Universitetet i Oslo
Institutt for Informatikk

Research group for Reliable Systems (PSY)
Andrea Pferscher

INF 5170: Models of Concurrency

Fall 2025 **Group Session 1** 29.08.2025

Topic: Warm-up: thinking concurrently and basic synchronization

Exercise 1 (Parallelism and concurrency) The notions of *parallelism* and *concurrency*, while related, are not identical.[1] Parallelism implies that executions "really" run at the same physical time which requires a multi-core CPU. Concurrent execution may happen on a single-core processor, where the fact that various processes seem to happen simultaneously is just an "illusion" (typically an illusion maintained by the operating system).

Assume you have a single-core processor, so the CPU does not contain parallel hardware. Under these circumstances, is it possible, that using concurrency makes programs run faster? Give reason for your opinion.

Solution: Concurrency can make programs run faster also on a single-core processor. The CPU often has to wait for a response to its requests. Using concurrency, the CPU can execute another thread while waiting for a response. For example, Thread 1 reads variable x. While waiting for the response, Thread 2 writes to variable y. Thus, concurrency can make programs run faster.

Exercise 2 (Synchronization) ([1, Exercise 2.1]) Consider the skeleton of the program in Listing 1 that prints all the lines in a file containing `pattern`.

1. Add the missing code for synchronizing access to `buffer`. Use the `await` statement for the synchronization code.
2. Extend your program so that it reads two files and prints all the lines that contain `pattern`. Identify the independent activities and use a separate process for each. Show all synchronization code that is required.

Solution: We may first identify local and global variables to check that only for `buffer`, synchronization code is needed. The variables `line1` and `line2` are local to Process 1, respectively Process 2, and therefore do not have to be protected since interference is only possible for global variables. The variable `done` is global, however, there cannot be interference. It is initialized with `false` and set to `true` exactly once, after which it stays `true`. So Process 1 cannot "miss" that fact. The variable `buffer` is global, Process 1 *reads* from the buffer and Process 2 *writes* to the buffer, thus the processes interact and synchronization is needed.

We may also check whether the AMO-Property holds to be sure that the assignments to and from the buffer can actually be treated as atomic (if not, we have to think whether that would be a problem and deal with that as well). Since `line1` and `line2` are local variables, the AMO-Property is satisfied for the assignments.

[1]The terminology is not 100% uniform across all fields. Nonetheless, the one we use in the lecture is the most common one.

Listing 1: Finding patterns in a file (skeleton)

```
string buffer;  # contains one line of the input
bool done := false;
process Finder { # find patterns
    string line1;
    while (true) {
      wait for buffer to be full or done to be true;
      if (done) break;
      line1 := buffer;
      signal that buffer is empty;
      look for pattern in line1;
      if (pattern is in line1)
         write line1;
    }
}
process Reader { # read new lines
    string line2;
    while (true) {
      read next line of input into line2 or set EOF after last line;
      if (EOF) {done := true; break;}
      wait for buffer to be empty;
      buffer := line2;
      signal that buffer is full;
    }
}
```

1. There are two keywords which indicate that this is where we have to add more concrete synchronization code, *wait* and *signal.*

   To implement synchronization with `await` statements, we can use additional shared variables. By changing the state of a shared variable, a process can *signal* to the other process. The other process uses the `await` statement with a condition over that shared variable to wait. Since all waiting and signaling in this example is done using either that `buffer` is full or empty, we can use a Boolean variable for synchronization. See Listing 2 for a solution.

2. The change now is to have *two readers.* If we leave everything unchanged, the change means: the two readers now *share the same buffer*, there is now two writers to that.

   The finder does not change much. Modulo two small changes, the code is the same. Since there is only one buffer, and the finder does not care who actually has produced the result, there is still just *one* Boolean variable (`bufferempty`) to organize the required synchronization. If one of the readers filled the buffer with a line, it sets `bufferempty` to false so the finder can look for `pattern` in the line. The other piece of synchronization from readers to finder, namely to signal termination, works analogous to the case with only one reader. With two readers, the finder needs the information from both so we need *two flags* for `done`.

   Apart from synchronization, the readers have the same code as in Listing 2.However, as far as synchronization is concerned, we have to be careful. In the single-reader setting, there was of course already a "dangerous" shared variable, the buffer, because it was read and repeatedly written. As for the one-reader solution, there is one fine point which becomes important now. We used the Boolean variable `bufferempty` to indicate that the buffer is full (respectively empty). A shared variable is of course not full or empty, but we may consider that in the reader, the assignment `buffer := line2` actually fills the buffer; the value is not there, and in the next line, `bufferempty` is used to signal that, see Lines 23–24 of Listing 2.

