Synchronization and Await Statement

- In general, if we cannot avoid the interaction between threads, we hope the interaction is good and won't cause any problems. To support good interaction between threads, we often have one thread wait for another one. For example, thread 1 should wait until thread 2 finishes executing a certain block of code.
- 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.

1. Here is a concrete 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.

   $$[x \coloneqq \cdots \parallel y \coloneqq \cdots \parallel z \coloneqq \cdots];\ u \coloneqq f(x,y); v \coloneqq g(u,z)$$

   But on the other hand, we cannot nest parallel programs, so it is not legal to write:

   $$\big[[x \coloneqq \cdots \parallel y \coloneqq \cdots];\ u \coloneqq f(x,y) \parallel z \coloneqq \cdots\big];\ v \coloneqq g(u,z)$$

   It is natural that we want u and z being calculated at the same time, but we need to wait for x and y in the thread of calculating u. We need something like:

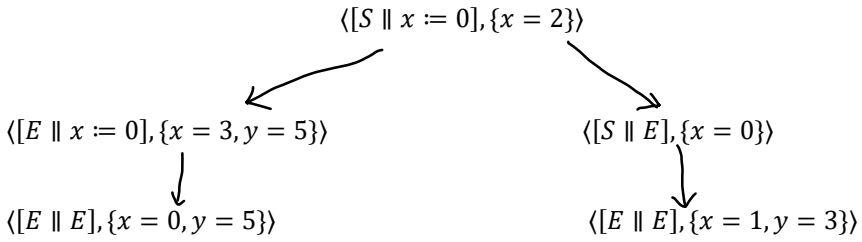
   $$[x \coloneqq \cdots \parallel y \coloneqq \cdots \parallel \text{wait for } x \text{ and } y;\ u \coloneqq f(x,y) \parallel z \coloneqq \cdots]; v \coloneqq g(u,z)$$

- Here we introduce a new statement await: **await** B **then** S **end,** where B is a Boolean expression and S is a statement.
  - S is not allowed to have loops, **await** statements, or atomic regions.
  - **await** statements can only appear in threads of parallel programs.

- An **await** statement itself is a conditional atomic region.
  - We randomly choose between all the available threads. In other words, there's no insistence that we must check the **await** before trying other threads.
    - For example, it is possible that $\langle[\textbf{await } B \textbf{ then } x \coloneqq x + x \textbf{ end} \parallel x \coloneqq 1]\ \{B = T, x = 2\}\rangle \rightarrow \langle[\textbf{await } B \textbf{ then } x \coloneqq x + x \textbf{ end} \parallel E]\ , \{B = T, x = 1\}\rangle$.
  - If we choose the thread that begins with **await** B **then** S **end**, and B is true, then immediately jump to S and execute all of it.
    - For example, in $\langle[\textbf{await } B \textbf{ then } x \coloneqq x + x \textbf{ end} \parallel x \coloneqq 1\ ], \{B = T, x = 2\}\rangle \rightarrow \langle[E \parallel x \coloneqq\ 1], \{b = T, x = 4\}\rangle$ is the only next configuration if we chose the first thread; there is no chance for $x \coloneqq 1$ to jump in after $B = T$ is evaluated.
  - If we choose the thread that begins with **await** B **then** S **end**, and B is false, then we block: We wait until B is true. Instead, we randomly choose among other threads and execute it.
    - For example, $\langle[\textbf{await } B \textbf{ then } S \textbf{ end} \parallel x \coloneqq 1]\ , \{B = F\}\ \rangle \rightarrow \langle[\textbf{await } B \textbf{ then } S \textbf{ end} \parallel E], \{B = F, x = 1\ \}\rangle$ and this is the only possible next step since the first thread will be blocked until B becomes true.

- An **await** is like an atomic **if** $-$ **then** statement, but not identical.
  - With $<$ **if** B **then** S **else skip fi** $>$, if B is false, we execute **skip** and complete the **if** $-$ **fi**.
  - With **await** B **then** S **end**, if B is false, nothing happens until B becomes true.

1. Show the evaluation graph of $\langle[S \parallel x \coloneqq 0],\ \{x = 2\}\rangle$, where $S \equiv \textbf{await } x \geq 0 \textbf{ then } x \coloneqq x + 1; y \coloneqq x + 2 \textbf{ end}$.



2. Let's consider some special situations in an await statement **await** B **then** S **end**.
   a. What if $B \equiv T$?
      We will always execute S atomically, thus we can optimize **await** T **then** S **end** to $< S >$.

   b. What if $S \equiv$ **skip**?
      [Definition] **wait** $B \equiv$ **await** B **then skip end.**

   c. What if $B \equiv F$?
      Then the thread will be blocked at **await** F **then** S **end** foreve .

3. What is the 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 be executed between the test B and running S.
   - With **wait** $B; < S >$, it allows another thread to be executed after the **wait** and before running S.

