# Pthreads Exercises

1- Create a program so that each thread receives an integer argument representing its ID and prints "Thread X is running". Use pthread_create() to pass the ID inside a **struct ThreadData { int id; }**, ensuring each thread correctly handles its data.

2- Create a program with three threads where each thread computes the square of its ID. Store results inside a struct ThreadResult { int id; int square; }, and ensure the main thread waits for all threads to finish before printing their squared results.

3- **Parallel Sum:** Implement a program that calculates the sum of an integer array using two threads. Each thread should sum half of the array, using a **struct SumData { int* array; int start; int end; int result; }** to pass array information. The main thread collects both partial sums to compute the final result.

4- **Vector Addition:** Given two arrays of size N, create N threads, where each thread computes one element of the resulting sum array, C[i] = A[i] + B[i]. Define **struct VectorData { int* A; int* B; int* C; int index; }** to pass individual indices to the threads.

5- **Race Condition and Mutex:** Implement a shared counter that multiple threads increment concurrently. Define **struct Counter { int value; pthread_mutex_t lock; }**, and use a mutex inside the struct to synchronize access to the counter, preventing race conditions.

6- **Parallel Prime Number Finder:** Given a range [L, R], divide it among N threads so each thread finds prime numbers in its assigned subrange. Use **struct PrimeData { int start; int end; int* primes; int count; pthread_mutex_t lock; }** to manage thread-safe access to the shared prime number list.

7- You are given an array of integers of size `N` and a target value `T`. Your task is to write a **multithreaded C program using pthreads** that searches for the target value in the array using **parallel simple (linear) search**.

8- **Multithreaded Queue Operation using Pthreads**

Use the concurrent queue implementation from the textbook to build a multithreaded C program where each thread performs enqueue or dequeue operations based on a given string of commands.

## Description

You are given a thread-safe (concurrent) queue implementation. Your task is to write a C program using **pthreads** that:

- Creates **N threads**.
- Each thread receives a command string (e.g., "E1E2E3DDE4") containing operations on the shared concurrent queue.
- Each character in the string corresponds to:
    - **'E' followed by a number:** enqueue the number into the queue.
    - **'D':** dequeue an element from the queue.

## Instructions

1. Use the concurrent queue implementation from the textbook.
2. Each thread should Perform enqueue and dequeue operations accordingly, using the shared queue.
3. Wait for all threads to finish.
4. After all threads complete, print the final contents of the queue.

9- Create two threads: Thread A should wait for a signal. Thread B should send the signal after 2 seconds.

