Instructions : Clone your GitHub Classroom repository for this assignment. Follow all steps below to complete the programming assignment portion of Homework 3. Push your changes to GitHub and check that all tests are passing in Actions.

After completing each portion of the assignment, compile your code with the command *sh compile.sh* and check correctness with *make test* from within the *build* directory. Note, if your local computer uses an alternative to *make* you will need to compile manually. **Do not change compile.sh as this could cause GitHub Actions to fail.**

If you get confused with GitHub/CMake logistics, refer back to Homework 0 here.

The Homework 3 repository provides you with multiple structs, described in Listings 1-4 below.

Listing 1: thread_data_t* data

1	int thread_id; // unique ID	associated with th	is $thread$,	starting	a t	0.
2	calc_t* pi_data; // struct,	described in Listin	g 2.			

Listing 2: calc_t* pi_data

1	int global_n_samples; // total number of samples of (x, y) coordinates
2	// to be computed
3	int global_n_threads; // total number of threads that will run this method
4	int global_sum; // Initialized to 0, will return the total number of
5	// samples that fall within the unit circle
6	lock_t* lock; // struct, described in Listing 3.

Listing 3: lock_t* lock

$\frac{1}{2}$	<pre>void init(lock_t* lock); // initializes the lock, called for you</pre>
3	void $lock(lock_t * lock);$ // waits until a thread receives the lock.
4	void unlock (lock_t * lock); // releases the lock.
5	void destroy(lock_t* lock); // destroys the struct, called for you
6	// after your threads complete.
7	
8	// Unset variables for you to use when implementing locks
9	<pre>int ticket;</pre>
10	int turn;
11	int flag;
12	int guard;
13	$\mathbf{int} \ \mathbf{S};$
14	pthread_mutex_t mutex; // For locking within semaphore
15	queue_t queue; // a queue struct, defined in Listing 4.

```
    \begin{array}{c}
      1 \\
      2 \\
      3 \\
      4 \\
      5 \\
      6 \\
      7 \\
      8
    \end{array}
```

Listing 4: queue_t* queue

1 Ticket Spin Lock

The goal of this portion of the programming assignment is to implement a ticket spin lock that spins while waiting to acquire the lock. To complete this part of the programming assignment, you will edit the methods within the file ticket_spin_lock.cpp. Some hints for creating this lock :

- Use the atomic method __sync_fetch_and_add(int* ticket, int addition). This method will atomically fetch the value in ticket and add 'addition' to this value.
- You may not need to do anything in the destroy method.

2 Ticket Yield Lock

The goal of this portion of the programming assignment is to implement a ticket yield lock that yields control of the CPU while waiting to acquire the lock. To complete this part of the programming assignment, you will edit the methods within the file ticket_yield_lock.cpp. Some hints for creating this lock :

- Use the atomic method __sync_fetch_and_add(int* ticket, int addition). This method will atomically fetch the value in ticket and add 'addition' to this value.
- To yield control of the CPU, call the method sched_yield()
- You may not need to do anything in the destroy method.

3 Queue Lock

The goal of this portion of the programming assignment is to create a lock that adds waiting threads to a queue and puts threads to sleep. When releasing the lock, if the queue is not empty, the first thread in the queue will be woken. To complete this part of the assignment, you will edit the methods within the file queue_lock.cpp. Some hints for creating this lock :

- The atomic method __sync_lock_test_and_set(int* ptr, int val) will atomically set ptr to the passed value if not already equal to val. It will return what was originally in the ptr.
- The method pause() will cause a thread to sleep indefinitely.
- The method pthread_kill(pthread_t thread, int signal) will wake up the given thread and pass the given signal to the thread. The signal SIGCONT tells the thread to continue (e.g. moves the thread back to the ready queue)
- The method pthread_self() returns the pthread_id of the calling thread, which is of type pthread_t (and should be added to the queue when needed).
- You may not need to do anything in the destroy method.

