Assignment 04 Programming (100 points)
Pair-programming up to two in a group or work alone.
Due: Beginning of the class, **Nov 19th**

NO Late Due for this assignment as I am giving **two weeks** for this assignment (as opposed to one and a half weeks for previous assignments). **Essentially**, I am **waiving the late penalty** for this one.

Restaurant flow control

A restaurant management system utilizes a **producer-consumer** model to control the customer flow in both **general table** areas and **VIP rooms** within the dining space.

Two **greeter robots receive (or produce)** customer seating requests, adding them to a shared seating request queue. Meanwhile, two **concierge robots retrieve (or consume)** requests from this queue, guiding customers to either **general tables** or **VIP rooms**. This seating **queue** acts as a **bounded buffer**, facilitating the flow of requests from the producers to the consumers and ensuring a steady and controlled seating process.

Functionality

Write a multi-threaded program to simulate this producer and consumer problem by using **POSIX pthreads**, **monitors** (with locks and condition variables), and **unnamed semaphores**.

Upon start, your program should create **two producers** (simulating **greeter** robots) and **two consumers** (simulating **concierge** robots) of **seating requests** (see spec below) as **pthreads**.

1. **Producer Threads**: Each producer thread receives and adds seating requests to a shared queue until a specified maximum number of total requests is reached. Once this limit is met, the producer threads will terminate.
2. **Consumer Threads**: Each consumer thread retrieves and processes requests from the queue (in FIFO order). When all requests have been handled, consumer threads will complete and exit.
3. **Main Thread**: The main thread will wait for the consumer threads to complete consuming the last request before exiting.

This setup ensures **synchronization** between producer and consumer threads, managing seating requests in a bounded queue.

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

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 **dineseating.**
- You are strongly recommended to set up your local development environment under a Linux environment (e.g., Ubuntu 22.04 or 20.04, or 24.04), develop and test your code there first. The gradescope autograder will use an Ubuntu environment and GNU compilers to compile and run autograding against your code.

Specifications of Multithreading Synchronization (mandatory)

1) The queue can hold up to a **maximum** of **18** seating requests at **any given** time.
    a. When the queue is **full**, **producers must wait** for consumers to consume a request before they can add another request to the queue.
    b. When the queue is **empty**, **consumers must wait** for producers to add a new request to the queue before they can retrieve another request from the queue and consume.
2) **Two producers** (**greeter robots**) are used:
    a. One receives (or produces) **general table requests** and **appends** them to the queue.
    b. One receives (or produces) **VIP room requests** and appends them to the queue.
    c. The system wants to limit the **VIP room requests** to **no more than 5** in the queue at **any given time**.
        i. When this limit is reached, the producer of VIP room requests must wait for consumers to take a **VIP room** request off from the queue before it can accept (or receive) and add another VIP room request.
    d. You will **simulate** the **production** of a request by having the **producer thread sleep** for a certain amount of time. See **optional arguments** below.
    e. Producers should **stop producing and adding** requests to the queue 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** (**concierge robots**) process seating requests from the queue and guide customers to the dining area:
    a. One concierge robot is of model **T-X**, the other is of model **Rev-9**. Don't worry, these Terminators have been reprogrammed and now specialize in serving customers at restaurants! They're fully focused on customer service.
    b. You will **simulate** the **consumption** of a request by having the **consumer thread sleep** for a certain amount of time. See optional arguments below.
    c. Requests are **taken out** of the request queue and processed (consumed) in the order that they are inserted.

        i. Maintain the ordering of the request production and consumption. Requests are removed in **first-in first-out** order.

4) The access to **the seating request queue must be protected** in a **mutually exclusive** way for adding or removing a request.
5) **Main thread must wait** for the **completion** of producer threads and consumer threads.
    a. Producer threads should finish first and exit after reaching the production limit.
    b. The **main thread should block until consumption is complete** then exit and let the OS kill consumer threads (or kill the consumer threads then exit).
        i. **Hint**: One of the challenges in this implementation is how to stop and exit the main thread. Imagine a scenario that the **T-X** thread consumes the last request; meanwhile, the **Rev-9 consumer** thread could be asleep waiting 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 semaphore** (see **precedence constraint** in lecture slide) in the main thread that is **signaled by the consumer that consumed the last request**.
        At the **end of consumer thread** logic, it would check **if the queue is empty and if the production limit is reached**; if so, **signal** the **barrier** to unlock the main thread. With the main thread exiting, it would automatically force all blocked (asleep) threads to exit.

