Assignment 1. Getting to know your system

[35L home > assignments]

Laboratory: Linux and Emacs scavenger hunt

Instructions: Use the commands that you learned in class to find answers to these questions. Don't use a search engine like Google, and don't ask your neighbor. If you need a hint, ask the TA. When you find a new command, run it so you can see exactly how it works. In addition to turning in the answers to these questions, turn in a description of your session discovering them. As you do actions, use a Linux-based editor to record your keystrokes and each answer in files `key1.txt` and `ans1.txt` that you will submit as part of the assignment.

1. How can you get `man` to print all the commands that have a specific word in their man page (or at least the description part of the man page)? (hint: `man man`)
2. Where are the `cp` and `wc` programs located in the file system?
3. What executable programs have names that are just one character long, and what do they do?
4. When you execute the command named by the symbolic link `/usr/bin/emacs`, which file actually is executed?
5. The `chmod` program changes permissions on a file. What does the symbolic mode `g+s,o-x` mean, in terms of permissions?
6. What option to `find` lets you search for files that have been modified in the last three weeks?
7. Use the previous answer to find all directories modified in the last three weeks.
8. Of the files in the same directory as `find`, how many of them are symbolic links?
9. What is the oldest regular file in the `/usr/lib` directory?
10. Where does the `locale` command get its data from?
11. In Emacs, what commands have `sort` in their name?
12. Briefly, what do the Emacs keystrokes `C-M-a` through `C-M-h` do? Can you list their actions concisely?
13. In more detail, what does the Emacs keystroke `C-g` do?
14. What does the Emacs `yank` function do?
15. When looking at the directory `/usr/bin`, what's the difference between the output of the `ls -l` command, and the directory listing of the Emacs `dired` command?

Homework: Learning to use Emacs

- Keith Waclena, A Tutorial Introduction to GNU Emacs (2009)
- The Emacs editor, version 25.1 (2016)
- An Introduction to Programming in Emacs Lisp, version 25.1 (2016)

For all the exercises, record the steps taken to accomplish the given tasks. Use intelligent ways of answering the questions. For example, if asked to move to the first occurrence of the word "scrumptious", do not merely use cursor keys to move the cursor by hand; instead, use the builtin search capabilities to find "scrumptious" quickly.

To start, download a copy of the web page you're looking at into a file named `assign1.html`. You can do this with Wget or curl. Use `cp` to make three copies of this file. Call the copies `exer1.html`, `exer2.html`, and `exer3.html`.

Exercise 1.1: Moving around in Emacs

1. Use Emacs to edit the file `exer1.html`.
2. Move the cursor to just after the first occurrence of the word "PUBLIC".
3. Now move the cursor to the start of the first later occurrence of the word "Laboratory".
4. Now move the cursor to the start of the first later occurrence of the word "self-referential".
5. Now move the cursor to the start of the first later occurrence of the word "arrow".
6. Now move the cursor to the end of the current line.
7. Now move the cursor to the beginning of the current line.
8. Doing the above tasks with the arrow keys takes many keystrokes, or it involves holding down keys for a long time. Can you think of a way to do it with fewer keystrokes by using some of the commands available in Emacs?
9. Did you move the cursor using the arrow keys? If so, repeat the above steps, without using the arrow keys.
10. When you are done, exit Emacs.

Exercise 1.2: Deleting text in Emacs

1. Use Emacs to edit the file `exer2.html`. The idea is to delete its HTML comments; the resulting page should display the same text as the original.
2. Delete the 18th line, which is an HTML comment. `<!-- HTML comments look like this. -->`
3. Delete the HTML comment containing the text "`DELETEME DELETEME DELETEME`".
4. Delete the HTML comment containing the text "`https://en.wikipedia.org/wiki/HTML_comment#Comments`".
5. There are two more HTML comments; delete them too.

Once again, try to accomplish the tasks using a small number of keystrokes. When you are done, save the file and exit back to the command line. You can check your work by using a browser to view `exer2.html`. Also, check that you haven't deleted something that you want to keep, by using the following command:

```
diff -u exer1.html exer2.html >exer2.diff
```

The output file `exer2.diff` should describe only text that you wanted to remove. Don't remove `exer2.diff`; you'll need it later.

Exercise 1.3: Inserting text in Emacs

1. Use Emacs to edit the file `exer3.html`.
2. Change the first two instances of "Assignment 1" to "Assignment 37".
3. Change the first instance of "UTF-8" to "US-ASCII".
4. Insert a blank line before the first line containing "`</ol>`".
5. When you finish, save the text file and exit Emacs. As before, use the `diff` command to check your work.

Exercise 1.4: Other editing tasks in Emacs

In addition to inserting and deleting text, there are other common tasks that you should know, like copy and paste, search and replace, and undo.

1. Execute the command "`cat exer2.html exer2.diff >exer4.html`" to create a file `exer4.html` that contains a copy of `exer2.html` followed by a copy of `exer2.diff`.
2. Use Emacs to edit the file `exer4.html`. The idea is to edit the file so that it looks identical to `exer1.html` on a browser, but the file itself is a little bit different internally.
3. Go to the end of the file. Copy the new lines in the last chunk of diff output, and paste them into the correct location earlier in the file.
4. Repeat the process, until the earlier part of the file is identical to what was in the original.
5. Delete the last part of the file, which contains the diff output.
6. … except we didn't really want to do that, so undo the deletion.
7. Turn the diff output into a comment, by surrounding it with "`<!--`" and "`-->`".
8. Now let's try some search and replaces. Search the text document for the pattern "`<ol>`". How many instances did you find? Use the search and replace function to replace them all with the initial-caps equivalent "`<Ol>`".
9. Check your work with viewing `exer4.html` with an HTML browser, and by running the shell command "`diff -u exer1.html exer4.html >exer4.diff`". The only differences should be changes from "`<ol>`" to "`<Ol>`", and a long HTML comment at the end.

Exercise 1.5: Doing commands in Emacs

Do these tasks all within Emacs. Don't use a shell subcommand if you can avoid it.

