SOLUTIONS

Suppose all atomic variables have been initialized to 0 before the threads have been created.

1. Thread 1:

```
int a = x.load(std::memory_order_acquire);
y.store(1, std::memory_order_release);
```

Thread 2:

```
int b = y.load(std::memory_order_acquire);
x.store(1, std::memory_order_release);
```

   (a) Which of the following are possible final values for the pair (`a`, `b`)?
   Options: (0, 0) / (0, 1) / (1, 0) / (1, 1)

> **Solution**: (0, 0), (0, 1) and (1, 0).
>
> - The modification order of `x` is $0 \to 1$ and the modification order of `y` is $0 \to 1$.
> - Suppose `a` is 1. Then, the `x.store()` call *synchronizes-with* ($\Rightarrow$ *happens-before*) the `x.load()` call. Hence, the `y.load()` call *happens-before* the `y.store()` call and by read-write coherence, `y.load()` must not read `1`. Therefore, `b` must be 0.
> - Similarly, when `b` is 1, `a` must be 0.
> - The output (0, 0) is possible: Both threads did not see 1 in the modification order when loading.

   (b) What if we change the `x.load()` operation to `std::memory_order_relaxed`?

> **Solution**: (0, 0), (0, 1) and (1, 0).
>
> - Using the same reasoning as (a), if `b` is 1, we can infer that `a` must be 0. This only requires the `y` stores and loads to form a release-acquire pair (`x.load()` being relaxed is fine).

   (c) What if we change both `x.load()` and `y.load()` to `std::memory_order_relaxed`?

> **Solution**: (0, 0), (0, 1), (1, 0) and (1, 1).
>
> - There are no more release-acquire pairs, so there are no more *synchronizes-with* relationships between the threads.
> - The execution (1, 1) is possible – Thread 1 sees `x = 1` and `y = 0` in the modification order when executing the two lines, and thread 2 sees `x = 0` and `y = 1` in the modification order when executing the two lines. Since each variable has an independent modification/coherence order, this is perfectly allowed by the C++ standard.

2. Thread 1:

```
x.store(1, std::memory_order_relaxed);
y.store(1, std::memory_order_release);
x.store(2, std::memory_order_relaxed);
```

Thread 2:

```
x.store(3, std::memory_order_release);
```

Thread 3:

```
while (y.load(std::memory_order_acquire) != 1);
cout << x.load(std::memory_order_acquire);
cout << x.load(std::memory_order_acquire);
```

(a) True or False: The two `cout`-s never print 0.

> **Solution**: True.
>
> - The acquire-load of `y` successfully reads the release-store to `y`. Therefore, `y.store()` *synchronizes-with* (⇒ *happens-before*) `y.load()`. Hence, the `x.store(1)` call *happens-before* the two `x.load()` calls.
> - The modification order of `x` could be $0 \to 1 \to 2 \to 3$, $0 \to 1 \to 3 \to 2$ or $0 \to 3 \to 1 \to 2$. In all cases, by write-read coherence, `x.load()` must see the value 1 or a later value in the modification order, so `x.load()` should never see 0.

(b) Which of the following are possible outputs of the two `cout`-s?
Options: 11 / 12 / 13 / 21 / 22 / 23 / 31 / 32 / 33

> **Solution**: 11, 12, 13, 22, 23, 32 and 33.
>
> - For modification order $0 \to 1 \to 2 \to 3$: Possible outputs are 11, 12, 13, 22, 23, 33.
> - For modification order $0 \to 1 \to 3 \to 2$: Possible outputs are 11, 13, 12, 33, 32, 22.
> - For modification order $0 \to 3 \to 1 \to 2$: Possible outputs are 11, 12, 22.

3. Thread 1:

```
x.store(1, std::memory_order_relaxed);
z.store(1, std::memory_order_release);
```

Thread 2:

```
z.fetch_add(1, std::memory_order_relaxed);
```

Threads 3:

```
while (z.load(std::memory_order_acquire) != 2);
cout << x.load(std::memory_order_relaxed);
```

(a) True or False: The program is guaranteed to terminate.

> **Solution**: False.
>
> - Consider the modification order for `z`: $0 \to 1 \to 1$. This is possible when `z.fetch_add(1)` writes first, followed by `z.store(1)`. `z.load()` can never read a 2 and hence the program runs into an infinite loop.

(b) True or False: There is a data race.

> **Solution**: False.
>
> - All operations are atomic operations, so there is no data race.

(c) True or False: The `cout` must print 1.

> **Solution**: True.
>
> - The while loop terminates only when `z.load()` reads 2. This implies the modification of `z` is $0 \to 1 \to 2$.
> - `z.fetch_add(1)` is a read-modify-write (RMW) operation, so it falls within the **release sequence** of `z.store(1)`. Since `z.load()` reads from `z.fetch_add(1)`, `z.store(1)` *synchronizes-with* `z.load()` due to release sequences.
> - Therefore, `x.store(1)` *happens-before* `x.load()`, and `x.load()` must read 1.

