Project 2

Unix Utilities

DUE MON 2015-11-09 @ 11:55 PM

Description has been updated; please reread this document.

CSCI 1730 – Fall 2015

Project Description

For this project, you are going to implement a collection of basic Unix utilities using low-level system calls. This is a natural extension to Breakout/Lab 09 where you implemented the `chmod` utility from scratch. You are NOT allowed to use the following system calls in any of your implementations: `fork`, `exec`, and `system` (or related functions). Here is the list of utilities that you must implement:

1. `./chmod`

   This is the same implementation as described in Breakout/Lab 09.

2. `./mkdir [-p] [-m OCTAL-MODE] DIRECTORY-NAME`

   The `mkdir` utility creates the directories named as operands, in the order specified, using mode `0755` (see `mkdir(2)`). The options are as follows:

   `-m OCTAL-MODE`

   Set the file permission bits of the final created directory to the specified mode. The mode argument should be specified using octal notation.

   `-p`

   Create intermediate directories as required. If this option is not specified, the full path prefix of the `DIRECTORY-NAME` must already exist. On the other hand, with this option specified, no error will be reported if a directory given as an operand already exists. Intermediate directories are created with permission bits of `0755`.

3. `./cp [-R] SOURCE-FILE TARGET-FILE`
   `./cp [-R] SOURCE-FILE TARGET-DIR/`

   In the first synopsis form, the `cp` utility copies the contents of the `SOURCE-FILE` to the `TARGET-FILE`. In the second synopsis form, the contents of the named `SOURCE-FILE` is copied to the destination `TARGET-DIR`. The names of the files themselves are not changed. If `cp` detects an attempt to copy a file to itself, the copy will fail. The following options are available:

   `-R`

   If `SOURCE-FILE` designates a directory, `cp` copies the directory and the entire subtree connected at that point. If the `SOURCE-FILE` ends in a `/`, the contents of the directory are copied rather than the directory itself. This option also causes symbolic links to be copied, rather than indirected through, and for `cp` to create special files rather than copying them as normal files. Created directories have the same mode as the corresponding source directory.

4. `./mv SOURCE-FILE TARGET-FILE`
   `./mv SOURCE-FILE TARGET-DIR/`

   In its first form, the `mv` utility renames the file named by the `SOURCE-FILE` operand to the destination path named by the `TARGET-FILE` operand (see `rename(2)`). This form is assumed when the last operand does not name an already existing directory. In its second form, `mv` moves each file named by the `SOURCE-FILE` operand to a destination file in the existing directory named by the `TARGET-DIR` operand. The destination path for each operand is the pathname produced by the concatenation of the last operand, a slash, and the final pathname component of the named file.

5. `./ls [FILE]`

   This implementation should produce output that matches GNU's `ls -l` utility. If `FILE` is not specified, then the present working directory is assumed.

6. `./cat FILE ...`

   The `cat` utility reads files sequentially, writing them to the standard output. The file operands are processed in command-line order.

7. `./rm [-R] FILE ...`

   The `rm` utility *attempts* to remove the non-directory type files specified on the command line (see `unlink(2)`). If the permissions of the file do not permit writing, and the standard input device is a terminal, the user is prompted (on the standard error output) for confirmation. The following options are available:

   `-R`

   Attempt to remove the file hierarchy rooted in each file argument. If the user does not respond affirmatively, the file hierarchy rooted in that directory is skipped.

8. `./ln [-S] OLD-FILE NEW-FILE`

   The `ln` utility creates a new link (also known as a hard link) to an existing file (see `link(2)`, `symlink(2)`). The following options are available:

   `-S`

   Instead of creating a hard link, a symbolic link is created instead (see `symlink(7)` for information on symbolic links).

9. `./penv`

   The `penv` utility prints all of the currently set environmental variables to standard output.

10. `./stat FILE`

    The `stat` utility displays the status of a file (see `stat(2)`). The output of this utility is the same as the GNU implementation (see `stat(1)`).

11. `./pwd`

    The `pwd` utility displays the present working directory (see `getcwd(3)`).

12. `./kill [-s SIGNAL] PID`

    The `kill` utility sends the specified signal to the specified process or process group `PID` (see `kill(2)`). If no signal is specified, the `SIGTERM` signal is sent. The `SIGTERM` signal will kill processes which do not catch this signal. For other processes, it may be necessary to use the `SIGKILL` signal, since this signal cannot be caught. If `PID` is positive, then the signal is sent to the process with the ID specified by `PID`. If `PID` equals 0, then the signal is sent to every process in the current process group. If `PID` equals -1, then the signal is sent to every process for which the utility has permission to send signals to, except for process 1 (init). If `PID` is less than -1, then the signal is sent to every process in the process group whose ID is `-PID`. The following options are available:

    `-s SIGNAL`

    Instead of sending `SIGTERM`, the specified signal is sent instead. `SIGNAL` can be provided as a signal number or a constant (e.g., `SIGTERM`).

Notes

- **User Input:** You may **NOT** assume valid user input.
- **Error Handling:** If a system call results in an error, then your implementation should display the error using `perror` (available in ⟨`cstdio`⟩). In general, if an error occurs (e.g., invalid input), then display a message to the user indicating what the error is and exit with status `EXIT_FAILURE` (available in ⟨`cstdlib`⟩).
- **Executable Names:** Make sure that your resulting executables have the same names as the ones presented above.
- **References:** You may find the following manual pages to be an interesting read: `intro(1)`, `intro(2)`, and `intro(3)`.

1 C++ Code

Make sure that all of your files are in a directory called `LastName-FirstName-p2`, where `LastName` and `FirstName` are replaced with your actual last and first names, respectively.

1.1 `Makefile` File

You need to include a `Makefile`. Your `Makefile` needs to compile and link separately. Make sure that your `.cpp` files compile to individual `.o` files. The resulting executables should match those presented in the Project Description.

1.2 `README` File

Make sure to include a `README` file that includes the following information presented in a reasonably formatted way:

- Your Name and 810/811#
- Instructions on how to compile and run your programs.

NOTE: Try to make sure that each line in your `README` file does not exceed 80 characters. Do not assume line-wrapping. Please manually insert a line break if a line exceeds 80 characters.

1.3 Compiler Warnings

Since you should be compiling with both the `-Wall` and `pedantic-error` options, your code is expected to compile without `g++` issuing any warnings. For this project, compiling without warnings will be one or more of the test cases.

1.4 Memory Leaks

You are expected to ensure that your implementation does not result in any memory leaks. We will test for memory leaks using the `valgrind` utility. For this project, having no memory leaks will be one or more of the test cases.

2 Submission

Before the due date, you need to submit your code via Nike. Make sure your work is on `nike.cs.uga.edu` in a directory called `LastName-FirstName-p2`. From within the parent directory, execute the following command:

```
$ submit LastName-FirstName-p2 cs1730a
```

It is also a good idea to email a copy to yourself. To do this, simply execute the following command, replacing the email address with your email address:

```
$ tar zcvf LastName-FirstName-p2.tar.gz LastName-FirstName-p2
$ mutt -s "p2" -a LastName-FirstName-p2.tar.gz -- your@email.com < /dev/null
```