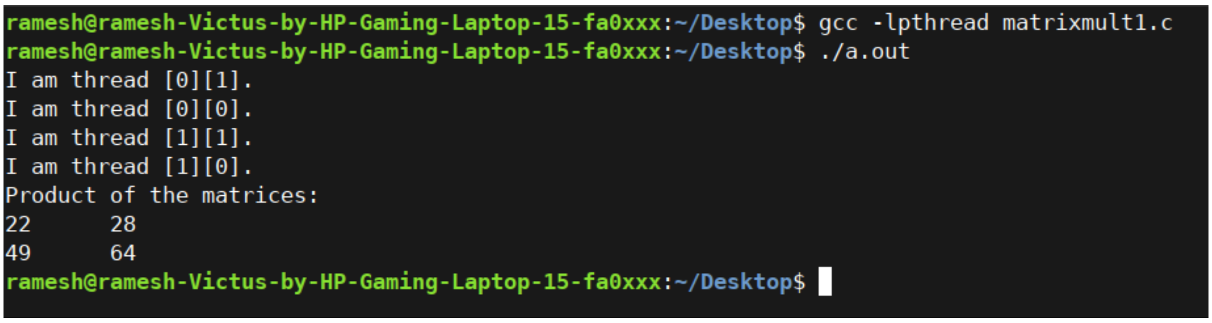
Assignment 7: Posix Threads (pthreads)

Name: Ramesh Chandra Soren Enrollment No: 2022CSB086

Submission of programs

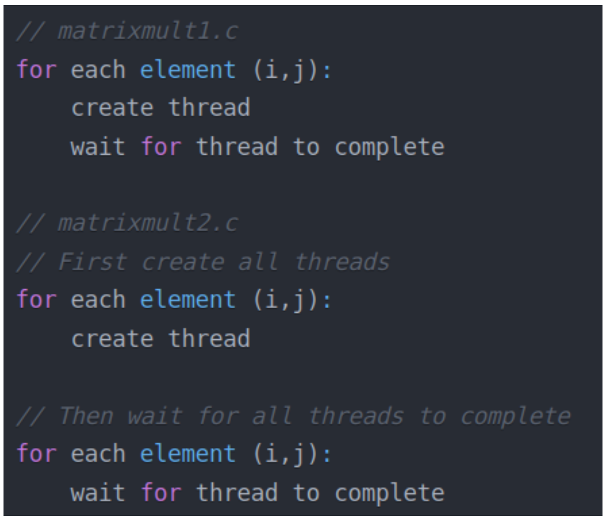


```
ramesh@ramesh-Victus-by-HP-Gaming-Laptop-15-fa0xxx:~/Desktop$ gcc -lpthread matrixmult1.c
ramesh@ramesh-Victus-by-HP-Gaming-Laptop-15-fa0xxx:~/Desktop$ ./a.out
I am thread [0][1].
I am thread [0][0].
I am thread [1][1].
I am thread [1][0].
Product of the matrices:
22      28
49      64
ramesh@ramesh-Victus-by-HP-Gaming-Laptop-15-fa0xxx:~/Desktop$ 
```

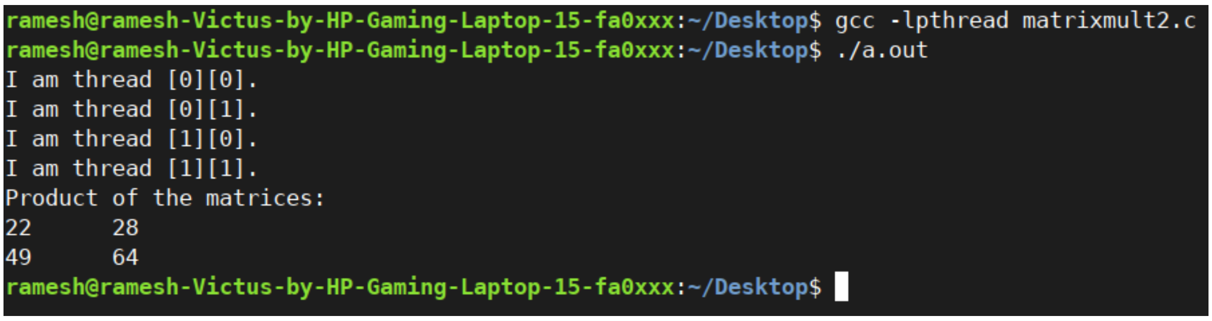
This version (matrixmult2.c) has a key difference in how it handles thread creation and synchronization compared to matrixmult1.c. Here are the main differences:

