Instructions : Clone your GitHub Classroom repository for this assignment. Follow all steps below to complete the programming assignment portion of Homework 4. 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.

1 Producer Consumer Problem

The goal of this portion of the programming assignment is to solve the producer consumer problem. The producer will put a value into the buffer (if the buffer has space), and the consumer thread will grab a value from the buffer (if the buffer has entries). The producer thread is passed a `data_t* data` struct (Listing 1), which contains the buffer and a value to insert. The consumer thread is passed only the buffer and should return an integer containing the value it pulls from the buffer. The buffer struct is defined in Listing 2.

Listing 1: data_t* data

```c
buffer_t* buf;
int val;
```

Listing 2: buffer_t* buf

```c
int limit;
int size;
pthread_cond_t empty;
pthread_cond_t full;
pthread_mutex_t mutex;
sem_t* sem_empty;
sem_t* sem_full;
```

To complete this portion of the homework, you will need to call the `put` and `get` methods, which are provided for you and defined in Listings 3 and 4. **Note, these methods are not currently thread safe, and it is your job to make sure that multiple threads are unable to call these methods at the same time.**

Listing 3: put

```c
void put(buffer_t* buf, int val)
{
    buf->list[buf->put_ctr] = val;
    buf->put_ctr = (buf->put_ctr+1) % buf->limit;
    buf->size++;
}
```

Listing 4: get

```
void get(buffer_t* buf, int* val_ptr)
{
    *val_ptr = buf->list[buf->get_ctr];
    buf->get_ctr = (buf->get_ctr+1) % buf->limit;
    buf->size--;
}
```

To get full credit on this portion of the assignment, you should edit the `producer_thread()` method to put the given value in the buffer, but only once the buffer has room. Similarly, you should edit the `consumer_thread()` method to get a value from the buffer, but only once the buffer has values. **You will do this in two different ways, described below.** The producer thread should return NULL, but the consumer thread must return the value that is received from the `get()` method. *Note, there may be multiple consumers, multiple producers, or both.*

1.1 Using Condition Variables:

To complete this portion of the homework, edit the file `condition_var.cpp`. Fill in the producer and consumer methods as described above, using the provided condition variables and mutex (Listing 2). You can assume all condition variables and locks have been initialized before the method is called.

Some hints:

- Use the methods `pthread_cond_wait` and `pthread_cond_signal`.
- The method `pthread_cond_wait` takes both a condition variable and a locked mutex.
- You have two condition variables to work with, and you should use both of them.
- The size and limit variables, within the `buffer_t*` struct, can be used to determine if there is buffer space available and if anything is in the buffer.
- Condition variables require while loops.

1.2 Using Semaphores:

To complete this portion of the homework, edit the file `semaphore.cpp`. Fill in the producer and consumer methods as described above, using the provided sempahore variables and mutex. You can assume all condition variables and locks have been initialized before this method was called.

Some hints:

- Use the methods `sem_wait` and `sem_post`.
- You have two semaphores to work with, and you should use both of them.
- The size and limit variables, within the `buffer_t*` struct, can be used to determine if there is buffer space available and if anything is in the buffer.

2 Dining Philosophers

The goal of this portion of the programming assignment is to implement the dining philosophers problem. The method is passed a `diners_t*` struct, defined in Listing 5.

Listing 5: diners_t* diner

```
sem_t** forks;
int size;
pthread_mutex_t mutex;
int philosopher;
bool* eat;
```

Cont.

To complete this portion of the assignment, you will need to use the methods defined in Listings 6-8. The method `left` returns the semaphore that controls the calling philosophers left fork, while the method `right` returns the semaphore that controls the calling philosophers right fork.

Listing 6: left

```
sem_t* left(diners_t* diner)
{
    int pos = diner->philosopher - 1;
    if (pos < 0)
        pos += diner->size;
    return diner->forks[pos];
}
```

Listing 7: right

```
sem_t* right(diners_t* diner)
{
    return diner->forks[diner->philosopher];
}
```

Listing 8: eat

```
void eat(diners_t* diner)
{
    if (sem_trywait(left(diner)) == 0)
    {
        sem_post(left(diner));
        return;
    }

    if (sem_trywait(right(diner)) == 0)
    {
        sem_post(right(diner));
        return;
    }

    diner->eat[diner->philosopher] = true;
}
```

A given fork can be acquired with the method `sem_wait()` and can be released with `sem_post()`. Once a philosopher has both forks, that philosopher can call eat by calling the method `eat(diners_t* diner)`. After the eat method returns, the philosopher should release both forks. **The standard approach for the philosopher problem has all philosophers grab their left fork before any grabs the right. This can result in a deadlock. You will prevent this deadlock in two different ways, described below.**

2.1 Avoiding the Circular Wait:

For this portion of the assignment, you will edit the file `circular.cpp`. Complete the dining philosophers problem described above by avoiding the circular wait. Make sure that all philosophers cannot grab the same fork first.

2.2 Avoiding the Hold-and-Wait:

For this portion of the assignment, you will edit the file `hold_and_wait.cpp`. Complete the dining philosophers problem described above by avoiding hold-and-wait. Use the mutex lock to have a single philosopher

Cont.

grab all necessary forks at once.

3 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

The End.