Synchronization and Await Statement

- We learned that the execution order of statements in different threads matters. To avoid race conditions while keeping parallelism, we learned to use atomic regions to avoid interleaving the critical section in a thread; it is not enough, we often want threads to synchronize: we want one thread to wait until some other thread makes a condition come true.

  Let's look at the following example.

1. In the following parallel program, the calculation of $u$ doesn't start until we finish calculating $z$, even though $u$ doesn't depend on $z$; this program can have "more parallelism".

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

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

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

   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:

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

- Here we introduce the await statement: **await** $B$ **then** $S$ **end,** where $B$ is a Boolean expression and $S$ is a statement.
  - **await** statements can only appear in threads of parallel programs, and they cannot appear inside of atomic regions.
  - Each **await** statement itself is an atomic region: once $B$ is evaluated to true, the execution of $S$ is done in one step.
  - $S$ cannot contain loops, **await** statements, or atomic regions.
    - Not allowing loops in an **await** statement is for the program designing reason. Loops are usually the important part of a program; we don't want the important part of the program to be potentially blocked forever.

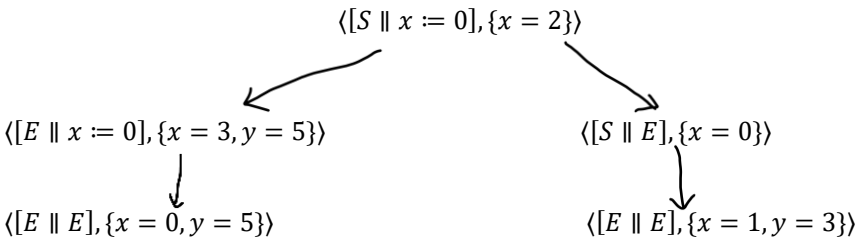
- An **await** is like an atomic **if** $-$ **then** statement, but not identical.
  - With $<$ **if** $B$ **then** $S$ **fi** $>$; if $B$ is false, we execute **skip** and complete the **if** $-$ **fi**.
  - With **await** $B$ **then** $S$ **end**; if $B$ is false, then this await statement blocks (no execution in this thread) until $B$ becomes true.

- Here is the semantic of parallel programs with await statements.
  - We still 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 $\langle[\textbf{await } x \geq 0 \textbf{ then } x \coloneqq x + x \textbf{ end} \parallel x \coloneqq -1]\ \{x = 2\}\rangle \rightarrow \langle[\textbf{await } x \geq 0 \textbf{ then } x \coloneqq x + x \textbf{ end} \parallel E]\ , \{x = -1\}\rangle$.

  - 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 $\langle[\textbf{await } x \geq 0 \textbf{ then } x \coloneqq x + x; x \coloneqq x + 1 \textbf{ end} \parallel x \coloneqq -1\ ], \{x = 2\}\rangle \rightarrow \langle[E \parallel x \coloneqq -1], \{x = 5\}\rangle$, the first thread is chosen to be execute first, and there is no chance for $x \coloneqq -1$ to jump in after $x \geq 0$ is evaluated and before the whole await statement is executed.

- If we choose the thread that begins with **await** $B$ **then** $S$ **end**, and $B$ is false, then this await statement blocks. Instead, we randomly choose among other threads to execute.
  - For example, $\langle [\textbf{await } x = 1 \textbf{ then } S \textbf{ end} \parallel x := 1], \{x = 0\} \rangle \rightarrow \langle [\textbf{await } x = 1 \textbf{ then } S \textbf{ end} \parallel E], \{ x = 1 \} \rangle$ and this is the only possible next evaluation step since the first thread blocks until $x = 1$ becomes true.

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




3. Let's consider some special situations in the await statement **await** $B$ **then** $S$ **end**.
   a. What if $B \equiv T$?
   The await statement never blocks and always executes $S$ 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 $B \equiv F$?
   The await statement blocks forever.

4. 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 executing $S$.

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.
- We represent await rule in the proof outline as follows:

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

5. How to understand the await rule for $\{p\}$ **await** $F$ **then** $S$ **end** $\{q\}$?
   - Since $\{p \wedge F\}\ S\ \{q\}$ is valid (under either correctness), so $\{p\}$ **await** $F$ **then** $S$ **end** $\{q\}$ is also valid. This sounds against our intuition, but $\{p\}$ **await** $F$ **then** $S$ **end** $\{q\}$ being valid doesn't mean it won't block; it

simply means "if precondition $p$ is satisfied and if the await condition $B$ is satisfied, then the execution of $S$ will satisfy the postcondition $q$."

- $wp(\textbf{await } B \textbf{ then } S \textbf{ end}, q) \equiv D(B) \wedge \big(B \rightarrow wp(S, q)\big)$
- $sp(p, \textbf{await } B \textbf{ then } S \textbf{ end}) \equiv sp(p \wedge B,\ S)$

Deadlock

- A parallel program is **deadlocked** if it has not finished the execution and there's no possible evaluation step to take.
  - In example 3.c, we can see that $\textbf{await } F \textbf{ then } S \textbf{ end}$ can block forever, thus if this await statement is in one thread then this parallel program is deadlocked.
  - Deadlock is a runtime error; so, if a parallel program is deadlocked, the execution terminates (unsuccessfully) in pseudo-state $\perp_e$.

6. When are the following programs deadlocked?
   a. $[\textbf{await } x \geq 0 \textbf{ then } S_1 \textbf{ end} \parallel \textbf{await } x \geq 0 \textbf{ then } S_2 \textbf{ end}]$
   If we execute it in a state $\sigma$ 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 $\sigma$ with $\sigma(x) = 0$ and $\sigma(y) = 0$.

   c. $x := 1; y := 1; [\textbf{wait } y \neq 0;\ x := 0 \parallel \textbf{wait } x \neq 0; y := 0]$
   Remember 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 disjointedness program, for disjoint conditions and for interference freedom, we also have a "static and too strict" test for "deadlock freedom".

