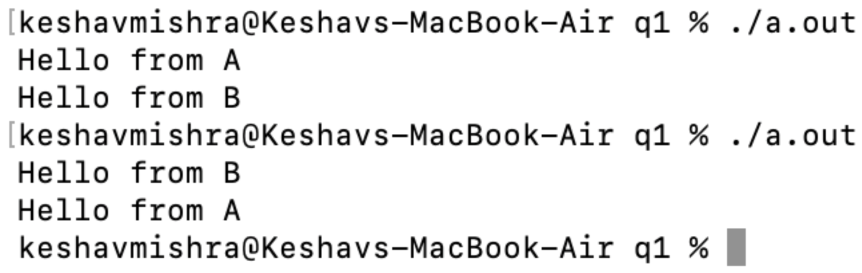
Q1 Part 1:
Code:

```
#include <stdio.h>
#include <pthread.h>

void* threadA(void* arg)
{
  printf("Hello from A\n");
  return NULL;
}

void* threadB(void* arg)
{
  printf("Hello from B\n");
  return NULL;
}

int main()
{
  pthread_t tA, tB;

  pthread_create(&tA, NULL, threadA, NULL);
  pthread_create(&tB, NULL, threadB, NULL);

  pthread_join(tA, NULL);
  pthread_join(tB, NULL);

  return 0;
}
```



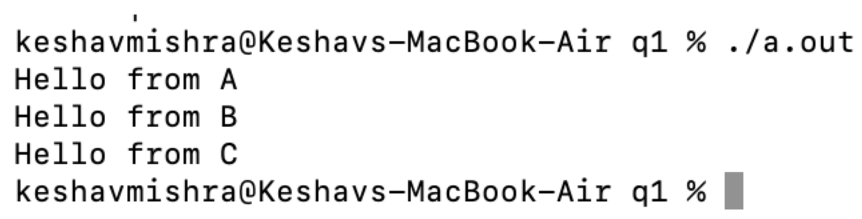
```
keshavmishra@Keshavs-MacBook-Air q1 % ./a.out
Hello from A
Hello from B
keshavmishra@Keshavs-MacBook-Air q1 % ./a.out
Hello from B
Hello from A
keshavmishra@Keshavs-MacBook-Air q1 %
```

Screenshot of output of Q!_12341140_part1.c

Explanation:
Because without synchronization both threads start concurrently. The OS scheduler decides which thread runs first. There is no guarantee of order, the CPU may run threadA first in one run and threadB first in another.

Q1 Part 2:

Code:
```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t semAB;  // To ensure B runs after A
sem_t semBC;  // To ensure C runs after B

void* threadA(void* arg)
{
  printf("Hello from A\n");
  sem_post(&semAB);  // Signal B to start
  return NULL;
}

void* threadB(void* arg)
{
  sem_wait(&semAB);  // Wait for A to finish
  printf("Hello from B\n");
  sem_post(&semBC);  // Signal C to start
  return NULL;
}

void* threadC(void* arg)
{
  sem_wait(&semBC);  // Wait for B to finish
  printf("Hello from C\n");
  return NULL;
}

int main()
{
  pthread_t tA, tB, tC;

  sem_init(&semAB, 0, 0);
  sem_init(&semBC, 0, 0);

  pthread_create(&tA, NULL, threadA, NULL);
  pthread_create(&tB, NULL, threadB, NULL);
  pthread_create(&tC, NULL, threadC, NULL);

  pthread_join(tA, NULL);
  pthread_join(tB, NULL);
  pthread_join(tC, NULL);

  sem_destroy(&semAB);
  sem_destroy(&semBC);

  return 0;
}
```


```
keshavmishra@Keshavs-MacBook-Air q1 % ./a.out
Hello from A
Hello from B
Hello from C
keshavmishra@Keshavs-MacBook-Air q1 %
```

