CPSC457 Winter 2023

Assignment 2

Due date is posted on D2L.
Individual assignment. Group work is NOT allowed.
Weight: 19% of the final grade.

For this assignment you will write code that recursively scans the current directory for all files and directories. During the scan, your code will need to collect some information, such as:

- the size and path to the largest file;
- the cumulative size of all files;
- the number of all files and directories;
- up to N most common words in .txt files;
- up to N largest images; and
- a list of all top-level vacant directories (defined below).

To finish this assignment, you will need to make use of several different system calls. You will also need to recall and implement one of the tree traversal algorithms you learned in earlier courses.

Starter code

You will be given a starter code for this assignment, which contains an incomplete implementation. The starter code contains a driver code in `main.cpp`, which parses the command line, calls the `analyzeDir()` function, and then prints the results on standard output. The `analyzeDir()` function is incomplete, and your job is to finish its implementation. It is defined in the `analyzeDir.cpp` file, which is **the only file** you should edit and submit for grading.

Start by cloning the starter code using git:

```
$ git clone https://gitlab.com/cpsc457w23/analyzedir.git
```

Now compile the code using the included `Makefile`:

```
$ cd analyzedir
$ make
```

This will create an executable called `analyzeDir`. The driver (`main.cpp`) accepts two command line arguments: `N` (a positive integer) and `directory_name` (a valid directory name). The driver then changes the current working directory to `directory_name` and calls `analyzeDir(N)`. After `analyzeDir()` returns results, the driver displays these results on standard output.

Here is how you can invoke the `analyzeDir` to analyze `"test1"` directory with `N=10`:

```
$ ./analyzeDir 10 test1

…
Largest file:      "some_dir/some_file.txt"
Largest file size: 123
Number of files:   321
Number of dirs:    333
Total file size:   1000000
Most common words from .txt files:
 - "hello" x 3
 - "world" x 1
Vacant directories:
 - "path1/subdir1"
 - "test2/xyz"
```

```
Largest images:
 - "img1" 640x480
 - "img2.png" 200x300
```

Naturally, the results above are incorrect, since `analyzeDir()` is not yet implemented.

The `analyzeDir()` function

The `analyzeDir()` is declared in `analyzeDir.h` and defined in `analyzeDir.cpp`:

```
Results analyzeDir(int N);
```

The `analyzeDir` function needs to recursively scan for all files and subdirectories in the current directory, compute some results during the scan, and then return the computed results. The results will be returned as an instance of a data structure called `struct Results`, which is also declared in `analyzeDir.h`. Your code will use the input parameter `N` to limit the number of reported most frequent words and largest images, as described below. Populate `Results` as follows:

```
std::string largest_file_path;
long largest_file_size;
```

During your recursive scan, you need to determine which of the files contains the most bytes. The fields `largest_file_path` and `largest_file_size` will contain the path and size of the largest file, respectively. If your scan does not find any files, set `largest_file_path=""` (empty string) and `largest_file_size=-1` (negative one).

```
long n_files, n_dirs;
```

The `n_files` fields should contain the total number of files encountered while recursively scanning the current directory. Similarly, `n_dirs` should contain the total number of directories encountered while scanning the directory, including the current directory.

```
long all_files_size;
```

This field should contain the sum of all sizes of all files encountered during the scan. Hint: use `stat()` to determine file sizes for each file, then sum them up.

```
std::vector<std::pair<std::string, int>> most_common_words;
```

Will contain a list of up to `N` most common words inside all text files, together with the number of occurrences of each word. The list must be sorted by the number of occurrences, in descending order. For this assignment, text file is any file that has extension `".txt"`. For this assignment, word is a sequence of 5 or more alphabetic characters, converted to lower case.

For example, the following string `"My name is Pavol, my password: is abc1ab2ZYZaa"` contains the words `"pavol"`, `"password"` and `"zyzaa"`. Another example, the string `"HeLLo,Hello"` contains one word `"hello"`, repeated twice.

