Project: Parallel Character Counting

Due Date: Friday, November 26, 2021 at **8 PM HKT**. Late submissions will be penalized as per syllabus.

I. Project Instructions

Overview

Huffman coding is a lossless data compression algorithm that uses the variable length coding table to encode the characters in the file. The variable-length coding table is obtained by evaluating the frequency of each character in the files. This project aims to count the number of different characters in each file using multiple threads, which is the first step to construct the variable-length coding table for Huffman coding.

There are four specific objectives to this project:

- To familiarize yourself with the Linux pthreads package for writing multi-threaded programs.
- To learn how to parallelize a program.
- To learn how to write the results concurrently.
- To learn how to program for performance.

Project Organization

Each group should do the following pieces to complete the project. Each piece is explained below:

- **Design** **30 points**
- **Implementation** **30 points**
- **Evaluation** **40 points**

You're required to submit a project report which concisely describes your design, implementation, and experimental results.

Design

The design space of building an effective counting characters tool is large. A practical character counting that achieves a better performance will require you to address (at least) the following issues (see part II. Project Description for more details):

- How to access the input file efficiently.
- How to parallelize the character counting.
- How to determine the number of threads to create.
- How to write the counting results for each input file concurrently.

- How to counting characters among multiple files in parallel efficiently.

In your project report, please describe the main techniques or mechanisms proposed to parallelize the counting, list and explain all the functions used in this project, summary and analyze the results. You should also discuss any difficulties encountered, what was learned. If you are working in a team, describe each team member's contribution.

Implementation

Your code should be nicely formatted with plenty of comments. Each function should be preceded by a header comment that describes what the function does. The code should be easy to read, properly indented, employ good naming standards and structure, and correctly implement the design. Your code should match your design.

Evaluation

We provide 10 test cases for you to test your code. Before submitting your work, please run your pcount on the test cases and check whether your pcount counts characters of input files correctly. Time limitation is set for each test case, and if this limit is exceeded, your test will fail. For each test case, your grade will be calculated as follows:

- **Correctness.** Your code will first be measured for correctness, ensuring that it counts characters of input files correctly. You will receive total points if your solution passes the correctness tests performed by the test script. You will receive **zero points** for this test case if your code is buggy.
- **Performance.** If you pass the correctness tests, your code will be tested for performance; The test script will record the running time of your program for performance evaluation. Shorter time/higher performance will lead to better scores.

In your project report, summary and analyze the results. You can also compare your solution with the provided baseline implementation.

Tips: Keep a log of work you have done. You may wish to list optimizations, what you tried and failed, etc. Keeping a good record will make it easy to put together your final writeup.

Bonus

You're encouraged to be creative and innovative, and this project award bonus points (**up to 10 points**) for additional or exemplary work.

- New ideas/designs are welcome to explore the parallelism of counting characters in each file entirely.
- To encourage healthy competition and desire to improve, we'll provide a scoreboard that shows scores for the top 10 groups. The latest scores are displayed, rank-ordered, on the scoreboard. We will award bonus points for the top 10 groups. Details will be posted on Canvas later.

Language/Platform

The project should be written in ANSI standard C.

This project can be done on Linux (recommended), macOS, or Windows using Cygwin. Since grading of this project will be done using the gateway Linux server, students who choose to develop their code on any other machine are strongly encouraged to run their programs on the gateway Linux server before turning it in. There will be no points for programs that do not compile and run on the gateway Linux server, even if they run somewhere else.

Handing In

The project can be done individually, in pairs, or in groups, where each group can have a maximum of three members. All students are required to join one project group in Canvas ("People" section > "Groups" tab > Project Group). Self-sign-up is enabled for these groups. Instead of all students submitting a solution to the project, Canvas allows one person from each group to submit on behalf of the team. Please be sure to indicate who is in your group when submitting the project report.

Before you hand it in, make sure to add the requested identifying information about your project group, which contains the project group number, full name, and e-mail address for each group member.

When you're ready to hand in your solution, go to the course site in Canvas, choose the "Assignments" section > "Project" group > "Project" item > "Submit Assignment" button, and upload your files, including the following:

1) A Microsoft Word or Adobe PDF document which concisely describes your design, implementation, and experimental results;
2) The source file, pcount.c.

Academic Honesty

All work must be developed by each group separately. Please write your own code. **All submitted source code will be scanned by anti-plagiarism software**. If the code does not work, please indicate it in the report.

Questions?

If you have questions, please first post them in discussions on Canvas so others can benefit from the TA's answer. Avoid posting code that will give away your solution or allow others to cheat. If this does not resolve your issue, contact the TA (Ms. REN Tinayu <tianyuren2-c@my.cityu.edu.hk>), or come to office hours.

