils we were unable to complete this option. Even with large amounts of upfront investment in researching ls and core utilities as a whole we were still unsuccessful at finding all the information we needed which would be a issue for anybody who is looking at this open source project for the first time.

4.3 - Ease of Use Study

Nielsen's Heuristics

1. Visibility of system status

The system status is not quite relevant here - however, in the terminal the CWD path is always shown, and previous commands are visible after they are entered. Users should always be informed about what is going on, where they are in the file system, and what information they've commanded to see, through appropriate feedback within reasonable time.

2. Match between system and the real world

The system command *'ls'* uses our *'-e'* flag to represent *'extend'* which is exactly related to its actual in practise functionality. The long term of this command would be *'list -extend'* which translates well to the english language rather than purely technical. Our modifier *'reach'* is also somewhat self explanatory; it represents the depth of which the *ls* recursive call will be made - in other words, how far the calls will *'reach'*.

It is important to Nielsen's Heuristics for the system to speak the users language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. We know that the terminal system, and dealing with system calls, this isn't something that is commonly known as familiar or human language. However, we think that we did hit this mark by ensuring any additional functionality is explained in the command itself, ex: above *'-e'* flag and *'reach'* explanation.

3. User control and freedom

A vital part of usability is for the user to be able to recover from mistakes when using the system. For example, having the ability to exit, or undo/redo.

The terminal system allows the up arrow key for redoing commands, and a simple 'clear' or enter command will essentially undo the call results. During the system call, it can be cancelled using the emergency exit signal command.

4. Consistency and standards

We stayed consistent to naming and structure conventions in the CoreUtils file throughout our modifications to the code. Furthermore, it is well documented and organized/formatted within the file.

5. Error prevention

We purposefully made a default value for '*reach*' as explained earlier in this report (section 1.3, 3.0). This was set in place for the purpose of preventing a massive tree of recursion.

We believe this heuristic was not met by '*ls -R*', which is why we added both a parameter for reach, and a default value of three.

6. Recognition rather than recall

Not able to make options always visible to user since this is an inline command line system. However, with '*ls*' help, and the naming conventions relative to the use case (extended, reach) we hope that the user is able to recognize which commands are appropriate when trying to extend the folder structure visibility.

7. Flexibility and efficiency of use

The functionality we added could be considered an accelerator which speeds up the interaction for the user with the system. It also allows the user to tailor their action depending on the requirement, for example, the depth of reach.

8. Aesthetic and minimalist design

Design is consistent with the rest of the terminal and system call structure. Use of color to emphasize file structure visibility. Nothing too interesting to mention

regarding visual design, as we mostly just remained consistent with what is and can be done, in the terminal.

9. Help users recognize, diagnose, and recover from errors

Error messages are displayed as compiler errors (if any) but have no situation where the system should resort to this. There are handlers for incorrect input, usually just redirecting the system to the default reach on the CWD.

10. Help and documentation

Though it is better if the system can be used without documentation, we felt compelled to include it for the simple fact that if we hadn't, the functionality may never be seen or noticed.

The code modifications in the CoreUtils package are heavily documented inline, as well as explained in this document. The report as a whole may be considered help and/or documentation.

5.0 Conclusion

5.1 - Goal Review

In review, our first goal was to add an input parameter to limit the depth of the recursive calls according to the user's use requirement. Our other goal was to ensure that when displayed, the file structure contents are shown nice, and in a non convoluted way. We hoped to use color, and tools to improve the overall aesthetic and user experience when navigating the filesystem.

Below is an outline of successes and failures after our goal review:

Successes

- Showing nested files with depth parameter option
- Used color tools to differentiate subfolders/files
- Added functionality to GNU Coreutils
- Improved file system navigation
- Made available for installation

Failures

- Display nice, but would have preferred a tree format
- Still making recursive calls which can be taxing on system

5.2 - Closing Statements

We feel that we managed to achieve our goals in a satisfactory way. Improving the functionality of 'ls' and making the user experience better while using the command line is something we can confidently say we did, and are proud of. In future, we plan to build a string message using separate ls calls so that we can display as tree rather than subfolders on newline.

