Assignment 04

Part II Programming (100 points)

Pair-programming up to two in a group or work alone.

Due: Beginning of the class, **Nov 16th**

Late Due: Beginning of the Class, **Nov 21st**

Real-time crypto brokerage

A crypto exchange platform connects traders of cryptocurrencies and fulfills transactions between them through a **real-time brokerage system** that employs a producer – consumer communication scheme. A trade request service produces trade requests (accepting buy/sell trade requests from traders) and publishes them to a request broker by adding the requests to a trade request queue. A request transaction service consumes the requests by taking each request off (remove) from the trade request queue and completing the transaction with another trader. The trade request queue serves as a **bounded** buffer for streaming the requests from producers (trade request services) to consumers (trade transaction services).

Functionality

Write a program to implement the following specifications for the brokerage system and simulate this producer and consumer problem, using POSIX threads, monitors (with locks and condition variables), and unnamed semaphores.

Upon start, your program should create **two** producers and **two** consumers of trade requests (see spec below) as pthreads.

1) Producer threads will accept and publish crypto trade requests to the broker until reaching the limit of the production (see specs below), then exit.
2) Consumer threads will consume all requests from the broker before exiting.
3) Main thread should wait for the consumer threads to complete consuming the last request before exiting.

Follow the **monitor** skeleton code for solving the **producer/consumer with bounded buffer** problem in TEXT and lecture slides.

Brokerage system specifications (mandatory)

1) The broker can hold up to a **maximum** of **15** crypto requests in its request queue at **any given** time.
    a. When the trade request queue is **full**, **producers must wait** for consumers to consume a request before they can publish another crypto transaction request to the queue.

   b. When the trade request queue is **empty**, **consumers must wait** for producers to add / insert a new request to the broker before they can consume another request off from the queue.
2) **Two producers** (**trade request** services) are offered:
   a. One produces (i.e., accepts) **Bitcoin trade requests** and publishes them to the broker.
   b. One produces (i.e., accepts) **Ethereum trade requests** and publishes them to the broker.
   c. You will **simulate** the production of a request by having the producer thread **sleep** for a certain amount of time. See optional arguments below.
   d. Publishing means **adding** a request to the **end** of the trade request **queue**.
   e. As a **Bitcoin** transaction costs more fees, there can be **no more than 6** transaction requests for **Bitcoins** in the broker request queue at **any given time**.
      i. When this limit is reached, producer of **Bitcoin** requests must wait for consumers to take a **Bitcoin** request off from the broker queue before it can accept and publish another **Bitcoin** request.
   f. Request services (producers) should **stop producing and publishing** requests to the broker once the limit of total number of requests is reached, which is specified by an optional command line argument (**-n**). The **default** is **120** if not specified.
3) **Two consumers** (**request transaction** services) are available for processing trade requests and completing trade transactions between traders:
   a. One uses **Blockchain X** to consume the request and complete the transaction for the request.
   b. One uses **Blockchain Y** to consume the request and complete the transaction for the request.
   c. You will **simulate** the **consumption** of a request by having the **consumer** thread **sleep** for a certain amount of time. See optional arguments below.
   d. Requests are taken off from the trade request queue and processed (consumed) in the order that they are published.
      i. Maintain the ordering of the request production and consumption. Requests are removed in **first-in first-out** order.
4) A producer or consumer must have protected access to trade request queue for adding or removing a request to or from the queue.
5) Main thread must wait for the completion of producer threads and consumer threads.
   a. Producer threads should finish first.
   b. The main thread should block until consumption is complete then exit and let the OS kill consumer threads (or kill the consumer threads the exit).
      i. **Hint**: One of the difficult problems is how to stop the program. Imagine that the Blockchain X consumer thread consumes the last request. The Blockchain Y consumer thread could be asleep then wait for a request to become available in the queue to consume, and thus never able to exit. The trick here is to use a **barrier** (see

precedence constraint in lecture slide) in the main thread that is signaled by the consumer that consumed the last request.

ii. At the end of consumer thread logic, you would want to check if the broker queue is empty and if the production limit is reached; if so, **signal** the **barrier**. Notice there could be the case that one of the consumer threads is blocked, the other consumer thread consumes the last request in the queue, and it sees both the broker queue being empty and the production limit was reached, then it can signal the barrier (main thread is waiting on) to unlock the main thread. Then with the main thread exiting, it would automatically force the blocked consumer thread to exit.

If you are interested to learn more of this kind of real-time messaging/streaming brokerage between producers and consumers of events, check out Apache Kafka (originally created by LinkedIn), a popular real-time data / message streaming platform connecting producers (publishers) and consumers (subscribers).

User Interface

When invoked, your program should accept the following optional arguments.
Implementing this interface correctly is critical to earning points.

