Universitetet i Oslo
Institutt for Informatikk

Research group for Reliable Systems (PSY)
Andrea Pferscher

INF 5170: Models of Concurrency

Fall 2025 | **Group Session 2** | 04.09.2024

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.

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.

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

```
int x := 10;
bool c := true;
co
   <await (x = 0)>;  c := false            # P1

||
    while (c) <x := x − 1>                 # P2

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:

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

Do we now have:

(a) Termination under weak fairness?

(b) Termination under strong fairness?

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

```
bool enter1 := false, enter2 := false;
int turn := 1;
process P1{
  while (true){
    enter1 := true                  # entry protocol
    while(enter2){
      if(turn = 2){
        enter1 := false;
        while(turn = 2) skip;
        enter1 := true;
      }
    }
    critical section;
    enter1 := false;                 # exit protocol
    turn := 2;
    non-critical section;
  }
}
process P2{
  while (true){
    enter2 := true                  # entry protocol
    while(enter1){
      if(turn = 1){
        enter2 := false;
        while(turn = 1) skip;
        enter2 := true;
      }
    }
    critical section;
    enter2 := false;                 # exit protocol
    turn := 1;
    non-critical section;
  }
}
```

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

```
int lock := 0;
process CS[i = 1 to n]{
  while(true){
    <await (lock = 0)>;
    lock := i;
    Delay;
    while(lock != i){
      <await (lock = 0)>;
      lock := i;
      Delay;
    }
    critical section;
    lock := 0;
    non-critical section;
  }
}
```

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.

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?

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.