# Shared Variables and Interference-Freedom

*CS 536: Science of Programming, Spring 2022*

## 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 class 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 ⟨ *[x := x+1; x := x*2 ‖ x := x+3], {x = 10}* ⟩. Since 11 + 1 + 3 = 11 + 3 + 1, two of the intermediate states are equal.

$$\begin{tikzcd}
 & \langle [x := x + \overline{1}; x := x * \overline{2} \| x := x + 3], \{x = 10\} \rangle \arrow[dl] \arrow[dr] & \\
\langle [x := x * \overline{2} \| x := x + 3], \{x = 11\} \rangle \arrow[d] \arrow[drr] & & \langle [x := x + \overline{1}; x := x * \overline{2} \| \mathsf{skip}], \{x = 13\} \rangle \arrow[d] \\
\langle [\mathsf{skip} \| x := x + 3], \{x = 22\} \rangle \arrow[d] & & \langle [x := x * \overline{2} \| \mathsf{skip}], \{x = 14\} \rangle \arrow[d] \\
\langle \mathsf{skip}, \{x = 25\} \rangle & & \langle \mathsf{skip}, \{x = 28\} \rangle
\end{tikzcd}$$

- 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. Critical Sections

- The basic *critical section* problem involves two threads, each with an identified subset of code (the *critical section*), where we must avoid both threads executing code in their critical sections simultaneously. Or said another way, if one thread is executing code in its critical section, then we must keep the other thread from entering its critical section. Race conditions caused by interleaving two pieces of code is a form of the critical section problem.
- Critical sections don't only involve what state changes pieces of code do, it also depends on the form of the code. For example, we normally don't think of there being a difference between $x := y+y$ and $x := y;\ x := x+y$; they both set $x$ to $2y$. But if their opportunities for interleaving are different, then these two pieces of code can behave very differently.
  - For us, $x := y+y$ cannot be interleaved but $x := y;\ x := x+y$ can be interleaved with at the semicolon (i.e., between statements). If we try to run both in parallel, there are three possible interleavings:
    - $\{y = n\}\ x := y+y;\ \{x = 2n\}\ x := y;\ \{x = n\}\ x := x+y\ \{x = 2n\}$
    - $\{y = n\}\ x := y;\ \{x = n\}\ x := x+y;\ \{x = 2n\}\ x := y+y\ \{x = 2n\}$
    - $\{y = n\}\ x := y;\ \{x = n\}\ x := y+y;\ \{x = 2n\}\ x := x+y\ \{x = 3n\}$
  - The third interleaving involves a race condition because running $x := y+y$ by overwriting $x$ after the other thread sets $x := y$ but before that thread gets to use x in $x := x+y$.
- Historically, the critical section problem was very difficult to solve using software alone. (Numerous proposed solutions were in fact wrong.) Eventually, the problem was solved by adding new hardware instructions ("test and set").

## ***E.*** *Atomic Regions*

- 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 $\langle S, \sigma\rangle \rightarrow^* \langle skip, \tau\rangle$ then $\langle < S >, \sigma\rangle \rightarrow \langle skip, \tau\rangle$.
- Our operational semantics definition makes assignment statements atomic automatically, so making $v := e$ its own atomic section causes no change. However, embedding $v := e$ within a larger atomic region does make a difference.
- A *normal assignment* is one not inside an atomic region.
- *Example 2*: For example, since it can't be interleaved, $< x := y; x := x+y >$ has the same effect as $x := y+y$. Indeed, the evaluation graph for both of them involve a single → arrow:
  - $\langle x := y+y, \sigma\rangle \rightarrow \langle skip, \sigma[x \mapsto 2\sigma(y)]\rangle$
  - $\langle < x := y; x := x+y >, \sigma\rangle \rightarrow \langle skip, \sigma[x \mapsto 2\sigma(y)]\rangle$
    - This is because $\langle x := y; x := x+y, \sigma\rangle \rightarrow^2 \langle skip, \sigma[x \mapsto 2\sigma(y)]\rangle$
- Using atomic regions gives us control over the size of pieces of code that can be interleaved *("granularity of code interleaving")*. However, making more or larger atomic sections is no panacea: The more or larger atomic regions code has, the less interleaving and hence parallelism we have.

## ***F.*** *Interleaving with skip, if, and while statements*

- There's no interleaving with a *skip* statement: *skip* executes atomically, so nothing can execute "in the middle of a" *skip*. Since *skip* doesn't change the state, interleaving it between two statements of other threads causes no change.
- For *if-else* and *while* statements (and nondeterministic branches and loops), although evaluation of a boolean expression is atomic, interleaving can occur between inspection of the result and the jump to the next configuration.
  - For example, sequentially, if $\sigma(e) = T$, then $\langle \textit{if } e \textit{ then } \{s_1\} \textit{ else } \{s_2\}, \sigma\rangle \rightarrow \langle S_1, \sigma\rangle$. So in parallel, another statement can be executed between the *if* test and the start of the true branch. If statement $U$ changes σ to σ', then we can have

$$\langle [\textit{if } e \textit{ then } \{s_1\} \textit{ else } \{s_2\} \parallel <U>], \sigma\rangle \rightarrow \langle [s_1 \parallel <U>], \sigma\rangle \rightarrow \langle [S_1 \parallel skip], \sigma'\rangle$$