Several functions are provided to help you **print output in the correct format**. These are located in report.c (report.h, tradecrypto.h), which will compile in C or C++. See function comments in report.c for output function details.

- Do **NOT** implement your own print functions, autograding is strictly dependent on the output formats from the provided print functions.

Optional arguments:

-n N	Total number of trade requests (production limit). The default is **120** if not specified.
-x N	Specifies the **number of milliseconds** N that the consumer using **Blockchain X** for processing a trade request and completing its transaction. You would simulate this time to consume a request by putting the consumer thread to sleep for N milliseconds. Other consumer and producer threads (consuming over Blockchain Y, producing Bitcoin request, and producing Ethereum request) are handled similarly.
-y N	Similar argument for consuming over Blockchain Y.
-b N	Specifies the number of milliseconds required to produce and publish a **Bitcoin** request.
-e N	Specifies the number of milliseconds required to produce and publish an **Ethereum** request.

Note: If an argument is **not given** for any one of the threads, that thread should incur **no delay**, **i.e., the defaults for -x, -y, -b, -e above should be 0.**
The programming reference FAQ (canvas) explains command line argument parsing and how to cause a thread to sleep for a given interval. You need not check for errors for these arguments. Also remember from previous assignments that using **getopt** can make command line arguments parsing much easier.

Important: when using sleep to simulate either production or consumption, you **must sleep outside the critical region** for accessing the broker request queue, so while the thread is sleeping, other threads would be able to access the broker request queue.

Design criteria (mandatory)

1. Requirements:
    a. The main thread must create and start the producer and consumer threads at the same time; violating it would result in a **ZERO grade** of a4 programming.
    b. For **synchronization between producers and consumers threads**, you **must** use the **pthreads monitor** programming support:
        i. Mutex lock: pthread_mutex_t
        ii. Condition variables: pthread_cond_t, and related functions: pthread_cond_wait, pthread_cond_signal, pthread_cond_init, and pthread_cond_destroy (if necessary) functions.
            1. Refer to https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html#SYNCHRONIZATION
        iii. Also see monitor pseudocode code for solving the producer-consumer problem in slides and TEXT for references.
    c. For **synchronization** between **producers and consumers threads**, **any** implementation **without using monitor-related constructs** would result in a **ZERO grade** of a4 programming.
    d. Any use of **busy wait** for ensuring the order of execution would result in a **ZERO grade** of a4 programming.
    e. **Note** as mentioned in the Brokerage specifications above, for having main thread wait for consumer threads to complete before exiting, you may use a **barrier**, which can be implemented by using a **pthread semaphore**.
        i. To avoid run time complexities, you **should only use sem_init, sem_wait and sem_post** operations from semaphore.h, you should **NOT** use the non blocking version of the wait in sem_trywait. Any use of sem_trywait would result in a **30%** penalty.
    f. **Note** the total production limit (see specifications 2) f. above) of the trade requests can be tracked by using an integer and used as a stopping condition to end production of requests.

2. Your program should be written using multiple files that have some type of logical coherency (e.g. producer, consumer, broker, etc.).
    a. Multiple source code files should be used to separate implementations of producer, consumer, and main thread logic.
3. The **producer** threads **must share** common code (the same pthread function) but must be created and executed as separate threads.
    a. If you choose to use separate code for the two producers, a **10% penalty** will be applied to the overall a4 programming grade.
    b. In **producer thread creation**, create the producer of **Bitcoin** request then the producer of **Ethereum** request. (This is for a more deterministic output sequence for autograding)
4. The **consumer** threads **must share** common code (the same pthread function) but must be created and executed as separate threads.
    a. If you choose to use separate code for the two consumers, a **10% penalty** will be applied to the overall a4 programming grade.
    b. In **consumer thread creation**, create **Blockchain X** consumer thread before **Blockchain Y** consumer thread. (This is for a more deterministic output sequence for autograding)
5. Each time the broker request queue is mutated (addition or removal), a message should be printed indicating which thread performed the action and the current state, i.e., descriptive output should be produced each time **right after a request is added** to or **removed** from the broker request queue.
    a. Functions in **report.c** will let you print output in a standard manner. Use **report_request_added** and **report_request_removed** to print messages. After the request production is complete (after the production limit is reached) and the last request is consumed, you should print out how many of each type of request was produced and how many of each type were processed by each of the consumer threads, use **report_production_history**.

You are recommended to write this project in **stages**. **First**, make a single producer and consumer function on a generic request type function. **Then**, add multiple producers and consumers. Finally, introduce the multiple types of requests.

