CPSC 457: Principles of Operating Systems

Assignment 2: more system calls, directory structures

Weight: 19%

Collaboration

Discussing the assignment requirements with others is a reasonable thing to do and an excellent way to learn. However, the work you hand in must ultimately be your work. This is essential for you to benefit from the learning experience and for the instructors and TAs to grade you fairly. Handing in work that is not your original work but is represented as such is plagiarism and academic misconduct. Penalties for academic misconduct are outlined in the university calendar.

Here are some tips to avoid plagiarism in your programming assignments.

1. Cite all sources of code you hand in that are not your original work. You can put the citations into comments in your program. For example, if you find and use code found on a website, include a comment that says, for example:

   # The following code is from https://www.quackit.com/python/tutorial/python_hello_world.cfm.

   Use the complete URL so that the marker can check the source.

2. A tool like chat-GPT can be used to improve small code blocks. For example, three lines of code. If you get help from code assistance like Chat-GPT, you should comment above the block of code you requested assistance on debugging or improving and cite the tool used to get that suggestion. Using a tool like chat-GPT to write the majority of your assignment requirements will be treated as plagiarism if found without citation, and with citation, it will be treated as 0 for the component the student did not complete. Code improvement of short length will get credit if commented/cited properly.
3. Citing sources avoids accusations of plagiarism and penalties for academic misconduct. **However, you may still get a low grade if you submit code not primarily developed by yourself. Cited material should never be used to complete core assignment specifications unless clearly approved. Before submitting, you can and should verify any code you are concerned about with your instructor/TA.**
4. Discuss and share ideas with other programmers as much as you like, but make sure that when you write your code, it is your own. A good rule of thumb is to wait 20 minutes after talking with somebody before writing your code. If you exchange code with another student, write code while discussing it with a fellow student, or copy code from another person's screen, this code is not yours.
5. **Collaborative coding is strictly prohibited. Your assignment submission must be strictly your code**. Discussing anything beyond assignment requirements and ideas is a strictly forbidden form of collaboration. This includes sharing code, discussing the code itself, or modelling code after another student's algorithm. **You can not use (even with citation) another student's code.**
6. Making your code available, even passively, for others to copy or potentially copy is also plagiarism.
7. We will look for plagiarism in all code submissions, possibly using automated software designed for the task. For example, see Measures of Software Similarity (MOSS - https://theory.stanford.edu/~aiken/moss/).
8. Remember, if you are having trouble with an assignment, it is always better to go to your TA and/or instructor for help rather than plagiarizing. A common penalty is an F on a plagiarized assignment.

Late Policy

All students will have 6 total days during the semester that can be used for none, one, some, or all the 6 assignments. Students can use these days all on one assignment, or some on each of the six.

Each 24-hour period late after an assignment deadline counts as one full day regardless of how many hours the assignment was late within that period. For example, deadlines are generally Fridays 11:59pm local time. That means an assignment submitted any time Saturday before 11:59pm local time will be considered as 1 day late and count against the students 5 total days.

As long as a student still has days left to use, their assignment will be graded without penalty. If a student has no more days left or their submission exceeds the days they have remaining, then they will receive a 0 grade for a late assignment. Students who use up late days on earlier assignments will not have them available for later assignments.

TAs will indicate the student's late day usage in grading feedback.

Due date is posted on D2L. Your D2L submission should include the files requested and a link to a Gitlab repository you used while completing the assignment with your TA added as a **Developer** role. Help with Gitlab Clone/Developer role access is available in D2L video.

Description

For this assignment you will write code that recursively a directory for all files and sub-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 of the largest directories (non-empty)
- up to N most common words in .txt files;
- a list of all top-level vacant directories (defined below); and
- up to N largest dimension images.

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.

There will be a provided code structure that you can work within and examples of a number of system calls being used. Looking in sub-folders and tracking the data returned from system calls is a large part of the assignment. Like assignment 1 there is some example projects you can cite code from to complete parts of the assignment not directly related to directory structure.

