Synchronization: Await

CS 536: Science of Programming, Fall 2019

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.

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

- 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 :=` … `||` `y :=` … `||` `z :=` …`]; u = f(x, y); v :=` `g(u, z)`

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

  `[[[x :=` … `||` `y :=` …`]; u = f(x, y)` `||` `z :=` …`]; 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 :=` … `||` `y :=` … `||` *wait for* `x` and `y; u = f(x, y)` `||` `z :=` …`]; 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.
  - Could use busy wait loops like **`while`** $\neg B$ **`do skip od`** $\{B\}$ but this is 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 parallel programs (in some thread S_i in an $[S_1 \parallel S_2 \parallel \ldots]$).
- An **`await`** statement is a **conditional atomic region**. If we choose to run the thread that begins with **`await`** *B* **`then`** *S* **`end`**,
  - If *B* is true, immediately and atomically execute *S*.
    - If $\sigma(B) = \text{T}$ and $\langle S, \sigma\rangle \rightarrow^* \langle E, \tau\rangle$, then $\langle$**`await`** *B* **`then`** *S* **`end`**$, \sigma\rangle \rightarrow \langle E, \tau\rangle$.
    - No other thread can run between the **`await`** noticing $\sigma(B) = \text{T}$ and evaluating *S*, nor can another thread run while *S* executes.
      - We can see this from the $\rightarrow$ notation because each $\rightarrow$ step is a indivisible unit of execution.
      - The nondeterministic choice of which thread to run is made after one arrow ends and the next arrow begins.
    - On the other hand, when we make the nondeterministic choice of which thread to run, $\sigma(B)$ being true doesn't force the choice of executing the **`await`**. If some other thread besides this one can also run, then that choice is allowed.
  - If *B* is not true, wait until it is, then continue as above.
    - If $\sigma(B) = \text{F}$, there's no transition rule and no out arrow from the current configuration.
    - Nothing occurs and the thread is **blocked** (= the thread waits) until *B* becomes true.
    - If *B* never becomes true, we wait forever and the program **deadlocks**.
- **Example 2**: Here are some examples of configurations and transitions.
  - This **`await`** is blocked, so the configuration has no out arrows.

    $\langle$**`await`** `z` $\neq$ `0` **`then`** `x := 0; y := 1` **`end`**$, \{\texttt{z} = 0\}\rangle$.
  - This **`await`** can execute; note the body is executed atomically with the test

    $\langle$**`await`** `z` $\neq$ `0` **`then`** `x := 0; y := 1` **`end`**$, \{\texttt{z} = 5\}\rangle$
    $\rightarrow \langle E, \{\texttt{z} = 5, \texttt{x} = 0, \texttt{y} = 1\}\rangle$
  - This is a conditional statement; note that one step of execution has us jumping to the start of the true branch, so another thread could execute before we continue with `x := 0`.

    $\langle$**`if`** `z` $\neq$ `0` **`then`** `x := 0; y := 1` **`fi`**$, \{\texttt{z} = 5\}\rangle$
    $\rightarrow \langle$`x := 0; y := 1`$, \{\texttt{z} = 5\}\rangle$

Abbreviations:

- $< S >$ (S as an atomic statement) is an abbreviation for **`await`** `T` **`then`** *S* **`end`**. (It doesn't wait, but it does execute S atomically.)
- **`wait`** *B* is an abbreviation for **`await`** *B* **`then skip end`**. (Once it finishes waiting, it's done.)
  - There's a important difference between **`wait`** *B*`;` *S* and **`await`** *B* **`then`** *S* **`end`**.

 © James Sasaki, 2019

- **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*. Because of this, you can't rely on *B* being true when you execute *S*, in general.

