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

Instructor: Abhishek Chandra

Interim Submission Due: 11:59 pm, Nov. 10 (Wed.), 2021
Final Submission Due: 11:59 pm, Nov. 17 (Wed.), 2021

1. Background

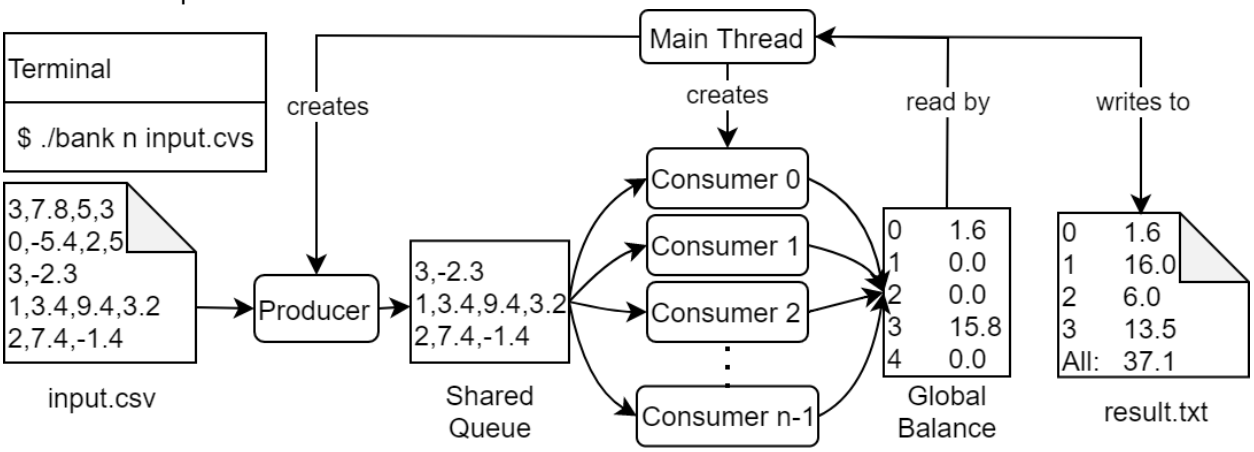
In multi-threads programming, threads can perform the same or different roles. In some multithreading scenarios like producer-consumer, producer and consumer threads have different functionality. In other multithreading scenarios like many parallel algorithms, threads have almost the same functionality. In this programming assignment, we want to work on both sides to process some bank transactions histories. 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 balance **array** synchronized by consumers.

The entire program contains 4 parts:

1. The main thread initializes the shared queue, result balance, one producer thread and many 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 balance change of a customer for its data pieces and synchronize the result to global balance.
4. After producer and all consumers complete their work, the main thread writes the final result into the output file.



3. Project Implementation and Specifics

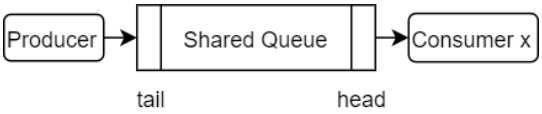
3.1 Main thread

At startup, the main 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 thread and the consumers threads. 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\t%lf\n" except the last line, and it will

loop through the balance change of all customers in global balance. The last line will be the assets change of the bank. An example of result.txt is provided in Section 6.

3.2 Shared Queue and Parallelism

The core of this multithreading program is a thread-safe shared queue (shared array is easier). 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 (they should run in parallel). Also, it should be implemented in a non-busy-waiting way (use “mutex lock + conditional variable” or “semaphore”).



3.3 Producer Thread

The main functionality of the producer thread is to read the input file and pass the data into the shared queue (by a packet, described later). 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. If the EOF is reached, the producer should send notifications to consumers specifying there will be no more data. To do so, the producer should append n (n is the number of consumer threads) special packets to the shared queue and each consumer will take one. The producer terminates after sending those notifications. Note that a packet is the information transferred between producer/consumers via shared queue. It should contain the data for consumers and other information if needed. You should design the structure of packet by yourself.

3.4 Consumer Threads

The consumer will check the queue for new data and parse it to calculate the balance change of a customer. Before the consumer takes a new packet from the queue, it needs to update the global balance array to reflect the balance change of current customer. This will repeat until receiving the EOF. After all consumers finish their work, the main thread will generate the output of the global balance change. Note that the consumer should not write its result to any output files. In the next section, we will see how a consumer parses a packet data, which is a line of the input file, and update the balance change to the global array.

3.4.1 Input File and Shared Array

We use a comma separated values file as an input file. For each line, the first number is a customer id, which is always an integer between 0 and 999. In this project, we assume there are totally 1000 customers. Thus, the length of global balance array is 1000 as well, and global array stores balance of all customers. The numbers after customer id are transactions. You will need to sum up those numbers to get the balance change in Section 3.4.

```
#id,t0,t1,t2...
3,-37.149,97.841,8.285
5,74.01,-29.517
9,49.658,37.089,81.541,-78.231
```

For example, `3` is the customer id. `-37.149,97.841,8.285` are transactions. The balance change of customer `3` is `68.977`. You will need to update it to the global balance by `balance[3] += 68.977` in a consumer thread. Where `balance` is a global double array shared by all consumers. Note that you do not need to check if the balance of a customer is negative or not.

```
balance:
#index  0       1       2       3       4   ... 998      999
#value  53.1    -4.05   0.0     1.0     0.7 ... 13.5     29.0
after a consumer processed the first line:
#value  53.1    -4.05   0.0     69.977  0.7 ... 13.5     29.0
```

3.5 Log File

The program will also print a log file if the argument option (-p) 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 file. You may want to flush the stream buffer after each write.

