Assignment 04

Part II Programming (100 points)

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

Due: Beginning of the class, **Apr 21st**

Real-time messaging for ridesharing

A toy ridesharing service provider connects riders and drivers through a real-time messaging system that employs a producer – consumer communication scheme. Rider services produce rider requests (i.e., accepting requests from riders) and publish them to a request broker by adding the requests to the broker request queue. Request dispatchers consume the requests by taking the requests off (remove) from the broker request queue and dispatch them to drivers. The broker request queue serves as a bounded buffer for streaming the requests from producers to consumers.

Functionality

Write a program to implement the following specifications for the messaging system and simulate this producer and consumer problem, using POSIX unnamed semaphores and POSIX threads. POSIX unnamed semaphores are covered in the course frequently asked questions.

Upon start:

1) Your program should create two producers and two consumers of rider requests (see spec below) as pthreads.
2) Producer threads will produce and publish rider requests to the broker until reaching the limit of the production (see spec below), then exit.
3) Consumer threads will consume all requests from the broker before exiting.
4) Main thread should wait for the consumer threads complete consuming the last request before exiting.

Follow the skeleton code of producer/consumer with bounded buffer problem solution in TEXT and lecture slides.

Messaging system specifications (mandatory)

1) Two rider request services (producers) are offered:
    a. One accepts (produces) requests for a **human driver** and publishes them to the broker.
    b. One accepts requests for an **autonomous car** 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 to insert a request to the end of the broker request queue.
    e. To promote the use of autonomous cars (as human drivers cost more), there can be **no more than 4** rider requests for a **human** driver in the broker request queue at **any given time**.

        i. When this limit is reached, producer of human driver requests must wait for consumers to take a human driver request off from the broker queue before it can publish another human driver request.
2) The broker can hold up to a maximum of **12** rider requests in its request queue at any given time.
    a. When the broker request queue is full, producers must wait for consumers to consume a request before they can publish another rider request.
3) Two ride request dispatcher services (consumers) are available using different driver search algorithms for dispatching:
    a. One consumes (dispatches) the requests by using a cost saving algorithm to find nearby drivers.
    b. One consumes (dispatches) the requests by using a fast-matching algorithm to find nearby drivers.
    c. You are not asked to implement these algorithms; you will simulate the consumption (dispatching) 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 broker request queue and dispatched (consumed) in the order that they are published.
        i. Maintain the ordering of the request production and consumption. Request are removed in first-in first-out order.
4) When the broker 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.
5) A producer or consumer must acquire a mutex to gain access to broker request queue for adding or removing a request to or from the queue.
6) Your 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). Default is **120** if not specified.
7) Must have main thread wait for the completion of producer threads and consumer threads.
    a. HINT: One of the difficult problems for students is how to stop the program. Imagine that the cost saving algo thread consumes the last request. The fast-matching algo thread could be asleep, 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. The main thread should block until consumption is complete and kill the child threads (or simply let the OS kill them).

For those of you who 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 message streaming platform connecting producers (publishers) and consumers (subscribers) of information.

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 put output in the correct format. These are located in io.c (io.h, ridesharing.h), which will compile in C or C++. See function comments in io.c for output function details.

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

Optional arguments:

-n N	Total number of requests of requests (production limit). Default is **120** if not specified.
-c N	Specifies the number of milliseconds N that the cost-saving dispatcher (consumer) requires dispatching a request and should be invoked each time the cost-saving dispatcher removes a request from queue regardless of the request type. You would simulate this time to consume a request by putting the consumer thread to sleep for N milliseconds. Other consumer and producer threads (fast-matching dispatcher, producing human driver request, and producing autonomous driver request) are handled similarly.
-f N	Similar argument for the fast-matching dispatcher.
-h N	Specifies the number of milliseconds required to produce a ride request for a human driver.
-a N	Specifies the number of milliseconds required to produce a ride request for an autonomous car.

Important: If an argument is not given for any one of the threads, that thread should incur no delay, i.e., the default for -c, -f, -h, -a above should be 0.
The class 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 when sleeping. Also remember from previous assignments that using getopt can make command line arguments parsing much easier.

Design criteria (mandatory)

1. 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.
2. 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, you will get partial credits. Points will be deducted from autograding portion as well as manual grading portion if this requirement is not satisfied.
    b. In producer thread creation, create the producer of human driver request then the producer of autonomous car.
3. 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, you will get partial credits. Points will be deducted from autograding portion as well as manual grading portion if this requirement is not satisfied.

    b. In consumer thread creation, create cost saving dispatcher thread before fast matching dispatcher thread.
4. 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 **io.c** will let you produce output in a standard manner. Use **io_add_type** and **io_remove_type** to print messages. When the request production is complete, 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 **io_production_report**.

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 for Makefile help.
    - The Makefile should use a -g compile flag to generate gdb debugging information
- The Makefile must create the executable with a name as **rideshare.**
- Your code **must compile and run correctly on edoras.sdsu.edu**, regardless of where you wrote, compiled, and tested your program.

Sample output

./rideshare -n 150 -h 5 -a 15 -c 35 -f 20

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 sequence of outputs should be more or less similar to the sample_output.txt.

The autograding test cases would 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.

Programming references

Please refer to Canvas Module class FAQ for coding help related to:

- 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.
- 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 as well as from online sources. We will also include solutions from the popular learning platforms (such as Chegg, etc.) 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.
- Refer to Syllabus for penalties on plagiarisms.
- Note the provided source code snippets for illustrating shared data structure would be excluded in the plagiarism check.