a. Identify the data involved in the race condition:

available _resources is the data involved. It's not protected/locked from other threads modifying its value if the current thread is switched out.

b. Identify the location (or locations) in the code where the race condition occurs.

In the decrease_count function, after the if/else check, if the thread is put on pause, other thread could come in and modify that value to be greater/smaller than 5 (max allowed). Or before the current thread can reduce the number of resources available, other thread could use the not updated available _resources value to make its decision about whether there are resources available.

c. Modify decrease_count() function using semaphore or mutex lock:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int decrease_count(int count){

    pthread_mutex_lock(&mutex); //before the CS, new

    if (available _resources <= count)

            return -1;

    else {

            available_resources -= count;

            return 0;

    }

    pthread_mutex_unlock(&mutex); //after the CS

}
```

// Note: I didn't include a busy wait for the thread. It is also assumed that functions like pthread_mutex_init(&mutex, NULL) and pthread_mutex_destroy(&mutex) are called in the main().

// pthread_mutex_lock will throw an error, so having a try catch around the decrease_count() function call seem wise for the actual implementation.

// Technically it feels like a lock should also be implemented in increase_count() function when the shared resources is modified. However, the problem implies that only decrease_count() function shall be modified. Just putting a note here about it.

// no other threads can access the available_resources variable while it's locked. Thus solving the problem of threads accessing/modifying its value before other threads finishes modifying the value.