CPSC 457 - Assignment 2

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

Q1. Multithreaded subset-sum problem [20 marks]

Write a multithreaded program `subset.[c|cpp]` that counts how many subsets of a given set of 32-bit signed integers sums to zero. A single-threaded solution to this problem is provided here: https://gitlab.com/cpsc457/public/subset-sum . It uses an exhaustive search for all possible subsets. In your solution you need to use the same exhaustive search, but spread the execution evenly between multiple threads.

You program should accept a single command line argument, indicating the number of threads to create. Your program should work for a number of threads in range 1..32.

Your program will read the integers from standard input. You can assume all numbers will be separated by white spaces, i.e. you can read them using `scanf("%ld")` or `std::cin`. You can assume that the total number of integers in the input will be in range 1..31. Here a sample interaction with your program:

```
$ g++ -O2 subset.cpp -o subset
$ cat test1.txt
1 2 4 8 16
32 64 -20 -10
$ ./subset 3 < test1.txt
Using 3 thread(s) on 9 numbers.
Subsets found: 3
```

The reason the program outputs 3 is because there are 3 subsets that sum to 0:

- 4 + 16 – 20 = 0
- 2 + 8 – 10 = 0
- 2 + 4 + 8 + 16 - 20 – 10 = 0

Here is another example:

```
$ echo "2 2 2 -4" | ./subset 1
Using 1 thread(s) on 4 numbers.
Subsets found: 3
```

Notice there is only 1 unique solution, but it repeats 3 times because there are repeated values in the input. You should not attempt to look for unique solutions.

You may not use any synchronization primitives for this question, such as mutexes or semaphores. Your solution should be entirely lock-free, and it should provide N times speedup where N is the number of threads. Your solution will be graded both on correctness and the speedup it achieves.

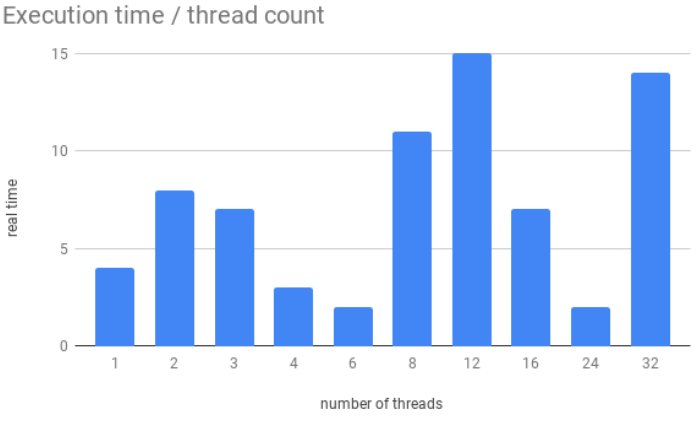
Your multithreaded solution must output correct results. If it does not, you will not receive marks. This means you need to find a way to avoid race conditions. Since you are not allowed to use any synchronization primitives, you need to make sure that each thread performs a task that only modifies memory for that task. Some hints on how to accomplish that are provided in the sample solution, and your TAs will try to help you with more hints.

Your program cannot crash for any reason. For example, if the command line arguments are invalid, report an error and quit. Programs that crash will receive a significant penalty.

Q2 – Written answer [5 marks]

Time your multithreaded solution from Q1 on the numbers in `test3.txt` file using the `time` command. Record the real-time for 1 thread, then 2, 3, 4, 6, 8, 12, 16, 24 and 32 threads. Also record the timings for the sample solution I provided. In your report, include a table of these timings, and a bar graph, both formatted similar to these:

Threads	Timing (s)
1 (original)	5
1	4
2	8
3	7
4	3
6	2
8	11
12	15
16	7
24	2
32	14



The numbers in the above table and graph are random; your timings will look diffferent.

Answer the following questions:

a) With N threads you should see N-times speed up compared to the original single threaded program. Do you observe this in your timings for all N?
b) Why do you stop seeing the speed up after some value of N?

Q3 – Finding empty directories [10 marks]

Write a program `findEmptyDirs.[c|cpp]` that recursively scans the current working directory and reports all empty directories. The output of your program should be identical to the output of '`find . -type d -empty`'.

