CSci 4061: Introduction to Operating Systems
Spring 2021
Project #3: Multithreading

Instructor: Abhishek Chandra

Due: 11:59 pm, Apr. 13, 2021

1 Background

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 length statistics", which is the multithreaded version of MapReduce application. You should work in groups as previous projects. `Please adhere to the output formats provided in each section.`
Topics covered: POSIX-threading, synchronization, producer-consumer model, file IO.

2 Project Overview

In this programming assignment, we will use multithreading to create **a producer thread** to read the file and **multiple consumer threads** 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.
The program contains: a master thread, one producer thread and many consumer threads (shown in Fig.1 below). For program execution flow, the entire program contains 4 parts:

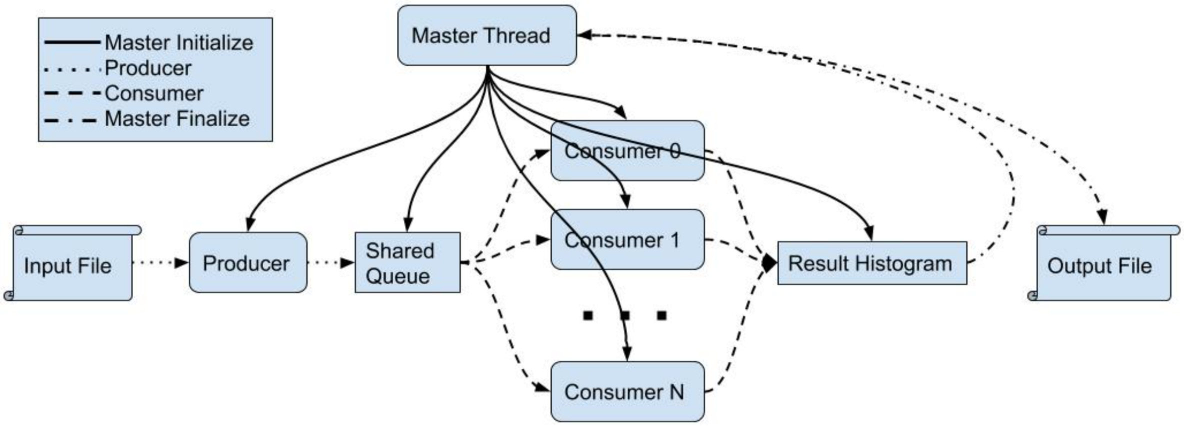


Figure 1: Overview of Assignment 3

1. The master thread initializes the shared queue, result histogram, one 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 length 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.

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

3 Project Implementation and Specifics

In this section, we will see the design details that will help you get started.

3.1 Master Phase

In this stage, the master thread needs to check the input arguments and print error messages if argument mismatch. Then it should perform the initialization on shared data structure and launch the producer/consumers thread. Then the master will wait for all threads to join back and write the final result to **one output file** "**output/result.txt**". The output format should be: "**%d %d\n**", and it will loop through the global histogram from length of 1 to 20.

> **Notice:** Even though we use the same application, there are several key differences in the input file and processing flows:
>
> 1. Assignment 2 works on different files located in different directories, but this assignment takes **only one file** as your input file.
> 2. You only have **one process but multiple threads**. Assignment 3 is mainly focused on the usage of threads and thread-safe data structures.
> 3. The final result will be written to one output file only rather than multiple files.

3.2 Shared data structure

The core of this multithreading program 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 while 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”).

3.3 Producer phase

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.
The producer phase is similar to the steam phase in Assignment 2. But the producer sends data to a shared queue instead of pipes.

3.4 Consumer phase

The consumer will repeatedly check the queue for new data, and parse it to perform word length count. This will repeat until receiving the EOF notification. You can reuse your code from Project 2 to count word lengths. Before the consumer terminates, it needs to synchronize its count result with **a global final**

histogram which is shared by all consumers. After all consumers finish their work, the master thread will generate the output of the global final histogram. Note that the consumer should not write its result to any output files.

> **Notice:** Words are separated by whitespaces as delimiters.
> Example: Thi's is. a text* 0h gr8
> The words in this sentence are {Thi's, is., a, text*, 0h, gr8}

3.5 Log

The program will also print a log file **if the argument option is specified**. The path name of the log file should be "output/log.txt". The producer and consumers should print their execution information into the log:
Producer:

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

Consumer:

- Print "consumer %d\n" when launched, with the consumer id (0 to number of consumers minus 1)
- 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.
- 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".

4 Extra Credit - Bounded buffer

We will provide extra credit (10%) if a bounded buffer shared queue is implemented. The application can 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.