  - In this last configuration, we're about to execute $s_1$ not in σ, but in σ'. We can't use annotations like $\{T\}$ *if e then* $\{\{e\}\ s_1\}$… because there's no guarantee that $e$ is still true when the true branch begins executing.
  - Exactly the same problem occurs with *while*, since we just step it to an *if.*

## ***G.*** *Interference Between Threads*

- *Interference* occurs when one thread invalidates a condition needed by another thread. In the previous class 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 $\{x \geq 0\}\ y := 0\ \{x \geq y = 0\}$, then the conditions would still not be disjoint, but $x := 1$ would not have caused them to become invalid.
- *Example 3*: Let $\{x > 0\}\ s_1\ \{x \geq 0\}$ and $\{x > 0\}\ s_2\ \{x > 0\}$ be two threads. If $x > 0$, then the preconditions for $s_1$ and $s_2$ hold, so we can run either one.
  - If $s_1$ runs before $s_2$, then $s_1$ terminates with $x \geq 0$, interfering with the precondition of $s_2$.
  - If $s_1$ runs after $s_2$, then $s_1$ again takes $x > 0$ to $x \geq 0$, interfering with the postcondition of $S_2$.
  - If $s_2$ runs before $s_1$, it terminates with $x > 0$, so it doesn't interfere with the precondition of $S_1$.
  - To remove the interference caused by $s_1$, we might strengthen its postcondition from $x \geq 0$ to $x > 0$ (if we can), or we might weaken the precondition of $s_2$ to $x \geq 0$ (again, if we can). Another possibility is to make $s_1$ run atomically with the code that follows it (and presumably requires $x \geq 0$).
    - $\{x > 0\}\ s_1\ \{x \geq 0\};\ U\ \{q\}$ could become $\{x > 0\} < S_1\ \{x \geq 0\};\ U > \{q\}$, so we wouldn't be able to run $s_2$ between $s_1$ and $U$.

## ***H.*** *The Interference-Freedom Checks*

- *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 what $< s >$ does if $p$ holds but $q$ does not: $\{p \wedge \neg q\}$ $< s > \{\ldots\}$.)
- *Example 4*: $\{p\}\ x := x+1\ \{\ldots\}$ does not interfere with $x \geq 0$, but it does interfere with $x < 0$.
- *Example 5*: $\{x \leq -1\}\ x := x+1\ \{\ldots\}$ does not interfere with $x \leq 0$ *or* $x > 0$ (because $x > 0 \wedge x \leq -1 = F$)
- *Example 6*: $\{x \% 4 = 2\}\ x := x+4\ \{\ldots\}$ does not interfere with *even(x)* (i.e., $x \% 2 = 0$).
- Note 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 $q$. Interference freedom is less restrictive than pairwise disjointedness of programs.

### *Failing an interference freedom check*

- Proving that the interference freedom check for $\{p\} < s > \{\ldots\}$ with $q$ is valid tells us that we know interference cannot occur at runtime. That means failing to prove the check only tells us that *we don't know that interference cannot occur at runtime*. Said another way, the negation of "*does not interfere with*" is "*possibly interferes with*".

- Specifically, failing an interference freedom check does not guarantee that interference will occur at runtime. Interference occurs if running the program uses $\langle <s>, \tau_0 \rangle \rightarrow \langle \text{skip}, \tau_1 \rangle$ for some $\tau_0 \vDash p \land q$ and some $\tau_1 \nvDash q$. If program execution along some path never involves $\langle <s>, \tau_0 \rangle \rightarrow \langle \text{skip}, \tau_1 \rangle$, then interference doesn't occur[1].

## *Checking for Interference-Freedom of larger structures*

- Once we have a notion of interference freedom of an atomic triple versus a predicate, we can build up to a notion of interference between threads.
- Because we care not just about interference with the variables and statements, but with the conditions itself, we need to introduce notations for conditions and proofs into the language we're using to talk about programs. This is weird and we haven't really done it before (except in a limited way last class). In fact, we're now leaving the realm of standard Hoare logic and entering a new logic based on it called Owicki-Gries logic after its inventors Susan Owicki and David Gries.
- *Notation*: $s^*$ is 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 $p_2$, nor with $q_2$, nor with any precondition before an atomic statement in $s_2^*$. I.e., $\{p_1\} < s_1 > \{\ldots\}$ does not interfere with any $r$ where $\{r\} < \ldots > \{\ldots\}$ appears in $s_2^*$.
- *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 $\{r\} < s > \{\ldots\}$ in $s_1^*$ does not interfere with the outline $\{p_2\}\ s_2^*\ \{q_2\}$.
  - By atomic statements, we mean <s> and x := e (technically, this also includes *skip*, but *skip* can't interfere with anything).
- It's sufficient to look at only the atomic statements in $s_1^*$ because complex statements don't cause state changes until they get to atomic sub-statements. e.g., with $\{p_1\}$ *while* $e$ *do* $\{\{p_2\}\ s\ \{p_3\}\}\ \{p_4\}$, when execution is at $p_1$ or $p_3$, the next execution step involves testing $e$ and jumping to $p_2$ or $p_4$; this doesn't change the state. When execution is at $p_3$, execution might cause a state change, but only if $s$ begins with (or is) an atomic statement. The situations with *if* statements and nondeterministic *if* and *do* statements are similar.
- *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.