Starter Code

Start by cloning the repository with starter code (type the following on the command line on the linux lab computers). You may need to update the repository if you pulled it before to see future assignment changes before starting an assignment:

```
$ git clone https://csgit.ucalgary.ca/jwhudson/cpsc457w24.git

$ cd cpsc457w24/analyzedir
```

The repository contains the following files:

analyzeDir.py	Python program that completes the assignment output correctly in python. Use this like you did the assignment 1 reference for correctness.
main.cpp	Do not change this file. It is how your program starts.
analyzeDir.h	main.cpp assumes you do not modify this header and that your analyzeDir.cpp fulfills the requirements of the header file.
analyzeDir.cpp	Where you complete your assignment. Modify the based function required by analyzeDir.h and add other functions it uses to complete assignment 2.
test*	Five different test directories that can be used for early testing.
runner.sh	**$ sh runner.sh** is quick way to run a full diff check between all 5 provided test directories with analyzeDir.py and the analyzeDir from make command
MakeFile	Used to make and clean your cpp code in right order (main.cpp, anaylzeDir.h, analyzeDir.cpp)

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. 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.

Here is how you can invoke the **analyzeDir** to analyze "**test1**" directory with **N=10**, but note that the results are incorrect, since **analyzeDir()** is incomplete:

```
$ python python./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
Largest directories:
 - "path1" x 150
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 - "dir1/down.png" 16x16
------------------------------------------------------------
```

The starter code includes a python solution called **analyzeDir.py** which is described in the appendix. You can use it to see what the correct output should look like. It is clearly different form this output.

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 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 scan, you need to determine which of the encountered files contains the most bytes, using the **stat()** system call. 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). If multiple files have the same maximum size, report the path to any of them.

long n_files, n_dirs;

The **n_files** field should contain the total number of files encountered during the scan. Similarly, **n_dirs** should contain the total number of sub-directories, 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.

std::vector<std::pair<std::string, int>> largest_dirs;

This vector should include a list of up to **N** largest directories (non-empty/size > 0). Here we will only determine a size of a directory by the size of all the files immediately in the directory (you have this size from stat() already if you done the prior. You should not add the size of the sub-directories or files within it to the size tracked for this list. List should be sorted by decreasing size first (largest size directory printed first), and then directory path second (std::string default order) for directories of the same size.

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

This field 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 first by the number of occurrences in descending order (largest printed first), and then word second (std::string default order).

For this assignment, text file is any file that has extension **".txt"**, and word is a sequence of **more than 5** alphabetic characters, converted to lower case.

Couple of examples: the string "My name is Jonathan, my password: is abc1ab2ZYZaaa" contains words "jonathan", "password" and "zyzaaa"; the string "HeLLLo,Helllo" contains one word "helllo", 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 **wordhistogram** example below for motivation.

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

This field will contain a list of all top-level 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.

A top-level vacant directory is a vacant directory, whose parent is not vacant. This means that once a directory is reported in **vacant_dirs**, none of its descendants can be included in this list. For example, if **"dir1/sub/xy"**, **"dir1/sub"** and **"dir1"** are all vacant directories, but **"."** is not vacant, then only **"dir1"** is top-level vacant, and **"dir1"** should be the only one included in the results.

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

Sort this list by order of directory path (std::string default order).

std::vector<ImageInfo> largest_images;

> This field will contain a list of up to **N** largest dimension 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 images largest dimension (largest number of pixels in image dimensions =max(width , height). If this first order sort is tied, then sort by width, then height, and finally sort by filepath last (std::string default order).
>
> IMPORTANT: All paths reported in any of the results must be relative to the current directory, but they must not begin with "./" (dot slash), nor contain any unnecessary "." (dot) and ".." (dot dot) parts. The included python solution follows this requirement. If you want, you can remove the leading "./" from your paths as final postprocessing step.

If you follow the correct sort orderings you should be able to run **analyzeDir.py** and retrieve exact results. If you don't follow the sort orderings you can get partial marks on tests if the primary (first) order is correct, but the tie break on second, third, fourth orders is not. For example, if the largest directories are sorted correctly by size but you did not complete sorting by naming for tied directories, then you will get partial marks. The penalty will only be minor for not completing the full sort order.

IMPORTANT: All paths reported in any of the results must be relative to the current directory, but they must not begin with **"./"** (dot slash), nor contain any unnecessary **"."** (dot) and **".."** (dot dot) parts. The included python solution follows this requirement. If you want, you can remove the leading **"./"** from your paths as final post-processing step

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 its 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 to 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. Other than calling the **identify** program, you may not use **popen()** for any other purpose.

