Await and Deadlocks

CS 536: Science of Programming, Spring 2023

A. Why?

- Avoiding interference isn't the same as coordinating desirable activities.
- It's common for one thread to wait for another thread to reach a desired state.
- Care needs to be taken to avoid a program that waits with no hope of completing.

B. Objectives

At the end of this lecture you should know

- The syntax and semantics of the ***await*** statement.
- How to draw an evaluation diagram for a parallel program that uses ***await***.
- How to recognize deadlocked configurations in an evaluation diagram.
- How to list the potential deadlock predicates for a parallel program that uses ***await***.

C. Synchronization

The Need for Synchronization

- We've looked at parallel programs whose threads avoid bad interactions.
- They don't interfere because they don't interact (disjoint programs/conditions).
- They interact but don't interfere (interference-freedom).
- To supporting good interaction between threads, we often have to have one thread wait for another one. Some examples:
  - Thread 1 should wait until thread 2 is finished executing a certain block of code.
  - Thread 1 has to wait until some buffer is not empty
  - Thread 2 has to wait until some buffer is not full.
- The general problem is that we often want threads to ***synchronize***: We want one thread to wait until some other thread makes a condition come true.
- ***Example 1***: For a more specific example, in the following program, the calculation of u doesn't start until we finish calculating z, even though u doesn't depend on z.

  $[x := \dots \,||\, y := \dots \,||\, z := \dots];\ u = f(x, y);\ v := g(u, z)$

  On the other hand, we can't nest parallel programs, so we can't write

  $[[[x := \dots \,||\, y := \dots];\ u = f(x, y) \,||\, z := \dots];\ v := g(u, z)$

  which would be a natural way to do the calculations of u and z in parallel. In some sense, what we'd like is to run something like

$[\,x := \ldots \,||\, y := \ldots \,||\,$ wait for x and $y;\ u = f(x, y) \,||\, z := \ldots\,];\ v := g(u, z)$

D. The Await Statement

- It's time to introduce a new statement, the ***await*** statement, whose semantics implements the notion of waiting until some condition is true.
    - Busy wait loops like ***while*** $\neg B$ ***do skip od*** $\{B\}$ work but are wasteful.
- ***Syntax***: ***await*** B ***then*** S ***end*** where B is a boolean expression and S is a statement.
    - S isn't allowed to have loops, ***await*** statements, or atomic regions.
    - ***await*** statements can only appear in sequential threads of parallel programs. (I.e., in some thread S_k in an $[\,S_1 \,||\, S_2 \,||\, \ldots\,]$.)
- An ***await*** statement is a ***conditional atomic region***. Suppose that some thread begins with ***await*** B ***then*** S ***end***, then
    - We nondeterministically choose between all the available threads. I.e., there's no insistence that we must check the ***await*** before trying other threads. (See case 1 of Example 2.)
    - If we choose the thread that begins with ***await*** B ***then*** S ***end***,
    - If B is true, then immediately jump to S and execute all of it.
        - The test, jump, and execution of S are atomic — the combination executes as one step. E.g., with the configuration below, we can't set x to 1 between looking up the two x's to use for calculating $x+x$. (See case 2 of Example 2.)
    - If B is false, we ***block***: We wait until B is true. Instead, we nondeterministically choose between the other threads and execute it. (See case 3 of Example 2.)
- An ***await*** is similar to an atomic ***if-then*** statement, but not identical.
    - With $\langle$ ***if*** B ***then*** S ***else skip fi*** $\rangle$, if B is false, we execute ***skip*** and complete the ***if-fi***. (See case 4, Example 2.)
    - With ***await*** B ***then*** S ***end***, if B is false, nothing happens until B becomes true. (See the note with case 3, Example 2.)