• SIGCONT seems to not work on some computers. If you have issues with this, you can create your own signal handler. To do so, create a method that does nothing (which is what you want threads to do when you wake them with this signal). Then, each thread should initialize the signal handler before adding itself to the queue with 'signal(SIGUSR1, your_sig_handler_method_name);' Instead of passing SIGCONT to your pthread_kill, you can now pass SIGUSR1.

4 Semaphore Lock

The goal of this portion of the programming assignment is to create a semaphore lock. To complete this part of the assignment, you will edit the methods in the file semaphore_lock.cpp. The lock(...) method will operate equivalently to a semaphore wait(...) method while the unlock(...) method will be equivalent to a semaphore signal(...). Some hints for completing this method :

- The method pthread_mutex_lock(...) can be used to lock the mutex (provided as a part of the lock struct).
- The method pthread_mutex_unlock(...) can be used to unlock the mutex (provided as a part of the lock struct).
- The atomic method __sync_fetch_and_add(int* ptr, int var) will atomically add var to the current value in ptr
- The atomic method __sync_fetch_and_sub(int* ptr, int var) will automatically subtract var from the current value in ptr
- The variable pthread_mutex_t will need to be initialized and destroyed (if you choose to use it).
- Do not include the semaphore header file. You are to implement your own semaphore rather than using that which is provided to you in C.

5 Concurrently Compute PI

The goal of this portion of the programming assignment is to compute the value of pi concurrently among many threads. The value of pi can be computed by randomly accessing points within a unit square, and computing how many of these points fall within the unit circle. To complete this task, you will edit the file compute_pi.cpp. You will add your implementation into the method compute_pi. This method is passed a struct of type thread_data_t, described in the Listings at the top of this assignment.

This code will be linked with all of the locks you created in previous steps. An initial implementation of the lock_t* struct is provided for you in the file mutex_lock.cpp within the submodule, allowing you to test this code before completing all previous tasks.

The method pthread_compute_pi within the submodule file src.cpp initializes a number of threads so that each calls the compute_pi method that you will be editing. This part of the programming assignment can be completed through the subtasks described below.

1. Calculate random coordinates: Each thread should calculate an approximately equal portion of (x,y) coordinates, so that in total the global number of samples are calculated among all threads. Each thread should compute an equal number of samples, with threads with lowest thread ids each calculating one extra sample if the number of samples does not evenly divide the number of threads.

For each local sample, compute two random values (one for x and one for y) between -1 and 1. The method rand() is not thread safe! To calculate a random sample, call thread_rand(), which is implemented for you in src.cpp.

2. Calculate global sum: The variable global_sum is shared by all threads. This variable should be incremented for each (x, y) coordinate that falls within the unit circle. The unit circle has radius 1, so a value is within the unit circle if the following is true:

$$x^2 + y^2 \le 1 \tag{1}$$

- 3. Avoid race conditions: Global sum is a shared variable that all threads will be updating. As a result, this will need to be updated atomically. Use the provided lock to atomically update this method. You must use the provided lock structure, and the methods lock(...) and unlock(...) to pass the autograder. If you were to use an atomic instruction instead, you will fail tests in later portions of the programming assignment.
- 4. **Guarantee performance:** Locks are typically expensive, so make sure to avoid locking within a for loop, otherwise the overhead will cause your program to time out within tests. Similarly, you do not want to lock around an entire for loop, as this will remove all concurrency from your program and also cause time outs. Use techniques discussed in class to optimize the performance of your method.
- 5. No return statement: This method should return NULL. After the method returns, π will be calculated for you with the following equation (you do not need to implement this equation).

 $\pi \leftarrow 4 * \frac{global_sum}{global_number_of_samples} \tag{2}$

6 Check for Correctness

To check that your code is working, do the following:

1. Make sure all tests are passing locally

sh compile.sh
cd build
make test

- 2. Push all changes to GitHub
- 3. Check that your GitHub actions are passing