Await and Deadlocks

CS 536: Science of Programming, Spring 2022

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 support 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)

© James Sasaki, 2021
Ed. Stefan Muller, 2022

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.
  - Busy wait loops like *while ¬e {skip} {e}* work but are wasteful.
- *Syntax*: *await e then{s}* where *e* 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 e then{s}*, 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 e then {s}*,
  - If *e* 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 *e* is false, we *block*: We wait until *e* is true. Instead, we nondeterministically choose between the other threads and execute one. (See case 3 of Example 2.)
- An *await* is similar to an atomic *if-then* statement, but not identical.
  - With *< if e then {s} else {skip} >*, if *e* is false, we execute *skip* and complete. (See case 4, Example 2.)
  - With *await e then {s}*, if *e* is false, nothing happens until *e* becomes true. (See the note with case 3, Example 2.)

Example 2:

- (See the discussion above)
  - Case 1: $\langle [\text{await } b \text{ then}\{ x := x+x \} \| x:=1], \{ b=T, x=2 \} \rangle$
    $\rightarrow \langle [\text{await } b \text{ then}\{ x := x+x \} \| skip], \{ b=T, x=1 \} \rangle$.
  - Case 2: $\langle [\text{await } b \text{ then}\{ x := x+x \} \| x:= 1], \{ b=T, x=2 \} \rangle \rightarrow \langle [skip \| x:=1], \{b=T, x=4\} \rangle$.
    (This is the only transition that executes the *await*.)

© James Sasaki, 2021
Ed. Stefan Muller, 2022

- Case 3: $<[\text{await } b \text{ then } \{s\} \parallel x := 1], \{b = F\}> \rightarrow <[\text{await } b \text{ then } \{s\} \parallel \text{skip}], \{b = F, x = 1\}>$.
  (The second configuration is blocked, with no other thread available to unblock it.)
- Case 4: $<[\text{if } b \text{ then } \{x := 0\} \text{ else } \{\text{skip}\} \parallel s'], \{b = F\}>$
  $\rightarrow <[\text{skip} \parallel s'], \{b = F, x = 0\}>$.

- *Example 3*: In the introduction, we looked at a situation where we want to wait for some calculations to finish before stating others.
  •*[x := ... || y := ... || wait for x and y; u = f(x, y) || z := ...]; v := g(u, z)*
  can be implemented using

```
x_done := F;  y_done:= F;
[ x := ... ; x_done := T || y := ... ; y_done := T
|| await x_done ∧ y_done then {u := f(x, y)} || z := ... ];
v := g(u, z);
```

E. *await, wait, if, and < s >*

The Abbreviations < s > and wait e

- With *await e then {s}*, there are two simple cases: When *e* is trivial and when *s* is trivial.
- *Definition*: We can redefine *< s >* to stand for *await T then {s}*. When the test is trivial, 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 e ≡ await e then {skip}*. When the body is trivial, we simply wait; when *e* is true, execution is complete.
- There's a important difference between *wait* e*; s* and *await e then {s}* (and even *wait e; <s>*).
  - With *await e then {s}*, once *e* is true, we immediately atomically execute *s*, so no other statement can interleave between the test and running *s*. Therefore *s* can rely on e being true when it starts executing.
  - *wait e; s* means *await e then {skip}; s*, so it allows another thread to be executed after the *wait* but before running *s*.
  - This is still the case with *wait e; <s>*. We can't interleave anything with *s*, but we can execute another thread between the test and *s*.
  - This is a common source of bugs in real parallel programs!

© James Sasaki, 2021
Ed. Stefan Muller, 2022

F. Await Statement Proof Rule and Outlines

- The proof rule for the *await* statement is similar to an *if*, but there's no false clause.

