CSCI415—Systems Programming

Spring 2014

Programming Assignment 2

1 Preliminaries

This programming assignment is intended to give you deeper exposure to C programming, the Unix standard i/o library, and a common Unix trick of creating new services by composing other services which already exist.

2 Grep

The `grep` utility is a very useful tool for serarching text files. You should check out the `grep` page in section 1 of the UPM, but in its most common use is of the form

```
 % grep somestring somefile
```

This will produce, on the standard output of `grep`, a file with all of the lines of `somefile` which contain the string `somestring`.

Note that `grep` can do far more powerful tricks than this example, as the manual page will make clear. One particular extra is the `-n` option of grep. If you invoke

```
 % grep -n somestring somefile
```

then `grep` outputs the lines in which `somestring` appear as before. But it also prefixes each line of output with the line number of that line in `somefile` with a colon separating the line number from the line.

Similarly, the `-H` option to `grep` prefixes a line of output with the name of the file in which a matching line occurs and a colon. If both `-n` and `-H` are asserted, grep outputs lines that have both file name and line number before the matching line.

Here's an example:

```
% cd ~kearns/public/415/p2
% grep -n -H elephant art
art:44:A fool-proof method for sculpting an elephant: first, get a huge block of
art:45:marble; then you chip away everything that doesn't look like an elephant.
art:426:God is really only another artist.  He invented the giraffe, the elephant
```

One final feature of `grep` that deserves explicit mention is that you may give it multiple files to search. This is best shown with another example:

```
% grep -n -H elephant art computers
art:44:A fool-proof method for sculpting an elephant: first, get a huge block of
art:45:marble; then you chip away everything that doesn't look like an elephant.
art:426:God is really only another artist.  He invented the giraffe, the elephant
computers:679:An elephant is a mouse with an operating system.
```

3 Fortunes

On some systems, when you login you are greeted with a witty (warning: value judgment here) saying which is randomly chosen from a collection of files of such witty sayings. These sayings conventionally go by the misnomer "fortunes" since the program which selected and output the saying is generally named `fortune`. The files in `~kearns/public/415/p2` are the fortunes files from an old Linux distribution. A single fortunes file is formatted as follows

```
first saying
%
second saying
%
. . .
%
last saying
%
```

Note that a `%` on a line by itself is a terminator of a witty saying.

4 Say: a Witty Sayings Helper Program

You should construct a program called `say` which:

- takes an optional "banner switch" as a command line argument,
- takes an optional "highlight switch" as a command line argument,
- takes a required string used to grep the fortunes files as its next command line argument, and
- takes the output of

  ```
  grep -n -H searchstring fortunesfile1 fortunesfile2 ...
  ```

  on its standard input.

The intent of this is to have `say` print out (on standard output) all the witty sayings which have the string `searchstring` in them. Note that `grep` operates at the line level; we want `say` to operate at the "saying level." (This is the major conceptual problem in this assignment.)

The `say` program will produce on its standard output:

- An optional "banner" with a suitable message and a count of the total number of sayings in the fortunes files which contain the specified string.
- This will be followed by the sayings which contain the `searchstring`. The output sayings should be separated by a line with `----------` on it for readability.
- If operating in the optional highlight mode, the `searchstring` will be displayed in inverse video in an xterm.

If there are no matches, the banner should state the fact, and no other output should be produced. If there are no matches and `say` is not in banner mode, no output should be produced.

Banner Mode The production of the banner is a user-specified option. If the `-b` switch is given as a command line argument to `say`, the banner is produced. It there is no `-b` switch, the matching sayings should be output without the banner. If there is no `-b` and there are no hits in the fortunes file, there should be no output generated on the standard output of `say`.

It is essential that the banner appears before the fortunes containing the search pattern.

Highlight Mode Highlight mode is indicated by a `-h` command line option. In this mode, we assume that the output is being directed to an xterm, and arrange for all matches of the search string be in inverse video in the output of the witty saying. The string `\e[7m` sent to an xterm will switch it into inverse video mode. The string `\e[0m` switches the xterm back to normal (non-inverted) video mode.

Usage You should be able to use `say` in the following contexts:

```
 % grep -n -H pattern f1 f2 > tmpfile ; say pattern < tmpfile
 . . .
 % grep -n -H pattern f1 f2 | say -b pattern > listing
```

Bear in mind that this code is to be written as a system utility, and, as such, should handle all possible errors (e.g., it is possible that what is believed to be `grep` output contains line numbers which are non-numeric or beyond the highest numbered line in a file, an unreadable fortunes file may be specified by an idiot user, etc.). You are responsible for producing code which is robust in the presence of ignorant, stupid, or even malicious, use by the user community. Your program will be tested against bizarre input when it is graded. Please note that there will be some stupid uses of **say** that you cannot reasonably handle. For example, a usage such as

```
 % grep -n -H xxx f1 | say yyy
```

is difficult to handle. The user obviously provided bad input (mismatched search strings). Garbage output is acceptable in this case. If you can produce a meaningful error message without extraordinary effort, you should do so. I suspect that you can produce an error message here with relatively little effort for all but the most perverse files. A cardinal rule is that if you allow garbage output, `say` should not crash (probably with a segfault).

You may *assume* that a file with symbolic name "-b" or "-h" is never used as the name of a fortunes file or a search string. This means that we will not test your code using these strings in the proscribed manner.

Here's an example of how it all should work:

```
 % grep -n -H  elephant art computers | say -b elephant

 3 witty sayings contain the string elephant

 A fool-proof method for sculpting an elephant: first, get a huge block of
 marble; then you chip away everything that doesn't look like an elephant.
-------------------------------------
 God is really only another artist.  He invented the giraffe, the elephant
 and the cat.  He has no real style, He just goes on trying other things.
               -- Pablo Picasso
-------------------------------------
 An elephant is a mouse with an operating system.
```

Note that the banner switch and the highlight switch can be asserted in either order, but before the search string. For example, the following two commands behave identically:

```
 % grep -n -H  elephant art computers | say -b -h elephant
 % grep -n -H  elephant art computers | say -h -b elephant
```

Your program must be able to deal with arbitrarily many hits on an arbitrary number of arbitrarily long fortunes files. You must use dynamically allocated data structures in the `say` program. You may assume that no line of a fortune is longer than 1023 characters; you may also assume that no file name is longer than 1023 characters. For general interest (not required for this project) You may want to check out the manual page for `limits.h` to see how one may interrogate the system for certain limits.

Your `say` program may not use any files except for the fortunes file (as specified on the command line), its standard input (assumed to be output from `grep`), its standard output (for its required output), and its standard error output (for diagnostic output).

5 Due

11:59pm on Monday, February 10. Be sure to keep your program in a single source file named `say.c`. To submit, do the following:

```
 gpg -r CSCI315 --sign -e say.c
 mv say.c.gpg ~
 chmod 644 ~/say.c.gpg
```

Note that you should not generate a new gpg key for the submission; the key you generated and used for the first programming project will be used for the entire course.