Shared Variables and Interference-Freedom

CS 536: Science of Programming, Fall 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).

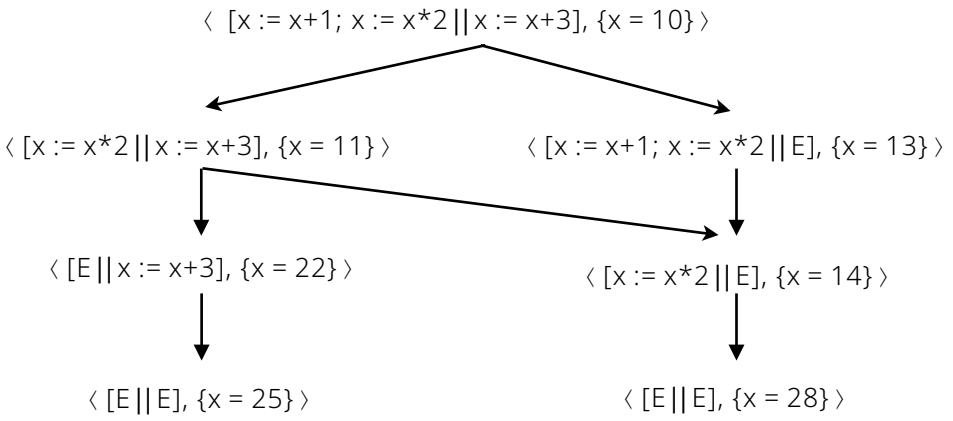
B. Objectives

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.



- 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). There are three possible interleavings:
    - {y = α} x := y+y; {x = 2α} x := y; {x = α} x := x+y {x = 2α}
    - {y = α} x := y; {x = α} x := x+y; {x = 2α} x := y+y {x = 2α}
    - {y = α} x := y; {x = α} x := y+y; {x = 2α} x := x+y {x = 3α}
  - 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 ⟨ S, σ ⟩ →* ⟨ E, τ ⟩ then ⟨ < S >, σ ⟩ → ⟨ E, τ ⟩.
- 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 area. We worry about interleaving of normal assignments; we don't worry about the non-normal ones.

- ***Example 2***: For example, since it can't be interleaved with $< x := y;\ x := x+y >$ has the same effect as $x := y+y$. Indeed, the evaluation graph for both of them involve a single $\rightarrow$ arrow:
  - $\langle x := y+y, \sigma\rangle \rightarrow \langle E, \sigma[x \mapsto 2\sigma(y)]\rangle$
  - $\langle <x := y;\ x := x+y >, \sigma\rangle \rightarrow \langle E, \sigma[x \mapsto 2\sigma(y)]\rangle$
    - This is because $\langle x := y;\ x := x+y, \sigma\rangle \rightarrow^2 \langle E, \sigma[x \mapsto 2\sigma(y)]\rangle$
- 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 ***if-fi*** and ***do-od***), 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 $\sigma(B) = T$, then $\langle$ ***if*** B ***then*** S_1 ***else*** S_2 ***fi***, $\sigma\rangle \rightarrow \langle S_1, \sigma\rangle$. 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 τ, then we can have

    $$\langle [\textbf{\textit{if}}\ B\ \textbf{\textit{then}}\ S_1\ \textbf{\textit{else}}\ S_2\ \textbf{\textit{fi}}\ ||\ < U >], \sigma\rangle \rightarrow \langle [S_1\ ||\ < U >], \sigma\rangle \rightarrow \langle [S_1\ ||\ E], \tau\rangle$$

  - In this last configuration, we're about to execute S_1 not in σ, but in τ. We can't use annotations like $\{T\}$ ***if*** B ***then*** $\{B\}\ S_1 \ldots$ ***fi*** because there's no guarantee that B is still true when the true branch begins executing.
  - Exactly the same problem occurs with ***while***. If $W \equiv$ ***while*** B ***do*** S ***od***, then

    $$\langle [\textbf{\textit{while}}\ B\ \textbf{\textit{do}}\ S\ \textbf{\textit{od}}\ ||\ < U >], \sigma\rangle \rightarrow \langle [S;\ W\ ||\ < U >], \sigma\rangle \rightarrow \langle [S;\ W\ ||\ E], \tau\rangle$$

So B may be false when the loop body S starts executing.

