Daniel Lounsbury
CS444
4/28/2024
Brian Hall
Project 4: Races and Deadlocks

Part 1: Race Conditions

For each of these, write a detailed description of how two or more threads could cause unexpected behavior (reference line numbers) **and** what could be done to fix it.

There might be more than one race in an example!

1. Expected result if 3 threads run it and `x` was initialized to `12`: `13`. How could it be different?

```
2.  1| if x == 12:
3.  2|     x++
```

Description: if 3 threads enter the if statement at the same time, x could become 15, when the desired value is 13. This is an example of Atomicity Violation, as the assumption is that the if statement conditional and its body are atomic. This could be fixed placing a lock around the entire if statement.

Solution:

```
1. Lock(m1)
2. 1| if x == 12
3. 2|     x++
4. Unlock(m1)
```

Logical maximum value:

D = desired/predicted value
I = initial value
Inc = increment value

X = thread count

v-max = max value

v-max = i+x or d-inc + x

Daniel Lounsbury
CS444
4/28/2024
Brian Hall
Project 4: Races and Deadlocks

2. Expected result if 3 threads run it and `x` was initialized to `12`: `13`. How could it be different?

```
3.  1| if x == 12:
4.  2|     lock(m1)
5.  3|     x++
6.  4|     unlock(m2)
```

Description: This barely differs from example one. The placement of the locks only ensures that the value will either be exactly 15. By failing to recognize the full critical section (lines 1-4), the threads may still all enter the body of the if statement, causing an unexpected result. Once again, this could be fixed placing a lock around the entire if statement, and removing the lock within the body of the if statement.

Solution:

```
2. lock(m1)
3.  1| if x == 12:
4.  2|
5.  3|     x++
6.  4|
7. unlock(m1)
```

3. In this problem, assume the hash itself is already threadsafe.

   Expected result if 3 threads run it and `y` was not in the hash to start: `14`. How could it be different?

```
1| if y not in hash:
2|     hash[y] = 12
3| else
```

Daniel Lounsbury
CS444
4/28/2024
Brian Hall
Project 4: Races and Deadlocks

```
 4|     hash[y]++
```

Description: As with scenario 1 and 2, assuming that 2 of the threads will wait outside the if statement while the first thread creates a hash entry for y is problematic, as there is no guarantee that this will happen. X could end up being 12, 13, or 14, depending on how the threads execute the code. To prevent this, a lock can be placed around the if else block, as it contains read and write references to memory, making it a critical section in this case.

Solution:

```
lock(m1)
1| if y not in hash:
 2|     hash[y] = 12
 3| else
 4|     hash[y]++
Unlock(m1)
```

5. Expected result if 3 threads run it and `x` was initialized to `0`: `36`. How could it be different?

   Hint: what is `+=` shorthand for?

```
x += 12
```

x+=12 is shorthand for x = x + 12, and it is not a truly atomic operation. It must evaluate x and then the addition between x's value and 12. While this is going on, another thread could be doing the exact same thing, causing the result to only be 24, or one less incrementation cycle than expected. By using either locks around the critical operation/section or by using atomic variables, this problem can be quite easily solved.

Solution:

```
Atomic int x += 12
Or
```

Daniel Lounsbury
CS444
4/28/2024
Brian Hall
Project 4: Races and Deadlocks

```
Lock(m1)

x += 12

Unlock(m1)
```

6. This is an implementation of a semaphore meant to be used by other code.

   `semaphore_init()` is documented as being not-threadsafe, so you don't have to worry about any races in that particular function.

   Assuming just one thread called `semaphore_init(1)`, then no matter how many threads call the signal and wait functions, `x` should never fall below `0`. How could that be violated?

   Hint: this one is a bit tricky. What other tools in addition to mutexes have we learned about?

```
1  semaphore_init(value):
2      x = value
3
4  semaphore_signal():
5      x++
6
7  semaphore_wait():
8      while x == 0:
9          do nothing  # spinlock
10
11     x--
```

Because Semaphore wait merely decrements x, if two threads enter semaphore wait at the exact same time, then x would be decremented twice in what looks like the same call to the semaphore, making x<0. This could be fixed by using compare and swap to ensure that a value has not been changed by another thread to ensure that they are synchronized, and only change the semaphore x if they are synched. Additionally, this could be fixed by locking the critical section in signal and wait.

Daniel Lounsbury
CS444
4/28/2024
Brian Hall
Project 4: Races and Deadlocks

Solution:

```
1| semaphore_init(value):
 2|     x = value
 3|
 4| semaphore_signal():
      Lock(m1)
 5|     x++
 6|    unlock(m1)
 7| semaphore_wait():
      Lock(m1)
 8|     while x == 0:
 9|         do nothing  # spinlock
10|
11|     x--
      Unlock(m1)
```

Part 2: Deadlocks

Multiple threads are running through the following code examples. For each, describe in detail (reference line numbers) how deadlock can occur **and** what can be done to fix it.

1. "Out of Order":

   Find a solution without removing the annoying code on lines 12-13, and without adding any more locks or unlocks.

Daniel Lounsbury
CS444
4/28/2024
Brian Hall
Project 4: Races and Deadlocks

```
 1| function1():
 2|     lock(m1)
 3|     lock(m2)
 4|
 5|     unlock(m2)
 6|     unlock(m1)
 7|
 8| function2():
 9|     lock(m1)
10|     lock(m2)
11|
12|     unlock(m1)
13|     lock(m1)
14|
15|     unlock(m2)
16|     unlock(m1)
```

Description: Function 2 can deadlock, as the locks from line 9 to line 16 are not acquired and released in the appropriate order. To fix this issue, I would simply move line 15 to line 11, as locks should be unlocked in the opposite order as they were locked in to avoid deadlock.

Solutions:

```
1| function1():
 2|     lock(m1)
 3|     lock(m2)
 4|
 5|     unlock(m2)
 6|     unlock(m1)
 7|
```

Daniel Lounsbury
CS444
4/28/2024
Brian Hall
Project 4: Races and Deadlocks

```
 8| function2():
 9|     lock(m1)
10|     lock(m2)
11|     unlock(m2)
12|     unlock(m1)
13|     lock(m1)
14|
16|     unlock(m1)
```

> "Twisting little passages, all different..."

In the following, assume you *cannot* make the mutexes `m1` or `m2` global or require the callers to pass them in any exact order.

```
 1| function1(m1, m2):  # Mutexes are passed in by caller
 2|     lock(m1)
 3|     lock(m2)
 4|
 5|     unlock(m2)
 6|     unlock(m1)
```

If function1 is called twice with the mutexes in the opposite order, ((m1,m2), (m2,m1)), the program would deadlock. To fix this problem, you could determine which lock is actually which my hashing the mutexes and then sorting them out by comparing the hashes, and resetting the variables into the proper order. (had to research this one a bit)

Solution:

```
1| function1(m1, m2):  # Mutexes are passed in by caller
If hash(m1) < hash(m2):
      mut1,mut2 = m1,m2
```

Daniel Lounsbury
CS444
4/28/2024
Brian Hall
Project 4: Races and Deadlocks

```
else:
      mut1,mut2 = m2,m1
 2|    lock(mut1)
 3|    lock(mut2)
 4|
 5|    unlock(mut2)
 6|    unlock(mut1)
```