Project 3

Unix Utility Collection

Pair Programming Project

DUE MON 2015-11-07 @ 11:55 PM

CSCI 1730 – Fall 2016

Project Description

For this project, you are tasked with implementing a collection of basic Unix utilities using low-level system calls. This is a natural extension to Breakout/Labs 08–12 where you implement some of the more involved utilities from scratch. Doug McIlroy, the inventor of Unix pipes and one of the founders of the Unix tradition states that one of the fundamental parts of the "Unix philosophy" is that you should write a program to do one thing and do it well. The ten (10) utility programs presented in this project embody this philosophy.

Not all utilities are the same difficulty. Please manage your time wisely.

1 Functional Requirements (80 points)

The functional requirements presented below describe the external characteristics and facilities of the utilities that are of importance to application developers, rather than the internal construction techniques employed to achieve these capabilities. These requirements are derived from a subset of POSIX.1-2008 (also known as IEEE Std 1003.1™-2008, The Open Group Technical Standard Base Specifications, Issue 7, and ISO/IEC 9945). Unless otherwise specified, your utility programs should match the output of the existing Unix utilities on Nike. Here is the list of utilities that you must implement:

1. `./cal [[month] year]`

   (10 points) The `cal` utility shall write a calendar to standard output using the Julian calendar for dates from January 1, 1 through September 2, 1752 and the Gregorian calendar for dates from September 14, 1752 through December 31, 9999 as though the Gregorian calendar had been adopted on September 14, 1752 (see Calendar (New Style) Act 1750 on Wikipedia).

   - `month`: Specify the month to be displayed, represented as a decimal integer from 1 (January) to 12 (December). The default shall be the current month.
   - `year`: Specify the year for which the calendar is displayed, represented as a decimal integer from 1 to 9999. The default shall be the current year.

2. `./date [+format]`

   (10 points) The `date` utility shall write the date and time to standard output. By default, the current date and time shall be written. If an operand beginning with '+' is specified, the output format of date shall be controlled by the conversion specifications and other text in the operand.

   - `+format`: When the format is specified, each conversion specifier shall be replaced in the standard output by its corresponding value. All other characters shall be copied to the output without change. The output shall always be terminated with a newline. Conversion specifications are introduced by a '%' character, and terminated by a conversion specifier character, and are replaced as described in `strftime(3)`.

3. `./head [-n number] [file...]`

(5 points) The `head` utility shall copy its input files to the standard output, ending the output for each file at a designated point. Copying shall end at the point in each input file indicated by the `-n` number option. The option-argument number shall be counted in units of lines. If no `file` operand is specified, or when `file` is `-`, then standard input is assumed.

- `-n number`: The first `number` lines of each input file shall be copied to standard output. The application shall ensure that the number option-argument is a positive decimal integer. When a file contains less than `number` lines, it shall be copied to standard output in its entirety. This shall not be an error. If no options are specified, `head` shall act as if `-n 10` had been specified.

4. `./ln [-s] target-file link-file`

(5 points) The `ln` utility creates a new link (also known as a hard link) to an existing target 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)`).

5. `./ls [-al] [file...]`

(10 points) For each operand that names a file of a type other than directory or symbolic link to a directory, `ls` shall write the name of the file as well as any requested, associated information. For each operand that names a file of type directory, `ls` shall write the names of files contained within the directory as well as any requested, associated information. If no operands are specified, `ls` shall write the contents of the current directory. If more than one operand is specified, `ls` shall write non-directory operands first; it shall sort directory and non-directory operands separately in ascending lexicographic order. The `ls` utility shall detect infinite loops; that is, entering a previously visited directory that is an ancestor of the last file encountered. When it detects an infinite loop, `ls` shall write a diagnostic message to standard error and shall either recover its position in the hierarchy or terminate.

- `-a`: Write out all directory entries, including those whose names begin with a period (`'.'`). Entries beginning with a period shall not be written out unless explicitly referenced, the `-a` option is supplied, or an implementation-defined condition shall cause them to be written.
- `-l`: (The letter ell.) Write out in long format (see the output of the existing `ls` utility).

6. `./mkdir [-p] [-m mode] dir...`

(10 points) The `mkdir` utility creates the directories named as operands, in the order specified, using mode `0755` (see `mkdir(2)`).

`-m mode`: Set the file permission bits of the final created directory to the specified mode. The mode argument should be specified using octal notation. You may or may not deed to modify the *umask* for the calling process.

`-p`: Create intermediate directories as required. If this option is not specified, the full path prefix of the `dir` 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`. You may need to modify the *umask* of the calling process using `umask(2)` to set the appropriate mode.

7. `./env`

(5 points) The `env` utility prints all of the currently set environmental variables to standard output.

8. `./tail [-f] [ -c number | -n number] [file]`

   (10 points) The `tail` utility shall copy its input file to the standard output beginning at a designated place. Copying shall begin at the point in the file indicated by the `-c number` or `-n number` options. The option-argument `number` shall be counted in units of lines or bytes, according to the options `-n` and `-c`, respectively. Both line and byte counts start from 1. Tails relative to the end of the file may be saved in an internal buffer, and thus may be limited in length. Such a buffer, if any, shall be no smaller than `2048*10` bytes. If no `file` operand is specified, or when `file` is `-`, then standard input is assumed.

   - `-c number`: The application shall ensure that the number option-argument is a decimal integer whose sign affects the location in the file, measured in bytes, to begin the copying relative to the beginning of the file.
   - `-f`: If the input file is a regular file or if the file operand specifies a FIFO, do not terminate after the last line of the input file has been copied, but read and copy further bytes from the input file when they become available. If no file operand is specified and standard input is a pipe, the `-f` option shall be ignored. If the input file is not a FIFO, pipe, or regular file, it is unspecified whether or not the `-f` option shall be ignored.
   - `-n number`: This option shall be equivalent to `-c number`, except the starting location in the file shall be measured in lines instead of bytes. If neither `-c` nor `-n` is specified, `-n 10` shall be assumed.