6.0 References and Appendices

6.1 - References

Using information from the GNU CoreUtils documentation, found online in detail at <https://www.gnu.org/doc/doc.html>, and course notes from Michael Barbeau's winter 2018 COMP 3000 class.

6.2 - Appendices

Screenshots of Code

```
for (i = cwd_n_used; i != 0; i-- )
{
    struct fileinfo *f = sorted_file[i];
    if (is_directory (f) && (! ignore_dot_and_dot_dot
        || ! basename_is_dot_or_dotdot (f->name)))
    {
        if (!dirname || f->name[0] == '/')
            queue_directory (f->name, f->linkname, command_line_arg);
        else
        {
            int s = 0;
            if(e_flag){
                char* c = strchr(dirname, '/');
                while(c != NULL){
                    s++;
                    c = strchr(c+1, '/');
                }
            }
            if(!e_flag || s < er_input){
                char *name = file_name_concat (dirname, f->name, NULL);
                queue_directory (name, f->linkname, command_line_arg);
                free (name);
            }
        }
        if (f->filetype == arg_directory)
            free_ent (f);
    }
}
```

```

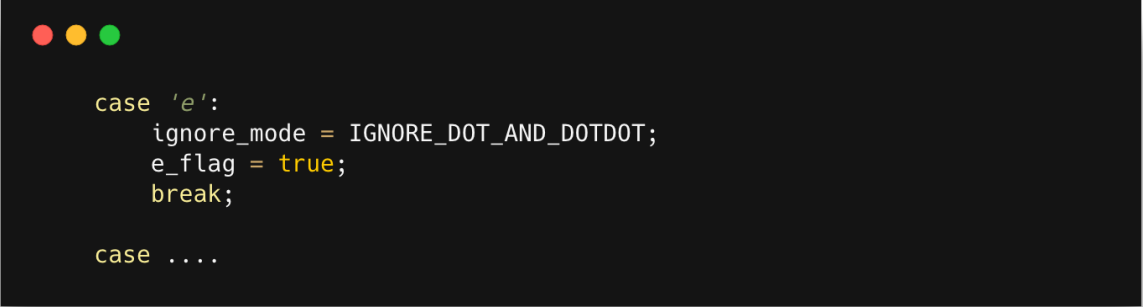
case e_format:
    for (i = 0; i < cwd_n_used; i++)
    {
        putchar( '\t' );
        if(((struct fileinfo*) sorted_file[i])->filetype == normal)
        {
            putchar( '/' );
            putchar( ' ' );
            print_file_name_and_frills (sorted_file[i], 0);
        }
        if(((struct fileinfo*) sorted_file[i])->filetype == directory)
        {
            set_normal_color ();
            putchar( '/' );
            putchar( '>' );
            print_file_name_and_frills (sorted_file[i], 0);
        }
        putchar ( '\n' );
    }
    break;
....

```

```

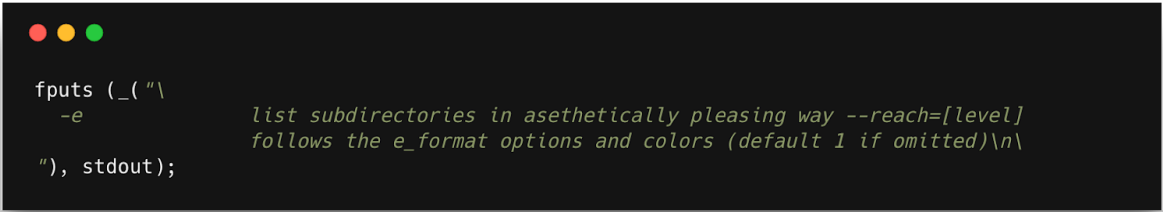
static struct option const long_options[] =
{
    {"extended", no_argument, NULL, 'e'},
    {"reach", optional_argument, NULL, 'E'},
    ...
    ...
}

```



```
case 'e':
    ignore_mode = IGNORE_DOT_AND_DOTDOT;
    e_flag = true;
    break;

case ....
```



```
fputs (_("\n
    -e          list subdirectories in aesthetically pleasing way --reach=[level]
                follows the e_format options and colors (default 1 if omitted)\n\n
"), stdout);
```