Example 2:

- (See the discussion above)
    - Case 1: Let $A \equiv$ ***await*** B ***then*** $x := x+x$ ***end*** in $\langle\, [$ ***await*** B ***then*** $x := x+x$ ***end*** $||\ x := 1\,]\ \{b=T, x=2\}\,\rangle$ $\rightarrow \langle\, [$ ***await*** B ***then*** $x := x+x$ ***end*** $||\ E\,],\ \{b=T, x=1\}\,\rangle$.
    - Case 2: $\langle\, [$ ***await*** B ***then*** $x := x+x$ ***end*** $||\ x := 1\,], \{b=T, x=2\}\,\rangle \rightarrow \langle\, [\,E \,||\, x := 1\,], \{b=T, x=4\}\,\rangle$. (This is the only transition that executes the ***await***.)
    - Case 3: $\langle\, [$ ***await*** B ***then*** S ***end*** $||\ x := 1\,], \{b=F\}\,\rangle \rightarrow \langle\, [$ ***await*** B ***then*** S ***end*** $||\ E\,], \{b=F, x=1\}\,\rangle$. (The second configuration is blocked, with no other thread available to unblock it.)
    - Case 4: $\langle\, [$ ***if*** B ***then*** $x := 0$ ***else skip fi*** $||\ S'\,], \{b=F\}\,\rangle \rightarrow \langle\, [\,E \,||\, S'\,], \{b=F, x=0\}\,\rangle$.

- ***Example 3:*** Execution of a non-atomic ***if-fi*** can be interleaved with. In Figure 1, the dashed red lines show how execution of ***await*** $x \ge 0$ ***then*** $x := x+1 \,; y := x+2$ end takes just one step to execute the entire body.

Solid black lines show execution steps taken only when $S \equiv$ ***if*** $x \ge 0$ ***then*** $x := x+1;\, y := x+2$ ***fi***

Dashed red lines show steps taken only when $S \equiv$ ***await*** $x \ge 0$ ***then*** $x := x+1;\, y := x+2$ ***end***

Dashed black lines are common to both executions.

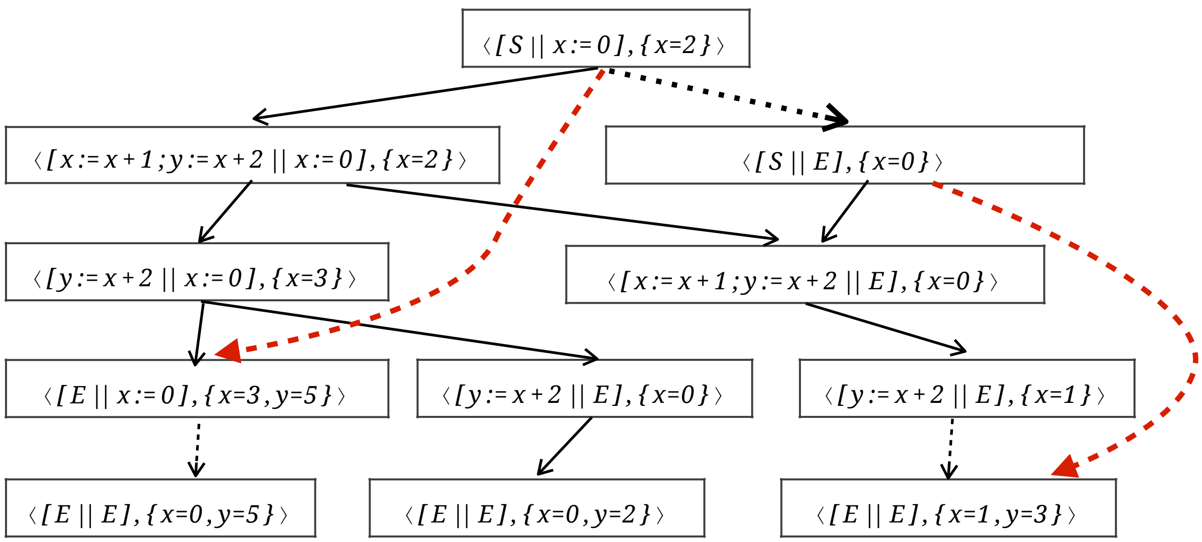

Figure 1: Execution of ***await*** vs ***if-fi***

- ***Example 4***: In the introduction, we looked at a situation where we want to wait for some calculations to finish before stating others.

  $[x := \ldots \,||\, y := \ldots \,||$ ***wait*** *for* x *and* $y;\ u = f(x,y) \,||\, z := \ldots];\ v := g(u,z)$

  can be implemented using

  $x\_done := F;\ y\_done := F;$
  $[x := \ldots;\ \ x\_done := T \,||\, y := \ldots;\ y\_done := T$
  $||$ ***await*** $x\_done \wedge y\_done$ ***then*** $u := f(x,y)$ ***end*** $||\, z := \ldots];$
  $v := g(u,z);$

E. await, wait, if, and ⟨ S ⟩

The Abbreviations ⟨ S ⟩ and wait B

- With ***await*** B ***then*** S ***end***, there are two simple cases: When B is trivial and when S is trivial.