1. Create a new directory named "`junk`" that's right under your home directory.
2. In that directory, create a C source file `hello.c` that contains the following text. Take care to get the text exactly right, with no trailing spaces or empty lines, with the initial `#` in the leftmost column of the first line, and with all other lines indented to match exactly as shown:

```
#include <stdio.h>
int
main (void)
{
  char n = '\n';
  char b = '\\';
  char q = '"';
  char const *p = "#include <stdio.h>%cint%cmain (void)%c{%c  char n = '%cn';%c  char b = '%c%c';%c  char q = '%c';%c  char const *p = %c%s%c;%c  printf (p, n, n, n, n, b, n, b, b, n, q, n, q, p, q, n, n, n, n);%c  return 0;%c}%c";
  printf (p, n, n, n, n, b, n, b, b, n, q, n, q, p, q, n, n, n, n);
  return 0;
}
```

3. Compile this file, using the Emacs `M-x compile` command.
4. Run the compiled program, and put its output into a new Emacs buffer named `hello-out`.
5. Copy this buffer's contents directly into the log that you're maintaining for this exercise. (You *are* using Emacs to maintain the log, aren't you?)

Exercise 1.6: Running Elisp code

1. Visit Emacs's `*scratch*` buffer.
2. In the buffer, compute a random integer by invoking the `random` function. Use `C-j` (`eval-print-last-sexp`) to invoke the `random` function.
3. In the buffer, assign two random integers to the global variables `x` and `y`. You can start by executing `(setq x (random))`. Again, use `C-j`.
4. What is the product of the two variables? You can find this out by executing `(* x y)`. What do you observe about the result? If the answer is the correct mathematical answer, keep trying again with a different pair of random integers until you get an answer that is not mathematically correct.

mathematical answer, keep trying again with a different pair of random integers until you get an answer that is not mathematically correct.
5. Try evaluating `(* x y)` again, but this time with `M-:` (`eval-expression`). What difference do you observe in the output?
6. Are the two random integers truly random in the mathematical sense? If not, what's not random about them?
7. Assuming `(random)` is truly random, what is the probability that the two-variable product mentioned above is mathematically incorrect? Explain how you calculated this.

Submit

Submit the following files.

`key1.txt`
: For each homework exercise, the set of keystrokes needed to do the exercise. Attempt to use as few keystrokes as possible. Do not bother to write down the keystrokes needed to start the editor (e.g., "`e m a c s SP e x e r 1 . h t m l Enter`") or to type in the complicated C program of Exercise 1.5. Write down the label of the key for each keystroke, e.g., "`a`", "`A`" (if you type "`a`" while holding down the shift key), "`Tab`", "`Enter`", "`Esc`". Use "`SP`" for space. Use prefix "`C-`" and "`M-`" for control and meta characters: e.g., "`C-f`" represents Control-F, and "`M-f`" represents Meta-F. Put a space or a newline between each pair of keystroke representations. For example: "`e m a c s < v i Backspace Backspace Backspace > v i`". If you use some key not described above, invent your own ASCII name for the key and explain what key you mean, but don't put spaces or newlines in your key name.

`ans1.txt`
: Answers to each lab question.

The `.txt` files should be ASCII text files, with no carriage returns, and with no more than 80 columns per line except when logging program output longer than that. The shell command:

```
awk '/\r/ || 80 < length' key1.txt ans1.txt
```

should output only a small number of log lines.

$Id: assign1.html,v 1.39 2017/01/04 00:45:52 eggert Exp $

Assignment 2. Shell scripting

Laboratory: Spell-checking Hawaiian

Keep a log in the file `lab2.log` of what you do in the lab so that you can reproduce the results later. This should not merely be a transcript of what you typed: it should be more like a true lab notebook, in which you briefly note down what you did and what happened.

For this laboratory we assume you're in the standard C or POSIX locale. The shell command `locale` should output `LC_CTYPE="C"` or `LC_CTYPE="POSIX"`. If it doesn't, use the following shell command:

```
export LC_ALL='C'
```

and make sure `locale` outputs the right thing afterwards.

We also assume the file `words` contains a sorted list of English words. Create such a file by sorting the contents of the file `/usr/share/dict/words` on the SEASnet GNU/Linux hosts, and putting the result into a file named `words` in your working directory. To do that, you can use the `sort` command.

Then, take a text file containing the HTML in this assignment's web page, and run the following commands with that text file being standard input. Describe generally what each command outputs (in particular, how its output differs from that of the previous command), and why.

```
tr -c 'A-Za-z' '[\n*]'
tr -cs 'A-Za-z' '[\n*]'
tr -cs 'A-Za-z' '[\n*]' | sort
tr -cs 'A-Za-z' '[\n*]' | sort -u
tr -cs 'A-Za-z' '[\n*]' | sort -u | comm - words
tr -cs 'A-Za-z' '[\n*]' | sort -u | comm -23 - words
```

Let's take the last command as the crude implementation of an English spelling checker. Suppose we want to change it to be a spelling checker for Hawaiian, a language whose traditional orthography has only the following letters (or their capitalized equivalents):

```
p k ' m n w l h a e i o u
```

In this lab for convenience we use ASCII apostrophe (') to represent the Hawaiian ‘okina (‘); it has no capitalized equivalent.

Create in the file `hwords` a simple Hawaiian dictionary containing a copy of all the Hawaiian words in the tables in "English to Hawaiian", an introductory list of words. Use Wget to obtain your copy of that web page. Extract these words systematically from the tables in "English to Hawaiian". Assume that each occurrence of "`<tr> <td>`*`Eword`*`</td> <td>`*`Hword`*`</td>`" contains a Hawaiian word in the *`Hword`* position. Treat upper case letters as if they were lower case; treat "`<u>a</u>`" as if it were "`a`", and similarly for other letters; and treat `` ` `` (ASCII grave accent) as if it were `'` (ASCII apostrophe, which we use to represent ‘okina). Some entries, for example "`H<u>a</u>lau, kula`", contain spaces or commas; treat them as multiple words (in this case, as "`halau`" and "`kula`"). You may find that some of the entries are improperly formatted and contain English rather than Hawaiian; to fix this problem reject any entries that contain non-Hawaiian letters after the abovementioned substitutions are performed. Sort the resulting list of words, removing any duplicates. Do not attempt to repair any remaining problems by hand; just use the systematic rules mentioned above. Automate the systematic rules into a shell script `buildwords`, which you should copy into your log; it should read the HTML from standard input and write a sorted list of unique words to standard output. For example, we should be able to run this script with a command like this:

```
cat foo.html bar.html | ./buildwords | less
```

If the shell script has bugs and doesn't do all the work, your log should record in detail each bug it has.

Modify the last shell command shown above so that it checks the spelling of Hawaiian rather than English, under the assumption that `hwords` is a Hawaiian dictionary. Input that is upper case should be lower-cased before it is checked against the dictionary, since the dictionary is in all lower case.

Check your work by running your Hawaiian spelling checker on this web page (which you should also fetch with Wget), and on the Hawaiian dictionary `hwords` itself. Count the number of "misspelled" English and Hawaiian words on this web page, using your spelling checkers. Are there any words that are "misspelled" as English, but not as Hawaiian? or "misspelled" as Hawaiian but not as English? If so, give examples.