You need to open and read the contents of every file you find and extract the words from the file contents. You need to create and maintain a histogram data structure as you extract the words. Look at the code in the word-histogram example below for motivation.

```
std::vector<ImageInfo> largest_images;
```

This field will contain a list of up to `N` largest images (size measured as number of pixels) encountered during the recursive scan. Each image will be listed using the `ImageInfo` struct, with its `path`, and its dimensions, `width` and `height`. The list will be sorted in descending order by the number of pixels in each image (number of pixels in image = width x height of the image).

```
std::vector<std::string> vacant_dirs;
```

Will contain a minimal list of all paths representing vacant directories. For this assignment, a vacant directory is a directory, such that the filesystem subtree starting at that directory contains no files. Here is a recursive definition: a vacant directory is either empty directory (no files, nor directories), or it contains only vacant directories. Another way to think of vacant directory is that if you deleted it recursively, no files would be deleted.

The `vacant_dirs` should only include the top-level vacant directories. This means that once a directory is listed here, none of its descendants should be listed. For example, if `"dir1/sub/xy"`, `"dir1/sub"` and `"dir1"` are all vacant directories, include only `"dir1"` in the results.

Special case: if the entire current directory is vacant, return `"."` (dot) as the only entry in `vacant_dirs`.

All paths reported in any of the results must be relative to the current directory. The paths must not begin with `"./"` (dot slash), nor contain any unnecessary `"."` (dot) and `".."` (dot dot) parts.

Using `identify` to determine image dimensions

You will need to call an external program `identify` to test whether a file is an image, and if it is an image, to determine its dimensions (width and height). `identify` takes a filename as input, and if the filename represents an image, it prints various information about the file on standard output. To reduce the amount of output it generates, and to simplify parsing, you should use the `-format '%w %h'` option to print out only the width and height of the image. For example:

```
$ identify -format '%w %h' test5/picasso.jpg
192 199
```

You will need to use `popen()` to run `identify` and collect its output. You will need to examine the exit code from `identify` to determine whether the file is an image or not. If a filename given to `identify` is an image, it will exit with status 0. If the filename is not an image, it will exit with a non-zero status. The exit status can be retrieved as a return value when you call `pclose()`. The starter code contains a short snippet of code illustrating how to accomplish this.

You **may not** use `popen()` for any other purpose, other than calling the `identify` program.

Miscellaneous hints

Remember to call the appropriate close functions for open file descriptors, e.g. `close()`, `fclose()`, `pclose()`.

Sample code showing how to recursively examine a directory:
https://gitlab.com/cpsc457/public/findLargestDir

Sample program illustrating how to use `std::unordered_map` to create a histogram of words, and how to extract the top N entries from it using 2 different approaches:
https://gitlab.com/cpsc457/public/word-histogram.

Feel free to re-use any code above, but please include appropriate citations.

Additional test directories are available on linuxlab machines in `~pfederl/public/cpsc457f22/a2/extra-tests`. You can run your analyzer or the python one on these test directories like this:

```
$ time ./analyzeDir 10 ~pfederl/public/cpsc457f22/a2/extra-tests/test3
$ time ./analyzeDir.py 10 ~pfederl/public/cpsc457f22/a2/extra-tests/test3
```

Allowed APIs

You are free to use the following APIs from the libc/libc++ libraries for this assignment:

- `popen()/pclose()` to get the output of the `identify` utility
- `stat(), opendir(), closedir(), readdir(), getcwd(), chdir()`
- `open(), close(), read(), fopen(), fread(), fclose(), fgets(), fgetc()`
- `any C++ containers & associated algorithms, C++ streams, C++ string related APIs`

If you want to use other APIs, please ask your instructor or TA.

Restricted APIs

- You may not use `system(3)` function at all.
- You may not call any external programs, other than `identify`. For example, you may not call `find`, `awk`, `grep`, `sort`, `uniq`, etc...

