Homework #6

Question 1 (5 pt.)

The program below implements a square 2D matrix multiplication algorithm, where matrices *A* and *B* are initialized with random integer values between 0 and 9, and matrix *C* is calculated as the product of the former two matrices. Notice that the 2D matrices are represented using a linearly projected, row-major format, where element *i*, *j* of matrix *X* can be accessed with the expression `X[i * DIM + j]`.

```
#include <stdio.h>

#define DIM 5

int A[DIM * DIM];
int B[DIM * DIM];
int C[DIM * DIM];

void InitializeMatrix(int *X)
{
	int i, j;
	for (i = 0; i < DIM; i++)
		for (j = 0; j < DIM; j++)
			X[i * DIM + j] = random() % 10;
}

void PrintMatrix(int *X)
{
	int i, j;
	for (i = 0; i < DIM; i++)
	{
		for (j = 0; j < DIM; j++)
			printf("%3d ", X[i * DIM + j]);
		printf("\n");
	}
	printf("\n");
}

void MultiplyMatrices()
{
	int i, j, k;
	for (i = 0; i < DIM; i++)
	{
		for (j = 0; j < DIM; j++)
		{
			int sum = 0;
			for (k = 0; k < DIM; k++)
				sum += A[i * DIM + k] * B[k * DIM + j];
			C[i * DIM + j] = sum;
		}
	}
}
```

```
int main()
{
       InitializeMatrix(A);
       InitializeMatrix(B);
       MultiplyMatrices();
       PrintMatrix(A);
       PrintMatrix(B);
       PrintMatrix(C);
       return 0;
}
```

Type this program, compile it, and run it, making sure that it provides the right results.

a) Replace the invocation to `MultiplyMatrices` with another function named `MultiplyMatricesParallel`. In this function, parallelize the matrix multiplication algorithm by creating one separate thread for each iteration of the outer loop (loop with induction variable *i*). Each child thread should do the work corresponding to the two inner loops with induction variables *j* and *k*. Verify that the result of the program is the same, and upload your code in a file named `q1.c`.

b) Do you need to include any locks in your parallel implementation of the matrix multiplication algorithm? Justify your answer.

Question 2 (5 pt.)

A barrier is a type of global synchronization mechanism where each thread has to reach a given point in the execution, before any of them can continue. When the first thread of a group of threads reaches a barrier, it gets suspended. When the second thread reaches the same barrier, it is suspended, too. And only when the last thread in the group reaches the barrier, all of them continue.

Consider the following header file, containing a data structure that represents a barrier, together with the prototype of functions `barrier_init` (which takes the number of threads in a group as an argument), and `barrier_wait` (which takes the unique identifier of the current thread). As you can see, the barrier structure stores the number of threads in the group (`count`), an array of boolean values indicating whether a given thread reached the barrier or not (`reached`), a mutex, and a set of condition variables (`cond`):

```
#ifndef BARRIER_H
#define BARRIER_H

#include <pthread.h>

#define BARRIER_MAX_THREADS 10

struct barrier_t
{
       int count;
       int reached[BARRIER_MAX_THREADS];
       pthread_mutex_t mutex;
       pthread_cond_t cond[BARRIER_MAX_THREADS];
};

void barrier_init(struct barrier_t *barrier, int count);
void barrier_wait(struct barrier_t *barrier, int id);

#endif
```

a) Write an implementation for function `barrier_init`. This function should initialize all fields of the barrier object passed by reference in the first argument. You can use functions `pthread_mutex_init` and `pthread_cond_init` to initialize mutexes and condition variables dynamically (see `man` pages).

b) Write an implementation for function `barrier_wait`. This function should leverage the mutex and the set of condition variables to implement the barrier behavior. You can assume that each thread will invoke this function by passing its own unique identifier in the second argument, which ranges between 0 and `barrier->count - 1`. Here are some hints on the steps to follow in the function:

   - Mark current thread as having reached the barrier.
   - Check if everyone arrived in the barrier.
   - If so, wake everyone up and continue.
   - If not, suspend the current thread in its associated condition variable.

   Upload your implementation for parts a) and b) in a file named `barrier.c`.

c) Write a main program that creates 5 child threads, where each thread performs the following actions:

- Prints a message notifying execution start, with a thread ID
- Suspends itself in a barrier
- Prints a message notifying execution end, with a thread ID

Notice that, if everything works well, you should see this program printing all execution start messages first, followed by all execution end messages at once. Upload your main program in a file named `q2.c`. Your code should compile correctly with command `gcc q2.c barrier.c -o q2 -lpthread`.