UiO : Department of Informatics
University of Oslo

IN5170 Models of Concurrency

Exercise #1-7

Author:
Kjetil K. Indrehus

November 21, 2024

Contents

1 Exercise #1

The topic of the exercise is thinking concurrently and learning basic synchronization.

1.1 Problem 1: Parallelism and concurrency

The notions of parallelism and concurrency, while related, are not identical. Parallelism implies that executions "really" run at the same physical time, whereas concurrent execution may happen on a mono-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 mono-processor (and a single-core machine), 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 still make the program faster. It still makes all processes work faster since it allows scheduling of tasks to be optimal. If a process would not run concurrently, it would execute instructions as seen. This would in many cases be very slow.

1.2 Problem 2: Synchronization

Consider the following skeleton code:

```
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;
        }
    }
```

1.2.1 Part A

Add missing code for synchronizing access to the buffer. Use *await* statements for the synchronization.

Solution
When we write to the buffer, we need to make sure that the finder read the content of the buffer. This is important so that we don't overwrite the content in the buffer before it is read. To solve this, we use *bufferEmpty* variable to signal when the buffer is empty. This will tell the *Reader* to put something in the buffer. We can use the same variable to signal that the buffer can be read. We do this with *atomic* read statements such that we read and evaluate the statement in a single atomic action.

```
string buffer;   # contains one line of the input
bool done := false;
bool bufferEmpty := false;

process Finder { # find patterns
    string line1;
    while (true) {
        <await bufferEmpty || done>
        if (done) break;
        line1 := buffer;

        # Signal empty buffer
        bufferEmpty:= true;

        if (pattern is in line1)
            write line1;
    }
}
process Reader { # read new lines
    string line2;
    while (true) {
        if (EOF) {done := true; break;}

        <await bufferEmpty>
        buffer := line2;

        # signal that buffer is full;
        bufferEmpty := false;
        }
    }
```

1.2.2 Part B

Extend your program so that it read 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
The solution now means that two readers read into a single shared buffer. If there were more than one buffer, then we could have used our solution from part a. But this task requires us to think about how to coordinate between the two reader. We introduce two *done* variables to signal termination for both. The two readers read one file each, and then but the content in the same buffer. But there is an important point to remember here. In the original solution we check if the buffer is empty and then write to the buffer. This will not work here. For example, the two readers can both go past the atomic statement, which would allow then to both write the buffer. To make this work, the checking as well as writing to the buffer must be atomic statements. This will ensure only one reader can add to the buffer.

```
    string buffer;   # contains one line of the input
    bool done1 := false;
    bool done2 := false;
    bool bufferEmpty := false;

    process Finder { # find patterns
        string line1;
        while (true) {
            <await bufferEmpty || (done1 && done2)>
            if (done) break;
            line1 := buffer;

            # Signal empty buffer
            bufferEmpty:= true;

            if (pattern is in line1)
                write line1;
        }
    }
    process Reader1 { # read new lines
        string line2;
        while (true) {
            if (EOF) {done1 := true; break;}
            <await bufferEmpty {
                 buffer := line2;
                 bufferEmpty := false;
            }>
        }
    }

    process Reader2 { # read new lines
        string line3;
        while (true) {
            if (EOF) {done2 := true; break;}

            <await bufferEmpty {
                 buffer := line3;
                 bufferEmpty := false;
            }>
        }
    }
```

1.3 Problem 3: Producer-Consumer

Consider the code of the simple producer-consumer problem below. 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.

```
    int buffer, p, 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;
        }
    }
```

Solution
The task makes variable p and c local. This means that we cannot use them for synchronization. After making them local, we see the *await* statements does not work anymore. To synchronize the access to the buffer we can use a boolean variable to control who should go next. The behavior from before was that the producer produces, but then waits for the consumer. It is alternating. This means that our solution will also work here.

```
int buffer;

# Variable to keep track of where the producer is
bool bufferEmpty := false;

process Producer{
    int p := 0;
    int a[N];
    while(p < N){
        <await (bufferEmpty = false)>
        buffer := a[p];
        p:= p + 1;
        bufferEmpty := true;

    }
}

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

1.4 Problem 4: Execution and atomicity

Consider the following program:

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

co
    x:= x + 1  # S1
    x:= x + 2; # S2
||
    x:= x + 2; # P1
    y:= y - x; # P2
oc
```

1.4.1 Part A

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?