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

$$\frac{\{p \wedge e\}\, s\, \{q\}}{\{p\}\, \mathit{await}\ e\ \mathit{then}\ \{s\}\ \{q\}}\ (\mathit{Await})$$

- *Minimal Proof Outline*: {*p*} *await e then {s}* {*q*}
- *Full Proof Outline*: {*p*} *await e then* {{*p* ∧ *e*} *s** {*q*}} {*q*} where *s** is a full proof outline for *s*.
- *Weakest Preconditions*: *wp(await e then {s}, q)* ≡ $e \rightarrow wp(s, q)$.
  - This guarantees *{e → wp(s, q)} await e then {{wp(s, q)} s* {q}}{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 {                          // Producer
    created := Create();
    await NotFull(buffer) then {
        BufferAdd(buffer, created)
    }
 }
|| while ¬done {                        // Consumer
    await NotEmpty(buffer) then
        removed := BufferRemove(buffer);
    };
    Consume(removed)
    }
]
```

- 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.

© James Sasaki, 2021
Ed. Stefan Muller, 2022

H. Deadlock

Blocked Threads; Deadlock

- Recall that $\langle [\textit{await } e \textit{ then } \{s\}; \ldots \| \ldots], \sigma \rangle$ is blocked (must wait) if $\sigma(e) = F$.
  - If some other thread can make e true, then the await may eventually unblock.
  - E.g., $\langle [\textit{await } x > y \textit{ then } \{s\}; \ldots \| \ldots; x := y+1; \ldots], \sigma \rangle$ 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., $\langle [\textit{await } e \textit{ then } \{s\} \textit{ end}; \ldots \| E], \sigma \rangle$ 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.
- *Example 6*: If $A \equiv \textit{await } x \geq 0 \ldots$, 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, $\langle [\textit{await } x \geq 0 \ldots \| \textit{skip}], \sigma[x \mapsto -1] \rangle$ is also deadlocked.

- Threads can block themselves (trivial example: $\textit{await false then } \{s\}$).
- 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: $\{p_1\}\ \textit{await } y \neq 0 \textit{ then } \{x := 1\}; \ldots$
  - Thread 2: $\{p_2\}\ \textit{await } x \neq 0 \textit{ then } \{y := 1\}; \ldots$
- A program might deadlock under all execution paths or only certain execution paths.

- *Example 7*: The program

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

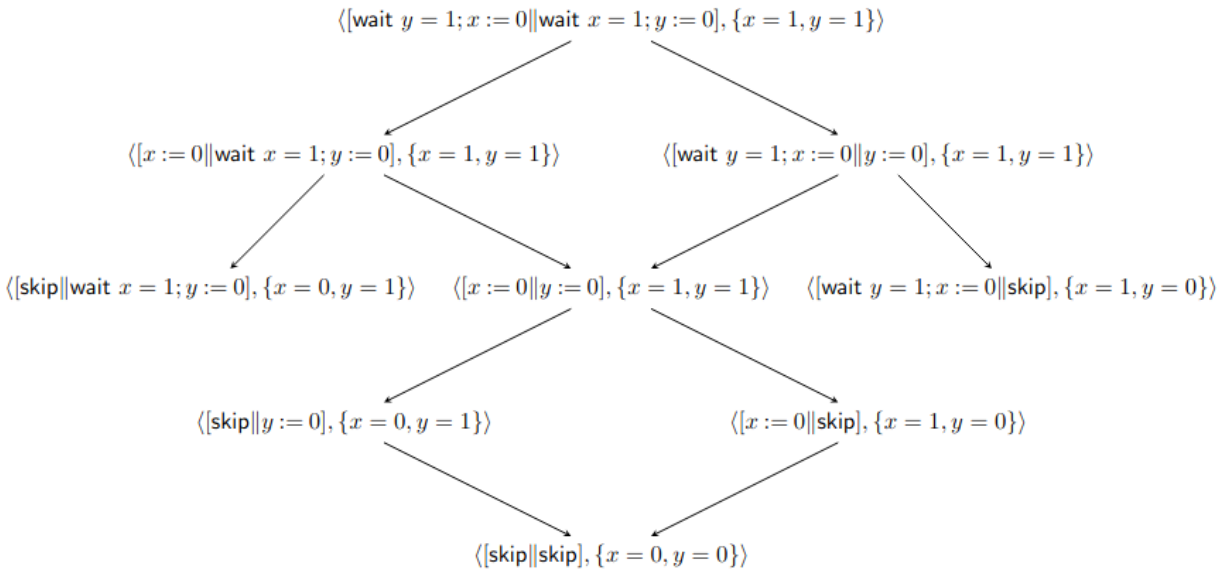
  deadlocks iff you execute in a state where x and y are both zero.