1 In actual execution, $<s>$ would be the next step of execution of a thread, so we would get something like $\langle\ [<s>;\ s' \| \ldots], \tau_0 \rangle \rightarrow \langle [s' \| \ldots],\ \tau_1 \rangle$.

- *Example 7: {x mod 4 = 0} x := x+3 {…}* interferes with *{even(x)} x := x+1 {odd(x)}* because it interferes with *even(x): {x mod 4 = 0 ∧ even(x)} x := x+3 {even(x)}* is not valid. However, the first triple doesn't interfere with *odd(x)*: *{x mod 4 = 0 ∧ odd(x)} x := x+3 {odd(x)}* is valid because the precondition implies false (and *{F} s {q}* is valid for all *s* and *q*).
- *Example 8*: Two different 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 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.
  - 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.

## *I. Normalized Proof Outlines*

- Say we have an outline that includes a sequence of conditions between two statements, as in $s_1$ $\{q_1\}$ => $\{q_2\}$ $s_2$, where $q_1$ implies $q_2$. It turns out we only need to prove interference freedom with respect to $q_2$.
- If $q_1$ passes an interference test $\models \{p \wedge q_1\}\ t\ \{q_1\}$, then because $q_1 \rightarrow q_2$, we know $\models \{p \wedge q_1\}\ t\ \{q_1\}$ => $\{q_2\}$, so $q_2$ will be satisfied before attempting to execute t.
- Definition: A standard proof outline {p} s* {q} has exactly one condition before each atomic statement in s*.
- We use standard outlines because we need the outline to have enough conditions for the interference-freedom checks to guarantee non-interference, but we don't want to have extra conditions lying around and making more work for us.
- If we have a full proof outline, we can get a standard one by shrinking sequences of internal conditions.
- Example 10: Instead of {T} $\{1 = 2^0\}$ x := 1 {x = 2n}, use {T} x := 1 {x = 2 n}
- Example 11: Instead of {T} x := 1; {x > 0} {inv x ≥ 0} while B …, use
  {T} x := 1 {inv x ≥ 0} while B …

## *J. Parallelism with Shared Variables and Interference-Freedom*

- *Theorem (Interference-Freedom)*: Let $\{p_1\}\ s_1^*\ \{q_1\}$, $\{p_2\}\ s_2^*\ \{q_2\}$, …, and $\{p_n\}\ s_n^*\ \{q_n\}$ be sequentially valid and pairwise interference-free. Then their parallel composition

$\{p_1 \wedge p_2 \wedge \dots \wedge p_n\}\ [s_1^* \| \dots \| s_n^*]\ \{q_1 \wedge q_2 \wedge \dots \wedge q_n\}$

  is also valid. (Proof omitted.)
- The interference freedom theorem enables the use of a new parallelism rule:

## *Parallelism with Owicki-Gries*

$$\frac{\{p_i\} s_i^* \{q_i\} \{p_i\} s_i^* \{q_i\}\ pairwise\ interference - free}{\{p_1 \wedge p_2 \wedge \ldots \wedge p_n\}[s_1^* \parallel \ldots \parallel s_n^*]\{q_1 \wedge q_2 \wedge \ldots \wedge q_n\}} (Parallelism - OG)$$

- One feature of this rule is that it talks about proof outlines, not correctness triples. (Before this, the rules only concerned triples.) We can no longer compose correctness triples because we can't guarantee correctness without knowing that the different threads don't invalidate conditions inside the other threads.

- *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 ∧ T} x := x+2 {T}*
    - *{x = 0 ∧ (x = 0 ∨ x = 2)} x := x+2 {x = 0 ∨ x = 2}*
  - *{T} x := 0 {…}* does not interfere with *x = 0* or *x = 0 ∨ x = 2*
    - *{T ∧ x = 0} x := 0 {x = 0}*
    - *{T ∧ (x = 0 ∨ x = 2)} x := 0 {x = 0 ∨ 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*.
- Sequentially, the disjunct *x = 0* is not needed in *{x = 0} x := x+2 {x = 0 ∨ x = 2}*, nor is *x = 2* needed in *{T} x := 0 {x = 0 ∨ x = 2}*. To run the threads in parallel, however, we need to add these disjuncts to account for the interactions that parallel execution causes. (Or said the other way, we add these disjuncts to avoid interference in the final result.)