Assignment 3: Concurrent Data Structures
CSE 130: Principles of Computer Systems Design

Due: November 12, 2024 at 11:30 PM

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

Overview You will be building both a bounded buffer and a reader-writer lock for this assignment. Both will fulfill defined APIs and meet certain behavior requirements. The function signatures and behavior are described below.

The code that you submit must be in your repository on `git.ucsc.edu`. In particular, your assignment must build two object files that can be linked, called `queue.o` and `rwlock.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 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.

Queue 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 `queue.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 which elements are popped should 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 their `q` argument is equal to NULL.

Thread Safety

This queue should support multiple concurrent producers and multiple concurrent consumers. What this means is that there may be multiple threads that will attempt to access `queue_push` and `queue_pop` at the same time. Despite 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 dequeued 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 upholding 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: $\langle A_1A_2B_1B_2\rangle$, $\langle A_1B_1A_2B_2\rangle$, $\langle A_1B_1B_2A_2\rangle$, $\langle B_1A_1A_2B_2\rangle$, $\langle B_1A_1B_2A_2\rangle$, or $\langle B_1B_2A_1A_2\rangle$. But, thread C could not observe any order in which it pops B_2 before B_1 nor any order in which it pops A_2 before A_1. In particular, none of the following should be observed: $\langle B_2B_1A_1A_2\rangle$, $\langle B_2A_1B_1A_2\rangle$, $\langle B_2A_1A_2B_1\rangle$, $\langle A_1B_2B_1A_2\rangle$, $\langle A_1B_2A_2B_1\rangle$, $\langle A_1A_2B_2B_1\rangle$, $\langle B_2B_1A_2A_1\rangle$, $\langle B_2A_2B_1A_1\rangle$, $\langle B_2A_2A_1B_1\rangle$, $\langle A_2B_2B_1A_1\rangle$, $\langle A_2B_2A_1B_1\rangle$, $\langle A_2A_1B_2B_1\rangle$, $\langle A_2A_1B_1B_2\rangle$, $\langle A_2B_1A_1B_2\rangle$, $\langle A_2B_1B_2A_1\rangle$, $\langle B_1A_2A_1B_2\rangle$, $\langle B_1A_2B_2A_1\rangle$, or $\langle B_1B_2A_2A_1\rangle$.
- 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 observe $\langle A_1B_1\rangle$.

Reader-Writer Lock Functionality

A reader/writer lock, `rwlock`, allows multiple readers to hold the lock at the same time, but only a single writer. You will need to implement 6 functions to fulfill the reader-writer lock API. The goal of a reader-writer lock is to ensure that the throughput of reading an item is *unchanged* if there are no writers present. The shortcoming of a traditional reader-writer lock is that they can starve readers or writers when there is contention. Your job will be to develop a scheme that allows for a `PRIORITY` argument that decides how the `rwlock` behaves when there is contention, favoring either: readers, writers, or n-way. We describe the `PRIORITY` in more detail below.

```
typedef struct rwlock rwlock_t;
typedef enum {READERS, WRITERS, N_WAY} PRIORITY;
rwlock_t* = rwlock_new(PRIORITY p, int n);
void rwlock_delete(rwlock_t **l);
void reader_lock(rwlock_t *rw);
void reader_unlock(rwlock _t *rw);
void writer_lock(rwlock_t *rw);
void writer_unlock(rwlock_t *rw);
```

Struct

You will define `struct rwlock` within your `rwlock.c` file. This means that `rwlock_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 reader-writer lock you will define the `rwlock_new` and `rwlock_delete` functions, respectively. In the `rwlock_new` function, you will allocate the required memory and initialize the necessary variables and locking mechanisms for a reader-writer lock. The resulting `rwlock` should be initialized with the `PRIORITY p`; `n` provides an argument for the `N_WAY` priority (see below). Your implementation should ignore `n` when `p` is `READERS` or `WRITERS`. In the `rwlock_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 rwlock`.

Reader Lock/Unlock

`reader_lock` and `reader_unlock` lock and unlock the shared resource for reading respectively. `reader_lock` will lock the shared resource and allow multiple readers to access the critical section simultaneously, but no writers. When the last reader calls `reader_unlock`, the lock is released, and a writer is given a chance to acquire the lock.

Writer Lock/Unlock

`writer_lock` and `writer_unlock` lock and unlock the shared resource for writing respectively. `writer_lock` will lock the shared resource and allow a single writer to access the critical section at a time, but no readers or other writers. `writer_unlock` will unlock the shared resource for the current writer and allow a group of one or more readers to take control of the lock.

Priority

The `PRIORITY` defines how the `rwlock` behaves when there is contention. If there is lock contention (i.e., multiple threads call `reader_lock` and/or `writer_lock` at the same time), then a `rwlock` initialized with `PRIORITY` of `READERS` should prioritize readers by allowing them to proceed before allowing writers. Similarly, a `rwlock` initialized with `PRIORITY` of `WRITERS` should allow writers to proceed before readers when there is contention.

