Shared Variables and Interference-Freedom

CS 536: Science of Programming, Fall 2019

A. Why

- Parallel programs can coordinate their work using shared variables, but it's important for threads to not interfere (to not invalidate conditions that the other thread relies on).

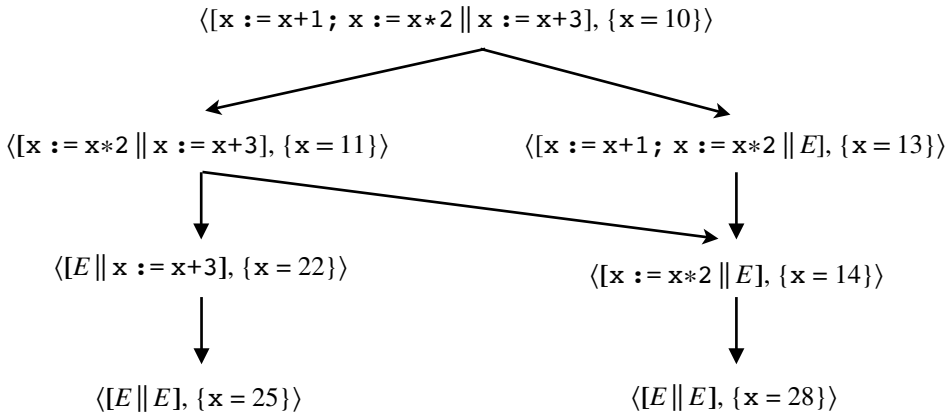
B. Objectives

At the end of this lecture you should know how to

- Check for interference between the correctness proofs of the sequential threads of a shared memory parallel program.

C. Parallel Programs with Shared Variables

- Disjoint parallel programs are nice because no thread interferes with another's work. They're bad because threads can't communicate or combine efforts.
- Let's start looking at programs that aren't disjoint parallel and allow threads to share variables. We've seen examples, but here's another one.
- **Example 1**: Below is the evaluation graph for $\langle$[`x := x+1; x := x*2` $\|$ `x := x+3`], {`x` = 10}$\rangle$. Since 11 + 1 + 3 = 11 + 3 + 1, two of the intermediate states are equal.



- The problem with shared variables is that threads that work correctly individually might stop working when you combine them in parallel.
  - Depending on the execution path it takes, some piece of code in one thread may invalidate a condition needed by a second thread.
- **Race condition**: A situation where correctness of a parallel program depends on the relative speeds of execution of the threads. (If different relative speeds produce different results but the results are correct, then we don't have a race condition.) To avoid race conditions,
  - We control where interleaving can occur by using "atomic regions".

- We ensure that when interleaving occurs, it causes no harm (threads are "interference-free").

D. Atomic Regions

- Interleaving problems occur in real life (not just theory :-).
  - Consider `x := y; x := x+y` — maybe we don't want to allow other threads to change `y` between two additions. By comparison, with `x := y+y`. No other thread can run while `x := y+y` is being done, so we're guaranteed that both `y`'s have the same values.
- Keeping interleaving from occurring when you don't want it to is called the "critical section" problem. Solving this problem using only software is hard; instead, people added new hardware instructions ("test and set") to solve it.
- More generally, people control the amount of possible interleaving of execution by declaring pieces of code to be **atomic**: Their execution cannot be interleaved with anything else.
- **Definition (Syntax)**: If S is a statement, then $< S >$ is an **atomic region** statement with body S.
- **Operational semantics of atomic regions:** Evaluation of $< S >$ behaves like a single step:
  - If $\langle S, \sigma\rangle \rightarrow^* \langle E, \tau\rangle$ then $\langle < S >, \sigma\rangle \rightarrow \langle E, \tau\rangle$.
  - Note $< v := e >$ and $v := e$ behave exactly the same.
- Our operational semantics definition automatically makes assignment statements, **`skip`**, and **`if`**/**`while`** tests atomic.
- A **normal assignment** is one not inside an atomic area. We worry about interleaving of normal assignments; we don't worry about the non-normal ones.
- **Example 2**: `x := y+y` and `< x := y; x := x+y >` are equivalent because we can't change the value of `y` between the `x := y` and `x := x+y` assignments. There are only two ways to evaluate [`< x := y; x := x+y >` $\|$ `y := 10`], and neither of them involve changing `y` between the two assignments to `x`.
  - $\langle$[`<x := y; x := x+y>` $\|$ `y := 10`], {`y = 6`}$\rangle$
    $\rightarrow \langle$[$E \,\|$ `y := 10`], {`x = 12, y = 6`}$\rangle$
    $\rightarrow \langle$[$E \,\|\, E$], {`x = 12, y = 10`}$\rangle$
  - $\langle$[`<x := y; x := x+y>` $\|$ `y := 10`], {`y = 6`}$\rangle$
    $\rightarrow \langle$[`<x := y; x := x+y>` $\| E$], {`y = 10`}$\rangle$
    $\rightarrow \langle$[$E \,\|\, E$], {`x = 20, y = 10`}$\rangle$
- Using atomic regions gives us control over the size of pieces of code that can be interleaved ("granularity of code interleaving"). The more/larger atomic regions, the less interleaving/less parallelism we have.