Solution
Since all statements are atomic we know there is a limited amount of execution. But, *S1* must execute before *S2*, and *P1* must execute before *P2*. The possible orders of executions then becomes:

1. S1, S2, P1, P2 → x:= 5, y:=-5
2. S1, P1, S2, P2 → x:= 5, y:=-5
3. S1, P1, P2, S2 → x:= 5, y:=-3
4. P1, S1, P2, S2 → x:= 5, y:=-3
5. P1, S1, S2, P2 → x:= 5, y:=-5
6. P1, P2, S1, S2 → x:= 5, y:=-2

There is a total of 6 executions, and the end result is:
$(x := 5 \wedge (y := -5 \vee y := -3 \vee y := -2))$

1.4.2 Part B

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?

Solution
We know know there is three operations for each statement. We can simplify them as Read and Write. Read must happen before write. We can calculate executions by using this formula:

$$E = \frac{(n * m)!}{m!^n}$$

Where n is the number of processes and m are the number of atomic steps. In this case we have 2 processes and each process have 6 atomic steps:

$$\begin{aligned} E &= \frac{(n*m)!}{m!^n} \\ &= \frac{(2*6)!}{6!^2} \\ &= 924 \end{aligned}$$

There are a total of 924 possible executions.
We know that X is never influenced by another processes other than S2. However, when the assignment is split into multiple instructions, we see that some instructions may be forgotten. This may happen when a thread read variables and is then about to write, but before this, another thread has read and then adds. But this new value is not read. This means x could be $x \in 5, 4, 3, 2$. The y values would then be then be based on those values. But note that y could also -1
The end values are:

$$(x, y) \in (3, -3), (4, -4), (5, -5), (2, -2), (3, -1), (5, -2), (4, -2), (3, -2)$$

For such a task: *hard to find all solutions, and know when you have all solutions.*

1.5 Problem 5: Interleaving, non-determinism, and atomicity

Consider the following program:

```
int x:= 2, y := 3;

co
    <x := x + y> # S1
||
    <y := x * y> # S2
oc
```

1.5.1 Part A

What are the possible values for X and Y?

Solution
Since both operation are atomic then we get the following executions:

1. S1, S2 → x:= 5, y:=15
2. S2, S1 → x:= 8, y:=6

Meaning the values are:
$(x := 5 \wedge y := 15) \vee (x := 8 \wedge y := 6)$

1.5.2 Part B

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 available. What are the possible final values of x and y now?

Solution
Since now we have to consider reading and writing operations, we get the following possible executions:

1. R1, W1, R2, W2 → x:= 5, y:= 15
2. R1, R2, W1, W2 → x:= 5, y:= 6
3. R1, R2, W2, W1 → x:= 5, y:= 6
4. R2, R1, W1, W2 → x:= 5, y:= 6
5. R2, R1, W2, W1 → x:= 5, y:= 6
6. R2, W2, R1, W1 → x:= 8, y:= 6

If both processes read before writing, then we get the same result, no matter what process reads first and what process writes first. (Both read, then write). In the case of writing and then another process read, then we get different results. This leads to the following end result:
$(x := 5 \wedge (y := 15 \vee y := 6)) \vee (x := 8 \wedge y := 6)$