A `rwlock` initialized with `PRIORITY` of `N_WAY` is a bit more complex. Intuitively, an `N_WAY` `rwlock` should allow `n` readers to proceed between each writer; this behavior should occur if an infinite number of readers and writers are attempting to acquire the lock. In particular, a `N_WAY` `rwlock` upholds the following guarantees:

- **Reader Non-Starvation**. If there is no thread waiting in `writer_lock` and no thread that holds the `rwlock` as a writer, then there should be no threads waiting in `reader_lock`.
- **Writer Non-Starvation**. If no threads are waiting in `reader_lock` and no threads that currently hold the `rwlock` as a reader or writer, then there should be no threads waiting in `writer_lock`.
- **N-Way Blocking**. Let `rwlock` be an `N_WAY` `rwlock` initialized with value n, thread w_1 releases a writer lock (i.e., call `writer_unlock`) at time t_1, and w_2 be the next writer that will acquire the `rwlock`, which calls `writer_lock` at time t_2. The `rwlock` should allow n threads to acquire the lock for reading before allowing w_2 to acquire the lock. However, note that the non-starvation properties take precedence over N-Way Blocking. So, if there are $r < n$ threads that call `reader_lock` while w_2 is blocked, then w_2 should only wait for r readers. Similarly, if $t_2 > t_1$ and $r > n$ threads call `reader_lock` between t_1 and t_2, then w_2 should wait for the existing readers to release the lock and then acquire it (in effect, r readers will acquire the lock between two writers).

We provide the following as an example of how to apply these principles, in which a reader-writer lock is initialized with n equal to 2:

(1) At time 0, a writer calls `writer_lock`. The `rwlock` gives the writer the lock by writer- non-starvation.
(2) At time 1, three readers call `reader_lock` and one writer call `writer_lock`. The `rwlock` blocks all of the readers and writers.
(3) At time 2, the writer with the lock releases it by calling `writer_unlock`. The `rwlock` gives the lock to two of the readers—it does not matter which two readers—in accordance with N-Way Blocking.
(4) At time 3, one of the readers releases the lock by calling `reader_unlock`. The `rwlock` continues to block the other readers and writers.
(5) At time 4, the other reader releases the lock by calling `reader_unlock`. The `rwlock` gives the lock to the waiting writer.
(6) At time 5, the writer releases the lock by calling `writer_unlock`. The `rwlock` gives the lock to the remaining waiting reader.
(7) At time 6, four more readers arrive. The `rwlock` lets the four readers acquire the lock (Reader Non-Starvation).
(8) At time 7, a writer arrives. The `rwlock` blocks the writer, but ensures that it will be the next to acquire the lock (N-Way Blocking).
(9) At time 8, a new reader calls `reader_lock`. The `rwlock` blocks the reader (N-Way Blocking).
(10) At time 9, all of the readers that currently hold the lock release it, the waiting writer acquires the lock.
(11) At time 9, the writer releases the lock and the blocked reader acquires it.
(12) At time 10, the reader releases the lock.
(13) At time 11, a writer calls `writer_lock`. The `rwlock` does not block the writer (Writer Non-Starvation).
(14) At time 11, the final writer releases the lock.

Additional Functionality

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

- You should not have `main` functions in either of your implementation files.
- Your implementations should not *busy-wait*.
- Your code should not cause segfaults.
- As you are only making two object files, to be linked later, you should have all the needed implementations in two separate files and use the `-c` flag to say you only want to create object files. Otherwise, the compiler will try to link the programs 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 and reader-writer lock 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`, `queue.o`, and `rwlock.o`, which produce the `queue.o` and `rwlock.o` object files, 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`, `rwlock.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 and reader-writer lock perform the functionality described above.

How to submit

1. Navigate to the `asgn3` directory in your repo.
2. Configure your `config.json` file (see Canvas or the README in the resources repo for more information).
3. `python3 -m autograder.run.submit [file1] [file2] ...`

Resources

Here are some resources to help you:

Testing

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

- An autograder, which you will run locally through the command line.
- 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.
- Some starter code. In this case, just two header files 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 useful :-).
- We expect you to spend some time reading resources to understand how to design a reader-writer lock. My favorite description is in Tom Anderson and Mike Dahlin's Operating Systems: Principle and Practice, Volume 2 (I wonder if you can find a free copy if you google the name of the textbook?). Section 5.6.1 discusses how to implement reader-writer locks using locks and condition variables. Note, they don't discuss the various policies in this assignment, but they help with the general idea.
- Remzi and Andrea give an alternative reader-writer approach that uses semaphores in their OSTEP book, in section 31.5 (see the free online book). Note, they don't discuss the various policies in this assignment, but they help with the general idea.