E. Interference Between Threads

- **What is interference?** Interference occurs when one thread invalidates a condition needed by another thread. In the previous lecture we had an example where `x := 1` interfered with {`x = 0`} `y := 0` {`x = y`} because we didn't have disjoint conditions. If the triple had been {`x ≥ 0`} `y := 0` {`x ≥ y = 0`}, then the conditions would still not be disjoint, but `x := 1` would not have caused them to become invalid.
- **Example 3**: Say we have three threads that contain {`x > 0`} S_1 {`x ≥ 0`},
  {`x > 0`} S_2 {…}, and {…} S_3 {`x > 0`} respectively.

- S_1 can interfere with the precondition of S_2:
  - If S_1 and S_2 are both ready to execute and S_1 executes first, it might invalidate the precondition that S_2 expects.
- S_1 can interfere with the postcondition of S_3:
  - If thread 3 has completed and then thread 1 executes, it might invalidate the postcondition of S_3.
- To avoid these problems, we need S_1 to behave like $\{\texttt{x > 0}\}\ S_1\ \{\texttt{x > 0}\}$.
- Or change the atomicity so that thread 1 doesn't see the `x > 0` condition:
  - I.e., <…; $\{\texttt{x > 0}\}\ S_2\ \{\ldots\}$; …> or <…; $\{\ldots\}\ S_3\ \{\texttt{x > 0}\}$; …>

- **Definition**: The **interference-freedom check** for $\{p\} < S > \{\ldots\}$ versus a predicate q is $\{p \wedge q\} < S > \{q\}$. If this check is valid, then we say that $\{p\} < S > \{\ldots\}$ **does not interfere with** q. (Note we don't care about $\{p \wedge \neg q\} < S > \{\ldots\}$.)
  - **Example 4**: $\{p\}$ `x := x+1` $\{\ldots\}$ does not interfere with `x ≥ 0`.
  - **Example 5**: `{x ≤ −1}` `x := x+1` $\{\ldots\}$ does not interfere with `x ≤ 0`.
  - **Example 6**: `{x ≤ 0}` `x := x+1` $\{\ldots\}$ (possibly) interferes with `x ≤ 0`.
- Unlike the disjoint condition requirement, general interference freedom of $\{p\} < S > \{\ldots\}$ with q doesn't mean that S can't change the values of variables free in q, it means that S is restricted to changes that maintain satisfaction of q.
- The opposite of "does not interfere with" is "possibly interferes with", not "is guaranteed to interfere with".
  - If $\{p\} < S > \{\ldots\}$ fails its interference-freedom check with q means that $\{p \wedge q\} < S > \{q\}$ is not valid: For one or more $\sigma_0 \vDash p \wedge q$, we have $\sigma_0 \nvDash \{p \wedge q\} < S > \{q\}$.
  - But that doesn't tell us much about how S will actually behave at runtime because there's no guarantee that the state we run S in is one of these states.
  - We might execute S and end in a state that satisfies q or in one that doesn't.
- Once we have a notion of interference freedom of an atomic triple versus a predicate, we can build up to interference between larger structures.
- **Notation**: By S^* we mean a proof outline of the program S.
- **Definition**: The atomic statement $\{p_1\} < S_1 > \{\ldots\}$ **does not interfere with** the proof outline $\{p_2\}\ S_2^*\ \{q_2\}$ if it doesn't interfere with the precondition or postcondition of any statement in S_2^* (including p_2 and q_2): For every $\{p_2'\} < S_2' > \{q_2'\}$ that appears in S_2^*, $\{p_1\} < S_1 > \{\ldots\}$ interferes with neither p_2' or q_2'.
- **Definition**: A proof outline $\{p_1\}\ S_1^*\ \{q_1\}$ **does not interfere with** another proof outline $\{p_2\}\ S_2^*\ \{q_2\}$ if every atomic statement $\{p_1'\} < S_1' > \{\ldots\}$ in S_1^* does not interfere with the outline $\{p_2\}\ S_2^*\ \{q_2\}$.
- **Definition**: Two proof outlines $\{p_1\}\ S_1^*\ \{q_1\}$ and $\{p_2\}\ S_2^*\ \{q_2\}$ are **interference-free** if neither interferes with the other: No atomic statement in S_1^* interferes with the conditions of any atomic statement in S_2^* and vice versa.
- **Example 7:** `{x mod 4 = 0}` `x := x+3` $\{\ldots\}$ interferes with `{even(x)}` `x := x+1` `{odd(x)}` because it interferes with `even(x)`: $\nvDash$ `{x mod 4 = 0 ∧ even(x)}` `x := x+3` `{even(x)}`. On the other hand, it doesn't interfere with `odd(x)` because we do have $\vDash$ `{x mod 4 = 0 ∧ odd(x)}` `x := x+3` `{odd(x)}`.