Listing 2: Finding a pattern using await

```
 1  string buffer;                 # contains one line of the input
 2  bool done := false;
 3  bool buffempty := true;  # shared variable, used for
 4                                 # signalling/wait synchronization
 5  process Finder { # find patterns
 6      string line1;
 7      while (true) {
 8        ⟨ await (buffempty = false ∨ done = true); ⟩ # wait
 9        if (done ∧ bufferempty = true) break;
10        line1 := buffer;
11        buffempty := true;          # signal
12        look for pattern in line1;
13        if (pattern is in line1)
14            write line1;
15      }
16  }
17  process Reader { # read new lines
18      string line2;
19      while (true) {
20        read next line of input into line2 or set EOF after last line;
21        if (EOF) {done := true; break;}
22        ⟨ await (buffempty = true); ⟩        # wait
23        buffer := line2;
24        buffempty := false;          # signal
25      }
26  }
```

A similar pattern is done analogously for the finder. The fine point here is: each of the two lines is in itself atomic, both together *are not!* In other words, there is a point where the buffer is factually non-empty, but the flag `bufferempty` is false! Another way of stating that is the following: we have a flag `buffempty` meant to indicate whether the buffer is full or empty. That means, that we expect conceptually that the buffer is factually empty if and only if `bufferempty = true` holds. That mental invariant was behind the way we solved the problem. However, as we just said, there is a point in the reader, *where this does not hold!* (There is a corresponding point in the finder where it is broken as well, of course).

Such points where the "intended" invariant is broken, said differently, where things that we consider in our head as atomic are factually *not* atomic, are always reasons to think carefully, because that's where things may go wrong! In the one-reader example, actually nothing went wrong. The reason for that ultimately was that we assumed that there was just *one* reader! More concretely; the dangerous point where the invariant was broken could not be *observed* by the (only) reader. This was achieved by the general pattern of synchronization: at the danger point between filling the buffer and flagging that to the finder could not be exploited by the finder because it guaranteed that at that time he is *waiting* (thanks to the await-statement).

Now, however, we have *two* readers, and, unlike the reader vs. finder, there is no synchronization *between the two readers*. In the design we made, the readers produce lines independently, they only have to coordinate with the (shared) finder. That independence of behavior exposes the described danger point of one reader to interference from the other. Concretely: a buffer which is full but looks empty (because `bufferempty` has not been set yet) may very well be overwritten by the competing readers, in which case the data item is "forgotten" (the corresponding line will not reach the finder). To prevent that, we can make Lines 23–24 in Listing 2 atomic, using a conditional critical section (see Listing 3).

Listing 3: Finding a pattern with two readers

```
string buffer;               # contains one line of the input
bool done1 := false;         # used to signal termination of process 2
bool done2 := false;         # used to signal termination of process 3
bool buffempty := true;      # shared variable, used for
                             # signalling/wait synchronization
process Finder { # find patterns
    string line1;
    while (true) {
      ⟨ await (buffempty = false ∨ (done1 = true ∧ done2 = true); ⟩ # wait
      if ((done1 = true ∧ done2 = true) ∧ bufferempty = true) break;
      line1 := buffer;
      buffempty := true;         # signal
      look for pattern in line1;
      if (pattern is in line1)
         write line1;
    }
}
process Reader1 { # read new lines
    string line2;
    while (true) {
      read next line of input into line2 or set EOF after last line;
      if (EOF) {done1 := true; break;}
      ⟨ await (buffempty = true) {       # wait
         buffer := line2;
         buffempty := false;} ⟩   # signal
    }
}
process Reader2 { # read new lines
    string line3;
    while (true) {
      read next line of input into line3 or set EOF after last line;
      if (EOF) {done2 := true; break;}
      ⟨ await (buffempty = true) {       # wait
         buffer := line3;
         buffempty := false;} ⟩   # signal
    }
}
```

Exercise 3 (Producer-consumer) ([1, Exercise 2.2]) Consider the code of the simple producer-consumer problem in Listing 4. Change it so that the variable `p` is local to the producer process and `c` is local to the consumer process, not global. Hence, those variables cannot be used to synchronize access to `buf`.

Solution: The solution to this problem, is similar to the one of the previous exercise. We need to introduce a global Boolean variable, called `bufferempty` again, to synchronize between the producer and the consumer. Intuitively, the variable `bufferempty` is `true` when the producer has not put an element in the buffer yet or the current element in the buffer has already been read by the consumer. Thus, the producer has to wait for `bufferempty` to be `true` before writing a new element to the buffer and signal afterwards that `bufferempty` is `false`. Conversely, the consumer has to wait for `bufferempty` to be `false` before reading a new element from the buffer and signal afterwards that `bufferempty` is `true`. See Listing 5 for a solution.