Await Statement Proof rule

- Here's the proof rule for the **await** statement:

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

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

- Proof Outlines with **await**
  - Minimal outline: $\{ p \}$ **await** B **then** S **end** $\{ q \}$
  - Full outline: $\{ p \}$ **await** B **then** $\{ p \wedge B \}\ S^*\ \{ q \}$ **end** $\{ q \}$ where S^* is a proof outline for S.
- Weakest Precondition rule: $wp($**await** B **then** S **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: Don't let $p \rightarrow \neg B$ because it guarantees blockage:

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

E. Example: 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 3**: The rough code to solve this is

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

 © James Sasaki, 2019

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

F. Deadlock

- 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.
  - Thread 1: $\{p_1\}$ **`await`** `y` $\neq 0$ **`then`** `x :=` 1`;`…
  - Thread 2: $\{p_2\}$ **`await`** `x` $\neq 0$ **`then`** `y :=` 1`;`…
  - Say we're running in a state where `y` $= 0$ and `x` $= 0$
- **Definition**: A parallel program is **deadlocked** if it has not finished execution and all its threads that have not completed are waiting at **`await`** statements. I.e., all of the threads are either complete or blocked, and at least one thread is blocked.
- A program might deadlock under all execution paths or only certain execution paths.
- **Example 4**: The program

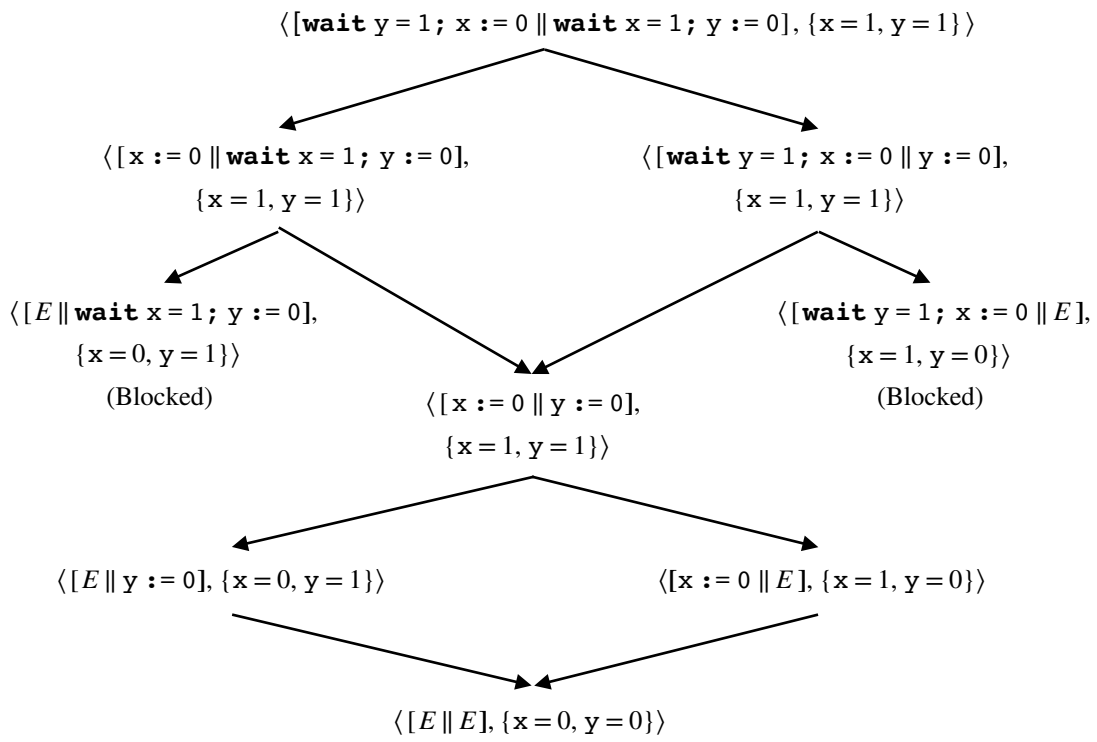
  [`await` `y` $\neq 0$ `then` `x :=` 1 `end` $\|$ `await` `x` $\neq 0$ `then` `y :=` 1 `end`]

  deadlocks iff you execute in a state where `x` and `y` are both zero.
- **Example 5**: If thread 1 sets `x :=` 0 before thread 2 evaluates its **`wait`** `x`, then thread 2 will block. (Recall **`wait`** `x` $\equiv$ **`await`** `x` **`then skip end.`**)

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

  Here's 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).