Homework: Find duplicate files

Suppose you're working in a project where software (or people) create lots of files, many of them duplicates. You don't want the duplicates: you want just one copy of each, to save disk space. Write a shell script `sameln` that takes a single argument naming a directory D, finds all regular files immediately under D that are duplicates, and replaces the duplicates with hard links. Your script should not recursively examine all files that are in subdirectories of D; it should examine only files that are immediately in D.

If your script finds two or more files that are duplicates, it should keep the file whose name is lexicographically first (for example, if the duplicates are named X, A, and B, it should keep A and replace X and B with hard links to A); however, it should prefer files whose name start with "." to other files (for example, if the duplicates are named .Y, .X, A, and B, it should keep .X and replace the others with hard links to .X).

If your script finds a file in D that is not a regular file, it should silently ignore it; for example, it should silently ignore all symbolic links and directories. If your script has a problem reading a file (for example, if the file not readable to you), it should report the error and not treat it as a duplicate of any file.

You need not worry about the cases where your script is given no arguments, or more than one argument. However, be prepared to handle files whose names contain special characters like spaces, "*", and leading "–".

Your script should be runnable as an ordinary user, and should be portable to any system that supports the standard POSIX shell and utilities; please see its list of utilities for the commands that your script may use. Hint: see the `cmp`, ln, and test utilities.

Submit

Submit the following files.

- The script `buildwords` as described in the lab.
- The file `lab2.log` as described in the lab.
- The file `sameln` as described in the homework.

All files should be ASCII text files, with no carriage returns, and with no more than 80 columns per line. The shell command:

```
awk '/\r/ || 80 < length' lab2.log sameln
```

should output nothing.

$Id: assign2.html,v 1.29 2013/04/08 16:46:24 eggert Exp $

Assignment 3. Modifying and rewriting software

Laboratory: Installing a small change to a big package

Keep a log in the file `lab3.txt` of what you do in the lab so that you can reproduce the results later. This should not merely be a transcript of what you typed: it should be more like a true lab notebook, in which you briefly note down what you did and what happened.

You're helping to build an application containing a shell script that invokes the `ls` command to get file status. Your application is running atop the Maroon Chapeau Enterprise Linux 5 distribution, which uses the `ls` implementation supplied by Coreutils 7.6. You've been running into the problem that for some users `ls` generates output that looks like this:

```
$ ls -l /bin/bash
-rwxr-xr-x 1 root root 729040 2009-03-02 06:22 /bin/bash
```

The users want the traditional Unix format, which looks like this:

```
$ ls -l /bin/bash
-rwxr-xr-x 1 root root 729040 Mar  2  2009 /bin/bash
```

You've been asked to look into the problem and fix it.

You discover that the problem is that in some cases users set their locale to a value like `en_US.UTF-8`, for example, by setting the `LC_ALL` environment variable to that value:

```
$ export LC_ALL='en_US.UTF-8'
```

Users who have done this get the YYYY-MM-DD date instead of the traditional Unix date.

You nose around on the net, and discover that the problem is that the locale files for Coreutils are not generated properly (see Jim Meyering's message of 2009-09-29, also archived in case the primary web site is down). Getting these files generated and distributed to all your clients seems like a bit of a hassle, so instead, you decide to patch the `ls` program instead, using a temporary workaround patch published by Pádraig Brady (also archived).

Try Brady's workaround, as follows:

1. Grab Coreutils 7.6.
2. Compile and install your copy of Coreutils into a temporary directory of your own. Note any problems you run into.
3. Reproduce the bug on your machine with the unmodified version of coreutils. You may need to use the `locale-gen` program to generate the `en_US.UTF-8` locale.
4. Use Emacs or Vim to apply Brady's patch.
5. Type the command `make` at the top level of your source tree, so that you build (but do not install) the fixed version. For each command that gets executed, explain why it needed to be executed (or say that it wasn't neeeded).
6. Make sure your change fixes the bug, by testing that the modified `ls` works on your test case and that the installed `ls` doesn't. Test on a file that has been recently modified, and on a file that is at least a year old. You can use the `touch` command to artficially mark a file as being a year old.

Q1. Why did Brady's patch remove the line "`case_long_iso_time_style:`"? Was it necessary to remove that line? Explain.

Q2. If your company adopts this patched version of Coreutils instead of the default one, what else should you watch out for? Might this new version of Coreutils introduce other problems with your application, perhaps in countries where users don't speak English and don't understand English-format dates?

Homework: Rewriting a script

Consider the Python script `randline.py`.

Q3. What happens when this script is invoked on an empty file like `/dev/null`, and why?

Q4. What happens when this script is invoked with Python 3 rather than Python 2, and why? (You can run Python 3 on the SEASnet hosts by using the command `python3` instead of `python`.)

Write a new script `comm.py` in the style of `randline.py`; your script should implement the POSIX `comm` command.

Your implementation should support all the options and operands required by POSIX. For example, if the file `words.txt` and standard input are both text files sorted in the current locale, `comm -12 words.txt -` should output a sorted file containing all lines that are in both `words.txt` and standard input. Your implementation should also support all the environment variables required by POSIX, except that some or all of its diagnostics can use English regardless of the values of the `LC_MESSAGES` and `NLSPATH` environment variables. As with the original `randline.py`, your program should report an error if given the wrong number of input operands.

Your implementation should also support an extra option `-u`, which causes `comm` to work even if its inputs are not sorted. Output lines should appear in the same order as the input lines. If a line appears in both input files, it should be output according to the first input file's order. Lines that appear only in the second input file should be output after all other lines.

Your implementation of `comm.py` should use only the `locale` and `string` modules and the modules that `randline.py` already uses (it should not import any other modules). Don't forget to change its usage message to accurately describe the modified behavior.

Port your `comm.py` implementation to Python 3. Make sure that it still works with Python 2. Don't rewrite it from scratch; make as few changes as is reasonable.

Submit

Submit the following files.

- The file `lab3.txt` as described in the lab.
- A file `hw3.txt` containing the answer to questions Q1 through Q4 noted above.
- The file `comm.py` as described in the homework.

All files should be ASCII text files, with no carriage returns, and with no more than 80 columns per line. The shell command:

```
expand lab3.txt hw3.txt comm.py | awk '/\r/ || 80 < length'
```

should output nothing.

