Project 3 - Multithreaded MapReduce

CSci 4061: Introduction to Operating Systems

Due Date: Wednesday, 13th Nov, 2019

1. Instructions

*(this section is modified from Programming Assignment 2 document)

- You could complete this project in a group of up to two students.
- Each group should turn in **one** copy with the names of all group members on it.
- The code must be originally written by your group. No code from outside the course texts and slides may be used—your code cannot be copied or derived from the Web, from past offerings, other students, programmer friends, etc.
- All submissions must compile and run on any CSE Labs machine located in KH 4-250.
- A zip file should be submitted through Canvas by 11:59pm on Friday, Nov 15th.
- Note: Do not publicize this assignment or your answer to the Internet, e.g., public GitHub repo. To share code between team members, create a private repo in github.umn.edu.

2. Background and Objective

In multi-threads programming, threads can perform the same or different roles. In some multithreading scenarios like producer-consumer, producer and consumer threads have different functionality. In other multithreading scenarios like many parallel algorithms, threads have almost the same functionality. In this programming assignment, we want to work on both sides in problem “word count statistics”, which is the multithreaded version of MapReduce application in programming assignment 2.

An example problem is shown in Fig. 1 below. A text file contains several lines of English characters, and this application will count the histogram of the **starting letter** for each word.

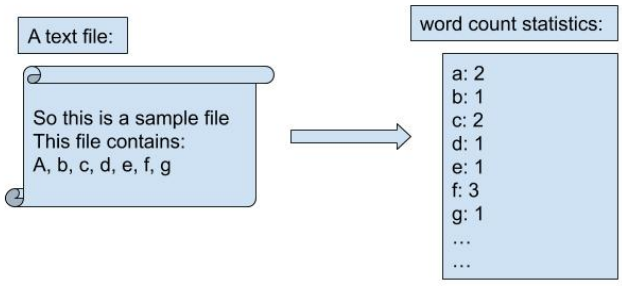

Fig. 1, an example of “word count statistics”

One idea to solve this problem is to cut the file into smaller pieces and hand it over to many workers. Then in this programming assignment, we will use multithreading to create a producer to read the file and multiple consumers to process the smaller piece data. We will have two forms of synchronization: a shared queue synchronized between producer and consumers, and a global result histogram synchronized by consumers.

Topics covered: POSIX-threading, synchronization, producer-consumer model, file IO and string processing.

3. Project Overview

In multithreading “word count statistics” application, the program contains: a master thread, one producer thread and many consumer threads (shown in Fig. 2 below). The producer and consumers will share a queue for data transfering. For program execution flow, the entire program contains 4 parts: 1) the master thread initializes the shared queue, result histogram, producer thread and consumer threads; 2) the producer thread will read the input file, cut the data into smaller pieces and feed into the shared queue; 3) the consumers will read from the shared queue, compute the “word count statistics” for its data pieces and synchronize the result with a global histogram; 4) after producer and all consumers complete their work, the master thread writes the final result into the output file.

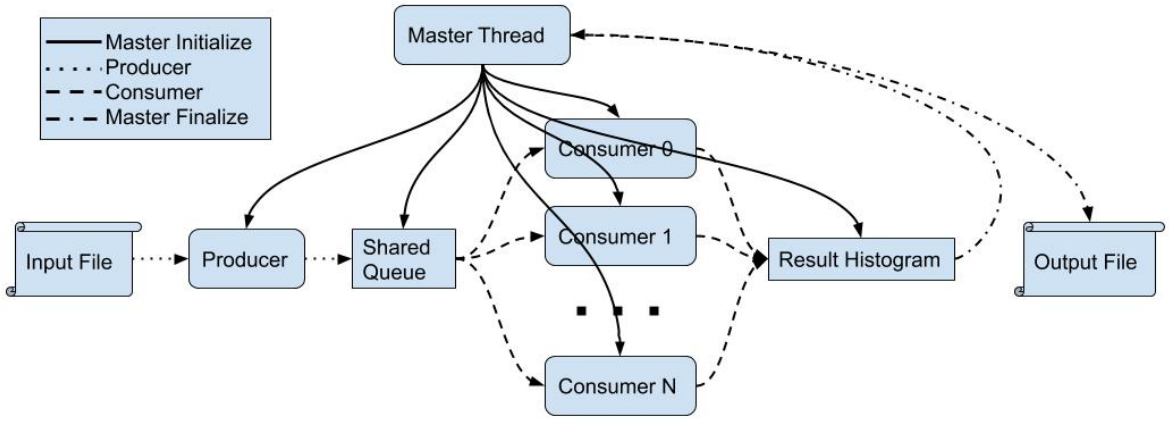

Fig. 2, 4 parts of multithreading “word count statistics”

In the next section, we will go through the implementation details of this project. The implementation requirements will be provided.