Producer:

- Print "producer\n" when the producer is launched
- Print "producer: line %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: line %d\n" for the consumer id and the line number it currently works on

Notes:

- 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”.

An example of log.txt is provided in Section 6.

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 command line argument option (See Section 5 for more details). You could use either a bounded linked list or an array to implement a bounded queue. An extra condition variable is needed to avoid busy waiting for the availability.

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:

```
$ cd PA3_Package
$ make
```

Execute:

Once the make is successful, run the program with the correct format.

```
$ ./bank #consumers inputFile [option] [#queueSize]
$ ./bank 100 input/t3.csv -bp 200
# alternatively, you could run ./bank 100 input/t1.csv with:
$ make run1
```

- 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”.
    - No need to write logs or limit buffer if no option is provided.
    - “-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

Here is an example output, your result will vary depending on the input file.

```
output/result.txt:                      output/log.txt:
0       -5558.894000                    producer
1       -6986.689000                    consumer 0
2       -5830.281000                    consumer 1
3       3807.207000                     producer: line 0
.                                       producer: line 1
.                                       .
.                                       .
997     219.322000                      consumer 7: line 9
998     -13041.560000                   producer: -1
999     -10138.835000                   producer: -1
All:    87805.667000                    consumer 13: line -1
```

The first line of `output/result.txt` means the balance of customer `0` is `-5558.894`.
The last line of `output/result.txt` means the assets change of the bank is `87805.667`.
`producer: line -1 in output/log.txt` means `producer` is sending EOF notification.
`consumer 13: line -1 in output/log.txt` means `consumer 13` received EOF notification.

7. Testing

You can run the testcase using the following command:

```
$ make t1       # run test on t1.csv
$ make t2       # run test on t2.csv
...
$ make t5
$ make test     # run all test cases
```

8. Assumptions and Notes

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.
- There are totally 1000 bank customers.
- All input files are valid, and no arithmetic overflow will happen.
- Balance of a customer can be negative.
- You are free to use helper functions defined in include/utils.h.
- You could define and implement shared queue in utils.h and utils.c

9. Deliverables

9.1 Interim Submission

Interim submission should be a zip file containing `consumer.c main.c producer.c README.md` and `terminal.png`. Specifically, your interim submission should satisfy following the requirements:

`main.c:`

- Create a producer thread and 3 consumer threads
- Print “launching producer\n” when creating the producer thread
- Print “launching consumer %d\n” when creating a consumer thread

`producer.c:`

- Print "producer\n" when the producer is launched

`consumer.c:`

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

`terminal.png:`

- A screenshot of the terminal after running the program

`README.md:`

- Group number, member names and x500 addresses
- Division of labor and plan.
- Embed your screenshot by adding this line: `![screenshot](terminal.png)`

Due for the interim submission is **Nov. 10, 2021 (Wed.), 11:59 pm.**

9.2 Final Submission

One student from each group should upload to Canvas, a zip file containing the project folder. README.md should include the following details:

- Lab machine name used to test your program
- Group number, member names and x500 addresses
- Whether to complete the extra task
- Members' individual contributions
- Any assumptions outside this document
- How to compile and run your program

Before compressing the project folder to a zip file please make sure:

The structure should be as follows:

```
PA3_Package
├── Makefile
├── README.md
├── include
│   ├── consumer.h
│   ├── header.h
│   ├── producer.h
│   └── utils.h
├── lib
│   └── utils.c
└── src
    ├── consumer.c
    ├── main.c
    └── producer.c
```

The following files should be removed:

```
├── expected
│   ├── result1.txt
│   ├── result2.txt
│   ├── result3.txt
│   ├── result4.txt
│   └── result5.txt
├── input
│   ├── t1.csv
│   ├── t2.csv
│   ├── t3.csv
│   ├── t4.csv
│   └── t5.csv
└── output
    ├── log.txt
    └── result.txt
```

Due for the final submission is **Nov. 17, 2021 (Wed.), 11:59 pm.**

10. Grading Policy

For the final submission

1. (10%) Interim submission
    a. (5%) Correct report content
    b. (5%) Completeness of minimum implementation requirements
2. (5%) Appropriate code style and comments

3. (5%) Conformance check
    a. (3%) Correct README.md content
    b. (1%) Folder structure and executable names
    c. (1%) Correct output file name
4. (50%) Testcases
    a. (10%) make t1: one consumer thread, each bank customer has one transaction
    b. (10%) make t2: one customer has many transactions
    c. (10%) make t3: medium input file
    d. (10%) make t4: large input file
    e. (10%) make t5: several customers have most transactions
    f. You will lose half points if you calculate the result balance array directly by the main thread. You could only calculate assets change of bank in main thread.
5. (25%) Code
    a. (5%) Correct use of thread-related functions with error handling.
    b. (5%) The producer thread and consumer threads are run in parallel.
    c. (5%) Correct and efficient use of mutexes and condition variables (or semaphores).
    d. (10%) Functional implementation of the shared queue
6. (5%) Log file
    a. (1%) Correct log file name
    b. (4%) Correct log file format
7. (10%) Extra credit
    a. (5%) Correct code to implement bounded queue
    b. (5%) Testcases with “-bp 200” flag