$Id: assign3.html,v 1.23 2015/04/16 06:24:49 eggert Exp $

Assignment 4. C programming and debugging

Useful pointers

- Ian Cooke, C for C++ Programmers (2004-06-08). Note that it describes C89; C11, the current version of C, supports `//` comments, declarations after statements, and (if you include `<stdbool.h>`) `bool`.
- Steve Holmes, C Programming (1995)
- Parlante, Zelenski, et. al, Unix Programming Tools (2001), section 3 — `gdb`.
- Valgrind Quick Start Guide (2016-10-20)
- Valgrind User Manual (2016-10-20)
- Richard Stallman, Roland Pesch, Stan Shebs, *et al.*, Debugging with GDB (2017)

Laboratory: Debugging a C program

As usual, keep a log in the file `lab4.txt` of what you do in the lab so that you can reproduce the results later. This should not merely be a transcript of what you typed: it should be more like a true lab notebook, in which you briefly note down what you did and what happened.

You're helping to maintain an old stable version of `coreutils`, but that version has a bug in its implementation of the `ls` program. (Actually, it has two bad bugs, but for now we'll just look at the first one.)

The bug is that `ls -t` mishandles files whose time stamps are very far in the past. It seems to act as if they are in the future. For example:

```
$ tmp=$(mktemp -d)
$ cd $tmp
$ touch -d '1918-11-11 11:00 GMT' wwi-armistice
$ touch now
$ sleep 1
$ touch now1
$ TZ=UTC0 ls -lt --full-time wwi-armistice now now1
-rw-r--r-- 1 eggert csfac 0 1918-11-11 11:00:00.000000000 +0000 wwi-armistice
-rw-r--r-- 1 eggert csfac 0 2017-01-25 00:11:55.528846902 +0000 now1
-rw-r--r-- 1 eggert csfac 0 2017-01-25 00:11:54.524820127 +0000 now
$ cd
$ rm -fr $tmp
```

Build this old version of coreutils as-is, and then again with this renaming patch. What problems did you have when building it as-is, and why did the renaming patch fix them?

Reproduce the problem. Use a debugger to figure out what went wrong and to fix the corresponding source file.

Construct a new patch file `lab4.diff` containing your coreutils fixes, in the form of a ChangeLog entry followed by a `diff -u` patch.

Also, try to reproduce the problem in your home directory on the SEASnet Linux servers, instead of using the `$tmp` directory. When running the above test case, use the already-installed `touch` and `ls` utilities instead of the old version of coreutils. How well does SEASnet do?

Homework: Sorting encrypted text

The basic idea is that we want to sort encoded data without decoding and encoding it. That is, our input is an encoded file, and we could compute the output by decoding the input, sorting the decoded data, and then encoding the resulting output—except that we do not want to encode or decode anything.

Write a C function `frobcmp` that takes two arguments `a` and `b` as input and returns an `int` result that is negative, zero, or positive depending on whether `a` is less than, equal to, or greater than `b`. Each argument is of type `char const *`, and each points to an array of non-space bytes that is followed by space byte. Use standard byte-by-byte lexicographic comparison on the non-space bytes, in the style of the `memcmp` function, except that you should assume that both arrays are frobnicated, (i.e., trivally encoded via `memfrob`) and should return the equivalent of running `memcmp` on the corresponding unfrobnicated arrays. If one unfrobnicated array is a prefix of the other, then consider the shorter to be less than the longer. The space bytes are not considered to be part of either array, so they do not participate in the comparison.

For example, `frobcmp ("*{_CIA\030\031 ", "*`_GZY\v ")` should return a positive `int` because `"*{_CIA\030\031"` is `"\0Quick23"` frobnicated and `"*`_GZY\v"` is `"\0Jumps!"` frobnicated, and `"\0Quick23"` is greater than `"\0Jumps!"` in the ASCII collating sequence. As the example demonstrates, null bytes `'\0'` are allowed in the byte arrays and do contribute to the comparison.

Your implementation should not invoke `memfrob`, as that would mean that memory would temporarily contain a copy of the unfrobnicated data. Instead, it should look only at frobnicated bytes one at a time, and unfrobnicate them "by hand", so to speak.

Use your C function to write a program `sfrob` that reads frobnicated text lines from standard input, and writes a sorted version to standard output in frobnicated form. Frobnicated text lines consist of a series of non-space bytes followed by a single space; the spaces represent newlines in the original text. Your program should do all the sorting work itself, by calling `frobcmp`. If standard input ends in a partial record that does not have a trailing space, your program should behave as if a space were appended to the input.

Use `<stdio.h>` functions to do I/O. Use `malloc`, `realloc` and `free` to allocate enough storage to hold all the input, and use `qsort` to sort the data. Do not assume that the input file is not growing: some other process may be appending to it while you're reading, and your program should continue to read until it reaches end of file. For example, your program should work on the file `/proc/self/maps`, a "file" that is constantly mutating: it always appears to be of size 0 when you `ls` it, but it always contains nonempty contents if you read it. You should make sure your program works on empty files, as well as on files that are relatively large, such as `/usr/local/cs/jdk*/jre/lib/rt.jar` on SEASnet.

If the program encounters an error of any kind (including input, output or memory allocation failures, it should report the error to `stderr` and exit with status 1; otherwise, the program should succeed and exit with status 0. The program need not report `stderr` output errors.

For example, the shell command:

```
printf '*~BO *{_CIA *hXE]D *LER #@_GZY #E\\OX #^BO #FKPS #NEM\4' |
./sfrob |
od -ta
```

should output:

```
0000000   *   h   X   E   ]   D  sp   *   {   _   C   I   A  sp   *   ~
0000020   B   O  sp   *   L   E   R  sp   #   N   E   M eot  sp   #   @
0000040   _   G   Z   Y  sp   #   F   K   P   S  sp   #   E   \   O   X
0000060  sp   #   ^   B   O  sp
0000066
```

because frobnicating `sfrob`'s input and then appending a trailing newline (because the last frobnicated byte is not a newline) yields:

```
^@The
^@Quick
^@Brown
^@fox
^Ijumps
^Iover
^Ithe
^Ilazy
^Idog.
```

where `^@` denotes a null byte `'\0'`, and `^I` denotes a tab byte `'\t'`. Sorting this yields:

```
^@Brown
^@Quick
^@The
^@fox
^Idog.
^Ijumps
^Ilazy
^Iover
^Ithe
```

and frobnicating this yields the output shown above.

Submit

Submit the following files.

- The files `lab4.txt` and `lab4.diff` as described in the lab.
- A single source file `sfrob.c` as described in the homework.

All files should be ASCII text files, with no carriage returns, and with no more than 200 columns per line. The C source file should contain no more than 132 columns per line. The shell commands

```
expand lab4.txt lab4.diff | awk '/\r/ || 200 < length'
expand sfrob.c | awk '/\r/ || 132 < length'
```

should output nothing.