Screenshot of output of Q1_12341140_part2.c

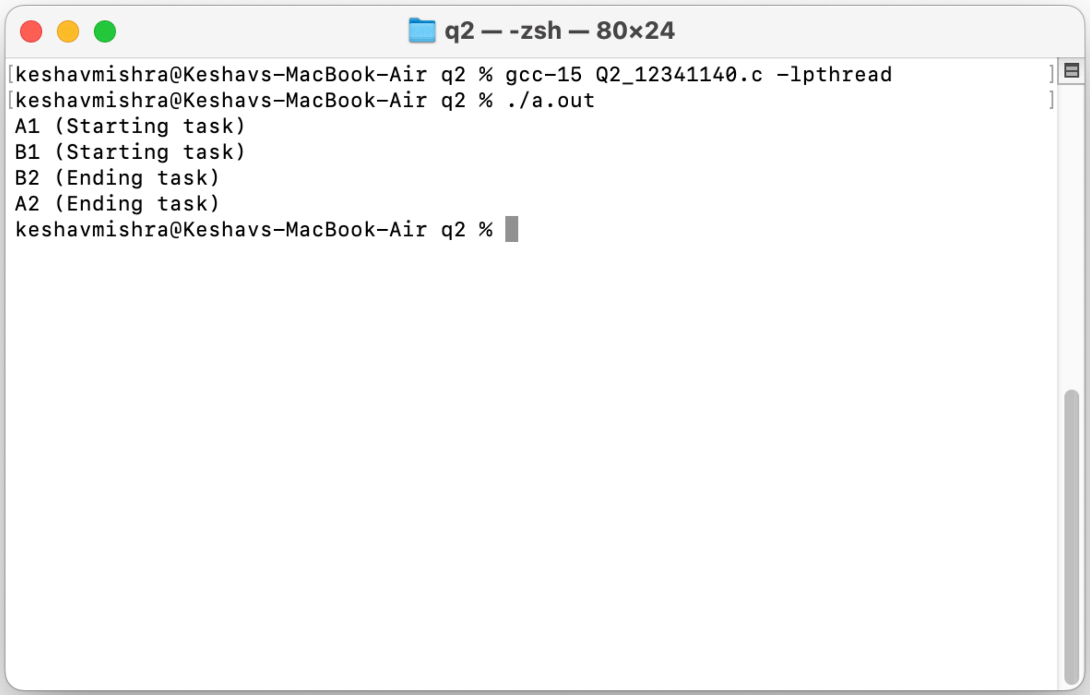
Explanation:
Semaphores are used to control the order of thread execution. Thread A prints first, signals thread B using sem_post, and then thread B runs after waiting on the semaphore with sem_wait. Adding thread C with another semaphore makes it run only after thread B finishes.

Q2

Code:

```
#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <fcntl.h>

#include <unistd.h>

sem_t *aArrived, *bArrived;

void* threadA(void* arg)

{

   printf("A1 (Starting task)\n");

   sem_post(aArrived);

   sem_wait(bArrived);

   printf("A2 (Ending task)\n");

   return NULL;

}

void* threadB(void* arg)
```

```
{
    printf("B1 (Starting task)\n");
    sem_post(bArrived);
    sem_wait(aArrived);
    printf("B2 (Ending task)\n");
    return NULL;
}
int main()
{
    pthread_t tA, tB;
    aArrived = sem_open("/aArrived", O_CREAT, 0644, 0);
    bArrived = sem_open("/bArrived", O_CREAT, 0644, 0);
    pthread_create(&tA, NULL, threadA, NULL);
    pthread_create(&tB, NULL, threadB, NULL);
    pthread_join(tA, NULL);
    pthread_join(tB, NULL);
    sem_close(aArrived);
    sem_close(bArrived);
    sem_unlink("/aArrived");
    sem_unlink("/bArrived");
    return 0;
}
```



```
keshavmishra@Keshavs-MacBook-Air q2 % gcc-15 Q2_12341140.c -lpthread
keshavmishra@Keshavs-MacBook-Air q2 % ./a.out
A1 (Starting task)
B1 (Starting task)
B2 (Ending task)
A2 (Ending task)
keshavmishra@Keshavs-MacBook-Air q2 %
```

Screenshot of output of Q2_12341140.c

Explanation:

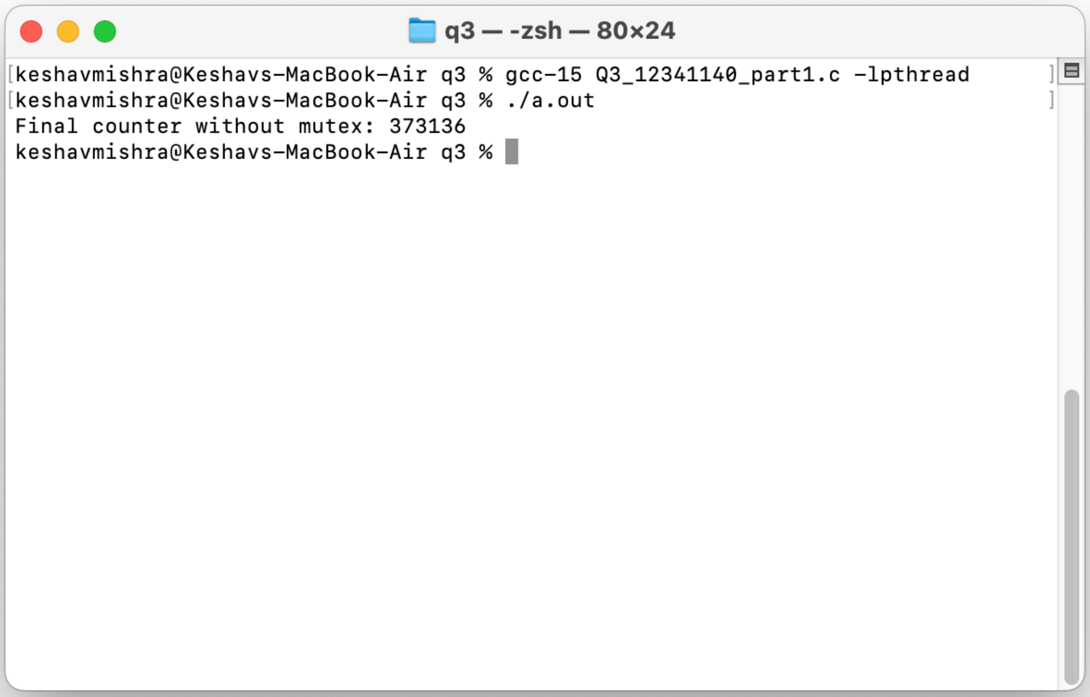
Two threads meet at a rendezvous point using two semaphores. Thread A prints A1 then waits for B1 before printing A2, and thread B prints B1 then waits for A1 before printing B2. This ensures A1 happens before B2 and B1 happens before A2.

Q3 Part 1

Code:

```
#include <stdio.h>
#include <pthread.h>

#define LIMIT 1000000
#define THREADS 10

int counter = 0;

void* increment(void* arg)
{
   for (int i = 0; i < LIMIT / THREADS; i++)
   {
      counter++; // race condition
   }
   return NULL;
}

int main()
{
   pthread_t t[THREADS];
```

```
    for (int i = 0; i < THREADS; i++)
        pthread_create(&t[i], NULL, increment, NULL);
    for (int i = 0; i < THREADS; i++)
        pthread_join(t[i], NULL);

    printf("Final counter without mutex: %d\n", counter);
    return 0;
}
```



```
keshavmishra@Keshavs-MacBook-Air q3 % gcc-15 Q3_12341140_part1.c -lpthread
keshavmishra@Keshavs-MacBook-Air q3 % ./a.out
Final counter without mutex: 373136
keshavmishra@Keshavs-MacBook-Air q3 %
```

