Universitetet i Oslo
Institutt for Informatikk
Research group for Reliable Systems (PSY)
Silvia Lizeth Tapia Tarifa, Riccardo Sieve

INF 5170: Models of Concurrency

Fall 2026 — **Group Session 2** — 09.09.2026

Topic: Synchronization, Critical Sections

Exercise 1 ([1, Exercise 2.17]) Consider the following program.

```
co
    <await (x ≥ 3) x := x − 3>   # P1
 ||
    <await (x ≥ 2) x := x − 2>   # P2
 ||
    <await (x = 1) x := x + 5>   # P3
oc
```

For which initial values of x does the program terminate (under weakly fair scheduling)? What are the corresponding final values? Explain your answer.

Solution: The program terminates for $x = 1$ or $x = 6$ as initial values and it may terminate for $x = 3$ and $x = 4$, depending on which process executes first.

```
x <= 0 : No termination, all await statements are blocked
x = 1 : Terminating with x = 1
x = 2 : No termination (x = 0 after P2)
x = 3 : Case 1: P2 executes first, terminating with x = 3
        Case 2: P1 executes first, no termination (x = 0 after P1)
x = 4 : Case 1: P1 executes first, terminating with x = 4
        Case 2: P2 executes first, no termination (x = 2 after P2)
x = 5 : No termination (x = 0 after P1 and P2)
x = 6 : Terminating with x = 6
x >= 7 : No termination, P3 will never be executed
```

Exercise 2 ([1, Exercise 2.18]) Consider the following program.

```
co
   <await (x > 0) x := x − 1;>  # P1
||
   <await (x < 0) x := x + 2;>  # P2
||
   <await (x = 0) x := x − 1;>  # P3
oc
```

For which initial values of x does the program terminate (under weakly fair scheduling)? What are the corresponding final values? Explain your answer.

Solution: The program terminates if x has -1,0 or 1 as initial value.

```
x < -1  :  No termination
x = -1  :  Terminating with x = -1
x = 0   :  Terminating with x = 0
x = 1   :  Terminating with x = 1
x > 1   :  No termination
```

Exercise 3 ([1, Exercise 2.33]) Consider the following program.

```
1  int x := 10;
2  bool c := true;
3  co
4     <await (x = 0)>;  c := false          # P1
5
6  ||
7     while (c) <x := x − 1>                # P2
8
9  oc
```

Do we have:

1. Termination under weak fairness?
2. Termination under strong fairness?
3. Add the following while statement as a 3rd branch of the **co**-statement:

```
1  while (c) {if (x < 0)   <x := 10> ;}    # P3
2
```

Do we now have:

(a) Termination under weak fairness?

(b) Termination under strong fairness?

Solution: As long as `P2` is enabled i.e., as long as c remains true (and as long as chosen by the scheduler), `P2` continuously decrements x. Since `P1` is initially disabled, the scheduler has to pick `P2`. Note that when $x = 0$, both `P1` and `P2` may be executed.

1. A necessary condition for termination is that c is set to false which means that `P1` must be executed. Weak fairness can guarantee that only if `P1` is enabled at some point *and stays enabled*. The latter is not the case: it may be that at the point where $x = 0$, `P2` is chosen again; the "chance" is missed for `P1`. In this case, the program will not terminate. Weak fairness cannot prevent this from happening. However, it is of course possible that `P1` is chosen when $x = 0$ and the program terminates.

2. In order to guarantee termination, the condition of `P1` must be true infinitely often. This will not happen, and the program is not guaranteed to terminate. But again, it may terminate.

3. If P3 is added, the program is guaranteed to terminate only if the scheduling is strongly fair. (As before, it "might terminate" with weak fairness, but this is not guaranteed.)

Exercise 4 ([1, Exercise 3.1]) Listing 1 shows Dekker's algorithm, a solution to the critical section problem for two processes. Are the following properties satisfied?

1. Mutual exclusion
2. Absence of deadlock
3. Absence of unnecessary delay
4. Eventual entry

How many times can one process that wants to enter its critical section be bypassed by the other before the first gets in?

Listing 1: Dekker's algorithm

```
1  bool enter1 := false, enter2 := false;
2  int turn := 1;
3  process P1{
4    while (true){
5      enter1 := true              # entry protocol
6      while(enter2){
7        if(turn = 2){
8          enter1 := false;
9          while(turn = 2) skip;
10         enter1 := true;
11       }
12     }
13     critical section;
14     enter1 := false;              # exit  protocol
15     turn := 2;
16     non-critical section;
17   }
18 }
19 process P2{
20   while (true){
21     enter2 := true              # entry protocol
22     while(enter1){
23       if(turn = 1){
24         enter2 := false;
25         while(turn = 1) skip;
26         enter2 := true;
27       }
28     }
29     critical section;
30     enter2 := false;              # exit  protocol
31     turn := 1;
32     non-critical section;
33   }
34 }
```

Solution: The variables `enter1` and `enter2` indicate the wish of process `P1`, respectively `P2`, to enter its critical section. If only one of the processes wants to enter its critical section, so only one `enter` variable is `true`, then the process can jump over the first while loop and enter the critical section directly. Note that this can be done irrespectively of the value of `turn`. If both processes want to enter their critical section, i.e., both `enter1` and `enter2` are `true`, then both processes enter the first while loop. Now the `turn` variable indicates which process can enter its critical section first. If `turn=1`, then process `P2` retracts its wish to enter the critical section by setting `enter2` to `false` and spins in the inner while loop. Since `enter2` is now `false`, process

`P1` can exit the outer while loop and enter its critical section. When it exits its critical section, it sets `enter1` to `false` and `turn` to 2 so process `P2` can exit its entry protocol and proceed into its critical section. It works similarly if `turn=2`.