1. Thread Creation and Joining Pattern:
   - In matrixmult1.c: The program creates one thread and waits for it to complete (join) before creating the next thread, making it essentially sequential.
   - In matrixmult2.c: All threads are created first in one loop, and then joined in a separate loop. This allows true parallel execution.
2. Thread ID Storage:
   - matrixmult1.c: Uses a single thread ID variable that gets reused
   - matrixmult2.c: Uses an array of thread IDs pthread_t tids[m][r] to store all thread IDs
3. pthread_attr_init() Usage:
   - matrixmult1.c: Incorrectly initializes thread attributes inside the loops (potentially causing undefined behavior)
   - matrixmult2.c: Correctly initializes thread attributes once before the loops

Here's the structural difference in pseudo-code:



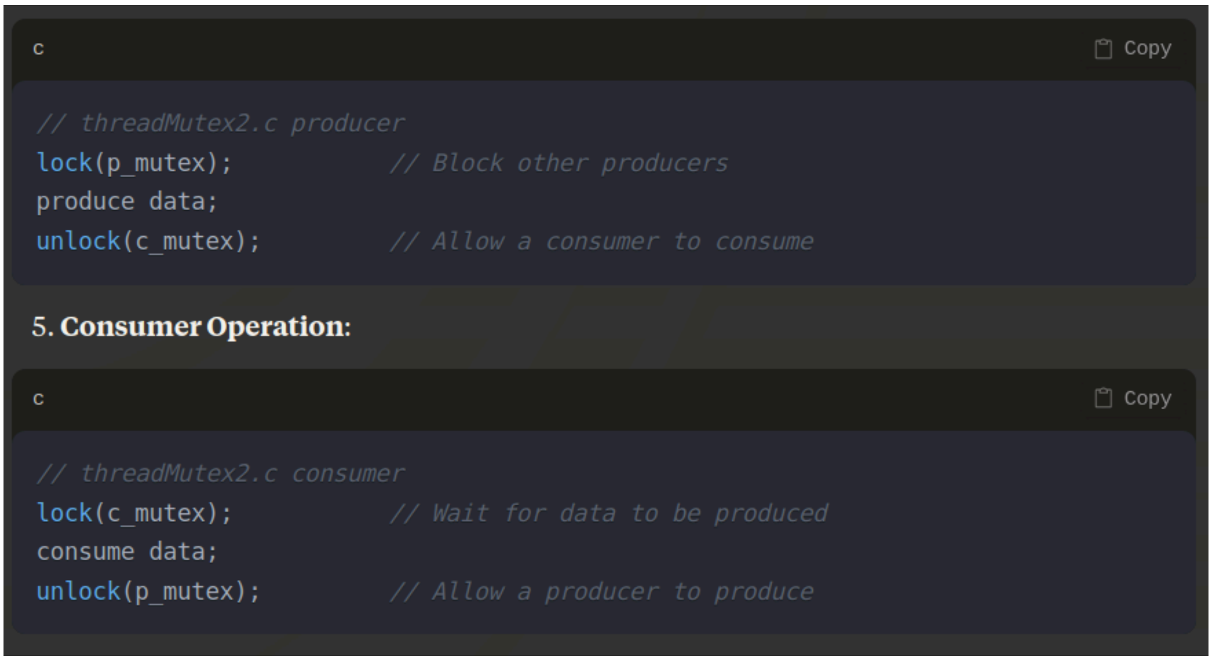
```
// matrixmult1.c
for each element (i,j):
    create thread
    wait for thread to complete

// matrixmult2.c
// First create all threads
for each element (i,j):
    create thread

// Then wait for all threads to complete
for each element (i,j):
    wait for thread to complete
```



```
ramesh@ramesh-Victus-by-HP-Gaming-Laptop-15-fa0xxx:~/Desktop$ gcc -lpthread matrixmult2.c
ramesh@ramesh-Victus-by-HP-Gaming-Laptop-15-fa0xxx:~/Desktop$ ./a.out
I am thread [0][0].
I am thread [0][1].
I am thread [1][0].
I am thread [1][1].
Product of the matrices:
22      28
49      64
ramesh@ramesh-Victus-by-HP-Gaming-Laptop-15-fa0xxx:~/Desktop$
```

This version (threadMutex2.c) implements a proper producer-consumer synchronization mechanism, which is the main difference from threadMutex1.c. Here are the key differences:

1. Mutex Usage:
   - threadMutex1.c: Used only one mutex (pc_mutex) to protect producers
   - threadMutex2.c: Uses two mutexes:
     - p_mutex: for producers
     - c_mutex: for consumers
2. Synchronization Pattern:
   - In threadMutex2.c:
     - Producers must wait for consumers to consume before producing new data
     - Consumers must wait for producers to produce before consuming
     - Each piece of data is consumed exactly once
3. Initial State:
   - threadMutex2.c initializes by locking c_mutex at the start, ensuring consumers wait until the first data is produced
4. Producer Operation:



```c
// threadMutex2.c producer
lock(p_mutex);          // Block other producers
produce data;
unlock(c_mutex);        // Allow a consumer to consume
```