Compilation and execution

- You must create and submit a Makefile for compiling your source code. Refer to the Programming page in Canvas and past assignments for Makefile help.
    - The Makefile must use a **-g** (or -g3) compile flag to generate gdb debugging information.
- The Makefile must create the executable with a name as **tradecrypto.**
- You are strongly recommended to set up your local development environment under a Linux environment (e.g., Ubuntu 18.04 or 20.04, or CentOS 8), develop and test your code there first, then port your code to Edoras (e.g., filezilla or winscp) to compile and test to verify. The gradescope autograder will use a similar environment as Edoras to compile and autograde your code.

Sample output

./tradecrypto -n 100 -x 12 -y 18 -b 14 -e 9

See **sample_output**.txt for the execution of the above command line.

Important: due to non-deterministic thread scheduling at run-time, the exact sequence of the execution cannot be guaranteed. Your execution outputs should satisfy the number of crypto constraints in the request queue though, and they should be like the sample_output.txt.

The auto grading test cases will be based on the constraints specified in the specifications above.

Turning In

For each pair programming group, submit the following artifacts by **ONLY ONE** group member in your group through **Gradescope**. Make sure you use the **Group Submission** feature in Gradescope submission to **add your partner** to the submission.

Make sure that all files mentioned below (Source code files, Makefile, Affidavit) contain each team member's name and Red ID!

- Program Artifacts
  - Source code files (.h, .hpp, .cpp, .C, or .cc files, etc.), Makefile
  - Do NOT **compress / zip** files into a ZIP file and submit, submit all files separately.
  - Do NOT submit any .o files or test files
- Academic Honesty Affidavit (no digital signature is required, type all student names and their Red IDs as signature)
  - Pair programming Equitable Participation and Honesty Affidavit with the members' names listed on it. Use the pair-programmer affidavit template.
  - If you worked alone on the assignment, use the single-programmer affidavit template.

- **Number of submissions**
  - The autograder counts the number of your submissions when you submit to Gradescope. For this assignment, you will be allowed a total of **10** submissions. As stressed in the class, you are supposed to do the testing in your own dev environment instead of using the autograder for testing your code. It is also the responsibility of you as the programmer to sort out the test cases based on the requirement specifications instead of relying on the autograder to give the test cases.
  - Note as a group, you would **ONLY have a total of 10 submissions** for the group. If the two members of a group submit separately and the total submissions together between the two members exceed 10, a **30%** penalty on your overall grade will be applied. This will be verified during grading.

Programming references

Please refer to Canvas Module "Programming Resources" for coding help related to:

- Pthread
- Using POSIX condition variables
- Using POSIX unnamed semaphores
- Processing command line arguments.
- Having a thread sleep for a specified amount of time.
- Code structure – what goes into a C/C++ header file.
- Many other tips for c/c++ programming in Unix / Linux.

Grading

Passing 100% auto-grading may NOT give you a perfect score on the assignment. The structure, coding style, and commenting will also be part of the rubrics for the final grade (**see Syllabus Course Design - assignments**). Your code shall follow industry best practices:

- Multiple source code files should be used to separate implementations of producer, consumer, and main thread logic.
- Be sure to comment your code appropriately. Code with no or minimal comments are automatically lowered one grade category.
- Design and implement clean interfaces between modules.
- Meaningful variable names.
- Do not use global variables to communicate information to your threads. Pass information through data structures (recall assignment 2).
- NO hard code – Magic numbers, etc.
- Have proper code structure between .h and .c / .cpp files, do not #include .cpp files.

Academic honesty

Posting this assignment to any online learning platform and asking for help is considered academic dishonesty and will be reported.

An automated program structure comparison algorithm will be used to detect code plagiarism.

- The plagiarism detection generates **similarity reports of your code with your peers code as well as code from online sources and submissions from past semesters**. It would be purely based on similarity check, two submissions being similar to each other could be due to both copied from the same source, whichever that source is, even the two students did NOT copy each other.
- We will also include solutions from the popular learning platforms (such as Chegg, github, etc.) as well as code from **ChatGPT** (see below) as part of the online sources used for the plagiarism similarity detection. Note not only the plagiarism detection checks for matching content, but also it checks the structure of your code.

- **If plagiarism is found in your code**, you will be automatically disenrolled from the class. You will also be reported for plagiarism.
- Note the provided source code snippets for illustrating proposed implementation would be ignored in the plagiarism check.

Recently, SDSU's Center for Student Rights and Responsibilities have officially added plagiarism policies of using ChatGPT for academic work, as put below:

- "Use of ChatGPT or similar entities [to represent human-authored work] is considered academic dishonesty and is a violation of the Student Code of Conduct. Students who utilize this technology will be referred to the Center for Student Rights and Responsibilities and will face student conduct consequences up to, and including, suspension."