- **Example 8**: Two copies of {even(x)} x := x+1 {odd(x)} interfere with each other. I.e., {even(x)} x := x+1 {odd(x)} and {even(x)} x := x+1 {odd(x)} interfere with each other.
- **Example 9**: {even(x)} x := x+1 {odd(x)} and {x ≥ 0} x := x+2 {x > 1} are interference-free.
  - Precondition of triple 2: {even(x) ∧ x ≥ 0} x := x+1 {x ≥ 0} is valid.
  - Postcondition of triple 2: {even(x) ∧ x > 1} x := x+1 {x > 1} is valid.
  - Precondition of triple 1: {x ≥ 0 ∧ even(x)} x:=x+2 {even(x)} is valid.
  - Postcondition of triple 1: {x ≥ 0 ∧ odd(x)} x := x+2 {odd(x)} is valid.

F. Parallelism with Shared Variables and Interference-Freedom

- **Theorem (Interference-Freedom)**: Let $\{p_1\}\ S_1^*\ \{q_1\}$ and $\{p_2\}\ S_2^*\ \{q_2\}$ be interference-free proof outlines. If both outlines are valid, then their parallel composition $\{p_1 \wedge p_2\}\ [S_1^* \parallel S_2^*]\ \{q_1 \wedge q_2\}$ is also valid.
  - Proof omitted.
- The interference freedom theorem enables the use of a new parallelism rule:

Disjoint Parallelism with Shared Variables Rule

1. $\{p_1\}\ S_1^*\ \{q_1\}$
2. $\{p_2\}\ S_2^*\ \{q_2\}$

…

n. $\{p_n\}\ S_n^*\ \{q_n\}$

n+1. $\{p_1 \wedge p_2 \wedge \ldots \wedge p_n\}$ D.P. Shared Vars, 1, 2, …, n
$[S_1^* \parallel \ldots \parallel S_n^*]$
$\{q_1 \wedge q_2 \wedge \ldots \wedge q_n\}$

where the $\{p_i\}\ S_i^*\ \{q_i\}$ are pairwise interference-free.

- One feature of this rule is that it talks about proof outlines, not correctness triples. (Before this, the rules only concerned triples.)
  - The outlines here are necessary because we can't guarantee correctness without knowing that the different threads don't invalidate conditions **inside** the other threads.
  - We can no longer compose correctness triples, we have to compose entire outlines.
- **Example 12**: A proof outline for {x = 0} [x := x+2 ‖ x := 0] {x = 0 ∨ x = 2} using parallelism with shared variables is below:

  {x = 0}
  {x = 0 ∧ T}
  [{x = 0} x := x+2 {x = 0 ∨ x = 2}
  ‖ {T} x := 0 {x = 0 ∨ x = 2}]
  {(x = 0 ∨ x = 2) ∧ (x = 0 ∨ x = 2)}
  {x = 0 ∨ x = 2}

- The side conditions are
  - {x = 0} x := x+2 {…} does not interfere with T or x = 0 ∨ x = 2

- $\{x = 0 \wedge T\}\ x := x+2\ \{T\}$
- $\{x = 0 \wedge (x = 0 \vee x = 2)\}\ x := x+2\ \{x = 0 \vee x = 2\}$

- $\{T\}\ x := 0\ \{\ldots\}$ does not interfere with $x = 0$ or $x = 0 \vee x = 2$
  - $\{T \wedge x = 0\}\ x := 0\ \{x = 0\}$
  - $\{T \wedge (x = 0 \vee x = 2)\}\ x := 0\ \{x = 0 \vee x = 2\}$

- No matter which assignment executes first, when $x := x+2$ runs, it sees $x = 0$ and sets it to 2. When $x := 0$ runs, it sees $x = 0$ or 2 and makes it 0.

