Await and Deadlocks

CS 536: Science of Programming, Spring 2023

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 practice assignment you should be able to

- Describe informally what deadlock is and recognize runtime configurations that are deadlocked.
- If a program fails one or more of its deadlock-freedom tests, what can happen at runtime?
- 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

The final exam will only include questions based on the problems below.

1. Informally, what is deadlock and when does it occur? Say $\langle\ [S_1; \ldots \parallel S_2 \parallel E], \sigma\ \rangle$ is deadlocked, and $S_1 \equiv$ ***await*** B ***then*** S ***end***. What property does σ have? What kind of statement can S_2 be? Can $S_2 \equiv E$ here?

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

   $C_1 = \langle\ [$ ***wait*** $x \geq 0\ ;\ y := \text{sqrt}(x) \parallel x := x - 1\ ],\ \sigma[x \mapsto 0]\ \rangle$

   $C_2 = \langle\ [$ ***await*** $x \geq 0$ ***then*** $y := \text{sqrt}(x) \parallel x := x - 1\ ],\ \sigma[x \mapsto 0]\ \rangle$

   Let's write $\rightarrow_1$ and $\rightarrow_2$ to mean we evaluate the first and second thread respectively.

   a. $C_1 \rightarrow_2$??? Can we continue this execution path? If so, how?

   b. $C_2 \rightarrow_2$??? Can we continue this execution path? If so, how?

   c. $C_1 \rightarrow_1$??? $\rightarrow_2$??? Can we continue this execution path? If so, how?

   d. $C_2 \rightarrow_1$??? $\rightarrow_2$??? Can we continue this execution path? If so, how?

   e. Why is $C_2 \rightarrow \langle\ [y := \text{sqrt}(x) \parallel x := x\text{-}1]\ \rangle$ incorrect?

3. Let S be a parallel program 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?

4. 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?

5. 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*** $\{q_3\}$
   $\|\ \{r_1\}\ U_1\ \{r_2\}\ ]$

Solution to Practice 27 (Synchronization: Await)

1. Informally, deadlock is the inability for execution of a parallel program to continue because of waiting. It occurs when at least one thread is waiting at an ***await*** statement and the other threads have either finished execution or are waiting at ***await*** statements themselves. In the sample, the ***await*** statement can't continue because $\sigma(B) = F$. S_2 can be an ***await*** statement that is similarly unable to continue ($S_2 \equiv$ ***await*** B' ***then*** S' ***end*** where $\sigma(B') = F$). S_2 can't be any other kind of statement because then thread 2 would not be waiting. We can have $S_2 \equiv E$, which means S_2 has finished execution.

2. (***wait*** vs ***await***)
   a. $C_1 = \langle [$***wait*** $x \geq 0 ;\ y := \text{sqrt}(x) \| x := x - 1\ ],\ \sigma[x \mapsto 0] \rangle$
      $\rightarrow_2 \langle [$***wait*** $x \geq 0 ;\ y := \text{sqrt}(x) \| E\ ],\ \sigma[x \mapsto -1] \rangle$
      and execution is deadlocked because $x \geq 0$ is not satisfied by $\sigma[x \mapsto -1]$.
   b. Similarly, $C_2 = \langle [$***await*** $x \geq 0$ ***then*** $y := \text{sqrt}(x)$ ***end*** $\| x := x - 1\ ],\ \sigma[x \mapsto 0] \rangle$
      $\rightarrow_2 \langle [$***await*** $x \geq 0$ ***then*** $y := \text{sqrt}(x)$ ***end*** $\| E\ ],\ \sigma[x \mapsto -1] \rangle$
      and execution is deadlocked for the same reason.
   c. $C_1 = \langle [$***wait*** $x \geq 0 ;\ \ y := \text{sqrt}(x) \| x := x - 1\ ],\ \sigma[x \mapsto 0] \rangle$
      $\rightarrow_1 \langle [y := \text{sqrt}(x) \| x := x - 1\ ],\ \sigma[x \mapsto 0] \rangle$
      $\rightarrow_2 \langle [y := \text{sqrt}(x) \| E\ ],\ \sigma[x \mapsto -1] \rangle$
      which generates a runtime error, i.e., $\rightarrow_1 \langle [E \| E],\ \perp_e \rangle$.
      (The first transition doesn't execute $y := \text{sqrt}(x)$ because ***wait*** $x \geq 0$ means ***await*** $x \geq 0$ ***then skip end***, so execution of ***wait*** $x \geq 0$ is complete once $x \geq 0$.