Rules and Triples for Await Statement

- Await Rule:
  1. $\{p \wedge B\}\ S\ \{q\}$
  2. $\{p\}$ **await** B **then** S **end** $\{q\}$ await 1

  - The await rule is very similar to conditional rule 1 with only the true branch.

4. What if $p \wedge B$ is a contradiction and $\{p \wedge B\}\ S\ \{q\}$ is trivially true?
   - It is not a problem, since **await** B **then** S **end** doesn't guarantee the execution of S either. For example, if B is evaluated to F then we get $\{p\}$ **await** F **then** S **end** $\{q\}$ is provable from $\{p \wedge F\}\ S\ \{q\}$; this is totally okay since we don't need to prove that $\{p\}$ **await** F **then** S **end** $\{q\}$ won't block.

- We represent await rule in the proof outline as follows:
  $\{p\}$ **await** B **then** $\{p \wedge B\}\ S\ \{q\}$ **end** $\{q\}$

- $wp($**await** B **then** S **end**$, q) \equiv B \rightarrow wp(S, q)$

Deadlock

- A parallel program is deadlocked if it has not finished execution and there's no possible evaluation step to take.
  - In example 2.c, we can see that **await** F **then** S **end** can block a thread forever, thus if this await statement is one thread then this parallel program is deadlocked.
  - Deadlock can be considered as a runtime error ($\perp_e$) .

5. When are the following programs deadlocked?
   a. $[A \parallel A]$ where $A \equiv \textbf{await}\ x \geq 0\ \textbf{then}\ S\ \textbf{end}$.
      If we execute it in a state σ with $\sigma(x) < 0$.

   b. $[\textbf{await}\ y \neq 0\ \textbf{then}\ x := 1\ \textbf{end} \parallel \textbf{await}\ x \neq 0\ \textbf{then}\ y := 1\ \textbf{end}]$
      If we execute it in a state σ with $\sigma(x) = 0$ and $\sigma(y) = 0$.

   c. $x := 1; y := 1; [\textbf{wait}\ y \neq 0;\ x := 0 \parallel \textbf{wait}\ x \neq 0; y := 0]$
      Remind that, $\textbf{wait}\ y \neq 0 \equiv \textbf{await}\ y \neq 0\ \textbf{then skip end}$. If we execute $x := 0$ before the test $x \neq 0$ in thread 2, then thread 2 will be blocked and the program is deadlocked; similarly, if we execute $y := 0$ before the test $y \neq 0$ in thread 1, then thread 1 will be blocked and the program is deadlocked.

- It is obvious that we want to know whether a program is deadlock-free or not. Like the test for "disjoint program", "disjoint condition" and "interference free", we also have a test for "deadlock free". Again, this test is static and too strong: if a parallel program past the test, then it must be deadlock-free; otherwise, there might be deadlocks during the execution.

- For program outline $\{p_1 \wedge p_2 \wedge \ldots \wedge p_n\}\ [\{p_1\}\ S_1^*\ \{q_1\} \parallel \{p_2\}\ S_2^*\ \{q_2\} \parallel \cdots \parallel \{p_n\}\ S_n^*\ \{q_n\}]\ \{q_1 \wedge q_2 \wedge \ldots \wedge q_n\}$, where S_k^* is the full proof outline for program S_k, we define one of its potential deadlock condition as $r_1 \wedge r_2 \wedge \ldots \wedge r_n$, where each r_k is one of the follows:
  - $p \wedge \neg B$, if $\{p\}\ \textbf{await}\ B\ \ldots$ appears in S_k^*;
  - q_k, otherwise.
  - In addition, at least one r_k must be $p \wedge \neg B$. If $r_1 \wedge r_2 \wedge \ldots \wedge r_n \equiv q_1 \wedge q_2 \wedge \ldots \wedge q_n$, then we didn't look at any possible blocks for an await statements, so we don't consider $q_1 \wedge q_2 \wedge \ldots \wedge q_n$ as a potential deadlock condition.

- A program outline is deadlock-free if every one of its potential deadlock conditions is unsatisfiable; in other words, every one of its potential deadlock conditions is a contradiction.

- With the definition of deadlock-free, we can expand the domain of the parallelism rule to programs with await statements now.
  Parallelism Rule with shared variables and await statements:
  If for $k = 1 \ldots n$, we have $\{p_k\}\ S_k^*\ \{q_k\}$ are all interference free proof outlines, and the proof outline is deadlock-free:

  $1\ \{p_1\}\ S_1^*\ \{q_1\}$
  $2\ \{p_2\}\ S_2^*\ \{q_2\}$
  ...
  $n\ \{p_n\}\ S_n^*\ \{q_n\}$
  $n+1\ \{p_1 \wedge p_2 \wedge \ldots \wedge p_n\}\ [S_1^* \parallel S_2^* \parallel \cdots \parallel S_n^*]\ \{q_1 \wedge q_2 \wedge \ldots \wedge q_n\}$ parallelism $1,2, \ldots, n$

6. The following program $\{T\}[\textbf{await}\ y \neq 0\ \textbf{then}\ x := 1\ \textbf{end} \parallel \textbf{await}\ x \neq 0\ \textbf{then}\ y := 1\ \textbf{end}]\{q\}$ is from example 5.b. We use the following proof outlines for its two threads:

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

   Is this program outline deadlock-free?