G. An Example With Shared and Auxiliary Variables

- Recall the program

  $$\{x = 0\}\ [x := x+2 \parallel x := 0]\ \{x = 0 \vee x = 2\}$$

  which we proved correct using parallelism with shared variables. Sequentially, we had $\{x = 0\}\ x := x+2\ \{x = 0 \vee x = 2\}$ and $\{T\}\ x := 0\ \{x = 0 \vee x = 2\}$, and interference freedom allowed us to compose these threads in parallel.

- We can weaken the precondition $x = 0$ to just true and the program still works, but it's annoyingly difficult to verify. If we try to annotate the program using

  $$\{T \wedge x = x_0\}\ [\{T\}\ x := 0\ \{x = 0\} \parallel \{x = x_0\}\ x := x+2\ \{x = x_0+2\}]\ \{\ldots\}$$

  we find that each thread's assignment to x interferes with one or both conditions of the other thread.

- Just because two proof outlines interfere, that doesn't mean the programs are wrong. It may just be that one or more of the proofs need to be modified in order to prove interference freedom.

- We could try weakening the $x = x_0$ precondition for thread 1:

  $\{T \wedge x = x_0\}$
  $[\ \{T\}\ x := 0\ \{x = 0\}$
  $\parallel \{x = x_0\}\ \{x = 0 \vee x = x_0\}\ x := x+2\ \{x = 2 \vee x = x_0+2\}\ ]$
  $\{x = 0 \wedge (x = 2 \vee x = x_0+2)\}$

- But if thread 2 runs first, it interferes with thread 1's $x = 0$. In addition, when thread 1's runs $x := 0$, it interferes with thread 2's ($x = 2 \vee x = x_0+2$).

- We could make the first thread $\{T\}\ x := 0\ \{x = 0 \vee x = 2\}$, which reflects the possibility that $x := 0$ runs and then $x := x+2$ runs.
  - But thread 2's $x := x+2$ interferes with $x = 0 \vee x = 2$.

- We could make the first thread $\{T\}\ x := 0\ \{x = 0 \vee x = 2 \vee x = 4\}$, but in the general case we shouldn't get $x = 4$, and $x := x+2$ still interferes anyway.

- And so on.

- The problem here is that we don't know which case runs first. To solve it, we can add a new boolean variable `inc` that says whether or not the $x := x+2$ increment has been done.

  $$\{T\}\ \texttt{inc} := \texttt{F};\ [x := 0 \parallel \langle x := x+2;\ \texttt{inc} := \texttt{T}\rangle]\ \{x = 0 \vee x = 2\}$$

- This is an example of an "auxiliary" variable — we haven't seen them yet, but under certain conditions, a program variable can be removed from the code without invaliding correctness.

- The increment of `x` and setting of `inc` are done atomically so we don't have to worry about `x := 0` being done between them. Since `inc` will be removed from the program, there's no actual increase in granularity of atomicity.
- The annotation of the first thread, is key: $\{\texttt{T}\}\ \texttt{x := 0}\ \{\texttt{x} = 0 \vee (\texttt{inc} \wedge \texttt{x} = 2)\}$. The postcondition can't just be $\texttt{x} = 0$, since that's interfered with by thread 2. If thread 2 runs, however, it sees $\texttt{x} = 0$ and sets $\texttt{x} = 2$ and `inc` to true, so we can make that a second disjunct of the postcondition.
- For the second thread, $\{\neg \texttt{inc}\} < \texttt{x := x+2; inc := T} > \{\texttt{inc}\}$ is all we need. The important information about the value of `x` is held in the conditions of thread 1.
  - $\{\texttt{x} = \texttt{x}_0 \wedge \neg \texttt{inc}\} < \texttt{x := x+2; inc := T} > \{\texttt{x} = \texttt{x}_0\texttt{+2} \wedge \texttt{inc}\}$ but also works but isn't necessary.
- If we add $\texttt{x} = \texttt{x}_0$ to the precondition of thread 2, then the postcondition of thread 2 can be $\texttt{inc} \wedge (\texttt{x} = \texttt{x}_0\texttt{+2} \vee \texttt{x} = 0 \vee \texttt{x} = 2)$. But the postcondition of thread 1 says $\texttt{x} = 0 \vee (\texttt{inc} \wedge \texttt{x} = 2)$, so adding $(\texttt{x} = \texttt{x}_0 \vee \texttt{x} = 0 \vee \texttt{x} = 2)$ to thread 2 doesn't give us any new information.
- Here is a full proof annotation for our program.