5. **Consumer Operation:**

```c
// threadMutex2.c consumer
lock(c_mutex);          // Wait for data to be produced
consume data;
unlock(p_mutex);        // Allow a producer to produce
```

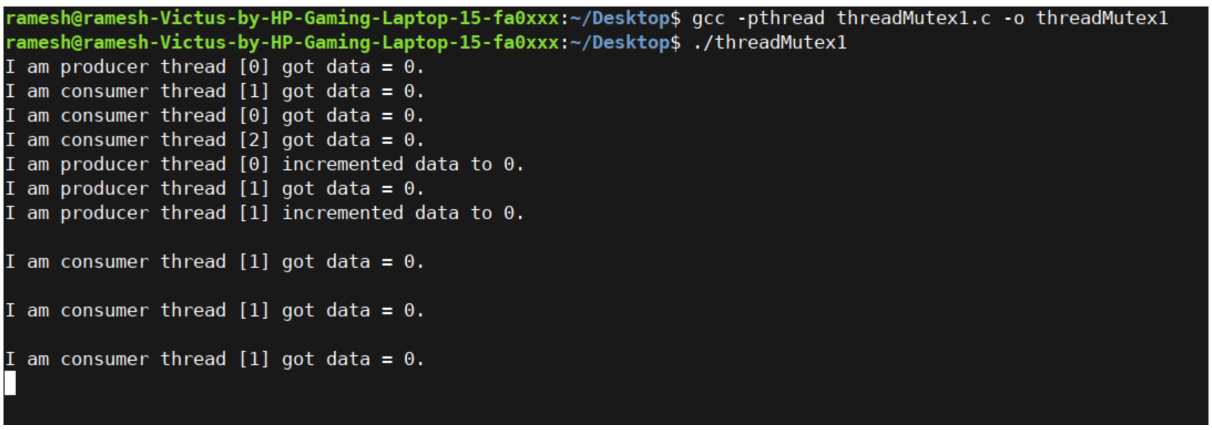
This creates an alternating pattern where:

1. Producer produces data
2. Consumer consumes it
3. Producer produces new data
4. And so on...

The main improvements over threadMutex1.c are:

- No data is lost (overwritten before consumption)
- No data is consumed multiple times
- Proper synchronization between producers and consumers
- Guaranteed one-to-one production and consumption of data

This version implements a more correct producer-consumer pattern, while threadMutex1.c was more of a demonstration of basic mutex usage with known flaws.