$Id: assign4.html,v 1.29 2017/01/25 00:24:31 eggert Exp $

Assignment 5. System call programming and debugging

Useful pointers

- Franco Callari, Block-oriented I/O in Unix (1996)
- The Open Group Base Specifications Issue 7, IEEE Std 1003.1-2008, 2016 Edition is the official standard for commands, system calls and some higher-level library calls.
- `man strace`
- strace on Wikipedia

Laboratory: Buffered versus unbuffered I/O

As usual, keep a log in the file `lab.txt` of what you do in the lab so that you can reproduce the results later. This should not merely be a transcript of what you typed: it should be more like a true lab notebook, in which you briefly note down what you did and what happened.

For this laboratory, you will implement transliteration programs `tr2b` and `tr2u` that use buffered and unbuffered I/O respectively, and compare the resulting implementations and performance. Each implementation should be a main program that takes two operands *from* and *to* that are byte strings of the same length, and that copies standard input to standard output, transliterating every byte in *from* to the corresponding byte in *to*. Your implementations should report an error *from* and *to* are not the same length, or if *from* has duplicate bytes. To summarize, your implementations should like the standard utility `tr` does, except that they have no options, characters like `[`, `-` and `\` have no special meaning in the operands, operand errors must be diagnosed, and your implementations act on bytes rather than on (possibly multibyte) characters.

1. Write a C transliteration program `tr2b.c` that uses `getchar` and `putchar` to transliterate bytes as described above.
2. Write a C program `tr2u.c` that uses `read` and `write` to transliterate bytes, instead of using `getchar` and `putchar`. The *nbyte* arguments to `read` and `write` should be 1, so that the program reads and writes single bytes at a time.
3. Use the `strace` command to compare the system calls issued by your `tr2b` and `tr2u` commands (a) when copying one file to another, and (b) when copying a file to your terminal. Use a file that contains at least 5,000,000 bytes.
4. Use the `time` command to measure how much faster one program is, compared to the other, when copying the same amount of data.

Homework: Encrypted sort revisited

Rewrite the `sfrob` program you wrote previously so that it uses system calls rather than `<stdio.h>` to read standard input and write standard output. If standard input is a regular file, your program should initially allocate enough memory to hold all the data in that file all at once, rather than the usual algorithm of reallocating memory as you go. However, if the regular file grows while you are reading it, your program should still work, by allocating more memory after the initial file size has been read.

Your program should do one thing in addition to `sfrob`. If given the `-f` option, your program should ignore case while sorting, by using the standard `toupper` function to upper-case each byte after decrypting and before comparing the byte. You can assume that each input byte represents a separate character; that is, you need not worry about multi-byte encodings. As noted in its specification, `toupper`'s argument should be either `EOF` or a

nonnegative value that is at most `UCHAR_MAX` (as defined in `<limits.h>`); hence one cannot simply pass a `char` value to `toupper`, as `char` is in the range `CHAR_MIN..CHAR_MAX`.

Call the rewritten program `sfrobu`. Measure any differences in performance between `sfrob` and `sfrobu` using the `time` command. Run your program on inputs of varying numbers of input lines, and estimate the number of comparisons as a function of the number of input lines.

Also, write a shell script `sfrobs` that uses standard `tr` and `sort` to sort encrypted files using a pipeline (that is, your script should not create any temporary files). Your shell script should also accept an `-f` option, with the same meaning as with `sfrobu`. Use the `time` command to compare the overall performance of `sfrob`, `sfrobu`, `sfrobs`, `sfrobu -f`, and `sfrobs -f`.

Submit

Submit a compressed tarball `syscall.tgz` containing the following files.

- The files `lab.txt`, `tr2b.c`, and `tr2u.c` as described in the lab.
- A single source file `sfrobu.c` as described in the homework.
- A single shell script `sfrobs` as described in the homework.
- A text file `sfrob.txt` containing the results of your sfrob performance comparison as described in the homework.

All files should be ASCII text files, with no carriage returns, and with no more than 200 columns per line. The C source file should contain no more than 132 columns per line. The shell commands

```
tar xf syscall.tgz
expand lab.txt sfrob.txt |
  awk '/\r/ || 200 < length'
expand tr2b.c tr2u.c sfrobu.c sfrobs |
  awk '/\r/ || 132 < length'
```

should output nothing.

$Id: assign5.html,v 1.34 2017/02/04 22:45:48 eggert Exp $

Assignment 6. Multithreaded performance

Laboratory

Ordinarily when processes run in Linux, each gets its own virtual processor. For example, when you run the command:

```
tr -cs 'A-Za-z' '[\n*]' | sort -u | comm -23 - words
```

there are three processes, one each for `tr`, `sort`, and `comm`. Each process has its own virtual memory, and processes can communicate to each other only via system calls such as `read` and `write`.

This lab focuses on a different way to gain performance: multithreading. In this approach, a process can have more than one thread of execution. Each thread has its own instruction pointer, registers and stack, so that each thread can be executing a different function and the functions' local variables are accessed only by that thread. However, threads can directly access shared memory, and can communicate results to each other efficiently via the shared memory, so long as they take care not to step on each others' toes.

Synchronization is the Achilles' heel of multithreading, in that it's easy to write buggy programs that have race conditions, where one thread is reading from an area that another thread is simultaneously writing to, and therefore reads inconsistent data (a polite term for "garbage"). This lab does not attack that problem: you will use a prebuilt application that should not have internal race conditions, and you will write an application that is embarrassingly parallel, so that there's no need for subthreads to synchronize with each other.

Lab

Starting with coreutils 8.6, released 2010-10-15, GNU `sort` can use multiple threads to improve performance. This improvement to GNU `sort` was contributed by UCLA students as part of Computer Science 130, the undergraduate software engineering course. This improvement is in the current version of GNU `sort` installed as `/usr/local/cs/bin/sort` in the SEASnet GNU/Linux servers.

Run the command `sort --version` to make sure you're using a new-enough version. Investigate how well the multithreaded sort works, by measuring its performance. First, generate a file containing 10,000,000 random single-precision floating point numbers, in textual form, one per line with no white space. Do this by running the the `od` command with standard input taken from `/dev/urandom`, interpreting the bytes read from standard input as single-precision floating point numbers. (Almost certainly you will occasionally get NaNs, but that's OK; just leave them in there.) Process the output of `od` using standard tools such as `sed` and `tr` so that each floating-point number is on a separate line, without any white space.