4. Project Implementation and Specifics

Before we go into each part, the application comparison between PA2 (Programming Assignment 2) and PA3 (Programming Assignment 3) needs to be clarified. Even though we use the same application “word count statistics” in PA2, there are several key differences in the input file and processing flows: 1) PA2 works on different files located in different directories, but PA3 takes **only one file** as your input file. 2) In PA2, each file contains multiple lines and each line contains only one word. But PA3’s input file **is more “natural”**, which contains multiple words in each line and the file contains multiple lines. 3) PA2 uses multi-processing, which means the parent forks several children and seperate the workload. But in PA3, you only **have one process but multiple threads**. PA3 is mainly focused on the usage of threads and thread-safe data structures.

4.0 Shared Queue

The core of this multithreading “word count statistic” application is a thread-safe shared queue. This shared queue should be implemented as a **linked-list unbounded buffer**. The producer inserts the data in the tail of the linked-list and the consumer extracts the data from the head. Also, it should be implemented in a **non-busy-waiting** way (use “mutex lock + conditional variable” or “semaphore”).

4.1 Master Initialization

In this stage, the program needs to **check the input arguments** and print error messages if argument mismatch (see Section 6 execution for arguments). Then it should perform the **initialization** on data structure and launch the producer/consumers thread. Then the master will wait for all threads to join back (4.4 Master Finalize).

4.2 Producer functionality

The main functionality of the producer thread is to read the input file and pass the data into the shared queue (by a package). The file is required to be read by **line granularity** (one line at a time), thus each consumer will work on one line at a time. Since there are multiple consumers, if the EOF is reached, the producer should **send notifications** to consumers specifying there will be no more data. The producer terminates after sending those notifications. Note that the **package** is the information transferred between producer/consumers via shared queue. It should contain the data for consumers and other information if needed.

4.3 Consumer functionality

The consumer will repeatedly check the queue for a new package, work on it for **word count statistics**, and then go back to get the next package. This will continue until receiving the EOF notification, then it will terminate. Besides package handling, it's the consumer's responsibility to **synchronize the result** with the global final histogram. Then after all consumers finish their work, the master thread could summarize it and generate the output.

The word count statistics will check the beginning character of a word. The definition of a word is a continuous a-zA-Z character (shown in Fig. 3 below). Note that, **all other** characters are treated as space. The characters like "-", "_" are not connecting words. Same as PA2, we don't differentiate between uppercase and lowercase letters.

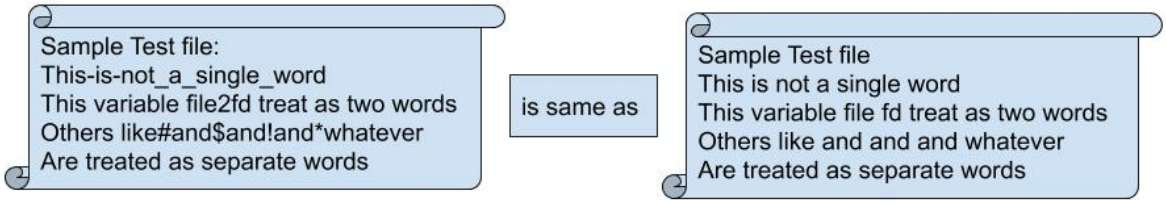

Fig. 3, the word count statistics function

4.4 Master Finalize

After the producer and consumers have joined back, the master thread will write the final result (global histogram) into the output file **named "result.txt"**. The output format should be: `"%c: %d\n"`, and it will loop through the global histogram from 'a' to 'z' (shown in Fig. 4 below).