Acknowledgments

This project is taken (and modified) from the OSTEP book written by Remzi and Andrea Arpaci-Dusseau at the University of Wisconsin. This free OS book is available at http://www.ostep.org. Automated testing scripts are from Kyle C. Hale at the Illinois Institute of Technology.

Disclaimer

The information in this document is subject to change **with** notice via Canvas. Be sure to download the latest version from Canvas.

II. Project Description

For this project, you will implement a parallel version of counting characters using threads. This is the first step to construct the variable-length coding table for Huffman coding.

You will use your `pcount` program to count the number of each character in the file. Also, you may process more than one file.

Thus, if we had two files named 2-0.in and 2-1.in, with the following contents:

2-0.in

```
aaaabbbbbbddddffffgggggggg
```

2-1.in

```
ccccoooffffkkkkzzzznnnnnnnnnn
```

Your `pcount` program will use POSIX threads to parallelize the counting characters process. The command line will be as follows:

```
$ ./pcount 2-0.in 2-1.in
```

The tool would turn it (logically) into:

```
2-0.in
a 4
b 6
d 4
f 4
g 7
2-1.in
c 4
f 5
k 4
n 11
o 3
z 4
```

All input files include **only** 26 lowercase English characters. Moreover, the exact output format is quite important; here, for each file, you need to print the **file name** before the results of counting different characters of that file. The counting results should be **formatted**: the first output is a character, the second output is a space, and the final output is an integer of counting results corresponding to that character. Moreover, the counting result of each character needs to be output in a separate line. You can use `printf("%c %d\n",*,*)`

`to` print out your results. It is important to note that the output results need to be in **dictionary order**[1]. (i.e., if you have two files, 2-9.in and 2-10.in, you need to print the result of 2-10.in first). For every file, you should also print the characters from *a* to *z*, and the characters with a counting result of 0 are **not** output. For pcount, all the output should be written to standard output (the stdout file stream). If your result cannot match the standard output, you will receive zero points for those failed tests.

Doing so effectively and with high performance will require you to address (at least) the following issues:

- **How to parallelize the character counting.** The central challenge of this project is to parallelize the character counting process. You are required to think about whether the character counting process can be separated into several sub-processes, what sub-processes can be done in parallel, and what sub-processes must be done serially by a single thread. Then, you are required to design your parallel counting characters tool as appropriate. For example, does it possible to count characters in several small sub-files using multiple threads instead of counting characters in a large file using only one thread? If it's possible, how to divide the large file? How to count characters in those small sub-files using multiple threads? How to merge the results of several small sub-files? One interesting issue that the "best" implementations will handle is this: what happens if one thread runs much slower than another? Does the process of counting different characters give more work to faster threads? This issue is often referred to as the *straggler problem*.

- **How to determine the number of threads to create.** On Linux, the determination of the number of threads may refer to some interfaces like `get_nprocs()` and `get_nprocs_conf()`; You are suggested to read the man pages for more details. Then, you are required to create an appropriate number of threads to match the number of CPUs available on whichever system your program is running.

- **How to concurrently write the counting results for each input file.** You are required to think about merging the results from the multiple threads since multiple threads will process a file. You are suggested to set a global variable to save the results of various threads, and there may need to design a lock to protect this variable.

- **How to efficiently count different characters in multiple files in parallel.** You are required to think about how to parallelize the character counting processes of multiple files. A naïve way is to process the parallel character counting process of each file sequentially. However, this method cannot fully explore the parallelism of the character counting processes of multiple files. You are required to investigate the parallelism between the character counting processes of various files and design an efficient and fast parallel method to count characters in multiple files. Note that when the input contains directories, you can first obtain the paths of all files in the directories recursively using readdir(), then count them in **dictionary order**.

[1] Dictionary order(or lexicographical order), https://en.wikipedia.org/wiki/Lexicographic_order

- **How to access the input file efficiently.** On Linux, there are many ways to read from a file, including C standard library calls like `fread()` and raw system calls like `read()`. One particularly efficient way is to use memory-mapped files, available via `mmap()`. By mapping the input file into the address space, you can access bytes of the input file via pointers and do so quite efficiently.

To tackle these problems, you should first understand the basics of thread creation and perhaps locking and signaling via mutex locks and condition variables. Review the tutorials and read the following chapters from the OSTEP book carefully to prepare yourself for this project.