Once you have your test data, perhaps in a pipe or perhaps in a file, use `time -p` to time the command `sort -g` on that data, with the output sent to `/dev/null`. Do not time `od` or any of the rest of your test harness; time just `sort` itself.

Invoke `sort` with the `--parallel` option as well as the `-g` option, and run your benchmark with 1, 2, 4, and 8 threads, in each case recording the real, user, and system time. Assuming your `PATH` environment variable is set properly so that `/usr/local/cs/bin` is at its start, you can use `sort --help` or `info sort` for details about how to use the `--parallel` option.

Keep a log of every step you personally took during the laboratory to measure the speed of `sort`, and what the results of the step were. The idea behind recording your steps is that you should be able to reproduce your work later, if need be.

Homework

Modify the simple ray tracer code in Brian Allen's SRT implementation so that the code is multithreaded and runs several times faster on a multicore machine, such as one of the SEASnet Linux servers.

SRT is made of several components. You need to modify only the code in `main.c` and the `Makefile` in order to multithread it. Your implementation should use POSIX threads, so you'll need to include `<pthread.h>` and link with the `-lpthread` library. You'll need to modify the `main` function so that it does something useful with the `nthreads` variable that it computes from the leading digit string in the first operand of the program; currently it errors out unless `nthreads` is 1.

Your new code should invoke `pthread_create` and `pthread_join` to create your threads, and to wait for them to finish. It need not use any other part of the POSIX threads interface; the rest of this (complicated) interface is not needed for this application. It is OK if your new code needs to allocate some additional memory via `malloc` or similar primitives.

Benchmark and test your program with the command "`make clean check`"; this command should output a file `1-test.ppm` that is byte-for-byte identical with the similarly-named file that is output by the unmodified SRT code, a copy of which is in the file `baseline.ppm`. This file is in standard Netpbm format and can be displayed as an image by GIMP and many other graphics tools.

Submit

Submit the following files.

1. A copy of your lab log, as a file `log.txt`.
2. The output of the command `make clean check`, as a text file `make-log.txt`.
3. A gzipped tar file `srt.tgz`, generated by `make dist`.
4. A brief after-action report of your homework, as a text file `readme.txt`. This should discuss any issues that you ran into, and your conclusions about how well your implementation of SRT improves its performance.

If the above files are all in the working directory, the following shell commands should work:

```
gunzip <srt.tgz | tar xf -
(cd srt && make clean check) 2>&1 | diff -u - make-log.txt
awk '200 < length' log.txt readme.txt
```

These commands should output only minor timing differences.

$Id: assign6.html,v 1.17 2017/01/25 00:24:31 eggert Exp $

Assignment 7. SSH setup and use in applications

Laboratory

When you initially set up your host in the lab, you will be running an operating system, and will be able to connect to the outside world, but you won't be able to connect to the hosts of the other students in the class except in trivial ways. What you'd like to do is to be able to run processes on the other students' hosts. For example, you'd like to be able to log into a neighbor's host, and run a program there that displays on your host.

To do that, you need to set up an account on your neighbor's host, and vice versa so that your neighbor can log into your host and do the same. Unfortunately, the obvious ways to do that involve an initial step that exchanges passwords over the Internet in the clear. We'd like to avoid that.

In this laboratory, the class will divide into teams. Your team will assume that the other teams have all tapped the network connection and can observe the contents of all the packets going back and forth among all your team's computers. Your job is to set up your computers so that you can log into each other's hosts, without letting the other teams into your hosts.

Do not try to actually break into the other team's hosts; this is an exercise in defense, not offense!

Use OpenSSH to establish trusted connections among your teams' hosts. You want to make your logins convenient, so you should use ssh-agent on your host to manage authentication. That is, you should be able to log out of your host (dropping all your connections to the outside world), then log back in, type your passphrase once to `ssh-agent`, and then be able to use `ssh` to connect to any of your colleagues' hosts, without typing any passwords or passphrases.

You should also use port forwarding so that you can run a command on a remote host that displays on your host. For example, you should be able to log into a remote host, type the command `xterm`, and get a shell window on your host.

Keep a log of every step you personally took during the laboratory to configure your or your team members' hosts, and what the results of the step were. The idea behind recording your steps is that you should be able to reproduce your work later, if need be.

Homework

Use GNU Privacy Guard's shell commands to create a key pair. Export the public key, in ASCII format, into a file `hw-pubkey.asc`. Use this key to create a detached signature for your submission so that the commands described below can successfully verify it.

If you are creating a key pair on the SEASnet GNU/Linux servers, you may exhaust its entropy pool as described in Launchpad bug 706011. The symptom will be a diagnostic saying "It is a good idea to perform some other action (type on the keyboard, move the mouse, utilize the disks) during the prime generation; this gives the random number generator a better chance to gain enough entropy." Since you can't use the keyboard or mouse on the SEASnet servers, you'll have to use the disks, for example, by using the `find` command to copy every readable file to `/dev/null`; this is something that you can do in another session that is logged into the same machine. Please remember to interrupt the `find` once the key pair is generated, so that you don't tie up the server unnecessarily.

Briefly answer the following questions.

1. Suppose the other teams really had been observing all the bytes going across the network. Is your resulting network still secure? If so, explain why, and explain whether your answer would change if (1) you assumed the other teams had also tapped your keyboards and had observed all of your team's keystrokes, or (2) you are booting off USB and you assume the other teams temporarily had physical control of the USB. If not, explain any weaknesses of your team's setups, focusing on possible attacks by such outside observers.
2. Explain why the `gpg --verify` command in the following instructions doesn't really verify that you personally created the tar file in question. How would you go about fixing this problem?

Submit

Submit two files. The first should be a single gzipped tar file `hw.tar.gz` containing the following files:

1. The file `hw-pubkey.asc` as described above.
2. A copy of your lab log, as a file `log.txt`.
3. The answers to the homework, as a file `hw.txt`. This and `log.txt` should both be ASCII text files, with with no more than 200 columns per line.

The second file `hw.tar.gz.sig` should be a detached cleartext signature, in ASCII form, for `hw.tar.gz`. It should use the key of `hw-pubkey.asc`.

The following shell commands should work:

```
gunzip <hw.tar.gz | tar xf -
mkdir -m go-rwx .gnupg
gpg --homedir .gnupg --import hw-pubkey.asc
gpg --verify hw.tar.gz.sig hw.tar.gz
awk '200 < length' log.txt hw.txt
```

The `gpg --verify` command should say "Good signature". The last `awk` command should output nothing.