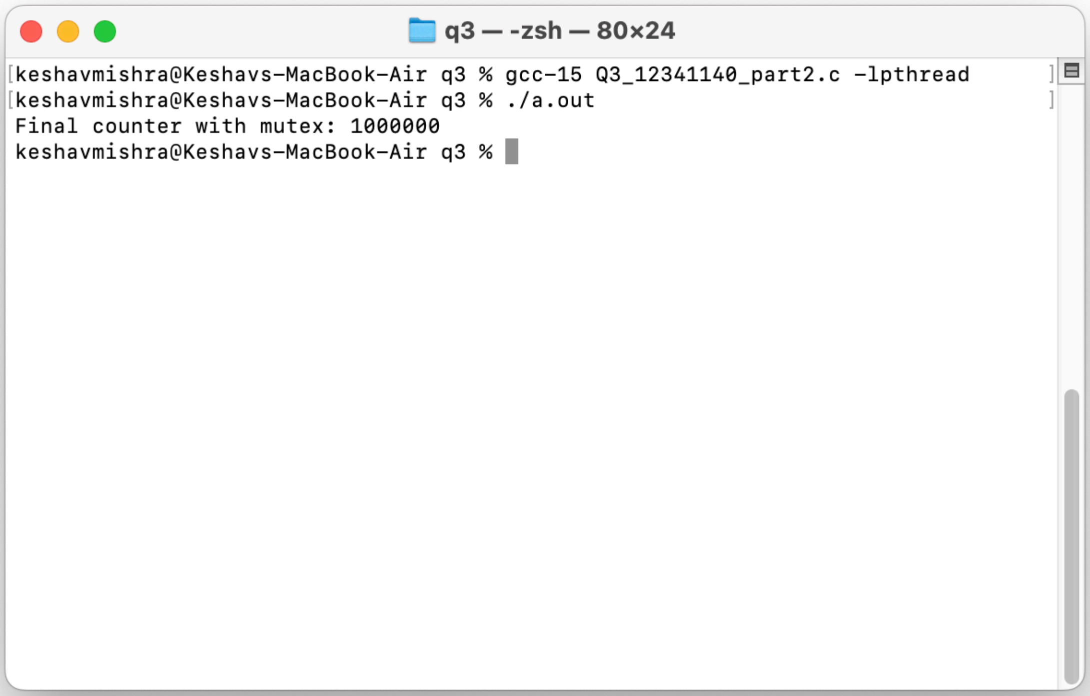
Screenshot of output of Q3_12341140_part1.c

Explanation:
When multiple threads increment the counter simultaneously without mutex, they access and modify the shared variable at the same time. This causes race conditions where updates overlap, leading to an incorrect final count.

Q3 Part 2

Code:
```
#include <stdio.h>
#include <pthread.h>

#define LIMIT 1000000
#define THREADS 10

int counter = 0;
pthread_mutex_t lock;
```

```
void* increment(void* arg)
{
   for (int i = 0; i < LIMIT / THREADS; i++)
   {
      pthread_mutex_lock(&lock);
      counter++; // protected
      pthread_mutex_unlock(&lock);
   }
   return NULL;
}

int main()
{
   pthread_t t[THREADS];
   pthread_mutex_init(&lock, NULL);

   for (int i = 0; i < THREADS; i++)
      pthread_create(&t[i], NULL, increment, NULL);
   for (int i = 0; i < THREADS; i++)
      pthread_join(t[i], NULL);

   pthread_mutex_destroy(&lock);
   printf("Final counter with mutex: %d\n", counter);
   return 0;
}
```



```
keshavmishra@Keshavs-MacBook-Air q3 % gcc-15 Q3_12341140_part2.c -lpthread
keshavmishra@Keshavs-MacBook-Air q3 % ./a.out
Final counter with mutex: 1000000
keshavmishra@Keshavs-MacBook-Air q3 %
```

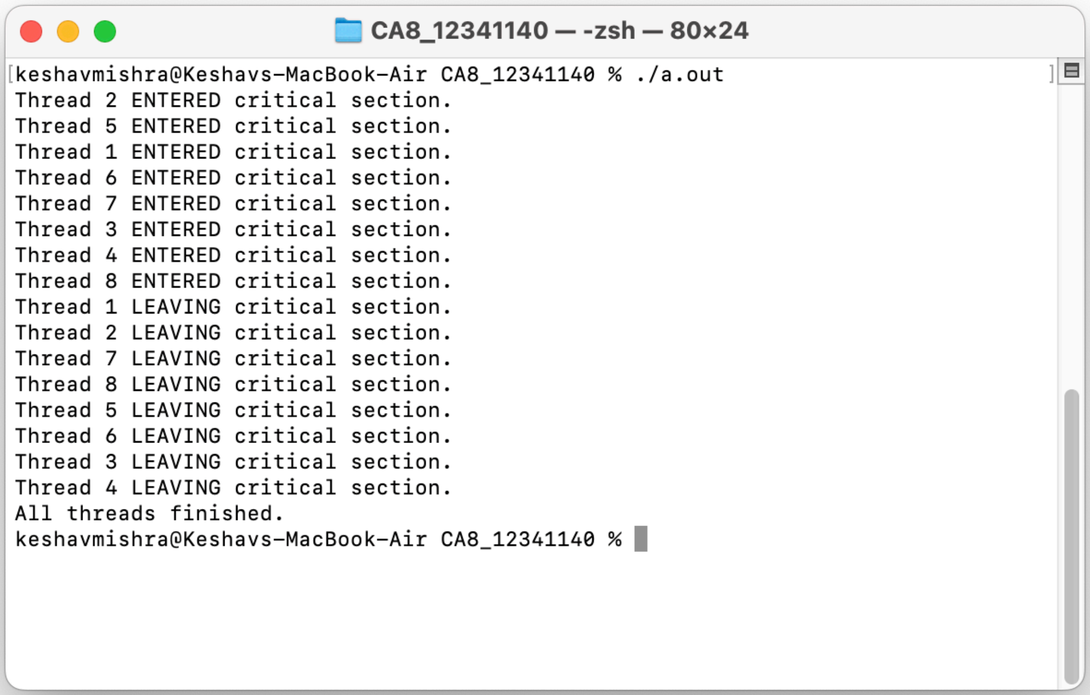
Screenshot of output of Q3_12341140_part2.c