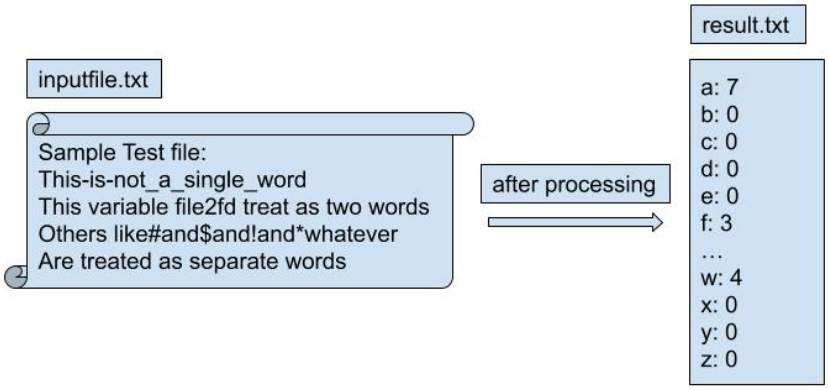

Fig. 4, the output file format

4.5 Log Printout

The program will also print a log file **if the argument option is specified**. The log file should be named as **“log.txt”**. The producer and consumers should print their execution information into the log:

Producer:

1. Print `"producer\n"` when the producer is launched
2. Print `"producer: %d\n"` for the current line number (starts from 0)

Consumer:

1. Print `"consumer %d\n"` when launched, with the consumer id (0 to number of consumers minus 1)
2. Print `"consumer %d: %d\n"` for the consumer id and the line number it currently works on.

There are some other notes when writing the log:

- The print library functions are usually thread-safe, so you don’t need to use a lock or worry about messy printing (unless you meet that).
- Since the execution order of threads is nondeterministic, you usually will not get a stable log print out.
- EOF notification should be printed out as line number “-1”.

5. Extra Credits

The extra credits will be granted if a **bounded buffer** shared queue is implemented. The application could choose the unbounded/bounded buffer by the commandline argument option. Also the bounded buffer should have a configurable buffer size specified by commandline argument option.

6. Execution Syntax, Options

The “word count statistic” will take the arguments syntax:

```
$ ./wcs #consumer filename [option] [#queue_size]
```

- The second argument “#consumer” is the number of consumers the program will create.
- The third argument “filename” is the input file name
- Options have only three possibilities: “-p”, “-b”, “-bp”.
  1) “-p” means printing, the program will generate log in this case.
  2) “-b” means bounded buffer (extra credit), the program will use it instead of unbounded buffer.
  3) “-bp” means both bounded buffer and log printing.
- The last option argument is the queue size if using bounded buffer (extra credits).

7. Testing

There will be 4 testing files in the “testing” folder, each of them has different scenarios specified in the first line of testfile. Also, the solution for 4 test files is contained in the “testing/sol” folder.

8. Assumptions and Hints

- One line of input file has at most 1024 chars.
- You are allowed to use library file IO functions (instead of open/read/write syscall).
- Use the keyword “extern” in the header file for the global variable consistency.
- Check if the library function is thread-safe before using them (if you use them in concurrent threads scenario).

9. Submission

*(this section is modified from Programming Assignment 2 document)

One student from each group should upload to Canvas, a zip file containing their C source files, a makefile, and a **README that includes** the following details:

- Team names and x500
- How to compile the program
- Your and your partner's individual contributions
- Any assumptions outside this document
- What exactly your program does
- If you have attempted extra credit

The README file does not have to be long, but must properly describe the above points. Your source code should provide **appropriate comments** for functions and critical code sections (like the purpose of these code and logical execution flow). At the top of your README file and each C source file please include the following comment:

```
/*test machine: CSELAB_machine_name * date: mm/dd/yy
* name: full_name1 , [full_name2]
* x500: id_for_first_name , [id_for_second_name] */
```

10. Grading Policy

1. (5%) correct README content
2. (5%) appropriate code style and comments
3. (40%) passed all tests (there may be some other tests when grading)

4. (15%) correct shared queue data structure and functions (like insert, extract, non-busy-waiting, and etc.)
5. (5%) correct thread create and join usage
6. (10%) correct global histogram usage (like synchronization)
7. (10%) correct producer functionality
8. (10%) correct consumer functionality
9. (10%) extra credit: bounded buffer