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 $x > 0$, or we might weaken the precondition of S_2 to $x \geq 0$. Another possibility is to make running S_1 atomically with the code that follows it (and presumably requires $x \geq 0$).
  - $\{x > 0\}\ S_1\ \{x \geq 0\};\ U\ \{q\}$ could become $\{x > 0\}\ \langle S_1\ \{x \geq 0\};\ U \rangle\ \{q\}$, 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 $\{p\}\ \langle S \rangle\ \{\ldots\}$ ***versus a predicate*** q is $\{p \wedge q\}\ \langle S \rangle\ \{q\}$. If this check is valid, then we say that $\{p\}\ \langle S \rangle\ \{\ldots\}$ ***does not interfere with*** q. (Note we don't care what $\langle S \rangle$ does if p holds but q does not: $\{p \wedge \neg q\}\ \langle S \rangle\ \{\ldots\}$.)
- ***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 $x \leq 0$, but it does interfere with $x > 0$.
- ***Example 6***: $\{x \,\%\, 4 = 2\}\ x := x+4\ \{\ldots\}$ does not interfere with $\text{even}(x)$ (i.e., $x \,\%\, 2 = 0$).
- Note interference freedom of $\{p\}\ \langle S \rangle\ \{\ldots\}$ 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 $\{p\}\ \langle S \rangle\ \{\ldots\}$ 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 $\langle \langle S \rangle, \tau_0 \rangle \rightarrow \langle E, \tau_1 \rangle$ for some $\tau_0 \vDash p \wedge q$ and some $\tau_1 \nvDash q$. If program execution along some path never involves $\langle \langle S \rangle, \tau_0 \rangle \rightarrow \langle E, \tau_1 \rangle$, 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.
- ***Notation***: S^* is a proof outline of the program S.
- ***Definition***: The atomic statement $\{p_1\}\ \langle S_1 \rangle\ \{\ldots\}$ ***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., $\{p_1\}\ \langle S_1 \rangle\ \{\ldots\}$ does not interfere with any r where $\{r\}\ \langle \ldots \rangle\ \{\ldots\}$ appears in S_2^*.

---

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

- ***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 $\{r\}\ <S>\ \{...\}$ in S_1^* does not interfere with the outline $\{p_2\}\ S_2^*\ \{q_2\}$.
- 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*** B ***do*** $\{p_2\}\ S\ \{p_3\}$ ***od*** $\{p_4\}$, when execution is at p_1 or p_3, the next execution step involves testing B 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.
- ***Example 7:*** $\{x \bmod 4 = 0\}\ x := x+3\ \{...\}$ interferes with $\{even(x)\}\ x := x+1\ \{odd(x)\}$ because it interferes with $even(x)$: $\{x \bmod 4 = 0 \wedge even(x)\}\ x := x+3\ \{even(x)\}$ is not valid. However, the first triple doesn't interfere with $odd(x)$: $\{x \bmod 4 = 0 \wedge odd(x)\}\ x := x+3\ \{odd(x)\}$ 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 $\{x \geq 0\}\ x := x+2\ \{x > 1\}$ are interference-free.
  - Precondition of triple 1: $\{x \geq 0 \wedge even(x)\}\ x:=x+2\ \{even(x)\}$ is valid.
  - Postcondition of triple 1: $\{x \geq 0 \wedge odd(x)\}\ x := x+2\ \{odd(x)\}$ is valid.
  - Precondition of triple 2: $\{even(x) \wedge x \geq 0\}\ x := x+1\ \{x \geq 0\}$ is valid.
  - Postcondition of triple 2: $\{even(x) \wedge x > 1\}\ x := x+1\ \{x > 1\}$ is valid.

I. 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

$$\{p_1 \wedge p_2 \wedge ... \wedge p_n\}\ [S_1^* \,||\, ... \,||\, S_n^*]\ \{q_1 \wedge q_2 \wedge ... \wedge q_n\}$$

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

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. $\{p_1 \wedge p_2 \wedge ... \wedge p_n\}\ [S_1^* \,\|\, ... \,\|\, S_n^*]\ \{q_1 \wedge q_2 \wedge ... \wedge q_n\}$ Parallelism w/ Shared Vars, 1, 2, ..., n

where the $\{p_k\}\ S_k^*\ \{q_k\}$ 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.) 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.)