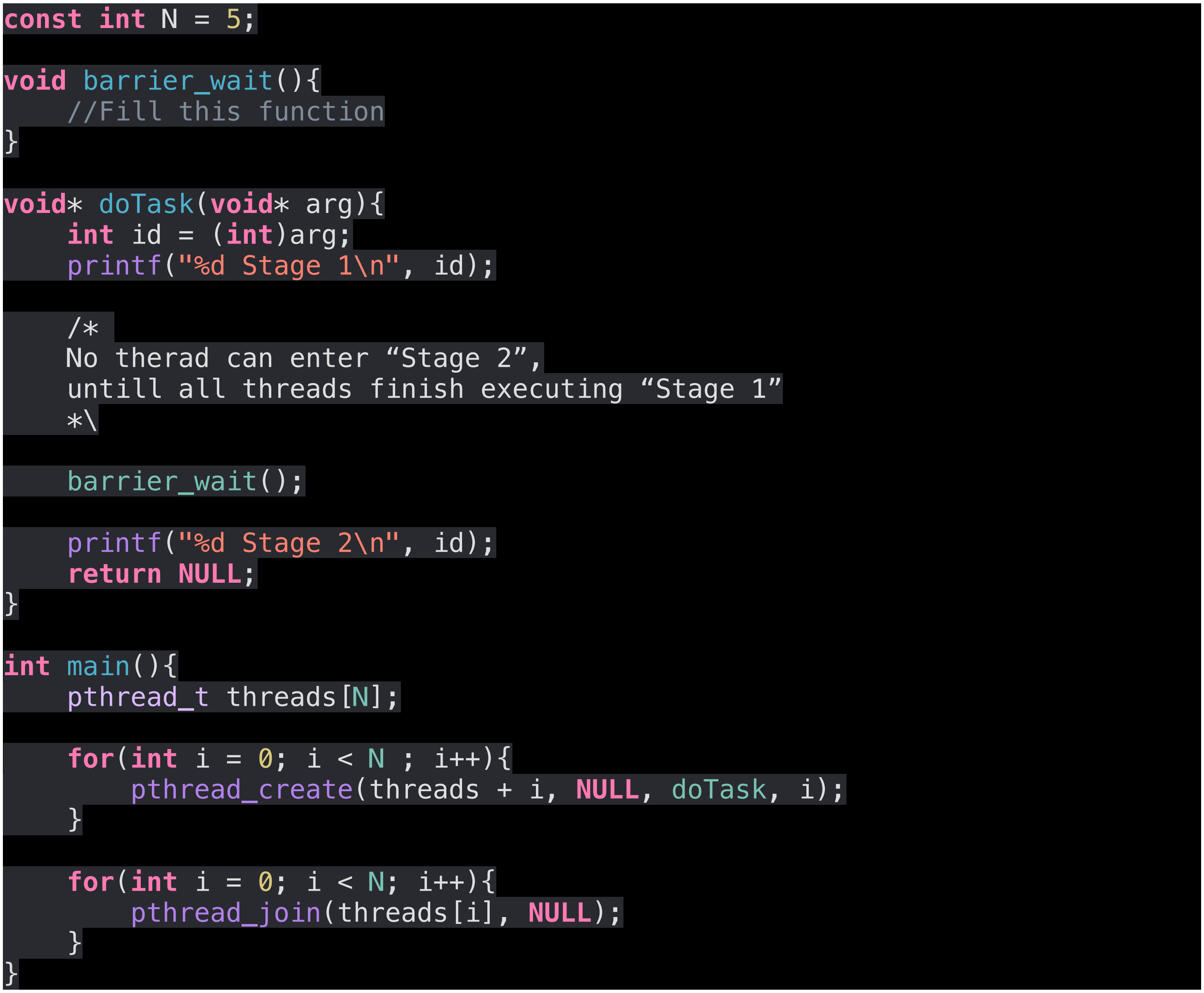
10- Create two threads that print "Ping" and "Pong" alternatively, 5 times.

11- Complete the producer-consumer program shown in Figure 30.14 of the textbook Operating Systems: Three Easy Pieces. Your task is to write a main() function that creates one or more producer and consumer threads. Each producer should produce a sequence of integers (for example, from 0 to 99) and insert them into the buffer using the put() function. Consumers should retrieve these integers using the get() function and print each consumed value.

You must also implement a proper termination mechanism. For example, if the number of items to be produced is known, consumers may iterate a fixed number of times accordingly. Alternatively, producers can send a sentinel value such as -1 to signal consumers to terminate.

Make sure the program runs correctly without data races, infinite loops, or lost values.

12- In concurrent programming, a **barrier** is a synchronization primitive used to make a group of threads wait until all threads reach a certain point in their execution. Only once all threads have reached the barrier can they proceed further. Given the code in **Figure 1** below, Fill the **barrier_wait()** function that must do the barrier job.



```
const int N = 5;

void barrier_wait(){
    //Fill this function
}

void* doTask(void* arg){
    int id = (int)arg;
    printf("%d Stage 1\n", id);

    /*
    No therad can enter “Stage 2”,
    untill all threads finish executing “Stage 1”
    *\

    barrier_wait();

    printf("%d Stage 2\n", id);
    return NULL;
}

int main(){
    pthread_t threads[N];

    for(int i = 0; i < N ; i++){
        pthread_create(threads + i, NULL, doTask, i);
    }

    for(int i = 0; i < N; i++){
        pthread_join(threads[i], NULL);
    }
}
```

Figure 1: Starting code for implementing a simple barrier