 © James Sasaki, 2019



- **Note**: There's a big difference between **wait** y = 1; x := 0 (i.e., **await** y = 1 **then skip end**; x := 0) and **await** y = 1 **then** x := 0 **end**. (See the activity questions for what happens with the example above if we change each **wait** *B*; *assignment* to **await** *B* **then** *assignment* **end**.)
- 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 $\{ p \}\ [\{ p_1 \}\ S_1^*\ \{ q_1 \} \parallel \{ p_2 \}\ S_2^*\ \{ q_2 \} \parallel \ldots \parallel \{ p_n \}\ S_n^*\ \{ q_n \}]\ \{ q \}$ 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 $r_1' \wedge r_2' \wedge \ldots \wedge r_n'$ where each r_i' is either
  - q_i, the postcondition for thread S_i or
  - $p \wedge \neg B$ where $\{ p \}$ **await** B … appears in the proof outline for thread S_i.
  - In addition, at least one of the r_i' 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 $\vDash \neg r'$ (or the equivalent $\vDash r' \rightarrow \mathtt{F}$).

 © James Sasaki, 2019

Parallelism with Deadlock Freedom

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

   …

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

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

$[S_1 \parallel S_2 \parallel \ldots \parallel S_n]$

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

where the $\{p_i\}\ S_i^*\ \{q_i\}$ are pairwise interference-free standard proof outlines and the parallel program outline is deadlock-free.

G. Example 6: An Example of Deadlock Conditions

- Let's take the program from Example 4:

  [await y ≠ 0 **then** x := 1 **end** || **await** x ≠ 0 **then** y := 1 **end]**

  and develop an annotation for it:

  {T}
  [{T} **await** y ≠ 0 **then** {y ≠ 0} x := 1 {x ≠ 0 ∧ y ≠ 0} **end** {x ≠ 0 ∧ y ≠ 0}
  || {T} **await** x ≠ 0 **then** {x ≠ 0} y := 1 {x ≠ 0 ∧ y ≠ 0} **end** {x ≠ 0 ∧ y ≠ 0}
  ] {x ≠ 0 ∧ y ≠ 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 $r_1' \wedge r_2'$:
  - $(x \neq 0 \wedge y \neq 0) \wedge (x = 0)$, 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 ≠ 0 **then** {$p \wedge$ y ≠ 0} x := 1 {x ≠ 0 ∧ y ≠ 0} **end** {x ≠ 0 ∧ y ≠ 0}
  || {p} **await** x ≠ 0 **then** {$p \wedge$ x ≠ 0} y := 1 {x ≠ 0 ∧ y ≠ 0} **end** {x ≠ 0 ∧ y ≠ 0}
  ] {x ≠ 0 ∧ y ≠ 0}

 © James Sasaki, 2019

- Let $D_1 = \{ \mathtt{x} \neq 0 \wedge \mathtt{y} \neq 0, p \wedge \mathtt{y} = 0 \}$ and let $D_2 = \{ \mathtt{x} \neq 0 \wedge \mathtt{y} \neq 0, p \wedge \mathtt{x} = 0 \}$.
- The three potential deadlock predicates are now contradictory
  - $(\mathtt{x} \neq 0 \wedge \mathtt{y} \neq 0) \wedge (p \wedge \mathtt{x} = 0)$ (is false because of $\mathtt{x} \neq 0 \wedge \mathtt{x} = 0$)
  - $(p \wedge \mathtt{y} = 0) \wedge (\mathtt{x} \neq 0 \wedge \mathtt{y} \neq 0)$ (is false because of $\mathtt{y} = 0 \wedge \mathtt{y} \neq 0$)
  - $(p \wedge \mathtt{y} = 0) \wedge (p \wedge \mathtt{x} = 0)$
    $\equiv ((\mathtt{x} \neq 0 \vee \mathtt{y} \neq 0) \wedge \mathtt{y} = 0) \wedge ((\mathtt{x} \neq 0 \vee \mathtt{y} \neq 0) \wedge \mathtt{x} = 0)$
    $\Rightarrow (\mathtt{x} \neq 0 \wedge \mathtt{y} = 0) \wedge (\mathtt{y} \neq 0 \wedge \mathtt{x} = 0)$
    $\Rightarrow \mathtt{F}$

H. Example 7: Another Example of Deadlock Conditions

- Here's an example with three threads: 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`** $B_{22} \ldots \{ q_2 \}$
  $\| \ldots \{ q_3 \} \ ]$