J. An Example With Shared and Auxiliary Variables[2]

- Recall the program

  {x = 0} [x := x+2 || x := 0] {x = 0 ∨ x = 2}

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

---

[2] We'll look at auxiliary variables in detail in the next class.

- 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

  $$\{T \wedge x = x_0\}\ [\ \{T\}\ x := 0\ \{x = 0\}\ ||\ \{x = x_0\}\ x := x+2\ \{x = x_0+2\}\ ]\ \{...\}$$

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

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

But if thread 2 runs first, it interferes with thread 1's $x = 0$. When thread 1 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 add $x = 4$ as a third disjunct (and get $x = 0 \vee x = 2 \vee x = 4$), but interference with it leads us to add $x = 6$ as a disjunct, which leads to $x = 8$, ad infinitum. In addition, adding $x = 4$ or 6 or 8 ... doesn't reflect the reality of what the program does: Either $x = x_0$ or 0 or x_0+2 or 2. It should never be 4 (unless $x_0 = 2$).

- The problem here is that we don't know which case ran first: If thread 1 runs first then we have $x = x_0$ and then $x = 0$ and then thread 2 sets $x = 2$. If thread 2 runs first then we have $x = x_0$ and then $x = x_0+2$ and then $x = 0$. We can solve this problem by adding a new boolean variable inc that tells us whether or not the $x:= x+2$ increment has been done.

  $$\{T\}\ inc := F;\ [x := 0\ ||\ <x := x+2;\ inc := T>]\ \{x = 0 \vee x = 2\}$$

- First, let's look at how adding inc lets us prove correctness, then we'll look at (and eliminate) the inefficiency caused by adding a new (what we'll eventually call an ***auxiliary***) variable.
  - 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: $\{T\}\ x := 0\ \{x = 0 \vee (inc \wedge x = 2)\}$. The postcondition can't just be $x = 0$, since that's interfered with by thread 2. If thread 2 runs, however, it sees $x = 0$ and sets $x = 2$ and inc to true, so we can make that a second disjunct of the postcondition.
  - For the second thread, $\{\neg inc\}\ <x := x+2;\ inc := T>\ \{inc\}$ is all we need. The important information about the value of x is held in the conditions of thread 1. We could add

information to the conditions of the second thread, but it just makes for more complicated interference-freedom checks.

- Here is a full annotation for our program, with inc added

  {T} inc := F; {T ∧ ¬inc}
  [{T} x := 0 {x = 0 ∨ (inc ∧ x = 2)}
  || {¬inc} <x := x+2; inc := T > {inc}
  ]
  {(x = 0 ∨ (inc ∧ x = 2)) ∧ inc}
  {x = 0 ∨ x = 2}

- Adding inc lets us know whether or not the increment of x has occurred. There's a symmetric alternative, which is to use a variable that tells us whether or not x := 0 has executed; let's call it z (short for zeroed). To avoid interference, we need to make z := T atomic with x := 0 so that thread 2 can never observe x = 0 ∧ ¬ z.

  {T} z:= F; {¬z ∧ (z → x = 0)}
  [{¬z} <x:= 0; z:= T > {z}
  || {z → x = 0} x:= x+2 {z → x = 0 ∨ x = 2}
  ]
  {z ∧ (z → x = 0 ∨ x = 2)}
  {x = 0 ∨ x = 2}

  Once again, we don't need to include all the variables in all the conditions. The postcondition of thread 1 doesn't need to mention x because all the relevant information is contained in the postcondition of thread 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 z → x = 0 or with z → x = 0 ∨ x = 2, and thread 2 doesn't modify z, so it can't interfere with z.

Auxiliary variables

- Whether we add inc or z, the resulting program works correctly, but we've also added to its computation, which doesn't seem efficient. On the other hand, assignments to x don't rely on inc or z: We don't have x := ... expression involving inc or z ..., so for purposes of calculating x, the actual value of inc or z in memory isn't relevant. When we look at auxiliary variables in the next class, we'll see that the code involving inc or z can be removed without affecting the overall correctness of the program. This will get us back to the program and annotation we wanted,

  {T} [x := 0 || x := x+2] {x = 0 ∨ x = 2}