Exercise 4 (Executions and atomicity) ([1, Exercise 2.10]) Consider the program in Listing 6.

1. Suppose each assignment statement is implemented by a single machine instruction and hence is atomic. How many possible executions are there? What are the possible final values of `x` and `y`?

Listing 4: Copying an array from a producer to a consumer; global `p` and `c`

```
int buffer, p := 0; c := 0;

process Producer {
  int a[N];
  while (p < N) {
    ⟨ await (p = c); ⟩
    buffer := a[p];
    p := p+1;
  }
}
process Consumer {
  int b[N];
  while (c < N) {
    ⟨ await (p > c); ⟩
    b[c] := buffer;
    c := c+1;
  }
}
```

Listing 5: Copying an array from a producer to a consumer; local `p` and `c`

```
int buffer;
bool bufferempty := true;

process Producer {
  int a[N], p := 0;
  while(p < N){
    ⟨ await (bufferempty = true); ⟩
    buffer := a[p];
    p := p+1;
    bufferempty := false;
  }
}

process Consumer {
  int b[N], c := 0;
  while(c < N){
    ⟨ await (bufferempty = false); ⟩
    b[c] := buffer;
    c := c+1;
    bufferempty := true;
  }
}
```

2. Suppose each assignment statement is implemented by three atomic actions that load a register, add or subtract a value from that register, then store the result. How many possible executions are there now? What are the possible final values of `x` and `y`?

Listing 6: A concurrent program with different executions

```
int x := 0, y := 0;
co
   x := x + 1; # S1
   x := x + 2; # S2
||
   x := x + 2; # P1
   y := y − x; # P2
oc
```

Solution: We can try to enumerate them by hand but for larger ones the number of possible executions can be determined by

$$\frac{(n \cdot m)!}{m!^n} \tag{1}$$

where n is the number of processes and m the number of atomic steps.

1. If each assignment statement is atomic, we have $n = 2$ and $m = 2 \Rightarrow 6$ possible executions. The following shows the possible final results.

```
S1;S2;P1;P2 -> x = 5, y = -5
S1;P1;S2;P2 -> x = 5, y = -5
S1;P1;P2;S2 -> x = 5, y = -3
P1;P2;S1;S2 -> x = 5, y = -2
P1;S1;P2;S2 -> x = 5, y = -3
P1;S1;S2;P2 -> x = 5, y = -5
```

In a formulaic way, a post-condition can be written for instance as

$$(x = 5) \wedge ((y = -5) \vee (y = -3) \vee (y = -2)) \tag{2}$$

The fact that there is only one result for x is a consequence somehow of the fact that there are only increments on x and $+$ is commutative.

2. If each assignment statement is implemented by three atomic actions, we have $n = 2$ and $m = 6 \Rightarrow 924$ possible executions.

The solutions for atomic assignment statements are still possible. Now that the statements are non-atomic, commutativity does not help anymore. Since x is not influenced by y, we concentrate on that first. Some assignments may now be "forgotten". Consider for example the (very informal) sequence diagram in Figure 1. Both Thread 1 and Thread 2 read the initial value of x, which is 0. Thread 2 is faster than Thread 1 and writes $x = 2$ before Thread 1 writes $x = 1$. Thus, the write $x = 2$ is "forgotten". In this way, it is possible to obtain that $x \in \{5, 4, 3, 2\}$. Based on that, the possible values for y are $y \in \{-2, -3, -4, -5\}$. However, considering the example in Figure 1, Thread 2 could also read x after Thread 1 wrote $x = 1$. This would lead to $y = -1$. Writing to y could also happen before $S2$ is executed, which leads to $y = -2$ in two cases as shown in Figure 2. Therefore, we have to add the following to Equation 2 .

$$(x, y) \in \{(4, -4), (3, -3), (3, -2), (2, -2), (3, -1)\}$$

That means that if the statements are *not* atomic, then they also don't appear to be atomic, because the set of results changes.