Below is an example of how your program should behave:

```
$ mkdir -p a/b/c a/b/x a/c a/d
$ find . -type d -empty
./a/d
./a/b/x
```

```
./a/b/c
./a/c
$ g++ findEmptyDirs.c -o findEmptyDirs
$ ./findEmptyDirs
./a/b/x
./a/b/c
./a/c
./a/d
$ touch a/d/f a/b/c/f
$ ./findEmptyDirs
./a/b/x
./a/c
$ cd a/c
$ ../../findEmptyDirs
.
```

Your program must match the output of the `find` command precisely, although it is OK if your program ouputs the results in different order. A sample program that recursively finds and prints all files and directories in the current directory is provided here:

https://gitlab.com/cpsc457/public/find-empty-directories

You can use this program as a starting point.

Q4 – Finding duplicate files [20 marks]

Write a program `findDupFiles.[c|cpp]` that detects and reports duplicate files for a given set of filenames. Your program will read the list of filenames from standard input, where each filename will be specified on a separate line. For each filename your program will compute a SHA256 message digest for the file, and then compare these digests to each other to find the duplicates. It will then report these duplicates to standard output.

To get the digest of the file, you should use `popen()` in your program to call the external program `sha256sum`. A sample program that does this and detects duplicates on first two files is provided here: https://gitlab.com/cpsc457/public/finddupfiles . Feel free to use it as a starting point.

Here is an example of how to use `sha256sum` from command line to get a digest of a file `1.txt`:

```
$ echo hello > 1.txt # create a file 1.txt with the word “hello”
$ sha256sum 1.txt # compute the digest
5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f6be03
1.txt
```

The long hexadecimal string is the digest of the file contents. The idea behind the digest is that it will be different for every file – as long as their contents are different. Here is an example of a digest for a slightly different file:

```
$ echo hello! | sha256sum # compute the digest from standard input
c8a31cb076b21999bd2cdcfa5f446a7a6644de88037087112fa18bd90cc13984 -
```

As you can see, the digest is completely different, yet all we added was an extra character “!”.

If your program is unable to obtain a digest (eg. the filename points to a directory, unreadable file or non-existant file), you should report those. This is indicated by the `sha256sum` program by returning a non 0 value, which you can read when you close the stream using `pclose()`. The sample program I provided already does that.

Example of expected output of your program is shown below. The output of your program should match it as closely as possible. Please note that the files are reported with the paths, exactly as they are specified on standard input.

```
$ g++ findDupFiles.cpp

$ mkdir -p tests/1 tests/2 tests/33
$ echo hello > tests/hello.txt
$ echo hello > tests/1/hello2.txt
$ echo hellp > tests/2/hello3.txt
$ cp tests/1/hello2.txt tests/hello4.txt
$ cp tests/2/hello3.txt tests/1/hello5.txt
$ cp tests/2/hello3.txt tests/hello6.txt
$ chmod -r tests/hello6.txt
$ echo world > tests/2/hello7.txt
$ mkdir tests/33/hello8.txt
$ find tests -name "hello*.txt" | ./a.out
Match 1:
  - tests/1/hello2.txt
  - tests/hello4.txt
  - tests/hello.txt
Match 2:
  - tests/1/hello5.txt
  - tests/2/hello3.txt
Could not compute digests for files:
  - tests/hello6.txt
  - tests/33/hello8.txt
```

Digest for `hello8.txt` could not be computed because it is a directory. Digest for `hello6.txt` could not be computed because it does not have read permissions. File `hello7.txt` is not mentioned in the results at all because it does not have a duplicate.

Submission

Submit 4 files to D2L:

`report.[pdf|txt]`	answers to all written questions
`subset.[c|cpp]`	solution to Q1 in C or C++
`findEmptyDirs.[c|cpp]`	solution to Q3 in C or C++
`findDupFiles.[c|cpp]`	solution to Q4 in C or C++

General information about all assignments:

1. All assignments are due on the date listed on D2L. Late sumissions will be not be marked.
2. Extensions may be granted only by the course instructor.
3. After you submit your work to D2L, verify your submission by re-downloading it.
4. You can submit many times before the due date. D2L will simply overwrite previous submissions with newer ones. It's 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.
5. 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.
6. All programs you submit must run on `linux.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.
7. 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.
8. 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 with anyone else; you are not allowed to sell or purchase a solution. This list is not exclusive.
9. We will use automated similarity detection software to check for plagiarism. You 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.