CSC 209 Assignment 2, Summer 2019

Due by the end of Friday July 5, 2019; no late assignments without written explanation.

1. Derep

Write a standard unix filter, which processes its standard input if there are no arguments, or processes a sequence of one or more file names on the command-line. As usual, if one of these "file names" is actually the single-character string "–", stdin should be read at that point. However, there are no command-line options, so you needn't use getopt().

The transformation on the file will be to eliminate large repetitions of a single byte. If a single byte occurs more than ten times in a row, after the first ten occurrences, instead of the remaining occurrences output "[%d more %d bytes]", where the first %d is the number of remaining occurrences and the second %d is the byte in question. See the Q&A file for additional clarification and examples, and/or try my sample solution in /u/csc209h/summer/pub/a2/derep.

Since you are not processing any command-line options, you will never emit a usage message (there is no way to run your program improperly—every argument is considered to be a file name). However, when a file cannot be opened, you must call perror() properly to output a standard error message.

2. Parsing the PATH variable

Write a function in C (not a complete program) named parsepath(). It takes three parameters, as follows:

```
int parsepath(char *path, char **result, int size)
```

The first parameter is the string to be parsed, in the format of the *sh* PATH variable (colon-separated directory names).

The second parameter is a pointer to the beginning of an array of pointers-to-char to hold the separately parsed-out directory path names. To avoid exceeding array bounds, the third parameter identifies the size of this array in the usual way.

parsepath() assigns the individual directory path names to successive members of the result array, in order. To do this in some cases it will need to call malloc() to create a new string array. Remember that an empty string counts as the current directory; for this, put `"."` in the array.

parsepath() returns the number of items put in the result array, or –1 for error (which is either a failed malloc() or exceeding the size of the result array).

For testing, you can use any main() you devise, but you can also use my example such code in /u/csc209h/summer/pub/a2/testparsepath.c.

3. Funny times on a file

In the unix filesystem, each file has three times, known as the atime (last accessed time), mtime (last modified time), and ctime (last inode change time). Since modifying a file generally changes the inode information, the mtime of a file is often equal to the ctime. Other operations such as "chmod" change the inode information but do not update the mtime, so at these points the ctime is generally greater than the mtime.

However, there is also a system call "utimes()" to set the mtime and atime of a file to arbitrary values. This is used by things such as the '–p' option to *cp*. But for security reasons, users are not able to set the ctime to arbitrary values (only to the current time, by actually modifying inode information). So for example if you modify a file but claim you didn't, people can tell with an "ls –lc", and this normally can't be disguised.

Write a tool called 'funnytimes' to find files for which the mtime and ctime are not equal. It will search a particular directory and below, recursively, similar to *find*. The successful exit status is 0 whether such files are found or not.

The command-line arguments are one or more directories to search, preceded by two possible options: –s (for **s**trictly-less-than) makes it report only files for which the mtime is less than the ctime, not files for which it is greater (which probably represent some incorrect time on some other computer

(over)

from which the files were transferred with something like scp –p and probably don't represent a security anomaly); and –n *limit* makes it exit with exit status 2 after reporting *limit* files if there are more files yet to report. (See example command-lines on the Q&A web page.)

You must parse the command-line options with getopt() so as to accommodate all standard allowable command-line variations. There is an example getopt()-using program in the "Course notes" section of the CSC 209 web site, also linked to from the Q&A page.

You may not use ftw() or fts().

Other notes

All of your programs must be in standard C. They must compile on the teach.cs linux machines with "gcc –Wall" with no errors or warning messages, and may not use linux-specific or GNU-specific features.

Pay attention to process exit statuses. Your programs must return exit status 0 or 1 as appropriate (or sometimes 2 in the case of funnytimes).

Call your assignment files derep.c, parsepath.c, and funnytimes.c. Your files must have the correct names so as to be processed correctly by the grading software, and auxiliary files are not permitted. Submit with commands such as

```
submit -c csc209h -a a2 derep.c parsepath.c funnytimes.c
```

and the other 'submit' commands are as in assignment one and the labs.

Please see the assignment Q&A web page at

```
https://www.teach.cs.toronto.edu/~ajr/209/a2/qna.html
```

for other reminders, and answers to common questions.

Remember:

This assignment is due at the end of Friday, July 5, by midnight. Late assignments are not ordinarily accepted, and *always* require a written explanation. If you are not finished your assignment by the submission deadline, you should just submit what you have, for partial marks.

Despite the above, I'd like to be clear that if there *is* a legitimate reason for lateness, please do submit your assignment late and send me that written explanation.

And above all: Do not commit an academic offence. Your work must be your own. Do not look at other students' assignments, and do not show your assignment (complete or partial) to other students. Collaborate with other students only on material which is not being submitted for course credit.