Note that if a process is in its critical section, then its `enter` variable is `true`.

Mutual Exclusion: Mutual exclusion is guaranteed. If a process is in its critical section, then its `enter` variable is `true`, as described before. Therefore, the other process cannot exit the outer while loop of its entry protocol. Exiting the while loop is only possible if the other process sets its `enter` variable to `false`.

Deadlock: Deadlock is impossible. As described above, if both processes want to enter its critical section and set their `enter` variable to `true`, the process indicated by the `turn` variable will be able to exit its entry protocol because the other process sets its `enter` variable to `false` before waiting in the inner while loop.

Unnecessary Delay: If `P1` is in its non-critical section or terminated, then `enter1` = `false`. Thus, `P2` can go directly into its critical section. It works similarly if `P2` is in its non-critical section or terminated.

Eventual Entry: If both processes compete for entry to the critical section, one of them has to wait in the inner while loop. Assume `P1` has to wait in the inner while loop, so `enter2=true` and `turn=2`. Once process `P2` exits its critical section, it sets `enter2:=false` and `turn:=1`. Therefore, even if process `P2` sets `enter2:=true` before process `P1` can exit the outer while loop of its entry protocol, process `P1` has now precedence over process `P2` because `turn=1`. Process `P2` will enter the if statement in the outer while loop and set `enter2:=false` before waiting in the inner while loop. Therefore, process `P1` can proceed into its critical section. It works similarly if process `P2` has to wait.

Waiting "time"?: A process can be bypassed at most once, see above.

Exercise 5 ([1, Exercise 3.7]) Consider the following code snippet (Lamport [2]).

```
1  int lock := 0;
2  process CS[i = 1 to n]{
3    while(true){
4      <await (lock = 0)>;
5      lock := i;
6      Delay;
7      while(lock != i){
8        <await (lock = 0)>;
9        lock := i;
10       Delay;
11     }
12     critical section;
13     lock := 0;
14     non-critical section;
15   }
16 }
```

1. Suppose the delay code is deleted. Are the following properties satisfied?
   (a) Mutual exclusion
   (b) Absence of deadlock
   (c) Absence of unnecessary delay
   (d) Eventual entry
2. Suppose the Delay code spins for long enough to ensure that every process `i` that waits for `lock` to be `0` has time to execute the assignment statement that sets `lock` to `i`. Reconsider your answers under that circumstances.

Solution:

1. **Mutual Exclusion** Mutex is *not* guaranteed. Assume that `CS1` and `CS2` wait for the first await-condition: that one will be passed by both, because the initial value is 0 (lock is free). Now `CS1` sets `lock` to `1`, immediately checks the while loop (without interruption), and thus avoids the while loop, entering straight into its critical section. Process `CS2` does the same and mutual exclusion is violated.

   The point is somehow opposite of what we are often arguing:. The point is *not*: what happens if an reschedule *destroys* a value (by interference), here is the opposite:

   > What happens if there is (unfortunately) *no* reschedule?

   The functionality of `Delay` is to suspend the processes such that all processes executing the await statement will have time to execute the assignment before any of the processes enters the while-loop.

   Deadlock: Deadlock cannot happen since there must always exist one process `i` with `lock` = `i`. The remaining processes must wait for `lock` = `0`.

   Unnecessary Delay: Unnecessary delay is avoided. If only one processes wants access to the critical section, and all other processes are in their non-critical section, `lock` must equal `0`. Thus, the process can enter its critical section directly.

   Eventual Entry: Eventual entry cannot be guaranteed. It is possible for a process `CS`i to be bypassed infinitely. For instance, if there always exists another processes doing an assignment to `lock` immediately after `CS`i.

   Even with weak fair scheduling, it may happen that `CS`i tests lock only when it is different from `0`.

2. (a) Yes, mutual exclusion guaranteed now.
   (b) Yes, deadlock is avoided.
   (c) Yes, unnecessary delay is avoided.
   (d) No, eventual entry is not guaranteed.

Exercise 6 ([1, Exercise 3.8]) Suppose your machine has the following atomic instructions.

```
flip (lock)
  <lock := (lock + 1) % 2;        # flip the lock
  return (lock);>                 # return the new value
```

Someone suggests the following solution to the critical section problem for two processes.

```
int lock := 0;                    # shared variable
process CS[i = 1 to 2]{
  while(true){
    while(flip(lock) != 1)
      {while(lock != 0) skip;}
    critical section;
    lock := 0;
    non-critical section;
  }
}
```

1. Spot the defect in the code, violating mutual exclusion. That is, give an execution order that results in both processes being in their critical sections at the same time.

2. Suppose that the first line in the body of `flip` is changed to do addition modulo 3 instead of modulo 2. Will the solution now ensure mutual exclusion for two processes?

Solution:

1. The "idea" seems to be that `lock = 0` means the lock is free. That can be seen on the use of the while-loop (as last defense in front of the critical section).

   Consider the following scenario. Assume `CS1` starts first, making the first atomic check that also sets `lock` to `1`. Then it enters its critical section. Now `CS2` starts and executes the test `flip(lock)!=1`, which is true and sets `lock` back to `0` (which in our picture means "free"). Process `CS2` then enters the while loop and waits for `lock=0`. Since it has set it to `0` itself (and assuming that `CS1` had not left its critical section and changed `lock` again), it then enters its critical section as well, which breaks mutual exclusion.

2. The solution will now ensure mutual exclusion for two processes. In the situation described above, the process `CS2` cannot get out of the innermost while loop until `CS1` has finished its critical section and set `lock` to `0`.

References

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

[2] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1), 1987.