  $\{\texttt{T}\}\ \texttt{inc := F;}\ \{\texttt{T} \wedge \neg \texttt{inc}\}$
  $[\ \{\texttt{T}\}\ \texttt{x := 0}\ \{\texttt{x} = 0 \vee (\texttt{inc} \wedge \texttt{x} = 2)\}$
  $\|\ \{\neg \texttt{inc}\} < \texttt{x := x+2; inc := T} > \{\texttt{inc}\}\ ]$
  $\{(\texttt{x} = 0 \vee (\texttt{inc} \wedge \texttt{x} = 2)) \wedge \texttt{inc}\}$
  $\{\texttt{x} = 0 \vee \texttt{x} = 2\}$
- Since `inc` is auxiliary for this program, it turns out that we can remove it from the program and get just $\{\texttt{T}\}\ [\texttt{x := 0} \parallel \texttt{x := x+2}]\ \{\texttt{x} = 0 \vee \texttt{x} = 2\}$ as the result.
- A symmetric alternative to using `inc` is to use an auxiliary boolean variable `z` (short for "zeroed") that is true when we've run `x:= 0` in thread 1. Once again, to avoid interference, we need to make `z := T` atomic with `x:= 0` so that thread 2 can't see the moment where $\texttt{x} = 0 \wedge \neg \texttt{z}$.

  $\{\texttt{T}\}\ \texttt{z:= F;}\ \{\neg \texttt{z} \wedge (\texttt{z} \rightarrow \texttt{x} = 0)\}$
  $[\ \{\neg \texttt{z}\} < \texttt{x:= 0; z:= T} > \{\texttt{z}\}$
  $\|\ \{\texttt{z} \rightarrow \texttt{x} = 0\}\ \texttt{x:= x+2}\ \{\texttt{z} \rightarrow \texttt{x} = 0 \vee \texttt{x} = 2\}\ ]$
  $\{\texttt{z} \wedge (\texttt{z} \rightarrow \texttt{x} = 0 \vee \texttt{x} = 2)\}$
  $\{\texttt{x} = 0 \vee \texttt{x} = 2\}$
- Sequential correctness of the threads is easy to verify, as is interference freedom: Thread 1's `x:= 0; z:= T` doesn't interfere with $\texttt{z} \rightarrow \texttt{x} = 0$ or with $\texttt{z} \rightarrow \texttt{x} = 0 \vee \texttt{x} = 2$, and thread 2 doesn't modify `z`.
  - Note the postcondition of thread 1 doesn't need to mention `x` because all the relevant information is contained in the postcondition of thread 2.
- Since `z` only appears in the contexts `z := T` and `z := F`, it is auxiliary for the program, so removing it won't change the computation of `x` and we get what we wanted, $\{\texttt{T}\}\ [\texttt{x := 0} \parallel \texttt{x:= x+2}]\ \{\texttt{x} = 0 \vee \texttt{x} = 2\}$.

Shared Variables & Interference

CS 536: Science of Programming

A. Why

- Parallel programs can coordinate their work using shared variables, but it's important for threads to not interfere (to not invalidate conditions that the other thread relies on).

B. Objectives

At the end of this activity you should be able to

- Check for interference between the correctness proofs of the sequential threads of a shared memory parallel program.

C. Problems

1. Say we have a parallel program that fails one of its interference checks.
   a. What does that tell us about how the program will behave at runtime? (Assume we start it in a state satisfying the precondition.)
   b. Is it impossible to get a proof of correctness for the program?

(Problems 2 – 4): For each of the following sets of threads, (a) list the triples to check for interference freedom, (b) say whether each of the interference triples is true or false, and (c) say whether the threads interfere or not.

