1. Thread 1:

```
x.store(true, stdmo::seq_cst);      // (a)
```

Thread 2:

```
y.store(true, stdmo::seq_cst);      // (b)
```

Thread 3:

```
while (!x.load(stdmo::seq_cst)) {}   // (p)
if (y.load(stdmo::seq_cst)           // (q)
  z.fetch_add(1, stdmo::seq_cst);    // (r)
```

Thread 4:

```
while (!y.load(stdmo::seq_cst)) {}   // (t)
if (x.load(stdmo::seq_cst)           // (u)
  z.fetch_add(1, stdmo::seq_cst);    // (v)
```

   (a) Can the final value of `z` be 0?

   Answer: No. Suppose the condition in *(q)* returns false. Then, since *(t)* reads `y == true`, it must appear strictly after *(q)* in the timeline. Also, *(p)* must be strictly before *(q)* in the timeline due to program order, where we read `x == true`. Hence, at *(u)*, `x` must be true so the value of `z` gets added to 1.

   (b) What if we use acquire/release semantics instead?

   Answer: Yes, `z` can be 0. Acquire/release only introduces synchronizes-with relationships between *(a)* and *(p)* / *(u)*, as well as between *(b)* and *(t)* / *(q)*. Since the coherence orders of different variables are independent, there is no happens-before relationship that requires us to read `y == true` or `z == true`.

   (c) How about relaxed memory ordering?

   Answer: This is even more relaxed, so yes, `z` can be 0.

   (d) What if `z` is a non-atomic variable?

   Answer: There is a data race between the two writes to `z`.

2. Thread 1:

```
x.fetch_add(1, stdmo::relaxed); // a
y.store(2, stdmo::release);     // b
```

Thread 2:

```
x.fetch_add(1, stdmo::relaxed); // p
y.store(2, stdmo::release)?     // q
```

Thread 3:

```
while (y.load(stdmo::acquire) != 2) {} //x
x.load(stdmo::relaxed);               //y
```

Is there a synchronizes-with relationship for both (b)-(x) and (q)-(x)?

Answer: No, it's just one of them – and we don't know which one. But then in this case, `x.load` should read either 1 or 2, but never 0.

3. Thread 1:

```
x.store(1, stdmo::seq_cst);  // (a)
y.store(1, stdmo::release);  // (b)
```

Thread 2:

```
r1 = y.fetch_add(1, stdmo::seq_cst); // (p)
r2 = y.load(stdmo::relaxed);         // (q)
```

Thread 3:

```
y.store(3, stdmo::seq_cst);   // (x)
r3 = x.load(stdmo::seq_cst);  // (y)
```

Is it possible that `r1 = 1`, `r2 = 3` and `r3 = 0`?

Answer: Yes.

Let's first infer the modification order of `y` – note that *(q)* loads 3 *after* the fetch-add in *(p)*, so the 3 must have come from the store in *(x)*. Therefore, *(p)* is before *(x)* in the modification order of `y`. Also, since `r1` reads 1, it must have come from the store in *(b)*, so *(b)* synchronizes-with *(p)*. The modification order of `y` must be *(b)* → *(p)* → *(x)*.

Next, note that there are 4 sequentially consistent operations here – *(a)*, *(p)*, *(x)* and *(y)*. For sequentially consistent operations, they must be laid on a single global timeline agreed by all threads. Moreover, this timeline must respect the strongly happens-before relationship – if an operation k strongly happens-before another operation ℓ, then k must be placed before ℓ on this timeline.
Since *(y)* read a value of 0, we should place *(y)* after *(a)* in this timeline. And since the modification order of `y` is *(b)* → *(p)* → *(x)*, the only possible total order is *(p)* → *(x)* → *(y)* → *(a)*.

This ordering sounds counterintuitive – and the reason is that this ordering places *(a)* after *(p)*. Let's determine if *(a)* strongly happens-before *(p)* here. The answer is **no** – while *(a)* is sequenced-before *(b)* and *(b)* synchronizes-with *(p)*, and this implies that *(a)* simply happens-before *(p)*, this does not fit the definitions of strongly happens-before. Either we need both *(b)* and *(p)* to be sequentially consistent operations (as in definition B), or we need one operation after *(p)* to propagate the effects (as in definition C). So it's okay to place *(a)* after *(p)*.

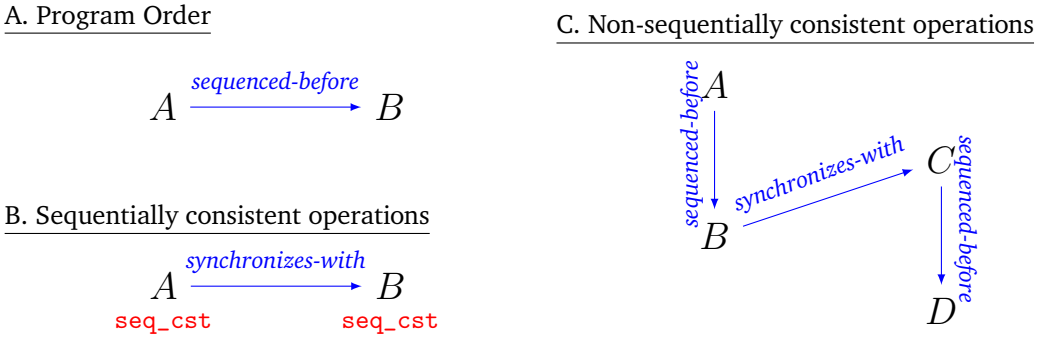

Figure 1: The three definitions of strongly happens before (+ transitivity).