9. `./true`
   `./false`

   (5 points) The `true` utility shall return with exit code `EXIT_SUCCESS`. The `false` utility shall return with exit code `EXIT_FAILURE`.

10. `./wc [-c | -m] [-lw] [file...]`

    (10 points) The `wc` utility shall read one or more input files and, by default, write the number of newlines, words, and bytes contained in each input file to the standard output. The utility also shall write a total count for all named files, if more than one input file is specified. The `wc` utility shall consider a word to be a non-zero-length string of characters delimited by white space. If no `file` operand is specified, or when `file` is `-`, then standard input is assumed.

    - `-c`: Write to the standard output the number of bytes in each input file.
    - `-l`: Write to the standard output the number of newlines in each input file.
    - `-m`: Write to the standard output the number of characters in each input file.
    - `-w`: Write to the standard output the number of words in each input file.

2 Extra Credit Requirements (10 points)

You may gain some extra credit points if your submission includes the following additional utilties:

- `./pwd`

  (5 points) The `pwd` utility shall write to standard output an absolute pathname of the current working directory, which does not contain the filenames dot or dot-dot (see `opendir(3)`, `readdir(3)`, and `closedir(3)`).

- `/cksum [file...]`

  (5 points) The `cksum` utility shall calculate and write to standard output a cyclic redundancy check (CRC) for each input file, and also write to standard output the number of octets in each file. The CRC used is based on the polynomial used for CRC error checking in the ISO/IEC 8802-3:1996 standard (Ethernet). You should consult IEEE Std 1003.1-2008, 2016 Edition `cksum` page ↗ for implementation details. If no `file` operands are specified, the standard input shall be used.

3 Nonfunctional Requirements (20 points)

Your submission needs to satisfy the following functional requirements:

- **Directory Setup:** Make sure that all of your files are in a directory called `LastNameA-LastNameB-p3`, where `LastNameA` and `LastNameB` are replaced with the actual last names of you and your pair programming partner.
- **Libraries:** You are allowed to use any of the C or C++ standard libraries. When reading or writing to a file are concerned, you need to use low-level calls to `read(2)` and `write(2)` and related functions. Whenever possible, program output should be unbuffered. You are NOT allowed to use the following system calls in any of your implementations: `fork(2)`, `execve(2)`, `exec(3)`, `popen(3)`, and `system(3)` (or related functions). Failure to adhere to this non-functional requirement will result in an automatic 50 point deduction.
- **Documentation (5 points):** All classes, structs, and functions must be documented using Javadoc (or Doxygen) style comments. Use inline documentation, as needed, to explain ambiguous or tricky parts of your code.
- `Makefile` **File (5 points):** You need to include a `Makefile`. Your `Makefile` needs to compile and link separately. That is, make sure that your `Makefile` is setup so that your `.cpp` files each compile to individual `.o` files. This is very important. The resulting executable names should correspond to the ones presented in the functional requirements.
- **Standards & Flags (5 points):** Make sure that when you compile, you pass the following options to `g++` in addition to the `-c` option:

  ```
  -Wall -std=c++14 -g -O0 -pedantic-errors
  ```

  Other compiler/linker options may be needed in addition to the ones mentioned above. The expectation is that the grader should be able to type `make clean` and `make` in the following to clean and compile/link your submission, respectively.
- `README` **File (5 points):** 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 program.

  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.
- **Compiler Warnings:** Since you should be compiling with both the `-Wall` and `-pedantic-errors` options, your code is expected to compile without `g++` issuing any warnings. Failure to adhere to this non-functional requirement will result in an automatic 5 point deduction.
- **Memory Leaks:** Since this project may make use of dynamic memory allocation, you are expected to ensure that your project implementation does not result in any memory leaks. We will test for memory leaks using the `valgrind` utility. Failure to adhere to this non-functional requirement will result in an automatic 5 point deduction.

4 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 `LastNameA-LastNameB-p3`. From within the parent directory, execute the following command:

```
$ submit LastNameA-LastNameB-p3 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 LastNameA-LastNameB-p3.tar.gz LastNameA-LastNameB-p3
$ mutt -s "p1" -a LastNameA-LastNameB-p3.tar.gz -- your@email.com < /dev/null
```

Appendix A - Utility Argument Syntax

The notation used for the arguments of a utility imposes requirements on the implementors of the utility and provides a simple reference for the application developer or system user.

1. Arguments or option-arguments enclosed in the `'['` and `']'` notation are optional and can be omitted. Conforming applications shall not include the `'['` and `']'` symbols in data submitted to the utility.
2. Arguments separated by the `'|'` (`<vertical-line>`) bar notation are mutually-exclusive. Conforming applications shall not include the `'|'` symbol in data submitted to the utility.
3. Ellipses (`"..."`) are used to denote that one or more occurrences of an operand are allowed. When an option or an operand followed by ellipses is enclosed in brackets, zero or more options or operands can be specified.

Appendix B - Parsing Utility Options

The `getopt(3)` function can be used to handle options and operands that conform to the command argument descriptions provided in this project description. In addition to the manual page for `getopt(3)`, a description of the function and its parameters as well as an example of how to use it are provided APUE 17.6.

Appendix C - Unbuffered Output

The best way to guarantee that output is unbuffered (i.e., characters at the destination as soon as possible) is to directly call `write(2)`. If you are using `printf(3)`, you should disable output buffering using `setvbuf(3)`. If you are using C++ output streams (e.g., `cout`), then you should disable output buffering using `setf` and `unitbuf` .