2. $\{\texttt{even(x)}\}\ \texttt{x:=x+1}\ \{\texttt{odd(x)}\}$ and $\{\texttt{x} \geq 0\}\ \texttt{x:=x+2}\ \{\texttt{x} \geq 0\}$.
3. $\{\texttt{T}\}\ \texttt{x:=0}\ \{\texttt{x=0}\}$ and $\{\texttt{x=0}\}\ \texttt{x:=x+2}\ \{\texttt{x=2}\}$.
4. $\{\texttt{T}\}\ \texttt{x:=0}\ \{\texttt{x=0}\}$ and $\{\texttt{x=0}\}\ \texttt{x:=x+2}\ \{\texttt{x=0} \vee \texttt{x=2}\}$.

5. List the interference freedom checks for the following two standard proof outlines. Remember, you only need to see if atomic statements cause interference.
   - $\{p_1\}$ **if** B_1 **then** $\{p_2\} < S_1 >$ **else** $\{p_3\}$ **skip fi** $\{p_4\}$
   - $\{q_1\} < T_1 >$**;** $\{$**inv** $q_2\}$ **while** C_1 **do** $\{q_3\} < T_2 >$ **od** $\{q_4\}$

Solution to Activity 22 (Shared Variables and Interference)

1. (Parallel program fails an interference check.)

   a. Failing an interference check only tells us that if we execute the failed triple, interference is not impossible. It doesn't guarantee that interference will occur at runtime; it just says its possible.

   b. It may or may not be possible to prove correctness. It's possible that if the proof outline's conditions are modified, we'd be able to prove interference freedom. It's also possible that proof outline can't be modified to prove interference freedom without modifying the given final postcondition. E.g., in Example 1, if we start with $x = 10$ and insist on showing that $x = 25$ when the program finishes, then no proof of correctness is possible because we might end with $x = 28$.

2. No interference:

   - $\{\texttt{even(x)} \wedge \texttt{x} \geq 0\}$ `x := x+1` $\{\texttt{x} \geq 0\}$ // the precondition of `x := x+2`
   - $\{\texttt{even(x)} \wedge \texttt{x} \geq 0\}$ `x := x+1` $\{\texttt{x} \geq 0\}$ // the postcondition of `x := x+2`
   - $\{\texttt{x} \geq 0 \wedge \texttt{even(x)}\}$ `x := x+2` $\{\texttt{even(x)}\}$
   - $\{\texttt{x} \geq 0 \wedge \texttt{odd(x)}\}$ `x := x+2` $\{\texttt{odd(x)}\}$

3. Interference (2nd and 4th triples are not valid)

   - $\{\texttt{T} \wedge \texttt{x} = 0\}$ `x := 0` $\{\texttt{x} = 0\}$
   - $\{\texttt{T} \wedge \texttt{x} = 2\}$ `x := 0` $\{\texttt{x} = 2\}$ // fails
   - $\{\texttt{x} = 0 \wedge \texttt{T}\}$ `x := x+2` $\{\texttt{T}\}$
   - $\{\texttt{x} = 0 \wedge \texttt{x} = 0\}$ `x := x+2` $\{\texttt{x} = 0\}$ // fails

4. Interference: The last triple is invalid.

   - $\{\texttt{T} \wedge \texttt{x} = 0\}$ `x := 0` $\{\texttt{x} = 0\}$
   - $\{\texttt{T} \wedge (\texttt{x} = 0 \vee \texttt{x} = 2)\ 0\}$ `x := 0` $\{\texttt{x} = 0 \vee \texttt{x} = 2\}$
   - $\{\texttt{x} = 0 \wedge \texttt{T}\}$ `x := x+2` $\{\texttt{T}\}$
   - $\{\texttt{x} = 0 \wedge \texttt{x} = 0\}$ `x := x+2` $\{\texttt{x} = 0\}$ // fails

5. (Interference checks)

   Thread 1 vs thread 2:

   - $\{ p_2 \wedge q \}\ S_1\ \{ q \}$ where $q = q_1, q_3, q_4$. (Since q_2 is not in front of an atomic statement, it's not necessary, but including it is pretty harmless.)
   - Technically, we also need $\{ p_3 \wedge q \}$ **`skip`** $\{ q \}$ for the same q, but these are trivial.

   Thread 2 vs thread 1:

   - $\{ q_1 \wedge p \}\ T_1\ \{ p \}$ where $p = p_1, p_2, p_3$, and p_4.
   - $\{ q_3 \wedge p \}\ T_2\ \{ p \}$ for the same p.