$Id: assign7.html,v 1.28 2017/01/25 00:24:31 eggert Exp $

Assignment 8. Dynamic linking

Useful pointers

- M. Tim Jones, Anatomy of Linux dynamic libraries. IBM developerWorks (2008).
- David A. Wheeler, Program Library HOWTO 1.20 (2003).
- vDSO – overview of the virtual ELF dynamic shared object (2014).
- libffi – A Portable Foreign Function Interface Library (2014).

Laboratory: Who's linked to what?

As usual, keep a log in the file `lab.txt` of what you do in the lab so that you can reproduce the results later. This should not merely be a transcript of what you typed: it should be more like a true lab notebook, in which you briefly note down what you did and what happened.

For this laboratory, you will find out about which programs are linked to which libraries.

1. Compile, build and run a trivial program in C on the SEASnet GNU/Linux servers. Your program should compute `cos(sqrt(3.0))` and print it using the `printf` format `"%.17g"`.
2. Use the `ldd` command to see which dynamic libraries your trivial program uses.
3. Use the `strace` command to see which system calls your trivial program makes. Which of these calls are related to dynamic linking and what is the relationship?
4. Suppose your student ID is the 9-digit number *nnnnnnnnn*. On a SEASnet GNU/Linux server, run the shell command "`ls /usr/bin | awk 'NR%101==`*nnnnnnnnn*`%101'`" to get a list of two dozen or so commands to investigate.
5. Invoke `ldd` on each command in your list. If there are error messages, investigate why they're occurring.
6. Get a sorted list of every dynamic library that is used by any of the commands on your list (omitting duplicates from your list).

Homework: Split an application into dynamically linked modules

In this homework you will divide a small example application into dynamically linked modules and a main program, so that the resulting executable does not need to load code that it doesn't need. Although this is just a toy example which would probably not be worth optimizing in this way, in real life many applications use dynamic linking to improve performance in common cases, and the skills used in this small exercise can be helpful in larger programs.

The skeleton tarball contains the following:

- A file `randall.c` that is a single main program, which you are going to split apart.
- A `Makefile` that builds the program `randall`.
- Two files `randcpuid.h` and `randlib.h` that specify two interfaces for libraries that you need to implement when you split `randall.c` apart.

First, read and understand the code in `randall.c`. Do not modify it, or any of the other files in the skeleton tarball.

Second, split the `randall` implementation by copying its source code into the following modules, which you will need to modify to get everything to work:

1. `randcpuid.c` should contain the code that determines whether the current CPU has the RDRAND instruction. It should start by including `randcpuid.h` and should implement the interface described by `randcpuid.h`.
2. `randlibhw.c` should contain the hardware implementation of the random number generator. It should start by including `randlib.h` and should implement the interface described by `randlib.h`.
3. `randlibsw.c` should contain the software implementation of the random number generator. Like `randlibhw.c`, it should start by including `randlib.h` and should implement the interface described by `randlib.h`. Since the software implementation needs initialization and finalization, this implementation should also define an initializer and a finalizer function, using GCC's "`__attribute__ ((constructor))`" and "`__attribute__ ((destructor))`" declaration specifiers.
4. `randmain.c` should contain the main program that glues together everything else. It should include `randcpuid.h` (as the corresponding module should be linked statically) but not `randlib.h` (as the corresponding module should be linked after `main` starts up). Depending on whether `randcpuid` reports that the hardware supports the RDRAND instruction, this main program should dynamically link the hardware-oriented or software-oriented implementation of `randlib`, doing the dynamic linking via `dlopen` and `dlsym`. Also, the main program should call `dlclose` to clean up before exiting. Like `randall`, if any function called by the main program fails, the main program should report an error and exit with nonzero status.

Each module should include the minimal number of include files; for example, since `randcpuid.c` doesn't need to do I/O, it shouldn't include `stdio.h`. Also, each module should keep as many symbols private as it can; for example, since `randcpuid` does not need to export the `cpuid` function, that function should be `static` and not `extern`.

Next, write a makefile include file `randmain.mk` that builds the program `randmain` using three types of linking. First, it should use static linking to combine `randmain.o` and `randcpuid.o` into a single program executable `randmain`. Second, it should use dynamic linking as usual to link the C library and any other necessary system-supplied files before its `main` function is called. Third, after `main` is called, it should use dynamic linking via `dlsym` as described above. `randmain.mk` should link `randmain` with the options "`-ldl -Wl,-rpath=$PWD`". It should compile `randlibhw.c` and `randlibsw.c` with the `-fPIC` options as well as the other GCC options already used. And it should build shared object files `randlibhw.so` and `randlibsw.so` by linking the corresponding object modules with the `-shared` option, e.g., "`gcc ... -shared randlibsw.o -o randlibsw.so`".

The supplied `Makefile` includes `randmain.mk`, so you should be able to type just `make` to build all four files: `randall`, `randmain`, `randlibhw.so`, and `randlibsw.so`. If `randmain` needs to generate any random numbers, it loads either `randlibhw.so` or `randlibsw.so` (but not both) to do its work. You can verify this by using "`strace ./randmain`" or by using a debugger.

Submit

Submit the file `dlsubmission.tgz`, which you can build by running the command `make submission`.

All files should be ASCII text files, with no carriage returns, and with no more than 132 columns per line. The shell command

```
expand lab.txt randmain.mk \
  randcpuid.c randlibhw.c randlibsw.c randmain.c |
  awk '/\r/ || 200 < length'
```

should output nothing.

$Id: assign8.html,v 1.38 2016/11/08 22:00:38 eggert Exp $

Assignment 9. Change management

Useful pointers

- Michael Johnson, Diff, Patch, and Friends, Linux Journal 28 (1996-08)
- Linus Torvalds, Jun Hamano *et al.*, Git - local branching on the cheap
- Scott Chacon, Pro Git (2009)
- Jacob Gube, Top 10 Git Tutorials for Beginners (2011)
- Sitaram Chamarty, The missing `gitk` documentation (2015)
- David MacKenzie, Paul Eggert, and Richard Stallman, Comparing and merging files, version 3.3 (2013-03-23)

You're helping to develop an operating system and command set that as part of its acceptance test is supposed to be used by a large government agency. The agency has lots of requirements, some sensible and some persnickety, and one of these requirements is that applications must use characters properly from the Unicode character set. In particular, applications must use the Unicode character "`" (grave accent, U+0060) only as a spacing accent character.