Additional requirements

- The total number of directories and files will be less than 10000.
- You may assume that none of the file names nor directory names will contain spaces or quotations.
- Each file path will contain less than 4096 characters.
- Words will have less than 1024 characters.
- If multiple words have the same number of occurrences, or multiple images have the same number of pixels, you can return those in arbitrary order.
- If multiple files have the same maximum size, arbitrarily pick one of them to report as the largest file.
- You need to consider all files as potential images, regardless of their extension.
- Use the filename extension only to identify which files to use for calculating the most common words, i.e. consider only files which have the extension `.txt`.

Grading

Your code will be graded on correctness and efficiency. Your code should be at least as efficient as the included Python solution (described in the appendix). You should design some of your own test cases to make sure your code is correct.

Submission

Submit a single file for grading: `analyzeDir.cpp`.

While the starter code contains many different files, the **only** file you are allowed to modify is `analyzeDir.cpp`. Do not modify any other files. All code you write must go in the `analyzeDir.cpp` file, and that should be the only file you will submit for grading. We will test your code by supplying our own `main()` function, which will be different from the `main()` function in the starter code. It is therefore vital that you maintain the same function signature as declared in `analyzeDir.h`. Before you submit `analyzeDir.cpp` to D2L, make sure it works with unmodified files from the starter code!

Appendix 3 – python solution

The repository includes a Python program `analyzeDir.py` that implements the assignment. This should help you design your own test cases and see what the expected output should look like. Your C++ code will need to run at least as fast as this Python program, and it should produce identical results.

Here is an example of running it on the `test3` directory with N=5:

```
$ ./analyzeDir.py 5 ~pfederl/public/cpsc457f22/a2/extra-tests/test3

Pre-counting files... 4
Analyzing: ################################################## (100.00%)
--------------------------------------------------------------
Largest file:      "happy.jpg"
Largest file size: 26155
Number of files:   4
Number of dirs:    9
Total file size:   35681
Most common words from .txt files:
 - "could" x 5
 - "elizabeth" x 4
 - "catherine" x 3
 - "marriage" x 3
 - "their" x 3
Vacant directories:
  - "empty2.txt/file2.jpg"
  - "empty2.txt/file1.png"
  - "empty1"
  - "e3"
Largest images:
 - "happy.jpg" 506x900
 - "what-is-this" 192x199
--------------------------------------------------------------
```

Please note that the python program displays a progress bar as it is scanning the directory, which is not something you should attempt to implement in your own code.

General information about all assignments:

- All assignments are due on the date listed on D2L. Late submissions will not be marked.
- Extensions may be granted only by the course instructor.
- After you submit your work to D2L, verify your submission by re-downloading it.
- You can submit many times before the due date. D2L will simply overwrite previous submissions with newer ones. It is better to submit incomplete work for a chance of getting partial marks, than not to submit anything. Please bear in mind that you cannot re-submit a single file if you have already submitted other files. Your new submission would delete the previous files you submitted. So please keep a copy of all files you intend to submit and resubmit all of them every time.
- Assignments will be marked by your TAs. If you have questions about assignment marking, contact your TA first. If you still have questions after you have talked to your TA, then you can contact your instructor.
- All programs you submit must run on `linuxlab.cpsc.ucalgary.ca`. If your TA is unable to run your code on the Linux machines, you will receive 0 marks for the relevant question.
- Unless specified otherwise, you must submit code that can finish on any valid input under 10s on linuxlab.cpsc.ucalgary.ca, when compiled with `-O2` optimization. Any code that runs longer than this may receive a deduction, and code that runs for too long (about 30s) will receive 0 marks.
- **Assignments must reflect individual work**. For further information on plagiarism, cheating and other academic misconduct, check the information at this link: http://www.ucalgary.ca/pubs/calendar/current/k-5.html.
- Here are some examples of what you are not allowed to do for individual assignments: you are not allowed to copy code, pseudo-code or written answers (in part, or in whole) from anyone else; you are not allowed to collaborate with anyone; you are not allowed to share your solutions with anyone else; you are not allowed to sell or purchase a solution. This list is not exclusive.
- We will use automated similarity detection software to check for plagiarism. Your submission will be compared to other students (from current and previous semesters), as well as to any known online sources. Any cases of detected plagiarism or any other academic misconduct will be investigated and reported.