d. $C_2 = \langle$ *[**await** x ≥ 0 **then** y := sqrt(x) **end** || x := x–1], σ[x ↦ 0]* $\rangle$
$\rightarrow_1 C_2 = \langle [\, E \parallel x := x - 1 \,], \sigma [\, x \mapsto 0 \,] [\, y \mapsto 0 \,] \rangle$
$\rightarrow_2 \langle$ *[*E*], σ[y ↦ 0] [x ↦ –1]* $\rangle$.
(Note the first transition goes from ***await*** ... directly to E.)

e. It's wrong to say $C_2 \rightarrow \langle [\, y := \mathrm{sqrt}(x) \parallel x := x - 1 \,] \rangle$ because ***await*** immediately and atomically executes its body after the test succeeds.

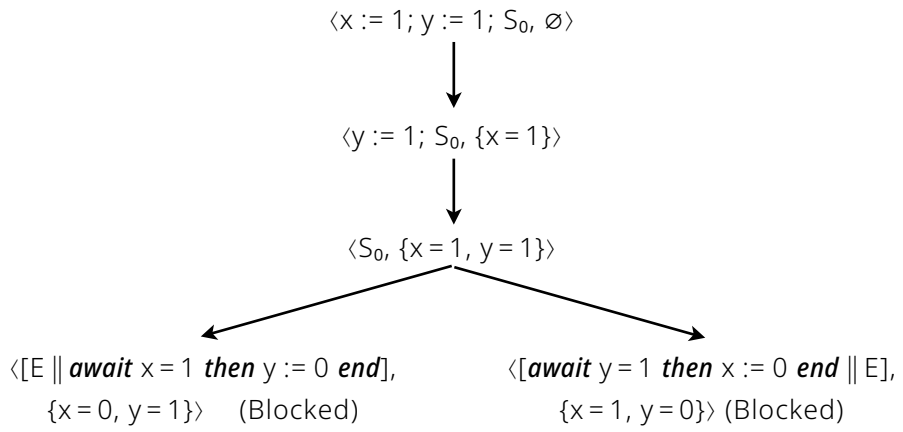
3. (Parallel program fails a deadlock check.)

a. Passing all the deadlock-freedom tests is sufficient to guarantee deadlock freedom, but passing all tests is not a necessary condition. A failed 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 are modified, we'd be able to prove deadlock freedom. (See Example 11.) 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.

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

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



b. There are two execution paths; both deadlock.

c. Let $D_1 = \{(y = 0 \vee y = 1) \wedge y \neq 1,\ x = 0\}$ Thread 1 is blocked or done
Let $D_2 = \{(x = 0 \vee x = 1) \wedge x \neq 1,\ y = 0\}$ Thread 2 is blocked or done

We form the deadlock conditions by taking joining one predicate each from D_1 and D_2, with the exception that $x = 0 \wedge y = 0$ is not a deadlock condition because it means execution has completed. There are three potential deadlock conditions; none are contradictions.

- Thread 1 blocked:
  - $((y = 0 \vee y = 1) \wedge y \neq 1) \wedge (y = 0) \Leftrightarrow y = 0$, which is satisfiable
- Thread 2 blocked:
  - $(x = 0) \wedge ((x = 0 \vee x = 1) \wedge x \neq 1) \Leftrightarrow x = 0$, which is satisfiable.
- Both threads blocked:

  $((y = 0 \vee y = 1) \wedge y \neq 1) \wedge ((x = 0 \vee x = 1) \wedge x \neq 1) \Leftrightarrow y = 0 \wedge x = 0$, which is satisfiable.

  (Note $y = 0 \wedge x = 0$ doesn't actually occur during execution.)

5. 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 1st ***await***, 2nd ***await***, 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.

   (Add "and thread 3 complete" to all 5 lines below.)
   - $(p_2 \wedge \neg B_1) \wedge (q_1 \wedge \neg C_1) \wedge (r_2)$ — Thread 1 blocked, thread 2 blocked at 1st ***await***
   - $(p_2 \wedge \neg B_1) \wedge (q_2 \wedge \neg C_2) \wedge (r_2)$ — Thread 1 blocked, thread 2 blocked at 2nd ***await***
   - $(p_2 \wedge \neg B_1) \wedge (q_3) \wedge (r_2)$ — Thread 1 blocked, thread 2 complete
   - $(p_3) \wedge (q_1 \wedge \neg C_1) \wedge (r_2)$ — Thread 1 complete, thread 2 blocked at 1st ***await***
   - $(p_3) \wedge (q_2 \wedge \neg C_2) \wedge (r_2)$ — Thread 1 complete, thread 2 blocked at 2nd ***await***