(d) True or False: Consider changing thread 2 to `z.load(std::memory_order_relaxed); z.store(2, std::memory_order_release)`. When the `z.load()` returns 1, the `cout` might print 0.

> **Solution**: True.
>
> - The modification order of `z` must be $0 \to 1 \to 2$.
> - In Thread 3, since `z.load()` reads from `z.store(2)`, `z.store(2)` *synchronizes-with* ($\Rightarrow$ *happens-before*) `z.load()`. However, if `z.load()` never reads from `z.store(1)` (i.e. only reading 0 or 2), there is no *synchronizes-with* relationship between thread 1 and thread 3.
> - Therefore, `x.store(1)` might not *happen-before* `x.load()` and the `cout` could possibly print 0.

4. Thread 1:

```
x.store(1, std::memory_order_relaxed);
y.store(1, std::memory_order_release);
```

Thread 2:

```
x.store(2, std::memory_order_relaxed);
y.store(2, std::memory_order_release);
```

Thread 3 & 4:

```
while (y.load(std::memory_order_acquire) == 1);
cout << x.load(std::memory_order_relaxed);
```

(a) True or False: Thread 3 is guaranteed to terminate.

> **Solution**: False.
>
> - Consider this modification order for `y`: $0 \to 2 \to 1$.
> - If thread 3 sees the value 1 for `y.load()` right away, by read-read coherence, it will never return to an older entry in the modification order. Therefore, the while loop gets stuck forever.

(b) Which of the following are possible outputs of the `cout`?
Options: 0 / 1 / 2

> **Solution**: 0, 1 and 2.
>
> - 0 is possible – if thread 3 only sees `x = 0` and `y = 0` in the modification order, it will pass the while loop and output 0.
> - 1 is possible – suppose the modification order of `x` is $0 \to 2 \to 1$. The `y.load()` reads in a 2, and hence `y.store(2)` *synchronizes-with* `y.load()`. Therefore, `x.store(2)` *happens-before* `x.load()` and by write-read coherence, `x.load()` must read 2 or a later entry in the modification order. It is legal for Thread 3 to read 1.
> - 2 is possible – using the same execution above, it is also legal for Thread 3 to read 2.

(c) True or False: It is possible that thread 3 prints 1 but thread 4 prints 2.

> **Solution**: True
>
> - Consider the same execution as above. It is legal for Thread 3 to read 1 while Thread 4 reads 2.

(d) True or False: If the while loop body executes at least once for both threads 3 and 4, it is possible that theead 3 prints 1 but thread 4 prints 2.

> **Solution**: False
>
> - For the while loop to read `y = 1` at least once and eventually terminating, the modification order of `y` must be $0 \to 1 \to 2$.
> - `y.load()` has read a value of 1, so `y.store(1)` *synchronizes-with* `y.load()`. `y.load()` has also read a value of 2 (to exit from the loop), so `y.store(2)` *synchronizes-with* `y.load()`. This implies that both `x.store(1)` and `x.store(2)` *happen-before* `x.load()`. The `x.load()` must see the effects of both stores in the modification order.
> - If the modification order of `x` is $0 \to 2 \to 1$ – both threads must print 1. If the modification order of `x` is $0 \to 1 \to 2$ – both threads must print 2.
> - Since both threads agree on the same modification order of `x`, it is impossible for both threads to output differently.

Thread 1:

```
x.store(1, std::memory_order_relaxed);
y.store(1, std::memory_order_release);
```

Thread 2:

```
x.store(2, std::memory_order_relaxed);
y.store(2, std::memory_order_release);
```

Thread 3 & 4:

```
while (y.load(std::memory_order_acquire) == 0);
cout << x.load(std::memory_order_relaxed);
```

(a) True or False: Thread 3 is guaranteed to terminate.

Solution: True.

- The modification order of `y` is $0 \to 1 \to 2$ or $0 \to 2 \to 1$.
- Thread 3 will eventually read 1 or 2, thus exiting from the while loop.

(b) Which of the following are possible outputs of the `cout`?
Options: 0 / 1 / 2

Solution: 1 and 2.

- The `y.load()` reads either 1 or 2 for the while loop to terminate.
- If `y.load()` reads 1, then `x.store(1)` *happens-before* `x.load()`. Therefore, `x.load()` must read either 1 or 2, but not 0.
- If `y.load()` reads 2, then `x.store(2)` *happens-before* `x.load()`. Therefore, `x.load()` must read either 1 or 2, but not 0.

(c) True or False: It is possible that thread 3 prints 1 but thread 4 prints 2.

Solution: True.

- Suppose the modification order of `x` is $0 \to 1 \to 2$, and the modification order of `y` is $0 \to 1 \to 2$.
- Thread 3 can load `y = 1` and `x = 1`. Thread 4 can load `y = 2` and `x = 2`.

(d) True or False: If the while loop body executes at least once for both threads 3 and 4, it is possible that theead 3 prints 1 but thread 4 prints 2.

Solution: True.

- Same as (c), reading some 0's at the start doesn't change the reasoning.