- ***Definition***: We can redefine $\langle S \rangle$ to stand for ***await*** T ***then*** S ***end***. When the test is trivially true, we don't need to wait, we simply execute the body atomically. So atomic execution is just conditional atomic execution with a trivial test.
- ***Definition***: ***wait*** $B \equiv$ ***await*** B ***then skip end***. When the body is trivial, we simply wait; when B is true, execution is complete.
- There's a important difference between ***wait*** B *;* S and ***await*** B ***then*** S ***end***.
  - With ***await*** B ***then*** S ***end***, once B is true, we immediately atomically execute S, so no other statement can interleave between the test and running S. Therefore S can rely on B being true when it starts executing. If $\sigma(B) = T$, then $\langle [\,\textbf{await}\ B\ \textbf{then}\ S\ \textbf{end} \,||\, \ldots], \sigma\rangle \rightarrow \langle [E \,||\, \ldots], \tau\rangle$, where $\tau \in M(S, \sigma)$.
  - ***wait*** B *;* S means ***await*** B ***then skip end;*** S, so it allows another thread to be executed after the ***wait*** but before running S. If $\sigma(B) = T$, then $\langle [\,\textbf{wait}\ B ; S \,||\, \ldots], \sigma\rangle \rightarrow \langle [S \,||\, \ldots], \sigma\rangle$ $\rightarrow^* \langle [E \,||\, \ldots], \tau\rangle$ (if no interleaving occurs). Since interleaving can occur, we rely on B being true when S starts execution.

F. Await Statement Proof Rule and Outlines

- The proof rule for the ***await*** statement is similar to an ***if fi***, but there's no false clause (not even ***else skip***).

await Statement (a.k.a. Synchronization Rule)

1.	$\{p \wedge B\}\ S\ \{q\}$	
2.	$\{p\}$ ***await*** B ***then*** S ***end*** $\{q\}$	***await***, *1*

- ***Minimal Proof Outline***: $\{p\}$ ***await*** B ***then*** S ***end*** $\{q\}$
- ***Full Proof Outline***: $\{p\}$ ***await*** B ***then*** $\{p \wedge B\}\ S^*\ \{q\}$ ***end*** $\{q\}$ where S^* is a full proof outline for program S.
- ***Weakest Preconditions***: $wp(\textbf{await}\ B\ \textbf{then}\ S\ \textbf{end}, q) \equiv B \rightarrow wp(S, q)$.
  - This guarantees $\{B \rightarrow wp(S, q)\}$ ***await*** B ***then*** $\{wp(S, q)\}\ S^*\ \{q\}$ ***end*** $\{q\}$
- Note: It may be tempting to write $\{p \wedge \neg B\}$ ***await*** B ***then*** …, but that's guaranteed to self-deadlock; the outline is

$$\{p \wedge \neg B\}\ \textbf{await}\ B\ \textbf{then}\ \{p \wedge \neg B \wedge B\}\ S^*\ \{q\}\ \textbf{end}\ \{q\}$$

G. The Producer/Consumer Problem

- The ***Producer/Consumer Problem*** (a.k.a. ***Bounded Buffer Problem***) is a standard problem in parallel programming.
- We have two threads running in parallel: The producer creates things and puts them into a buffer; the consumer removes things from the buffer and does something with them.

- The problem is that if the buffer is full, the producer shouldn't add anything to the buffer; if the buffer is empty, the consumer shouldn't remove anything from the buffer.
- ***Example 5***: The rough code to solve this problem is

```
Initialize ( buffer );
[ while ¬ done do                        // Producer
    created := Create ( );
    await NotFull ( buffer ) then
        BufferAdd ( buffer, created )
    end
 od
|| while ¬ done do                       // Consumer
    await NotEmpty ( buffer) then
        removed := BufferRemove ( buffer) ;
    end;
    Consume ( removed )
    od
]
```

- Buffer operations need to be synchronized because the threads share the buffer. The threads don't share the created or removed objects, so the *Create* and *Consume* calls can go outside the ***await*** and interleave execution.