- Let $D_1 = \{ p_{11} \wedge \neg B_{11}, q_1 \}$, let $D_2 = \{ p_{21} \wedge \neg B_{21}, p_{22} \wedge \neg B_{22}, q_2 \}$, and let $D_3 = \{ q_3 \}$.
- Let $D = \{ r_1 \wedge r_2 \wedge r_3 \mid r_1 \in D_1, r_2 \in D_2, r_3 \in D_3 \} - \{ q_1 \wedge q_2 \wedge q_3 \}$ be the set of deadlock conditions:

  $D = \{ (p_{11} \wedge \neg B_{11}) \wedge (p_{21} \wedge \neg B_{21}) \wedge q_3,$ -- Thread 1 blocked and 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 and 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 is deadlock-free if every predicate in D is a contradiction (i.e., unsatisfiable).

I. 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.

J. Example 8: Proving Deadlock Freedom by Strengthening Conditions

- This program doesn't deadlock:

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

- On the other hand, we can't prove deadlock freedom. In fact, all $2 \times 2 - 1 = 3$ deadlock conditions are satisfiable:
  - $n \neq 0 \wedge n \neq 1$ — Both threads blocked
  - $n \neq 0 \wedge n > 0$ — Thread 1 blocked
  - $n > 0 \wedge n \neq 1$ — 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} await n = 0 then {n = 0 ∧ n = 0} n := 1 end {n > 0}
|| {T} await n = 1 then {n = 1} skip {n = 1} end {n > 0} ]
{n > 0}
```

- The potential deadlock conditions for the proof outline above are now
  - $(n = 0 \wedge n \neq 0) \wedge n \neq 1$ — Both threads blocked (contradiction)
  - $(n = 0 \wedge n \neq 0) \wedge n > 0$ — Thread 1 blocked (contradiction)
  - $n > 0 \wedge n \neq 1$ — 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} await n = 0 then n := 1 end {n = 1}`
  - `{n = 1} wait n = 1 {n = 1}`
- Then the precondition of the parallel program has to be $(n = 0) \wedge (n = 1)$, which doesn't follow from the strongest postcondition of `n := 0` and worse yet, isn't possible anyway.