Explanation:
Using pthread_mutex_lock and pthread_mutex_unlock ensures only one thread modifies the counter at a time. This prevents race conditions and gives a consistent final count of 1000000 after all threads finish.

Q4

Code:
```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>


#define NUM_THREADS 8
#define MAX_CONCURRENT 3     // maximum threads allowed in critical section


sem_t multiplex;


void* worker(void* arg)
{
  int id = *(int*)arg;
  // Request to enter
  sem_wait(&multiplex); // take one token
  printf("Thread %d ENTERED critical section.\n", id);
  // Critical section (simulate some work)
  sleep(1); // simulate processing
  printf("Thread %d LEAVING critical section.\n", id);
  sem_post(&multiplex); // release token
  return NULL;
}


int main()
{
  pthread_t threads[NUM_THREADS];
  int ids[NUM_THREADS];

  // Initialize semaphore to MAX_CONCURRENT
  sem_init(&multiplex, 0, MAX_CONCURRENT);

  // Create threads
  for (int i = 0; i < NUM_THREADS; i++)
  {
    ids[i] = i + 1;
    pthread_create(&threads[i], NULL, worker, &ids[i]);
  }

  // Join threads
```

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

 sem_destroy(&multiplex);
 printf("All threads finished.\n");
 return 0;
}
```



```
keshavmishra@Keshavs-MacBook-Air CA8_12341140 % ./a.out
Thread 2 ENTERED critical section.
Thread 5 ENTERED critical section.
Thread 1 ENTERED critical section.
Thread 6 ENTERED critical section.
Thread 7 ENTERED critical section.
Thread 3 ENTERED critical section.
Thread 4 ENTERED critical section.
Thread 8 ENTERED critical section.
Thread 1 LEAVING critical section.
Thread 2 LEAVING critical section.
Thread 7 LEAVING critical section.
Thread 8 LEAVING critical section.
Thread 5 LEAVING critical section.
Thread 6 LEAVING critical section.
Thread 3 LEAVING critical section.
Thread 4 LEAVING critical section.
All threads finished.
keshavmishra@Keshavs-MacBook-Air CA8_12341140 %
```