H. Deadlock

Blocked Threads; Deadlock

- Recall that $\langle$ [***await*** *B* ***then*** *S* ***end;*** ... || ...] , $\sigma \rangle$ is blocked (must wait) if $\sigma(B) = F$.
  - If some other thread can make *B* true, then the await may eventually unblock.
  - E.g., $\langle$ [***await*** $x > y$ ***then*** *S* ***end;*** ... || ...; $x := y+1$; ...] , $\sigma \rangle$ could unblock.
  - But if all the other threads have either completed or are themselves blocked, then there's no way for our ***await*** to unblock. E.g., take $\langle$ [***await*** *B* ***then*** *S* ***end;*** ... || *E*] , $\sigma \rangle$. If thread 1 is stuck at the ***await*** but thread 2 has completed, the program can't evaluate further.
- ***Definition***: A parallel program is ***deadlocked*** if it has not finished execution and there's no possible evaluation step to take. I.e., all the threads are either complete or blocked and at least one thread is blocked.
- If all the other threads are complete or are blocked, the program is ***deadlocked:*** There's no possible evaluation step leaving from the configuration.
- ***Example 6***: If $A \equiv$ ***await*** $x \geq 0$ ... ***end***, then there's no arrow out of $\langle [A \,||\, A], \sigma[x \mapsto -1] \rangle$, so this configuration is deadlocked. If the value of *x* had been ≥ 0, then both ***await*** statements would have been eligible for execution.

Since only one blocked thread is required for deadlock, $\langle [\,\textbf{await}\ x \geq 0 \ldots \textbf{end} \,||\, E\,], \sigma[x \mapsto -1]\rangle$ is also deadlocked.

- Threads can block themselves (trivial example: ***await false then*** *S* ***end***).
  - More often, threads block because they're waiting for conditions they expect other threads to establish. E.g., if we're running in a state where $y=0$ and $x=0$, then these two threads deadlock:
  - Thread 1: $\{p_1\}$ ***await*** $y \neq 0$ ***then*** $x := 1$; …
  - Thread 2: $\{p2\}$ ***await*** $x \neq 0$ ***then*** $y := 1$; …
- A program might deadlock under all execution paths or only certain execution paths.
- ***Example 7***: The program

  [***await*** $y \neq 0$ ***then*** $x := 1$ ***end*** || ***await*** $x \neq 0$ ***then*** $y := 1$ ***end***]

  deadlocks iff you execute in a state where x and y are both zero
- ***Example 8***: If thread 1 sets $x := 0$ before thread2 evaluates its ***wait*** x, then thread 2 will block. (Recall ***wait*** x ≡ ***await*** x ***then skip end.***)

  $\{T\}$ $x := 1$; $y := 1$;
  [***wait*** $y=1$; $x := 0$ || ***wait*** $x=1$; $y := 0$]
  $\{x=0 \wedge y=0\}$

Figure 2 contains an execution graph for this program in state $\{x=1, y=1\}$ (somewhat abbreviated). There are two deadlocking paths (and four paths that terminate correctly).

- Obviously, we'd like to know if a program is going to deadlock. The following test identifies a set of predicates that indicate potential problems with a program; if none of these predicates is satisfiable, then deadlock is guaranteed not to occur.
- If one or more of these predicates is satisfiable, then we can't guarantee that deadlock will not occur, but we aren't guaranteeing that deadlock ***must*** occur. (So the deadlock conditions are sufficient to show deadlock is impossible but they are not necessary conditions.)
- Let $\{p\}\,[\,\{p_1\}\,S_1{}^*\{q1\} \,||\, \{p2\}\,S_2{}^*\{q2\} \,||\, \ldots \,||\, \{p_n\}\,S_n{}^*\{q_n\}\,]\,\{q\}$ be a full outline for a parallel program, where $p \equiv p_1 \wedge \ldots \wedge p_n$ and $q \equiv q_1 \wedge \ldots \wedge q_n$.
- ***Definition***: A (***potential***) ***deadlock condition*** for the program outline above is a predicate of the form $r1' \wedge r2' \wedge \ldots \wedge r_n'$ where each r_k' is either
  - q_k, the postcondition for thread S_k or
  - $p \wedge \neg B$ where $\{p\}$ ***await*** B… appears in the proof outline for thread S_k.
  - In addition, at least one of the r_k' must involve waiting. I.e., $q \equiv q_1 \wedge \ldots \wedge q_n$ is not a potential deadlock condition.
- A program outline is ***deadlock-free*** if every one of its potential deadlock conditions is unsatisfiable (i.e., a contradiction):
  - I.e., for each deadlock condition r', we have $\models \neg r'$ (or the equivalent $\models r' \rightarrow F$).