- Intro to Threads
- Threads API
- Locks
- Using Locks
- Condition Variables

III. Project Guidelines

1. Getting the Starter Code

The project is to be done on the CSLab SSH gateway server, to which you should already be able to log in. As before, follow the same copy procedure as you did in the previous tutorials to get the starter code. The start code is available at */public/cs3103/project* on the gateway server. `project.zip` contains the following files/directories:

```
/project
├── Makefile
├── pcount                  <- A sample solution (executable file)
├── pcount.c                <- modify and hand in pcount.c file
├── README.md
└── tests
    ├── bin
    │   ├── generator.py
    │   ├── generic-tester.py
    │   ├── serialized_runner.py
    │   └── test-pcount.csh
    ├── config
    │   ├── 1.json
    │   ├── ...
    │   └── 10.json
    ├── stdout
    │   ├── 1.err
    │   ├── 1.out
    │   ├── 1.rc
    │   ├── ...
    │   └── 10.rc
    └── tests-pcount
        ├── 1
        │   └── 1-0.in
        ├── 2
        │   ├── 2-0.in
        │   ├── 2-1.in
        │   ├── ...
        │   └── 2-11.in
        ├── ...
        └── 10
            ├── 10-0
            │   ├── 10-0-0.in
```

```
│   ├── 10-0-1.in
│   └── 10-0-2.in
├── 10-1
│   ├── 10-1-0.in
│   ├── 10-1-1.in
│   └── 10-1-2.in
├── 10-2.in
├── ...
└── 10-7.in

20 directories, 107 files
```

Start by copying the provided files to a directory in which you plan to do your work. For example, copy */public/cs3103/project/project.zip t*o your home directory and extract the ZIP file with the unzip command. Note that the file size of project.zip is only 10MB, but the uncompressed project directory has a size of 5GB. It takes about 120 seconds to unzip the project.zip file on our gateway server. After the unzip command extracts all files to the current directory, change to the project directory, and take a look at the directory contents:

```
$ cd ~
$ cp /public/cs3103/project/project.zip .
$ unzip project.zip
$ cd project
$ ls
```

A sample `pcount` (executable file only, no source codes) is also provided. This pcount uses pthread to support the parallel character counting of multiple files or directories. When the input files of the character counting process contain directories, the pcount first obtains the paths of all files in the directories recursively, then treats the character counting process as the separately counting characters of multiple files. For the various files' character counting, the pcount uses mmap to map files into numerous pages and counts characters for every page in parallel. The parallel character counting process can be treated as a producer-consumer problem. The pcount uses one thread of the producer to map files and multiple threads of consumers to count different characters of pages. You can use this one as a baseline implementation for performance evaluation. After building your pcount, the provided pcount will be overwritten. But don't worry, you can always copy it from */public/cs3103/project/pcount.*

2. Writing your pcount program

The `pcount.c` is the file you will be handing in and is the only one you should modify. Write your code from scratch or design your `pcount.c` according to the above requirements. Again, it's a good chance to learn (as a side effect) how to use a proper remote IDE such as VS Code[2][3].

3. Building your program

A simple makefile that describes how to compile `pcount` is provided for you. To compile your `pcount.c` and generate the executable file, use the make command within the directory containing your project. It will display the command used to compile the `pcount`.

```
$ make
gcc -Wall -Werror -pthread -O pcount.c -o pcount
```

Note that the `-Werror` compiler flag is specified. It causes all warnings to be treated as build errors. It would be better to fix the compiling issue instead of disabling `-Werror` flag.

If everything goes well, there would be an executable file `pcount` in it:

```
$ ls
Makefile  pcount  pcount.c  README.txt  tests
```

If you make some changes in `pcount.c` later, you should re-compile the project by running `make` command again.

To remove any files generated by the last `make`, use the `make clean` command.

```
$ make clean
rm -f pcount

$ ls
Makefile  pcount.c  README.txt  tests
```

4. Testing your C program

We also provide 10 test cases for you to test your code. You can find them in the directory *tests/tests-pcount/*.

- Test cases 1-6: Test cases 1, 2 are small files. For test case 3, the file is empty. For test case 4, there is no file. If no files are specified, the program should exit with return code 1 and print "pcount: file1 [file2 ...]" (followed by a newline).

  Test case 1) single file test – a small file.
  Test case 2) multiple files test – twelve small files of different file sizes.
  Test case 3) empty file test.
  Test case 4) no files test.

[2] Visual Studio Code documents, https://code.visualstudio.com/docs

[3] VS Code Remote Development, https://code.visualstudio.com/docs/remote/remote-overview

Test case 5) single large file test – a large file.
Test case 6) multiple large files test – six large files of different file sizes.

- Test case 7-10: Some files are stored in a directory, and you are required to count different characters in the directory and other files.

Test case 7) directory test – a directory that contains twelve small files of different file sizes.
Test case 8) mixed test 1 – a directory that contains six small files and six large files outside the directory.
Test case 9) mixed test 2 – a directory that contains six large files and six small files outside the directory.
Test case 10) mixed test 3 – two directories containing three large files, three small files, and six small files outside the directory.

