Assignment 3: Bounded Buffer
CSE 130: Principles of Computer Systems Design

Due: February 23, 2023 at 23:59

Goals This assignment will provide you with experience building a system that requires synchronization between multiple threads.

Overview You will be building a bounded buffer for this assignment. It will fulfill a defined API and meet certain behaviour requirements. The function signatures and overall behaviour will be laid out below.

The code that you submit must be in your repository on `git.ucsc.edu`. In particular, your assignment must build an object file that can be linked, called `queue.o`, when we execute the command `make` from the directory `asgn3` in your repository. Your repository must include a `README.md` file that includes information about the interesting design decisions and outlines the high-level functions, data-structures, and modules that you used in your code. The `README.md` should also include how your design implements any "vague" requirements listed in this document. Your repository should not include any executables or object files.

You must submit a 40 character commit ID hash on Canvas in order for us to identify which commit you want us to grade. We will grade the last hash that you submit to the assignment on Canvas and will use the timestamp of your last upload to determine grace days and bonus points. For example, if you post a commit hash 36 hours after the deadline, we will subtract 2 grace days from your total. Your submission should only contain the commit ID hash.

Functionality

You will need to implement 4 functions to fulfill the bounded buffer API. The API defines a queue, having FIFO properties, which will store and return arbitrary pointers to objects. This means that you can store anything in the queue. Their API is as follows:

```
typedef struct queue queue_t;
queue_t *queue_new(int size);
void queue_delete(queue_t **q);
bool queue_push(queue_t *q, void *elem);
bool queue_pop(queue_t *q, void **elem);
```

Struct

You will define `struct queue` within your c file. This means that `queue_t` is an opaque data structure. Tests will only interact with the structure via the defined API functions.

Constructor and Destructor

To create and delete your queue you will fill out the `queue_new` and `queue_delete` functions, respectively. In the `queue_new` function, you will allocate the required memory and initialize necessary variables for a queue that can hold at most `size` elements. In the `queue_delete` function you will free any allocated memory and set the passed in pointer to `NULL`.

A proper pairing of the two functions should result in no memory leaks. You will create your own definition for `struct queue`.

Push and Pop

Push and pop correspond to enqueue and dequeue, respectively, and are more generic names for adding and removing elements from a collection. For this queue we will be observing FIFO, First In First Out, semantics. That means that the order that elements will be popped in will correspond to the order in which they are pushed.

In your implementation `queue_push` should block if the queue is full and `queue_pop` should block if the queue is empty. Both functions should return `true` unless given an invalid pointer, which is a NULL pointer it is expected to dereference.

Thread Safety

This queue should support multiple concurrent producers and multiple concurrent consumers. What this means is that there may be multiple threads which will attempt to access `queue_push` and `queue_pop` at the same time. In spite of this, the queue still needs to behave like a queue! In particular, a thread-safe queue should make the following guarantees:

- **Thread order**. If a thread enqueues item a before some other thread enqueues item b, then item a must be dequened before item b is dequeued. This property should hold regardless of the thread(s) that enqueue and dequeue each item.
- **Validity**. If a consumer thread T_2 dequeues some item a, then a was enqueued by some producer thread T_1. Intuitively, no consumers may read junk.
- **Completeness**. If a producer thread T_1 enqueues item a, some consumer T_2 will dequeue a after a finite number of dequeues. Intuitively, no data is lost.

The real challenge in this assignment is uholding these properties in a thread-safe fashion. Consider the following examples:

- Thread A pushes A_1 and A_2 at the same time that thread B pushes B_1 and B_2. If thread C pops four times, then it could see any of: $A_1\ A_2\ B_1\ B_2$, $A_1\ B_1\ A_2\ B_2$, $A_1\ B_1\ B_2\ A_2$, $B_1\ A_1\ A_2\ B_2$, $B_1\ A_1\ B_2\ A_2$, or $B_1\ B_2\ A_1\ A_2$. But, thread C could not observe any ordering in which it pops B_2 before B_1 nor any ordering in which it pops A_2 before A_2 (i.e., none of the following should be observed: $B_2\ B_1\ A_1\ A_2$, $B_2\ A_1\ B_1\ A_2$, $B_2\ A_1\ A_2\ B_1$, $A_1\ B_2\ B_1\ A_2$, $A_1\ B_2\ A_2\ B_1$, $A_1\ A_2\ B_2\ B_1$, $B_2\ B_1\ A_2\ A_1$, $B_2\ A_2\ B_1\ A_1$, $B_2\ A_2\ A_1\ B_1$, $A_2\ B_2\ B_1\ A_1$, $A_2\ B_2\ A_1\ B_1$, $A_2\ A_1\ B_2\ B_1$, $A_2\ A_1\ B_1\ B_2$, $A_2\ B_1\ A_1\ B_2$, $A_2\ B_1\ B_2\ A_1$, $B_1\ A_2\ A_1\ B_2$, $B_1\ A_2\ B_2\ A_1$, or $B_1\ B_2\ A_2\ A_1$.
- Thread A pushes A_1. After A returns from push, Thread B pushes B_1. If thread C pops from the queue twice, it should return A_1 and then B_1.

Additional Functionality

In addition to supporting the methods listed above, your project must do the following:

- You should not have a `main` function in your implementation file.
- Your implementation should not *busy-wait*.
- Your code should not cause segfaults.
- As you are only making an object file, to be linked later, you should have all the needed implementations in a single file and use the `-c` flag to say you only want to create an object file. Otherwise the compiler will try to link the program and fail.
- Your code should be formatted according to the clang-format provided in your repository and it should compile using `clang` with the `-Wall -Werror -Wextra -pedantic` compiler flags.
- Your queue must be written using the 'C' programming language (*not C++!*).

Rubric

We will use the following rubric for this assignment:

Category	Point Value
Makefile	10
Clang-Format	5
Files	5
Functionality	80
Total	100

Makefile Your repository includes a Makefile with the rules `all` and `queue.o`, which produce the `queue.o` object file, and the rule `clean`, which removes all `.o` and binary files. Additionally, your Makefile should use clang (i.e., it should set `CC=clang`), and should use the `-Wall`, `-Wextra`, `-Werror`, and `-pedantic` flags (i.e., it should set `CFLAGS=-Wall -Wextra -Werror -pedantic`).

Clang-Format All `.c` and `.h` files in your repository are formatted in accordance with the `.clang-format` file included in your repository.

Files The following files are included in your repository: `queue.c`, `Makefile`, and `README.md`. Your repository should not include binary files nor any object files (i.e., `.o` files). To make it easier for you to maintain tests, you can also include binary files in any directory whose name starts with the phrase `test` (although, this is probably less useful for this assignment).

Functionality Your queue performs the functionality described above.

Resources

Here are some resources to help you:

Testing

We provided you with two resources to test your own code:

- An autograder, which is run each time you push code to gitlab, will show you the points that you will receive for your Makefile, Clang-Format, and Files.
- A set of test scripts in the resources repository to check your functionality. You can use the tests to see if your functionality is correct by running them on your Ubuntu 22.04 virtual machine. We provided you with a subset of the tests that we will run, but, I bet you can figure the other ones out by adapting what we have given you :-)
- Some starter code. In this case, just a header file for you to implement.

Hints

Here are some hints to help you get going:

- You will likely need to lookup how some synchronization functions (e.g., `pthread_mutex_lock`) work. You can always Google them, but you might also find the man pages useful (e.g., try typing `man pthread.h` on a terminal).
- Review the practica and course lectures that cover `pthread` operations. They will be very useful :-).