Part 1: Race Conditions

1. **Increment Race Condition (No Mutex):**
   * **What can go wrong:**
     + T1 calls the code and reads x as 12.
     + T1 is interrupted before incrementing x.
     + T2 calls the code and also reads x as 12.
     + T2 increments x.
     + T1 resumes and also increments x.
     + Both threads have now incremented x independently, leading to a final value less than 13.
   * **Solution:**
     + Wrap the critical section in a mutex.

1| lock(mutex)

2│ if x == 12:

3│ x++

4| unlock(mutex)

1. **Increment Race Condition (With Mutex):**
   * **What can go wrong:**
     + T1 calls the code and acquires the mutex.
     + T1 checks x and finds it as 12.
     + T2 is scheduled and also acquires the mutex.
     + T2 checks x and finds it as 12.
     + T1 increments x.
     + T2 increments x.
     + Both threads have now incremented x independently, leading to a final value less than 13.
   * **Solution:**
     + Move the mutex to include the entire critical section.
     + See Problem 1 Solution (above)
2. **Hash Update Race Condition:**
   * **What can go wrong:**
     + T1 checks if y is not in the hash and finds it's true.
     + T2 is scheduled and also checks if y is not in the hash, finding it true.
     + T1 and T2 simultaneously try to update the hash, potentially leading to lost updates or incorrect increments due to lack of synchronization.
   * **Solution:**
     + Wrap the hash manipulation in a mutex to ensure only one thread can update it at a time.

1| lock(mutex)

2│ if y not in hash:

3│ hash[y] = 12

4| else:

5| hash[y]++

6| unlock(mutex)

1. **Compound Assignment Race Condition:**
   * **What can go wrong:**
     + T1 reads the value of x.
     + T2 reads the value of x.
     + T1 adds 12 to its local copy of x.
     + T2 adds 12 to its local copy of x.
     + Both threads write their modified values of x back, potentially overwriting each other's changes.
   * **Solution:**
     + Wrap the compound assignment in a mutex to ensure atomicity.

1│ lock(m1)

2│ x += 12

3| unlock(m1)

1. **Semaphore Race Condition:**
   * **What can go wrong:**
     + Multiple threads concurrently execute semaphore\_signal() and semaphore\_wait().
     + Multiple threads could simultaneously read, modify, and update the semaphore x, leading to race conditions and violating the intended behavior.
   * **Solution:** 
     + Wrap semaphore operations in a mutex to avoid race conditions when modifying conditional value.

1│ semaphore\_init(value):

2| lock(m1)

3| x = value

4│ unlock(m1)

5|

6│ semaphore\_signal():

7| lock(m1)

8│ x++

9| unlock(m1)

10|

11│ semaphore\_wait():

12│ lock(m1)

13| while x == 0:

14│ do nothing # spinlock

15│ x—

16| unlock(m1)

Part 2: Deadlocks

1. **Out of Order Deadlock:**
   * **How deadlock can occur:**
     + T1 calls function1() and acquires m1.
     + T2 calls function2() and acquires m2.
     + T1 attempts to acquire m2 but blocks since T2 holds it.
     + T2 attempts to acquire m1 but blocks since T1 holds it.
     + Both threads are now deadlocked.
   * **Solution:**
     + Ensure that both functions acquire the locks in the same order to prevent a deadlock

function1():

lock(m2)

lock(m1)

unlock(m1)

unlock(m2)

function2():

lock(m2)

lock(m1)

unlock(m1)

unlock(m2)

1. **Twisting Little Passages, All Different... Deadlock:**
   * **How deadlock can occur:**
     + T1 calls function1(m1, m2) and acquires m1.
     + T2 calls function1(m2, m1) and acquires m2.
     + T1 attempts to acquire m2 but blocks since T2 holds it.
     + T2 attempts to acquire m1 but blocks since T1 holds it.
     + Both threads are now deadlocked.
   * **Solution:**
     + Implement a protocol to ensure exact ordering like using alphabetical order.

function1(m1, m2):

if m1 < m2:

lock(m1)

lock(m2)

unlock(m2)

unlock(m1)

else:

lock(m2)

lock(m1)

unlock(m1)

unlock(m2)