Extra test directories

Additional test directories are available on linuxlab machines in
~jwhudson/public/cpsc457w24/a2/extra-tests

You can run your code or the python solution on these test directories like this:

```
$ time ./analyzeDir 10 ~jwhudson/public/cpsc457w24/a2/extra-tests/test3

$ time python ./analyzeDir.py 10 ~jwhudson/public/cpsc457w24/a2/extra-tests/test3
```

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: **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: **word-histogram**

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

Allowed APIs

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

- **popen(), pclose(),** but only to get the output of the identify utility
- **stat(), opendir(), closedir(), readdir(), getcwd(), chdir()**
- **open(), close(), read(), fopen(), fread(), fclose(), fgets(), fgetc(), qsort()**
- any C++ containers & associated algorithms
- C++ streams, C++ string related APIs
- **std::sort, std::filesystem**

If you want to use other APIs, please check with your TA whether it would be allowed.

Not-allowed APIs

- You may not use **system(3**) at all.
- You may not call any external programs, other than identify. For example, you may not call **find, awk, grep, 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 entries in a list are tied in the primary ordering (ex. Size for directories), you can get partial marks if you return those in arbitrary order. However, full marks need you to return them in the request second, third, fourth orderings requested.
- 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), but read below for more explanation.

Sometimes your C++ solution will run a little bit slower than the Python solution. This is expected. In case you are wondering why: the **libc**'s implementation of **popen** is less efficient than the one used in the python solution. Essentially, the **libc**'s **popen** ends up calling **fork()** twice.

Consequently, on directories with many files, your C++ code will end up running a bit slower than the python version.

To give you an idea what I expect in terms of performance, here are some timings I obtained using my own quick student like C++ solution on the extra test directories:

- On test6, which contains 1342 files, the Python solution took 27s, my C++ solution took 29s;
- On test9, which contains only 72 files, Python finished in ~2.5s, and my C++ solution in ~2s.

Do not forget to design some of your own test cases to make sure your code is correct.

When timing your code, run your test at least twice, back-to-back, and use the best time. This will remove the effects of filesystem caching, and you will get more reliable timings.

Submission

Submit one file for this assignment to D2L:

- Your solution in a file called **analyzeDir.cpp.**

Submit this as a separate file. **Do not** submit an archive, such as ZIP or TAR. If you submit an archive, you will receive a penalty.

Submit the web address of your Gitlab repository you used for your assignment (I recommend one repository for all 6 of your assignments that you can re-use). Your TA must be added as a

Developer. There are penalties for submissions without the ability to access the corresponding students Gitlab by the TA.

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!

General information about all assignments

All assignments are due on the date listed on D2L. Late submissions without remaining late days banked will not be marked.

1. Extensions beyond the late day policy can be discussed more than 5 business days in advance and are granted only by the course instructor.
2. After you submit your work to D2L, verify your submission by re-downloading it.
3. 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.
4. Assignments are likely going to 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.
5. All programs you submit must run on **linux lab** or **cslinux.ucalgary.ca.** If your TA is unable to run your code on these, you will receive 0 marks.
6. Unless specified otherwise, you must submit code that can finish on any valid input under 10s on **linux lab** or **cslinux.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.
7. **Assignments must reflect individual work.** Here are some examples of what you are not allowed to do for individual assignments: you are not allowed to copy 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 (code or pseudocode) with anyone; you are not allowed to sell or purchase a solution; you are not allowed to make your code available publicly (e.g. via public git repositories). This list is not exclusive. 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 .

8. We will use automated similarity detection software to check for plagiarism. Your submission will be compared to other students (current and previous), as well as to any known online sources. Any cases of detected plagiarism or any other academic misconduct will be investigated and reported.

Appendix – Python Solution analyzeDir.py

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 must produce identical results. Here is an example of running it on the **test3** directory with **N=5**:

```
$ python ./analyzeDir.py 5 ~jwhudson/public/cpsc457w24/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
Largest directories:
 - "." x 31547
 - "empty2.txt/file2.cpp" x 4134
Most common words from .txt files:
 - "elizabeth" x 4
 - "catherine" x 3
 - "marriage" x 3
 - "appeared" x 2
 - "certain" x 2
Vacant directories:
 - "e3"
 - "empty1"
 - "empty2.txt/file1.png"
 - "empty2.txt/file2.jpg"
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. This can be turned off with a third argument False. Triggering this will let you create output you'd expect to match the output of your **cpp** program you complete in **analyzeDir.cpp**

```
$ python ./analyzeDir.py 5 ~jwhudson/public/cpsc457w24/a2/extra-tests/test3 False
```