- *Example 8*: If thread 1 sets $x := 0$ before thread 2 evaluates its *wait x*, then thread 2 will block. (Recall $\textit{wait } x \equiv \textit{await } x \textit{ then } \{\textit{skip}\}$.)

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

  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).

© James Sasaki, 2021
Ed. Stefan Muller, 2022

•



- Deadlock is another way for a program to fail to terminate, so M([*wait* y=1; x := 0 ‖ *wait* x=1; y := 0], {*x=1*, *y=1*}) = {{x = 1, y = 1}, $\perp_d$}
- 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_k' is either
    - q_k, the postcondition for thread s_k or
    - $p \wedge \neg e$ where $\{p\}$ *await* e ... 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 $\models r' \rightarrow F$).

© James Sasaki, 2021
Ed. Stefan Muller, 2022

Parallelism with Deadlock Freedom

$$\frac{\lceil p_i \rceil s_i^* \lceil q_i \rceil \qquad \lceil p_i \rceil s_i^* \lceil q_i \rceil \text{ are pairwise interference-free and program is deadlock-free}}{\lceil p_1 \wedge p_2 \wedge \ldots p_n \rceil \ \lceil s_1 \parallel \ldots \parallel s_n \rceil \ \lceil q_1 \wedge q_2 \wedge \ldots \wedge q_n \rceil}$$

I. Examples of Deadlock Conditions

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

  [await y≠0 then {x := 1} || await x≠0 then {y := 1}]

  and develop an annotation for it:

  {T}
  [{T} await y≠0 then {{y≠0} x := 1 {x≠0 ∧ y≠0}} {x≠0 ∧ y≠0}
  || {T} await x≠0 then {{x≠0} y := 1 {x≠0 ∧ y≠0}} {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 ∧ y≠0} x := 1 {x≠0 ∧ y≠0} end {x≠0 ∧ y≠0}
  || {p} await x≠0 then {p ∧ x≠0} y := 1 {x≠0 ∧ y≠0} end {x≠0 ∧ y≠0}
  ] {x≠0 ∧ y≠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 $y = 0 \wedge y \neq 0$)

© James Sasaki, 2021
Ed. Stefan Muller, 2022

- $(p \wedge y=0) \wedge (p \wedge x=0)$
  $\equiv ((x\neq 0 \vee y\neq 0) \wedge y=0) \wedge ((x\neq 0 \vee y\neq 0) \wedge x=0)$
  $\Rightarrow (x\neq 0 \wedge y=0) \wedge (y\neq 0 \wedge x=0)$
  $\Rightarrow F$
- (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* $e_{11} \ldots \{q_1\}$
  $\| \ldots \{p_{21}\}$ *await* $e_{21} \ldots \{p_{22}\}$ *await* $e_{22} \ldots \{q_2\}$
  $\| \ldots \{q_3\}\ ]$
- The deadlock conditions are built using the three sets
  - $D_1 = \{ p_{11} \wedge \neg e_{11}, q_1\}$
  - $D_2 = \{p_{21} \wedge \neg e_{21}, p_{22} \wedge \neg e_{22}, q_2\}$
  - $D_3 = \{ q_3 \}$.
- 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).

© James Sasaki, 2021
Ed. Stefan Muller, 2022