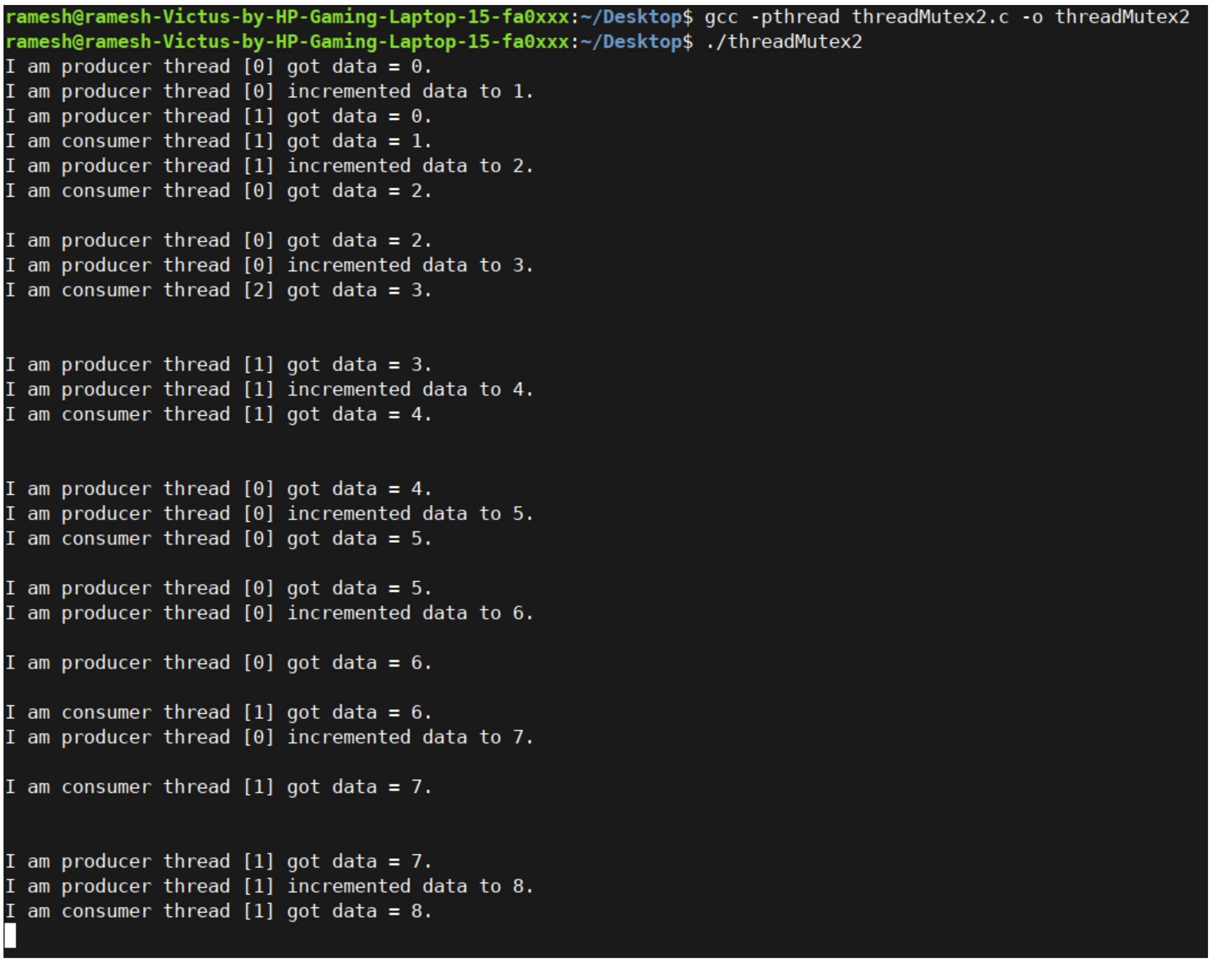


```
ramesh@ramesh-Victus-by-HP-Gaming-Laptop-15-fa0xxx:~/Desktop$ gcc -pthread threadMutex1.c -o threadMutex1
ramesh@ramesh-Victus-by-HP-Gaming-Laptop-15-fa0xxx:~/Desktop$ ./threadMutex1
I am producer thread [0] got data = 0.
I am consumer thread [1] got data = 0.
I am consumer thread [0] got data = 0.
I am consumer thread [2] got data = 0.
I am producer thread [0] incremented data to 0.
I am producer thread [1] got data = 0.
I am producer thread [1] incremented data to 0.

I am consumer thread [1] got data = 0.

I am consumer thread [1] got data = 0.

I am consumer thread [1] got data = 0.
```

This output demonstrates several concurrency issues:

- Race conditions between producers and consumers
- Multiple consumers reading same data
- Failed increment operations
- Lack of proper synchronization between threads

These issues occur because threadMutex1.c only implements basic mutex protection for producers but doesn't properly coordinate between producers and consumers. This is why threadMutex2.c was created with better synchronization mechanisms.



```
ramesh@ramesh-Victus-by-HP-Gaming-Laptop-15-fa0xxx:~/Desktop$ gcc -pthread threadMutex2.c -o threadMutex2
ramesh@ramesh-Victus-by-HP-Gaming-Laptop-15-fa0xxx:~/Desktop$ ./threadMutex2
I am producer thread [0] got data = 0.
I am producer thread [0] incremented data to 1.
I am producer thread [1] got data = 0.
I am consumer thread [1] got data = 1.
I am producer thread [1] incremented data to 2.
I am consumer thread [0] got data = 2.

I am producer thread [0] got data = 2.
I am producer thread [0] incremented data to 3.
I am consumer thread [2] got data = 3.


I am producer thread [1] got data = 3.
I am producer thread [1] incremented data to 4.
I am consumer thread [1] got data = 4.


I am producer thread [0] got data = 4.
I am producer thread [0] incremented data to 5.
I am consumer thread [0] got data = 5.

I am producer thread [0] got data = 5.
I am producer thread [0] incremented data to 6.

I am producer thread [0] got data = 6.

I am consumer thread [1] got data = 6.
I am producer thread [0] incremented data to 7.

I am consumer thread [1] got data = 7.


I am producer thread [1] got data = 7.
I am producer thread [1] incremented data to 8.
I am consumer thread [1] got data = 8.
```

This proper behavior is achieved through:

- Using two mutexes (p_mutex and c_mutex)
- Proper synchronization between producers and consumers
- Ensuring each value is consumed before next production
- Preventing multiple consumers from reading same value