Screenshot of output of Q4_12341140.c

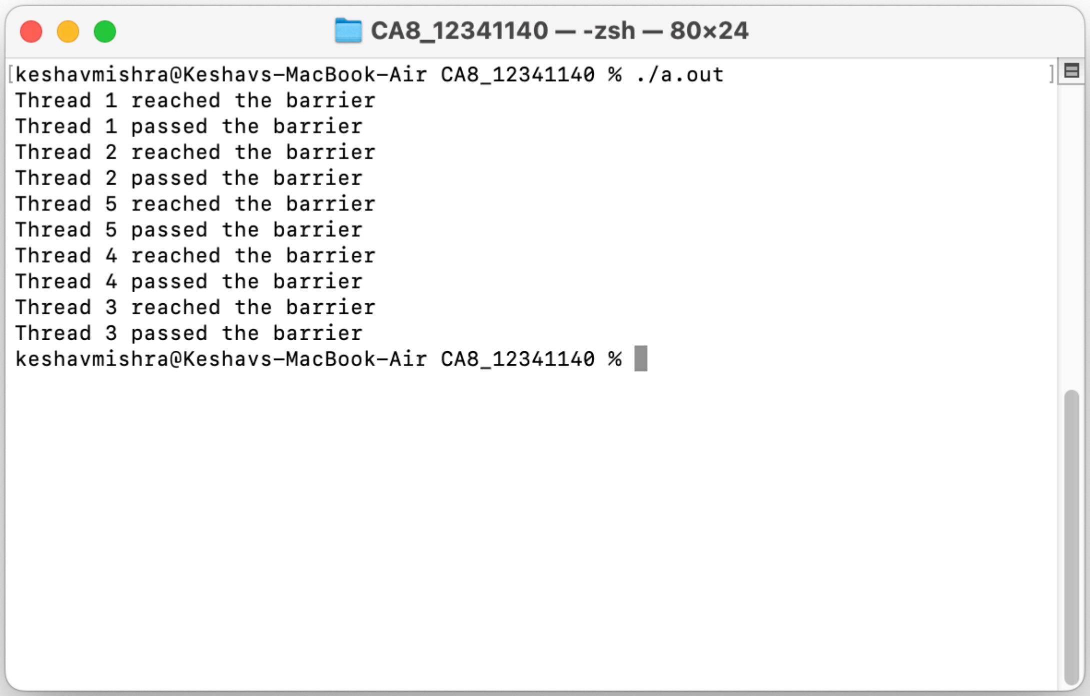
Explanation:
The semaphore is initialized with a value equal to the maximum number of threads allowed in the critical section. Each thread performs sem_wait before entering, which decreases the semaphore count, and sem_post after leaving, which increases it. When the semaphore count reaches zero, any new thread trying to enter must wait until another thread exits, ensuring that no more than the allowed number of threads are inside the critical section at the same time.

Q5 Part 1:

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
```

```
#define N 5

int count = 0;
pthread_mutex_t mutex;
sem_t barrier;

void* thread_func(void* arg)
{
    int id = *(int*)arg;
    printf("Thread %d reached the barrier\n", id);

    pthread_mutex_lock(&mutex);
    count++;
    if (count == N)
    {
        sem_post(&barrier); // Only one thread is released
    }
    pthread_mutex_unlock(&mutex);

    sem_wait(&barrier); // Deadlock for remaining threads
    printf("Thread %d passed the barrier\n", id);
    return NULL;
}

int main() {
    pthread_t threads[N];
    int ids[N];
    pthread_mutex_init(&mutex, NULL);
    sem_init(&barrier, 0, 0);

    for (int i = 0; i < N; i++) {
        ids[i] = i + 1;
        pthread_create(&threads[i], NULL, thread_func, &ids[i]);
    }
    for (int i = 0; i < N; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&mutex);
    sem_destroy(&barrier);
    return 0;
}
```



```
keshavmishra@Keshavs-MacBook-Air CA8_12341140 % ./a.out
Thread 1 reached the barrier
Thread 1 passed the barrier
Thread 2 reached the barrier
Thread 2 passed the barrier
Thread 5 reached the barrier
Thread 5 passed the barrier
Thread 4 reached the barrier
Thread 4 passed the barrier
Thread 3 reached the barrier
Thread 3 passed the barrier
keshavmishra@Keshavs-MacBook-Air CA8_12341140 %
```

Screenshot of output of Q5_12341140_part1.c

Explanation:

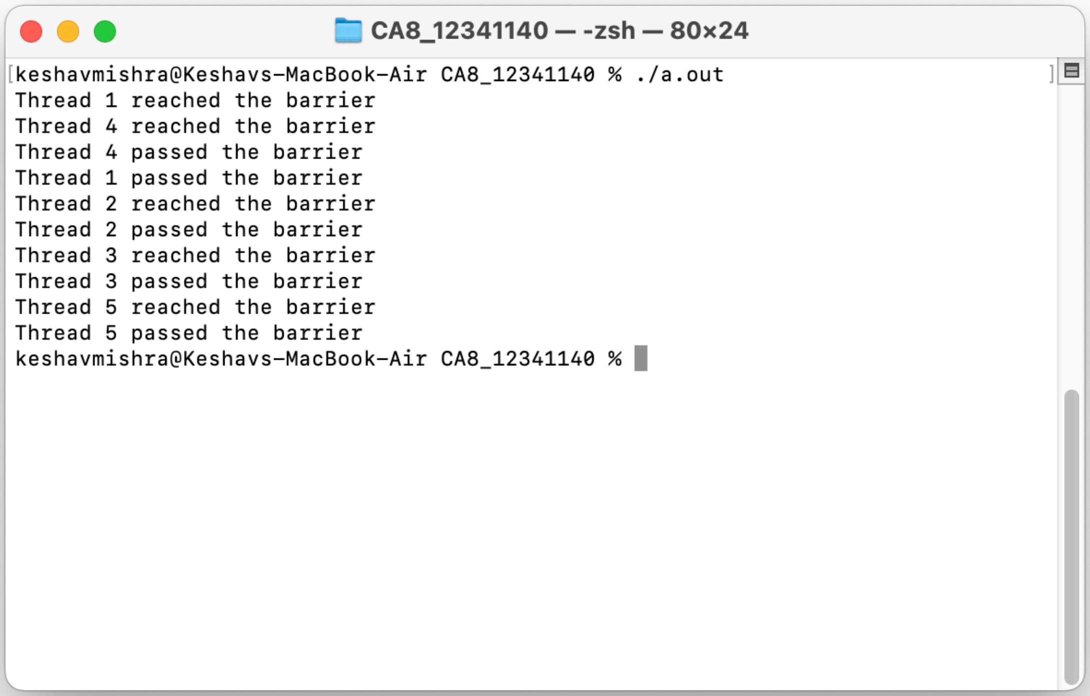
Only one thread signals the semaphore when the last thread arrives (count == N). All threads call sem_wait, but only one is released; the remaining threads are blocked forever, causing a deadlock. This is incorrect barrier implementation.

Q5 Part 2:

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define N 5

int count = 0;
pthread_mutex_t mutex;
sem_t turnstile1, turnstile2;

void* thread_func(void* arg)
{
   int id = *(int*)arg;
   printf("Thread %d reached the barrier\n", id);

   pthread_mutex_lock(&mutex);
   count++;
   if (count == N) {
      for (int i = 0; i < N; i++)
         sem_post(&turnstile1); // release all threads
```

```
    }
    pthread_mutex_unlock(&mutex);

    sem_wait(&turnstile1);

    printf("Thread %d passed the barrier\n", id);

    pthread_mutex_lock(&mutex);
    count--;
    if (count == 0) {
        for (int i = 0; i < N; i++)
            sem_post(&turnstile2); // reset barrier
    }
    pthread_mutex_unlock(&mutex);

    sem_wait(&turnstile2);

    return NULL;
}

int main() {
    pthread_t threads[N];
    int ids[N];
    pthread_mutex_init(&mutex, NULL);
    sem_init(&turnstile1, 0, 0);
    sem_init(&turnstile2, 0, 0);

    for (int i = 0; i < N; i++) {
        ids[i] = i + 1;
        pthread_create(&threads[i], NULL, thread_func, &ids[i]);
    }

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

    pthread_mutex_destroy(&mutex);
    sem_destroy(&turnstile1);
    sem_destroy(&turnstile2);

    return 0;
}
```


CA8_12341140 — -zsh — 80×24

```
keshavmishra@Keshavs-MacBook-Air CA8_12341140 % ./a.out
Thread 1 reached the barrier
Thread 4 reached the barrier
Thread 4 passed the barrier
Thread 1 passed the barrier
Thread 2 reached the barrier
Thread 2 passed the barrier
Thread 3 reached the barrier
Thread 3 passed the barrier
Thread 5 reached the barrier
Thread 5 passed the barrier
keshavmishra@Keshavs-MacBook-Air CA8_12341140 % 
```

Screenshot of output of Q5_1234110_part2.c

Explanation:

This code implements a reusable barrier for N threads using a mutex and two semaphores. Each thread waits at the barrier until all threads arrive, then all proceed together. The second semaphore resets the barrier so it can be used again safely.