```
{T} n := 0;
{n = 0 ∧ n = 1}      // ← error
[ {n = 0} await n = 0 then n := 1 end {n = 1}
|| {n = 1} wait n = 1 {n = 1}
] {n = 1 ∧ n = 1} {n = 1}
```

 © James Sasaki, 2019

- 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: $\{(n = 0 \vee n = 1) \wedge n = 0\}$ **wait** $n = 1$ $\{n = 0\}$
- We can solve this problem by adding an auxiliary variable to say whether or not the first thread has run and set $n = 1$.

 © James Sasaki, 2019

Synchronization: Await

CS 536: Science of Programming

A. Why

- It's common for one thread to wait for another thread to reach a desired state.

B. Objectives

At the end of this activity assignment you should be able to

- Draw an evaluation diagram for a parallel program that uses **`await`** and recognize any deadlocked configurations.
- List the potential deadlock predicates for a parallel program that uses **`await`**.

C. Questions

1. Let's investigate the difference between **`wait`** and **`await`**. Consider the following configurations

   $\langle$ [**`wait`** `x` $\geq$ `0; y :=` `sqrt(x)` $\|$ `x :=` `x-1`], $\sigma[\texttt{x} \mapsto 0]$ $\rangle$

   $\langle$ [**`await`** `x` $\geq$ `0` **`then`** `y :=` `sqrt(x)` **`end`** $\|$ `x :=` `x-1`], $\sigma[\texttt{x} \mapsto 0]$ $\rangle$

   a. For both configurations, what happens if we execute the right-hand thread first?

   b. Suppose we execute the first configuration and decrement `x` between the **`wait`** and `y :=` `sqrt(x)`. What evaluation sequence results? (Execute as many steps as possible.)

   c. Suppose we execute the second configuration and execute the entire **`await`** (including `y :=` `sqrt(x)`) and then decrement `x`. What evaluation sequence results? (Again, execute as many steps as possible.)

   d. We don't have to consider executing the second configuration but decrement `x` just before the **`then`** of the await statement. Why is that?

2. Let S be a parallel and suppose $\{ p \}\ S^* \{ q \}$ fails one of its deadlock tests.

   a. What do we know about the behavior of S if we run it in a $\sigma \vDash p$?

   b. Is it impossible to get a proof of correctness for the program?

3. The following program is a variant of one from the notes.

   a. Draw an evaluation diagram for this program, starting in the empty state.

   {`T`} `x :=` `1;` {`x = 1`} `y :=` `1;` {(`y = 0` $\vee$ `y = 1`) $\wedge$ (`x =  0` $\vee$ `x = 1`)}
   [{`y = 0` $\vee$ `y = 1`} **`await`** `y = 1` **`then`** {`y = 1`} `x :=` `0` **`end`** {`x = 0`}
   $\|$ {`x =  0` $\vee$ `x = 1`} **`await`** `x = 1` **`then`** {`x = 1`} `y :=` `0` **`end`** {`y = 0`}
   ] {`x = 0` $\wedge$ `y = 0`}

   b. Does the program deadlock always, sometimes, or never?

   c. What are the deadlock conditions for this program? Which (if any) are contradictory? Can all these conditions actually occur at runtime?

 © James Sasaki, 2019

4. Give the set of deadlock conditions for the proof outline below. (Assume that S_1, … U_1 do not include **await** statements.)

$[\ \{p_1\}\ S_1;\ \{p_2\}$ **await** B_1 **then** S_2 **end** $\{p_3\}$
$\|\ \{q_1\}$ **await** C_1 **then** T_1 **end;** $\{q_2\}$ **await** C_2 **then** T_2 **end** $\{q_3\}$
$\|\ \{r_1\}\ U_1\ \{r_2\}\ ]$

 © James Sasaki, 2019

Solution to Activity 23 (Synchronization: Await)

1. (**wait** vs **await**)

   a. With both configurations the `x := x-1` thread executes first, we get deadlock because `x` is now `-1` and there's not thread that can execute and make `x` nonnegative.

   b. $\langle$ [**wait** `x` $\geq$ `0; y := sqrt(x)` $\|$ `x := x-1`], $\sigma[\mathtt{x} \mapsto 0]$ $\rangle$
   $\rightarrow \langle$ [`y := sqrt(x)` $\|$ `x := x-1`], $\sigma[\mathtt{x} \mapsto 0]$ $\rangle$
   $\rightarrow \langle$ [`y := sqrt(x)` $\|$ E], $\sigma[\mathtt{x} \mapsto -1]\rangle$
   $\rightarrow \langle \bot_e, \sigma[\mathtt{x} \mapsto -1]\rangle$