5 Compile and Execute

Compile

The current structure of the `Template` folder should be maintained. If you want to add extra source(.c) files, add it to `src` folder and for headers user `include`. The current `Makefile` should be sufficient to execute the code, but if you are adding extra files, modify the `Makefile` accordingly. For compiling the code, the following steps should be taken:

Command Line

```
$ cd Template
$ make
```

The template code will not error out on compiling.

Execute

Once the *make* is successful, run the mapreduce code with the correct format.

Command Line

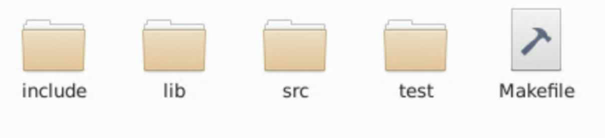
```
$ ./mapreduce #consumers inputFile [option] [#queueSize]
$ ./mapreduce 10 test/F1.txt -bp 20
```

- The first argument "#consumer" is the number of consumers the program will create.
- The second argument "filename" is the input file name
- The third, optional, argument has only three options: "-p", "-b", "-bp".
  - "-p" means printing, the program will generate log in this case.
  - "-b" means bounded buffer (extra credit), the program will use it instead of unbounded buffer.
  - "-bp" means both bounded buffer and log printing.
- #queue_Size means the queue size if using bounded buffer (extra credits).

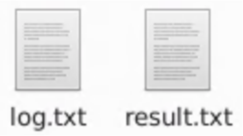
6 Expected Output

Please ensure to follow the guidelines listed below:

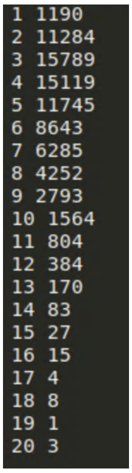
- Do not alter the folder structure. The structure should look as below before compiling via *make*:



- The output folder content (auto-created) will be as follows:

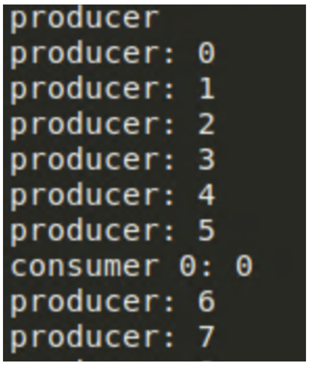


- The content of result.txt will be as follows:



```
1 1190
2 11284
3 15789
4 15119
5 11745
6 8643
7 6285
8 4252
9 2793
10 1564
11 804
12 384
13 170
14 83
15 27
16 15
17 4
18 8
19 1
20 3
```

- The content of log.txt will be as follows if you enable the log option::



```
producer
producer: 0
producer: 1
producer: 2
producer: 3
producer: 4
producer: 5
consumer 0: 0
producer: 6
producer: 7
```

7 Testing

A test folder is added to the template with one test case. You can run the testcase using the following command.

Command Line

```
      $ make test // run all test cases in the test directory

      $ make t1   // run a specific test case
```

8 Assumptions / Points to Note

The following points should be kept in mind when you design and code:

- The input file sizes can vary, there is no limit.
- Add error handling checks for all the system calls you use.
- You can assume the maximum length of a word to be 20.
- The chunk size to read lines will be at most 1024 bytes.
- Follow the expected output information provided in the previous section.

9 Deliverables

1. An Interim Submission. (Due by 03/31/2021, 11:59 PM)
Interim Submission should just contain **a src file** main.c that can create threads and wait for termination of threads.

2. A zip file containing **all source code in the template (including include/ and lib/)**, Makefile and a README that includes the following details:

- The purpose of your program
- How to compile the program
- What exactly your program does
- Any assumptions outside this document
- Team member names and x500
- Contribution by each member of the team

10 Rubric: Subject to change

- 5% README
- 10% Interim Submission
- 10% Documentation within code, coding, and style: indentations, readability of code, use of defined constants rather than numbers
- 75% Test cases: correctness, error handling, meeting the specifications
- Please make sure to pay attention to documentation and coding style. A perfectly working program will not receive full credit if it is undocumented and very difficult to read.
- A sample test case is provide to you upfront. Think about other corner cases that may occur in the code, for example, an empty input file. Your code should be able to handle such cases. Please make sure that you read the specifications very carefully. If there is anything that is not clear to you, you should ask for a clarification.
- We will use the GCC version installed on the CSELabs machines(i.e. 9.3.0) to compile your code. Make sure your code compiles and run on CSELabs.
- **Please make sure that your program works on the CSELabs machines** e.g., KH 4-250 (csel-kh4250-xx.cselabs.umn.edu). You will be graded on one of these machines.

Notice: To get full credits, please ensure the submission can compile and run on any CSE Labs machine located in KH 4-250, and follow the expected output.