Unfortunately, one of your applications, GNU Diffutils, regularly uses "`" as a quoting character in diagnostics. For example, the command "`diff . -`" outputs the diagnostic "`diff: cannot compare `-' to a directory`", and this misuse of "`" violates your customer's requirements. You need to change Diffutils so that it outputs "`diff: cannot compare '-' to a directory`" instead, using an apostrophe (U+0027) for both opening and closing quote. You don't want to use fancier quoting characters such as "‘" and "’" (left and right single quotation marks, U+2018 and U+2019) because they are not ASCII characters and another customer requirement is that the programs must work in ASCII-only environments.

The good news is that the Diffutils maintainers have run into a similar problem, and have a patch called "maint: quote 'like this' or "like this", not `like this'" that does what you want. The bad news is that your customer has specified Diffutils version 3.0, and the patch that you want is not in version 3.0. Also, your customer is conservative and wants a minimal patch so that it's easy to audit, whereas the Diffutils maintainers' patch also affects commentary and documentation which your customer doesn't need or want changed.

Laboratory: Managing a backported change

As usual, keep a log in the file `lab9.txt` of what you do in the lab so that you can reproduce the results later. This should not merely be a transcript of what you typed: it should be more like a true lab notebook, in which you briefly note down what you did and what happened.

1. Get a copy of the Diffutils repository, in Git format, from the file `~eggert/src/gnu/diffutils` on the SEASnet GNU/Linux servers, or from its main Savannah repository.
2. Get a log of changes to Diffutils' master branch using the "`git log`" command, and put it into the file `git-log.txt`.
3. Generate a list of tags used for Diffutils using the "`git tag`" command, and put it into the file `git-tags.txt`
4. Find the commit entitled "maint: quote 'like this' or "like this", not `like this'", and generate a patch for that commit, putting it into the file `quote-patch.txt`.
5. Check out version 3.0 of Diffutils from your repository.
6. Use the `patch` command to apply `quote-patch.txt` to version 3.0. In some cases it will not be able to figure out what file to patch; skip past those by typing RETURN. Record any problems you had in applying the patch.
7. Use the `git status` command to get an overview of what happened.

8. Learn how to use the Emacs functions `vc-diff` (`C-x v =`) and `vc-revert` (`C-x v u`). When you're in the `*vc-diff*` buffer generated by `vc-diff`, use `describe-mode` (`C-h m`) to find out the Emacs functions that you can use there, and in particular learn how to use the `diff-apply-hunk` (`C-c C-a`) and `diff-goto-source` (`C-c C-c`) functions.
9. Use Emacs to revert all the changes to files other than `.c` files, since you want only changes to `.c` files. Also, **and don't forget this part**, undo all the changes to `.c` files other than changes to character string constants, as the character-string changes are the only changes that you want; this may require editing some files by hand.
10. Use Emacs to examine the files `src/*.c.rej` carefully, and copy rejected patches into the corresponding `.c` files as needed.
11. Remove all untracked files that `git status` warns you about, since you don't plan on adding any files in your patch.
12. When you're done with the above, `git status` should report a half-dozen modified files, and `git diff` should output a patch that is three or four hundred lines long. Put that patch into a file `quote-3.0-patch.txt`.
13. Build the resulting modified version of Diffutils, using the commands described in the file `README-hacking`, skipping the part about CVS; CVS is obsolescent. (If you are building on `lnxsrv07` or `lnxsrv09` or any other host that is using version 2.16 or later of the GNU C Library, you will need to apply an additional patch after running `./bootstrap` and before running `./configure`, because glibc 2.16 removed the obsolete and dangerous `gets` function declared by a Diffutils header.) Verify that Diffutils does the right thing with the "`diff . -`" scenario, as well as with "`diff --help`".
14. Do a sanity test using the modified version of Diffutils that you just built, by using the just-built `diff` to compare the source code of Diffutils 3.0 to the source code of your modified version. Put the former source code into a directory `diffutils-3.0` and the latter source code into a directory `diffutils-3.0-patch`, and run your implementation of `diff` with the command "*`D`*`/diff -pru diffutils-3.0 diffutils-3.0-patch >quote-3.0-test.txt`", where the *`D`* is the directory containing your `diff` implementation.
15. Use `diff` to compare the contents of `quote-3.0-test.txt` and `quote-3.0-patch.txt`. Are the files identical? If not, are the differences innocuous?

Homework: Verifying and publishing a backported change

You're happy with the code that you wrote in your lab, but now you'd like to publish this patch, in a form similar to that presented in the original patch, so that others can use it.

1. Maintain a file `hw9.txt` that logs the actions you do in solving the homework. This is like your lab notebook `lab9.txt`, except it's for the homework instead of the lab.
2. Check out version 3.0 of Diffutils from your repository, into a new branch named "`quote`".
3. Install your change into this new branch, by running the `patch` command with your patch `quote-3.0-patch.txt` as input.
4. Learn how to use the Emacs function `add-change-log-entry-other-window` (`C-x 4 a`).
5. Use this Emacs function to compose an appropriate ChangeLog entry for your patch, by adapting the change log from the original patch.
6. Commit your changes to the new branch, using the ChangeLog entry as the commit message.
7. Use the command "`git format-patch`" to generate a file `formatted-patch.txt`. This patch should work without having to fix things by hand afterwards.
8. Your teaching assistant will assign you a partner, who will also generate a patch. Verify that your partner's patch works, by checking out version 3.0 again into a new temporary branch `partner`, applying the patch with the command "`git am`", and building the resulting system, checking that it works with "`make check`".
9. Verify that your ChangeLog entry works, by running the command "`make distdir`" and inspecting the resulting `diffutils*/ChangeLog` file.
10. There is a copy of the GNU Emacs git repository's master branch on SEASnet in the directory `~eggert/src/gnu/emacs`. Run the command `gitk` on it, and find the newest merge that is not newer than

2015-01-25. Take a screenshot `gitk-screenshot.png` of your view of the mergepoint, and in an ASCII text file `gitk-description.txt` briefly describe the roles of subwindows that you see in the screenshot.

Submit

Submit a compressed tarball `hw9.tgz` containing the following files.

- The files `lab9.txt`, `git-log.txt`, `git-tags.txt`, `quote-patch.txt`, and `quote-3.0-patch.txt` as described in the lab.
- The files `hw9.txt` `formatted-patch.txt`, `gitk-screenshot.png`, and `gitk-description.txt` as described in the homework.

All `.txt` files should be ASCII text files, with no carriage returns. You can create the tarball with the command:

```
tar czf hw9.tgz lab9.txt git-log.txt git-tags.txt \
  quote-patch.txt quote-3.0-patch.txt hw9.txt formatted-patch.txt \
  gitk-screenshot.png gitk-description.txt
```

$Id: assign9.html,v 1.28 2017/01/25 00:24:31 eggert Exp $