- For threads (written in proof outlines) in the parallel composition $[\{p_1\}\ S_1^*\ \{q_1\} \parallel \{p_2\}\ S_2^*\ \{q_2\} \parallel \cdots \parallel \{p_n\}\ S_n^*\ \{q_n\}]$ we define one of its **potential deadlock conditions** 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 $S_k^*$
  - $q_k$

  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('s proof 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 await statements):
  If for $k = 1 \ldots n$, we have $\{p_k\}\ S_k^*\ \{q_k\}$ are all interference-free and deadlock-free proof outlines:
  $1\ \{p_1\}\ S_1^*\ \{q_1\}$
  $2\ \{p_2\}\ S_2^*\ \{q_2\}$
  …

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, … , n

7. The following program $\{T\}[\textbf{await } y \neq 0 \textbf{ then } x := 1 \textbf{ end} \parallel \textbf{await } x \neq 0 \textbf{ then } y := 1 \textbf{ end}]\{\cdots\}$ is from example 6.b. We use the following full proof outlines (using the await rule and forward assignment) 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 deadlock-free?

- We need to test the following potential deadlock conditions:
  - $y = 0 \wedge (x \neq 0 \wedge y = 1)$ # $r_1$ is "$p \wedge \neg B$", $r_2$ is "$q_2$" ✓
  - $(x = 1 \wedge y \neq 0) \wedge x = 0$ # $r_2$ is "$p \wedge \neg B$", $r_1$ is "$q_1$" ✓
  - $y = 0 \wedge x = 0$ # $r_1$ is "$p \wedge \neg B$", $r_2$ is "$p \wedge \neg B$" ✗
  - (Note that, we don't test $q_1 \wedge q_2$)

  Among the above three conditions, the first two are contradictions and the last one is not. So, this program 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 $p \equiv x \neq 0 \vee y \neq 0$:

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

And the proof outline of this program becomes:

$$\{p\}\ [\{p\}\ \textbf{await } y \neq 0 \textbf{ then } \{p \wedge y \neq 0\}\ x := 1\ \{x = 1 \wedge y \neq 0\}\ \textbf{end}\ \{x = 1 \wedge y \neq 0\}$$
$$\parallel \{p\}\textbf{await } x \neq 0 \textbf{ then } \{p \wedge x \neq 0\}\ y := 1\ \{x \neq 0 \wedge y = 1\}\ \textbf{end}\ \{x \neq 0 \wedge y = 1\}]\ \{q\}$$

- We need to test the following potential deadlock conditions:
  - $(p \wedge y = 0) \wedge (x \neq 0 \wedge y = 1)$ # $r_1$ is "$p \wedge \neg B$", $r_2$ is "$q_2$" ✓
  - $(x = 1 \wedge y \neq 0) \wedge (p \wedge x = 0)$ # $r_2$ is "$p \wedge \neg B$", $r_1$ is "$q_1$" ✓
  - $(p \wedge y = 0) \wedge (p \wedge x = 0)$ # $r_1$ is "$p \wedge \neg B$", $r_2$ is "$p \wedge \neg B$" ✓

  $\#\ (p \wedge y = 0) \wedge (p \wedge x = 0) \Leftrightarrow (x \neq 0 \vee y \neq 0) \wedge (x = 0 \wedge y = 0)$
  $\Leftrightarrow (x \neq 0 \wedge x = 0 \wedge y = 0) \vee (y \neq 0 \wedge x = 0 \wedge y = 0)$
  $\Leftrightarrow F$

  All three conditions are contradictions; thus, this program is deadlock free.

- From example 7, we have the following observations.
  - Even if some potential deadlock conditions are not contradiction, we still might not have deadlock at the runtime: $\{T\}[\textbf{await } y \neq 0 \textbf{ then } x := 1 \textbf{ end} \parallel \textbf{await } x \neq 0 \textbf{ then } y := 1 \textbf{ end}]\{\ldots\}$ will be deadlocked only when $x = 0 \wedge y = 0$.
  - Strengthening the precondition can help us to avoid deadlocks: since in the above program deadlock only happens when $x = 0 \wedge y = 0$; by strengthening the precondition from $T$ to $x \neq 0 \vee y \neq 0$ (this is like we add the domain predicate of the program to the precondition to avoid runtime error) we can prove a program is deadlock free.

8. Consider the following partial proof outline:

$$\{\ldots\}[\{p_1\}\ \textbf{await } B_1 \textbf{ then } S_1 \textbf{ end}\ \{q_1\}$$

$\| \{p_{21}\}$ **await** $B_{21}$ **then** $S_{21}$ **end**; $\{p_{22}\}$ **await** $B_{22}$ **then** $S_{22}$ **end** $\{q_2\}$
$\| \{p_3\} < S_3 > \{q_3\}] \{\dots\}$

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:
  - $(p_1 \wedge \neg B_1) \wedge (p_{21} \wedge \neg B_{21}) \wedge q_3$
  - $(p_1 \wedge \neg B_1) \wedge (p_{22} \wedge \neg B_{22}) \wedge q_3$
  - $(p_1 \wedge \neg B_1) \wedge q_2 \wedge q_3$
  - $q_1 \wedge (p_{21} \wedge \neg B_{21}) \wedge q_3$
  - $q_1 \wedge (p_{22} \wedge \neg B_{22}) \wedge q_3$