Parallelism with Deadlock Freedom

$1.\quad \{p_1\}\ S_1{*}\ \{q_1\}$

$2.\quad \{p2\}\ S_2{*}\ \{q_2\}$

$\quad\ldots$

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

$n{+}1.\ \{p_1 \wedge p_2 \wedge \ldots \wedge p_n\}$

$[S_1 \,||\, S_2 \,||\, \ldots \,||\, S_n]$

$\{q_1 \wedge q_2 \wedge \ldots \wedge q_n\}$ D.P. w/o deadlock, *1, 2, …, n*

where the $\{p_k\}\ S_k{*}\ \{q_k\}$ are pairwise interference-free standard proof outlines and the parallel program outline is deadlock-free.

I. Examples of Deadlock Conditions

- ***Example 9***: Let's take the program from Example 7:

  [***await*** $y{\neq}0$ ***then*** $x := 1$ ***end*** || ***await*** $x{\neq}0$ ***then*** $y := 1$ ***end***]

  and develop an annotation for it:

  $\{T\}$

  $[\ \{T\}$ ***await*** $y{\neq}0$ ***then*** $\{y{\neq}0\}\ x := 1\ \{x{\neq}0 \wedge y{\neq}0\}$ ***end*** $\{x{\neq}0 \wedge y{\neq}0\}$

  $||\ \{T\}$ ***await*** $x{\neq}0$ ***then*** $\{x{\neq}0\}\ y := 1\ \{x{\neq}0 \wedge y{\neq}0\}$ ***end*** $\{x{\neq}0 \wedge y{\neq}0\}$

  $]\ \{x{\neq}0 \wedge y{\neq}0\}$
- Let set $D_1 = \{x{\neq}0 \wedge y{\neq}0,\ y{=}0\}$ be the choices for $p_1{}'$.
- $x{\neq}0 \wedge y{\neq}0$ is the thread postcondition
- $y{=}0$ indicates thread 1 is blocked at the ***await*** statement.
- Similarly, let set $D_2 = \{x{\neq}0 \wedge y{\neq}0,\ x{=}0\}$ be the choices for $p_2{}'$ (the postcondition of thread 2 and the blocking condition for its ***await***).
- There are three choices for the potential deadlock predicate $r1' \wedge r2'$:
- $(x{\neq}0 \wedge y{\neq}0) \wedge (x{=}0)$, which is a contradiction.
- $(y{=}0) \wedge (x{\neq}0 \wedge y{\neq}0)$, which is a contradiction.
- $(y{=}0) \wedge (x{=}0)$, which is not a contradiction, therefore, it's a potential deadlock condition, and our program does not pass the deadlock-freedom test.
- Recall $(x{\neq}0 \wedge y{\neq}0) \wedge (x{\neq}0 \wedge y{\neq}0)$ is not a potential deadlock predicate because it says that the two threads have both completed.
- One way out of this predicament is to make the initial precondition the negation of $y{=}0 \wedge x{=}0$. Let p be $(x{\neq}0 \vee y{\neq}0)$ in

  $\{p\}$

  $[\ \{p\}$ ***await*** $y{\neq}0$ ***then*** $\{p \wedge y{\neq}0\}\ x := 1\ \{x{\neq}0 \wedge y{\neq}0\}$ ***end*** $\{x{\neq}0 \wedge y{\neq}0\}$

  $||\ \{p\}$ ***await*** $x{\neq}0$ ***then*** $\{p \wedge x{\neq}0\}\ y := 1\ \{x{\neq}0 \wedge y{\neq}0\}$ ***end*** $\{x{\neq}0 \wedge y{\neq}0\}$