        You may use *pthread_join* for implementing precedence constraint, but with the non-deterministic execution by the OS scheduler between the threads, *pthread_join* may not work occasionally.

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

User Interface (Inputs and Outputs)

Inputs

Your program accepts the following **optional** arguments. Correct implementation of this interface is essential for earning points.

Optional arguments:

-s N	Total number (N) of seating requests (production limit) for the simulation. The default is **120** if not specified.
-x N	Specifies the **time** that **T-X robot** uses on average for processing a seating request and guiding customers to the proper seating. Your **code**

would **simulate this time to consume** a request by putting the consumer thread to **sleep** for **N milliseconds**. Other consumer and producer threads (Rev-9 consumption, producing general table request, and producing VIP room request) are handled similarly.

-r N — Similar argument for consuming by **Rev-9 robot**.

-g N — Specifies the average time for the **general table greeter robot** to produce and insert a **general table** request.

-v N — Specifies the average time for the **VIP room greeter robot** to produce and insert a **VIP room** request.

Note: If an argument is **not given** for any one of the threads, that thread should take **NO sleep** in simulating production or consumption, **i.e., the defaults for -x, -r, -g, or -v above would be 0.**

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

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.

Outputs

Each time the 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 request queue.

- Several helper functions are provided to assist you in producing output in the required format, found in the files **log.c** (change to log.cpp for C++), **log.h**, and **seating.h**. These files can be compiled in either C or C++. See function comments in log.c for print function details.
- Functions in **log.c** will let you print output in a standard manner. Use **output_request_added** and **output_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 **output_production_history**.
- **Important**: Do not create your own print functions, as the autograding process is strictly reliant on the output formats generated by the provided print functions.

Sample output

./dineseating -s 100 -x 20 -r 35 -g 15 -v 10

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 request constraints** in the request queue though, like the sample_output.txt.

The **auto-grading test cases** will be based on the **synchronization constraints** in the specifications above.

Design criteria (mandatory)

- **Requirements**:
    a. The **main thread must create and start ALL producer and consumer threads** at the **same time** (i.e., a sequence of pthread_create calls together).
        i. **Violating** it would result in a **ZERO grade** of a4 programming.
        ii. In **producer thread creation**, create the producer of **general table** request then the producer of **VIP room** request. (This is for a more deterministic output sequence for autograding)
        iii. In **consumer thread creation**, create **T-X** consumer thread before **Rev-9** consumer thread. (This is for a more deterministic output sequence for autograding)
    b. For **synchronization between producers and consumers threads**:
        i. You **must** use the **pthreads monitor**, do **NOT** use **semaphores** for this part of synchronization.
        ii. For this part, any implementation using programming constructs **other than the monitor constructs** would result in a **ZERO grade** of a4 programming.
    c. For **pthread monitor** use, please refer to:
        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.
    d. **Note** as mentioned in the specifications above, for having **main thread wait for consumer** threads to complete before exiting, you can use a **semaphore** for a **barrier** implementation.

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.

e. The total **production limit** (see specifications 2) e. above) of the requests can be tracked by using an integer and used as a stopping condition to end production of requests.

f. You can also use **integers** to track the **current** numbers of **general table** and **VIP room** requests in the queue, as well as use **integers** to track the number of **general table** and **VIP room requests** that **have been put** in the queue since the beginning of the simulation.

g. You may use std::atomic<> for mutually exclusive access to variables like shared integers, but do NOT use it for protected access to collection variables like the queue.

- Your program should be written using multiple files that have logical coherency.
  a. Multiple source code files should be used to separate implementations of producer, consumer, and main thread logic.

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.

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
    a. https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html#SYNCHRONIZATION
- 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 on your code appropriately. Code with no or minimal comments are automatically lowered one grade category.
    - Each **function in the header** should have **a comment section** above the function signature to give a brief description of the function, the inputs and output of the function.
- 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.

Any hardcoding to generate the correct output without implementing the specified program logic will automatically **result in a zero grade** of the assignment.

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.

- Plagiarism detection generates similarity reports of your code with your peers as well as from online sources. It would be purely based on similarity check, two submissions being like 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 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.

The Center for Teaching & Learning and Instructional Technology Services also shared information concerning the rapidly evolving impact of artificial intelligence (AI) within academia — e.g., ChatGPT, etc.

SDSU's Center for Student Rights and Responsibilities have officially added plagiarism policies of using AI generated content 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."