   c. $\langle$ [**await** `x` $\geq$ `0` **then** `y := sqrt(x)` **end** $\|$ `x := x-1`], $\sigma[\mathtt{x} \mapsto 0]$ $\rangle$
   $\rightarrow \langle$ [E $\|$ `x := x-1`], $\sigma[\mathtt{x} \mapsto 0][\mathtt{y} \mapsto 0]$ $\rangle$
   // because $\langle$ `y := sqrt(x)`, $\sigma[\mathtt{x} \mapsto 0]$ $\rangle \rightarrow^* \langle E, \sigma[\mathtt{x} \mapsto 0][\mathtt{y} \mapsto 0]$ $\rangle$
   $\rightarrow \langle$ [E $\|$ E], $\sigma[\mathtt{x} \mapsto -1][\mathtt{y} \mapsto 0]$ $\rangle$

   d. If the **await** statement notices `x` ≥ 0, then it immediately and atomically executes S, so there's no possibility of another thread executing just before the **await** statement's **then**.

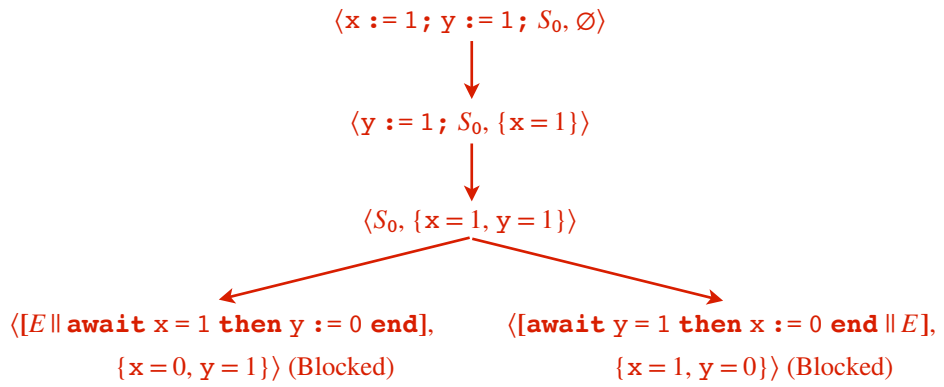
2. (Parallel program fails a deadlock check.)

   a. Passing all the deadlock-freedom tests is sufficient to guarantee deadlock freedom, but passing all tests is not a necessary condition. A deadlock test says nothing about whether the program can ever actually achieve the deadlocking configuration, so if we run the program in a state that satisfies the precondition, we might get deadlock or we might not. We might deadlock along every execution path or along only one possible execution path or along no execution paths.

   b. It may or may not be possible to prove deadlock freedom. It's possible that if the proof outline's internal conditions were modified, we'd be able to prove deadlock freedom. (See Example 8.) Or it might not be possible to prove deadlock freedom because the program really can deadlock. In that case we might be able to get deadlock freedom if we modify the initial precondition. Or we might have to actually change the program.

3. Let $S_0 \equiv$ [**await** `y` $= 1$ **then** `x := 0` **end** $\|$ **await** `x` $= 1$ **then** `y := 0` **end**]

   a. The evaluation graph for $\langle S_0, \varnothing\rangle$:



 © James Sasaki, 2019

b. There are two execution paths; both deadlock.

c. Let $D_1 = \{(y = 0 \vee y = 1) \wedge y \neq 1, x = 0\}$

Let $D_2 = \{(x = 0 \vee x = 1) \wedge x \neq 1, y = 0\}$

There are three potential deadlock conditions; none are contradictions.

- $((y = 0 \vee y = 1) \wedge y \neq 1) \wedge (y = 0)$ [Thread 1 blocked]
  $\Leftrightarrow y = 0$, which is satisfiable
- $(x = 0) \wedge ((x = 0 \vee x = 1) \wedge x \neq 1)$ [Thread 2 blocked]
  $\Leftrightarrow x = 0$, which is satisfiable
- $((y = 0 \vee y = 1) \wedge y \neq 1) \wedge ((x = 0 \vee x = 1) \wedge x \neq 1)$ [Both threads blocked]
  $\Leftrightarrow y = 0 \wedge x = 0$, which is satisfiable. (Note $y = 0 \wedge x = 0$ doesn't actually occur during execution.)

4. First, let's look at the sets of conditions to choose from:

- $D_1 = \{p_2 \wedge \neg B_1, p_3\}$ Thread 1: Blocked at its **`await`**, done
- $D_2 = \{q_1 \wedge \neg C_1, q_2 \wedge \neg C_2, q_3\}$ Thread 2: Blocked at **`await`** 1, blocked at **`await`** 2, or done
- $D_3 = \{r_2\}$ Thread 3: Done

We form each deadlock condition as the conjunction of three predicates, one from each set. There are five deadlock conditions, since $p_3 \wedge q_3 \wedge r_2$ is not a deadlock condition:

- $(p_2 \wedge \neg B_1) \wedge (q_1 \wedge \neg C_1) \wedge (r_2)$ member 1 of D_1, member 1 of D_2, only member of D_3
- $(p_2 \wedge \neg B_1) \wedge (q_2 \wedge \neg C_2) \wedge (r_2)$ member 1 of D_1, member 2 of D_2, only member of D_3
- $(p_2 \wedge \neg B_1) \wedge (q_3) \wedge (r_2)$ member 1 of D_1, member 3 of D_2, only member of D_3
- $(p_3) \wedge (q_1 \wedge \neg C_1) \wedge (r_2)$ member 2 of D_1, member 1 of D_2, only member of D_3
- $(p_3) \wedge (q_2 \wedge \neg C_2) \wedge (r_2)$ member 2 of D_1, member 2 of D_2, only member of D_3
- $p_3 \wedge q_3 \wedge r_2$ (isn't a d.l. condition) member 2 of D_1, member 3 of D_2, only member of D_3

 © James Sasaki, 2019