  $]\ \{x{\neq}0 \wedge y{\neq}0\}$

- Let $D_1 = \{x \neq 0 \wedge y \neq 0, p \wedge y = 0\}$ and let $D_2 = \{x \neq 0 \wedge y \neq 0, p \wedge x = 0\}$.
- The three potential deadlock predicates are now contradictory
  - $(x \neq 0 \wedge y \neq 0) \wedge (p \wedge x = 0)$ (is false because of $x \neq 0 \wedge x = 0$)
  - $(p \wedge y = 0) \wedge (x \neq 0 \wedge y \neq 0)$ (is false because of $y = 0 \wedge y \neq 0$)
  - $(p \wedge y = 0) \wedge (p \wedge x = 0)$
    $\equiv ((x \neq 0 \vee y \neq 0) \wedge y = 0) \wedge ((x \neq 0 \vee y \neq 0) \wedge x = 0)$
    $\Rightarrow (x \neq 0 \wedge y = 0) \wedge (y \neq 0 \wedge x = 0)$
    $\Rightarrow F$
- (end of example 9)
- ***Example 10***: Since it has three threads, the deadlock conditions for this program are a bit more involved than for Example 9. Thread 1 has one ***await*** statement, thread 2 has two ***await*** statements, and thread 3 has no ***await*** statements.

  $[\ \ldots \{p_{11}\}$ ***await*** $B_{11}\ \ldots \{q_1\}$
  $\|\ \ldots \{p_{21}\}$ ***await*** $B_{21}\ \ldots \{p_{22}\}$ ***await*** $B_{22} \ldots \{q_2\}$
  $\|\ \ldots \{q_3\}\ ]$
- The deadlock conditions are built using the three sets
  - $D_1 = \{p_{11} \wedge \neg B_{11}, q_1\}$
  - $D_2 = \{p_{21} \wedge \neg B_{21}, p_{22} \wedge \neg B_{22}, q_2\}$
  - $D_3 = \{q_3\}$.
- Let D be the set of deadlock conditions, $D = \{r_1 \wedge r_2 \wedge r_3 \mid r_1 \in D_1, r_2 \in D_2, r_3 \in D_3\} - \{q_1 \wedge q_2 \wedge q_3\}$. Specifically, we get the following $(2 \times 3 \times 1 - 1 = 5)$ conditions:

  $D = \{(p_{11} \wedge \neg B_{11}) \wedge (p_{21} \wedge \neg B_{21}) \wedge q_3,$ — Thread 1 blocked; thread 2 blocked at 1st await
  $(p_{11} \wedge \neg B_{11}) \wedge (p_{22} \wedge \neg B_{22}) \wedge q_3,$ — Thread 1 blocked; thread 2 blocked at 2nd await
  $(p_{11} \wedge \neg B_{11}) \wedge q_2 \wedge q_3,$ — Thread 1 blocked
  $q_1 \wedge (p_2 \wedge \neg B_{21}) \wedge q_3,$ — Thread 2 blocked at 1st await
  $q_1 \wedge (p_{22} \wedge \neg B_{22}) \wedge q_3\}$ — Thread 2 blocked at 2nd await
- The program will be deadlock-free if every predicate in D is a contradiction (i.e., unsatisfiable).

J. Strengthening Deadlock Conditions

- Having all deadlock conditions be contradictory is sufficient for guaranteeing that no program execution will deadlock.
- It's not a necessary condition, however. Just because some $r \in D$ is satisfiable, that doesn't mean that there exists a program execution that can get to the corresponding deadlocked configuration.

- ***Example 11***: Here's an example of strengthening conditions so that we can prove deadlock freedom. The program is small enough for us to be able to hand-verify that it never deadlocks (by figuring out all possible interleavings).

$\{T\}\ n := 0\ ;\ [\ \textbf{await}\ n{=}0\ \textbf{then}\ n := 1\ \textbf{end}\ \|\ \textbf{wait}\ n{=}1\ ]\ \{n > 0\}$

- If we annotate the program as below, we have sequential correctness for each thread, plus the threads are interference-free:

  $\{T\}\ n := 0\ ;\ \{T\}$
  $[\ \{T\}\ \textbf{await}\ n{=}0\ \textbf{then}\ n := 1\ \textbf{end}\ \{n > 0\}$
  $\|\ \{T\}\ \textbf{wait}\ n{=}1\ \{n > 0\}$
  $]\ \{n > 0\}$

- On the other hand, we can't prove deadlock freedom. There are $2 \times 2 - 1 = 3$ deadlock conditions and all of them are satisfiable:
  - $n{\ne}0 \wedge n{\ne}1$ — Both threads blocked
  - $n{\ne}0 \wedge n > 0$ — Thread 1 blocked
  - $n > 0 \wedge n{\ne}1$ — Thread 2 blocked
- The problem here is that the proof outline's conditions are too weak. We want each deadlock condition to be logically equivalent to false, the strongest predicate.
- To make a conjunctive formula stronger, we need to strengthen its conjuncts. For a deadlock-freedom test, we have two kinds of conjuncts:
  - (postcondition of thread)
  - (precondition of ***await*** statement)$\wedge\neg($ test of ***await*** statement)
- By strengthening the postcondition of the initial assignment of $n := 0$ from true to $n{=}0$, we can strengthen the precondition of the first ***await***:

  $\{T\}\ n := 0\ ;\ \{n{=}0\}$
  $[\ \{n{=}0\}\ \textbf{await}\ n{=}0\ \textbf{then}\ \{n{=}0 \wedge n{=}0\}\ n := 1\ \textbf{end}\ \{n > 0\}$
  $\|\ \{T\}\ \ \textbf{await}\ n{=}1\ \textbf{then}\ \{n{=}1\}\ \ \textbf{skip}\ \{n{=}1\}\ \textbf{end}\ \{n > 0\}\ ]$
  $\{n > 0\}$

- The potential deadlock conditions for the proof outline above are now
  - $(n{=}0 \wedge n{\ne}0) \wedge n{\ne}1$ — Both threads blocked (contradiction)
  - $(n{=}0 \wedge n{\ne}0) \wedge n > 0$ — Thread 1 blocked (contradiction)
  - $n > 0 \wedge n{\ne}1$ — Thread 2 blocked (satisfiable)
- So two of the conditions are contradictory, but one condition is still satisfiable. To prove deadlock-freedom, we need to strengthen the conditions even more to include the state we get to when the first thread has executed and the second thread hasn't.
- Unfortunately, if we annotate the two threads as
  - $\{n{=}0\}\ \textbf{await}\ n{=}0\ \textbf{then}\ n := 1\ \textbf{end}\ \{n{=}1\}$
  - $\{n{=}1\}\ \textbf{wait}\ n{=}1\ \{n{=}1\}$
- Then the precondition of the parallel program has to be $(n{=}0) \wedge (n{=}1)$, which isn't possible. Even if it were, we'd need to be sure it follows form the strongest postcondition of $n := 0$.

$\{T\}\ n := 0\,;$
$\{n=0 \wedge n=1\}$ // ← error
$[\,\{n=0\}$ ***await*** $n=0$ ***then*** $n := 1$ ***end*** $\{n=1\}$
$\|\ \{n=1\}$ ***wait*** $n=1\ \{n=1\}$
$]\ \{n=1 \wedge n=1\}\ \{n=1\}$

- Before thread 2 runs, it sees $n=0$ or $n=1$ depending on whether thread 1 has run yet. If we use that as the precondition for thread 2, then we get $n=0 \wedge (n=0 \vee n=1)$ as the precondition for the parallel program, which works:

$\{T\}\ n := 0\,;$
$n=0 \wedge (n=0 \vee n=1)$
$[\,\{n=0\}$ ***await*** $n=0$ ***then*** $n := 1$ ***end*** $\{n=1\}$
$\|\ \{n=0 \vee n=1\}$ ***wait*** $n=1\{n=1\}$
$\{n=1 \wedge n=1\}\ \{n=1\}$

- Better still, the deadlock conditions are now all contradictions, so we have deadlock-freedom
  - $(n=0 \wedge n\neq 0) \wedge ((n=0 \vee n=1) \wedge n\neq 1)$ — Both blocked (contradiction)
  - $(n=0 \wedge n\neq 0) \wedge n=1$ — Thread 1 blocked (contradiction)
  - $n=1 \wedge ((n=0 \vee n=1) \wedge n\neq 1)$ — Thread 2 blocked (contradiction)
- Unfortunately, one of the interference freedom tests now fails:
  - Pass: $\{n=0 \wedge (n=0 \vee n=1)\}$ ***await*** $n=0$ ***then*** $n := 1$ ***end*** $\{n=0 \vee n=1\}$
  - Pass: $\{n=0 \wedge n=1\}$ ***await*** $n=0$ ***then*** $n := 1$ ***end*** $\{n=1\}$
  - Fail: $\{(n=0 \wedge n=1) \wedge n=0\}$ ***wait*** $n=1\ \{n=0\}$ — ***wait*** $n=1$ definitely doesn't preserve $n=0$
- We can solve this problem by adding an auxiliary variable to say whether or not the first thread has run and set $n=1$. (end of Example 11)