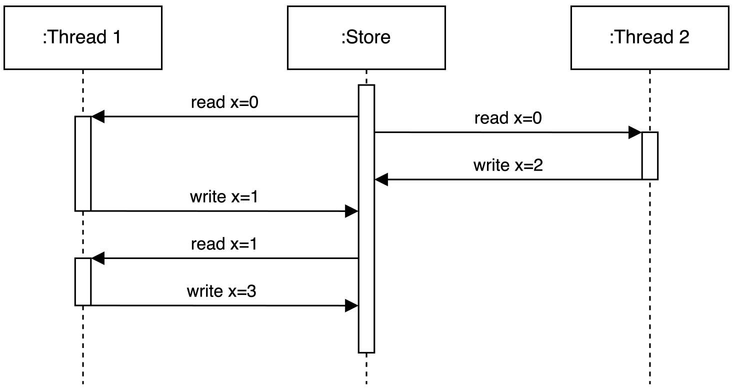

Figure 1: Example of interference of threads with non-atomic assignment

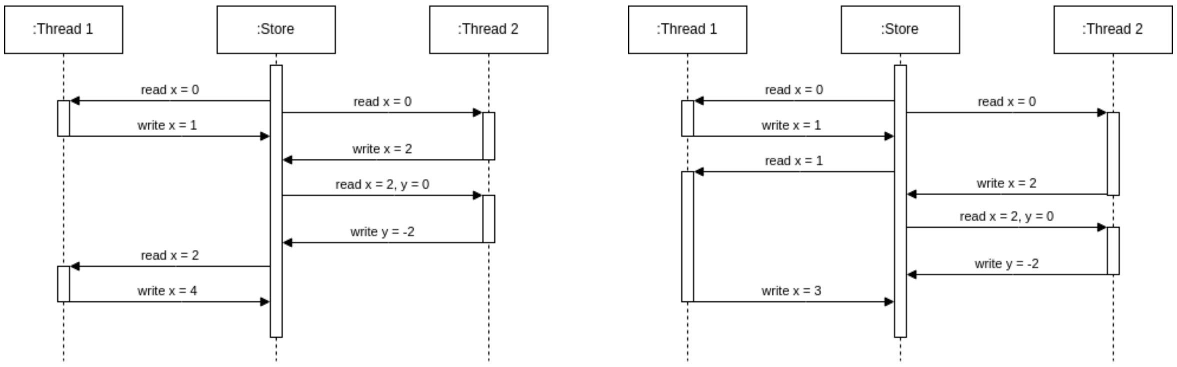

Figure 2: Two more examples of interference of threads with non-atomic assignment

Exercise 5 (Interleaving, non-determinism, and atomicity) ([1, Exercise 2.12]) Consider the following program.

```
int x := 2, y := 3;
co
   <x := x + y;>   #S1
||
   <y := x * y;>   #S2
oc
```

1. What are the possible final values of x and y?
2. Suppose the angle brackets are removed and each assignment statement is now implemented by three atomic actions: read the variables, add or multiply, and write to a variable. What are the possible final values of x and y now?

Solution:

1. The possible final results for x and y are obtained as follows.

```
S1;S2 -> x == 5, y == 15
S2;S1 -> x == 8, y== 6
```

2. If we drop the critical section, all results obtained with the critical section are still possible. Furthermore, if both threads read the variables first, then write (after the other thread already read both variables), the result $x = 5, y = 6$ can also be obtained. Thus, the solution is

$$(x, y) \in \{(5, 15), (5, 6), (8, 6)\}.$$

Exercise 6 (At most once) ([1, Exercise 2.14]) Consider the following program.

```
int x := 1, y := 1;
co
   <x := x + y;>      #S1
||
   y := 0;            #S2
||
   x := x − y;        #S3
oc
```

1. Do $S1, S2$ and $S3$ satisfy the requirements of the At-Most-Once Property?
2. What are the final values for x and y? Explain your answer.

Solution:

1. Remember that the AMO-Property identifies statements that can be considered atomic. Since $S1$ is already atomic, we do not have to check the AMO-Property for it. Since 0 is not a variable, it is also not a critical reference, and thus $S2$ satisfies the AMO-Property. However, $S3$ does not satisfy the AMO-Property because it contains two critical references.
2. The final values can be obtained as described in the previous exercises. The results can be $(x, y) \in \{(2, 0), (1, 0), (0, 0)\}$.

Exercise 7 (AMO, termination) ([1, Exercise 2.15]) Consider the following program.

```
int x := 0, y := 10;

co
    while (x != y) x := x + 1;
||
    while (x != y) y := y − 1;
oc
```

1. Do all parts of the program meet the requirements of the At-Most-Once-Property?
2. Will the program terminate? Always? Sometimes? Never?

Solution:

- Yes, because there is no critical reference in the assignments $x := x + 1$ or $y := y - 1$ (remember that a variable in an expression is only a critical reference if it is changed by *another* process).
- It sometimes terminates. However, it can happen that $x = y - 1$ and both threads conclude $x \neq y$ before the other one changed a variable. Thus, x is increased and y is decreased, leading to $x = y + 1$.

References

[1] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.