Each test consists of 5 files with different filename extensions:

- n.json (in *tests/config/* directory): The configuration of test case n.
  - binary: Indicates the data types of input and output are binary.
  - filename: The test case number.
  - filesize: The file size of each file. All numbers are included in a list. If an item is a list of numbers, it indicates it is a directory.
  - timeout: Time limitation (seconds).
  - seed: The seed used to generate the content of input files.
  - description: A short text description of the test.
  - preparation: Code to run before the test, to set something up.
- n.rc: The return code the program should return (usually 0 or 1)
- n.out: The standard output expected from the test
- n.in: The input file of the test case
- n.err: The standard error expected from the test

you can run and test your pcount manually. For example, to run your pcount to count different characters the input file 1-0.in in tests/tests-pcount and save the results file as 1.out, enter:

```
$ ./pcount ./tests/tests-pcount/1/1-0.in > 1.out
```

To run your pcount to count the different characters of multiple input files (the input files 2-0.in, 2-1.in, 2-2.in) in *tests/tests-pcount*, and save the counting results of multiple files as 2.out, enter:

```
$ ./pcount tests/tests-pcount/2/2-0.in tests/tests-pcount/2/2-1.in tests/tests-pcount/2/2-2.in > 2.out
```

To run your pcount to count the different characters of the input directory 7-0 in tests/tests-pcount and save the results of those files as 7.out, enter:

```
$ ./pcount tests/tests-pcount/7/7-0 > 7.out
```

To run your `pcount` to count the different characters of the input directory 8-0 and some input files (the input files 8-0.in, 8-1.in) in *tests/tests-count* and save the results of those files as 8.out, enter:

```
$ ./pcount tests/tests-count/8/8-0 tests/tests-pcount/8/8-0.in tests/tests-pcount/8/8-1.in > 8.out
```

To run other test cases, please run as follows:

```
$ ./pcount tests/tests-pcount/3/3-0.in > 3.out
$ ./pcount tests/tests-pcount/5/5-0.in > 5.out
$ ./pcount tests/tests-pcount/9/9-0 tests/tests-pcount/9/9-0.in tests/tests-pcount/9/9-1.in > 9.out
```

The makefile could also trigger automated testing scripts, type:

```
$ make test
TEST 0 - clean build (program should compile without errors or warnings)
Test finished in 0.232 seconds
RESULT passed

TEST 1 - single file test, a small file of 10 MB (2 sec timeout)
Test finished in 0.038 seconds
RESULT passed

TEST 2 - multiple files test, twelve small files of 10 MB, 20 MB, 30 MB, 10 MB, 20 MB, 30 MB, 10 MB, 20 MB, 30 MB, 10 MB, 20 MB, 30 MB (2 sec timeout)
Test finished in 0.085 seconds
RESULT passed

TEST 3 - empty file test (2 sec timeout)
Test finished in 0.014 seconds
RESULT passed

TEST 4 - no file test (2 sec timeout)
Test finished in 0.006 seconds
RESULT passed

TEST 5 - single large file test, a large file of 100 MB (2 sec timeout)
Test finished in 0.053 seconds
RESULT passed
```

```
TEST 6 - multiple large files test, six large files of 100 MB, 200 MB, 300 MB, 100 MB, 200 MB, 300 MB (2 sec timeout)
Test finished in 0.278 seconds
RESULT passed

TEST 7 - directory test, a directory that contains twelve small files of 10 MB, 20 MB, 30 MB, 40 MB, 10 MB, 20 MB, 30 MB, 40 MB, 10 MB, 20 MB, 30 MB, 40 MB (2 sec timeout)
Test finished in 0.084 seconds
RESULT passed

TEST 8 - mixed test 1, a directory that contains six small files of 10 MB, 20 MB, 10 MB, 20 MB, 10 MB, 20 MB and six large files outside directory of 100 MB, 200 MB, 300 MB, 100 MB, 200 MB, 300 MB (2 sec timeout)
Test finished in 0.359 seconds
RESULT passed

TEST 9 - mixed test 2, a directory that contains six large files of 100 MB, 200 MB, 100 MB, 200 MB, 100 MB, 200 MB, and six small files outside directory of 30 MB, 40 MB, 30 MB, 40 MB, 30 MB, 40 MB (2 sec timeout)
Test finished in 0.285 seconds
RESULT passed

TEST 10 - mixed test 3, two directories that contain three small files of 10 MB, 20 MB, 10 MB and three large files of 200 MB, 100 MB, 200 MB, and six small files outside directory of 10 MB, 20 MB, 10 MB, 20 MB, 10 MB, 20 MB (2 sec timeout)
Test finished in 0.157 seconds
RESULT passed
```

The job of those automated scripts is to orderly run your `pcount` on the test cases and check if your `pcount` counts the different characters of input files correctly. TEST 0 will fail if your program is compiled with errors or warnings. Time limitation is set for each test case, and if this limit is exceeded, your test will fail. Besides, the script will record the running time of your program for performance evaluation. Lower time/higher performance will lead to better scores.