   - We need test the following potential deadlock conditions:
     - $y = 0 \wedge (x \neq 0 \wedge y = 1)$ # r_1 is "$p \wedge \neg B$"
     - $(x = 1 \wedge y \neq 0) \wedge x = 0$ # r_2 is "$p \wedge \neg B$"

- $y = 0 \wedge x = 0$

(Note that, we don't test $(x = 1 \wedge y \neq 0) \wedge (x \neq 0 \wedge y = 1)$.)

- Among the above three conditions, the first two are contradictions and the last one is not. So, this program outline is not deadlock-free.

a. What if we only run the above program only when $x \neq 0 \vee y \neq 0$?
In this case, the precondition for both threads become $p \equiv x \neq 0 \vee y \neq 0$:

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

And the proof outline of this program becomes:

$\{p\}$ $[\{p\}$ **await** $y \neq 0$ **then** $\{p \wedge y \neq 0\}\ x \coloneqq 1\ \{x = 1 \wedge y \neq 0\}$ **end** $\{x = 1 \wedge y \neq 0\}$
$\quad \| \{p\}$**await** $x \neq 0$ **then** $\{p \wedge x \neq 0\}\ y \coloneqq 1\ \{x \neq 0 \wedge y = 1\}$ **end** $\{x \neq 0 \wedge y = 1\}]\ \{q\}$

- We need test the following potential deadlock conditions:
  - $(p \wedge y = 0) \wedge (x \neq 0 \wedge y = 1)$, which is a contradiction since $y = 0 \wedge y = 1$
  - $(x = 1 \wedge y \neq 0) \wedge (p \wedge x = 0)$, which is also a contradiction.
  - $(p \wedge y = 0) \wedge (p \wedge x = 0) \Leftrightarrow (x \neq 0 \vee y \neq 0) \wedge (x = 0 \wedge y = 0)$
    $\Leftrightarrow (x \neq 0 \wedge x = 0 \wedge y = 0) \vee (y \neq 0 \wedge x = 0 \wedge y = 0) \Leftrightarrow F$
- All three conditions are contradictions; thus, this program outline is deadlock free.

- From example 6, we have the following observations.
  - Even if some potential deadlock conditions are not contradiction, we still might not have deadlock at the runtime: $\{T\}$[**await** $y \neq 0$ **then** $x \coloneqq 1$ **end** ‖ **await** $x \neq 0$ **then** $y \coloneqq 1$ **end**]$\{\dots\}$ will be deadlocked only when $x = 0 \wedge y = 0$.
  - Strengthening the precondition can help us to prove deadlock-free: since in the above program deadlock only happens when $x = 0 \wedge y = 0$; by strengthening the precondition from T to $x \neq 0 \vee y \neq 0$ we can prove a program is deadlock free.

7. Consider the following partial proof outline:

$\{\dots\}[\{p_1\}$ **await** B_1 **then** S_1 **end** $\{q_1\}$
$\quad \| \{p_{21}\}$ **await** B_{21} **then** S_{21} **end**; $\{p_{22}\}$ **await** B_{22} **then** S_{22} **end** $\{q_2\}$
$\quad \| \{p_3\}\ S_3\ \{q_3\}]\ \{q\}$ # S_3 doesn't contain any await statement

What are the potential deadlock conditions we need to test?

- Let's denote D_1, D_2 and D_3 as the set of all possible expressions for r_1, r_2 and r_3 respectively. Then,
  $D_1 = \{p_1 \wedge \neg B_1,\ q_1\}$
  $D_2 = \{p_{21} \wedge \neg B_{21},\ p_{22} \wedge \neg B_{22}, q_2\}$
  $D_3 = \{q_3\}$

- Thus, there are $2 * 3 * 1 - 1$ different potential deadlock conditions:
  - $(p_1 \wedge \neg B_1) \wedge (p_{21} \wedge \neg B_{21}) \wedge q_3$
  - $(p_1 \wedge \neg B_1) \wedge (p_{22} \wedge \neg B_{22}) \wedge q_3$
  - $(p_1 \wedge \neg B_1) \wedge q_2 \wedge q_3$
  - $q_1 \wedge (p_{21} \wedge \neg B_{21}) \wedge q_3$
  - $q_1 \wedge (p_{22} \wedge \neg B_{22}) \wedge q_3$