1.6 Problem 6: AMO - At most once

Consider the following program:

```
int x:= 1, y:= 1;

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

1.6.1 Part A

Do S1, S2 and S3 satisfy the requirements of the At-Most-Once Property?

Solution
At most once property is used to check interleaving between the processes. *Statements that fulfills the AMO-property can be considered atomic.*

First *S1*, it is atomic, and we therefor does not need to check the AMO property. For *S2*, we see that it assigns a non-critical reference. This means that it satisfy the AMO property. For *S3*, it uses both x and y, which are both critical referees. S1, S2 meets the AMO property, but S3 does not.

1.6.2 Part B

What are the final values for x and y? Explain your answer.

Solution
First, we see that are 3 processes which has either 1 or 2 atomic operations. Since S2 meets the AMO property, we can assume that it is atomic!
This will lead to the following executions:

1. S1, S2, R3, W3 *to* x:= 2, y:= 0
2. S1, R3, S2, W3 *to* x:= 2, y:= 1
3. S1, R3, W3, S2 *to* x:= 2, y:= 0
4. S2, S1, R3, W3 *to* x:= 1, y:= 0
5. S2, R3, S1, W3 *to* x:= 1, y:= 0
6. S2, R3, W3, S1 *to* x:= 1, y:= 0
7. R3, W3, S1, S2 *to* x:= 1, y:= 0
8. R3, W3, S2, S1 *to* x:= 0, y:= 0

The result is then:
$(y := 0 \wedge (x := 0 \vee x := 1 \vee x := 2)) \vee (x := 2 \wedge y := 1)$

1.7 Problem 7: AMO and termination

Consider the following program:

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

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

1.7.1 Part A

Do all parts of the program satisfy the AMO-property?

Solution
All assignments meets the AMO property in the program.

1.7.2 Part B

Will the program terminate? Always? Sometimes? Never?

Solution
The program *may* terminate (sometimes). It can happen when both processes see that x equal to y. S1 approaches from the bottom, and S2 approaches from the top. But there is a case where x is increased above current y value. This will lead to both process incrementing/decrementing past each other. If this happens the program cannot terminate.

2 Exercise #2

The topic of this exercise is synchronization of critical sections

2.1 Problem 1: Weak scheduling

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
Weak fairness ensures that if a process is enabled and remains enabled, it must eventually execute.

We see that $x = 1$ will terminate because we do P3, then P1 or P2. It will also terminate when $x = 6$. Both cases enables statements or makes the state stay enabled when first enabled. Any other values would not enable P3. When $x = 3 \lor x = 4$ it may terminate based on the order of execution.
To summarize

1. $x <= 0$: Lower than 0, then all is blocking. No termination
2. $x = 1$: will start P3, and P2, P1 can execute at any order. Terminates
3. $x = 2$: will enter P2, and then x = 0. Then the other two will not terminate. No termination
4. $x = 3$: May terminate. If enter P1 then it will not terminate (x will then be set be 0). If enter P2 it will terminate.
5. $x = 4$; May terminate.
6. $x = 5$: No termination. X will be 0 after P1 and P2, and therefor will not execute P3
7. $x = 6$: Always terminates. P1 and P2 will happen at any order and P3 will determinate.
8. $x >= 7$: Never terminate because x will always be greater than 1, so P3 will not happen.

2.2 Problem 2: Weak scheduling

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

1. $x < -2$ does not terminate because after P2, then there is nothing that enables P1 and P3
2. $x = -2$ does not terminate because after P2, P3, then P1 will not be enabled.
3. $x = -1$ terminates under P2, P1, P3
4. $x = 0$ terminates under P3, P2, P1
5. $x = 1$ terminates under P1, P3, P2
6. $x = 2$ does not terminate, because after P1, x is 1 and there is no condition that gets enabled
7. $x > 2$ does not terminate for the same reasons as x equal to 2.

The final values are: $x \in [-1, 1]$

2.3 Problem 3: Termination under scheduling strategy

Consider the following program:

```
int x := 10;
bool x:= true;

co
    <await (x=0);> c:=false # P1
||
     while(c) <x:=x−1>         # P2
oc
```

2.3.1 Part A: termination under weak fairness

Does the program terminate under weak scheduling?

Solution
Weak scheduling ensures that if a condition is enabled and stays enabled, then it will terminate. In this case we see P2 is enables and starts to decrement x. P1 is only enabled when $x = 0$. This may happen, but we see that it get enabled until P2 executes again such that x is less than 0. This means that it may terminate, but it is not guaranteed.

2.3.2 Part B: termination under strong fairness

Does the program terminate under weak fairness?

Solution
Strong fairness ensures termination if it enables infinitely often. This is not the case. After x being decremented below 0, it may never become 0 again. This means that the program may terminate but there is no guarantee.

2.3.3 Part C

Program is not extended to:

```
int x := 10;
bool x:= true;

co
    <await (x=0);> c:=false          # P1
||
    while(c) <x:=x-1>                # P2
||
    while(c) {if (x<0) <x:=10>;}     # P3
oc
```

Does it now terminate under weak or strong fairness?

Solution
Under *weak fairness* it may terminate. Again for the same reasons as before. But again weak fairness only ensures termination when the condition is enabled and stays enabled. In this case it will be enabled, but not stay enabled.
However, under *strong fairness* it will terminate. This is because the new branch will ensure that the condition is infinitely enabled. Thus it terminates.

2.4 Problem 4: Dekker's Algorithm

The following is Dekker's Algorithm. It is a solution to the critical section problem for two processes.

```
bool enter1:= false, enter2:= false;
int turn := 1;

process P1{
    while(true){
        # Entry protocol
        enter1:= true;
        while(enter2){
            if (turn = 2){
                enter1 := false;
                while(turn = 2) skip;
                enter1 := true;
            }
        }

        # CS

        # Exit protocol
        enter1:= false;
        turn := 2;

        # Non CS
    }
}


process P2{
    while(true){
        # Entry protocol
        enter2 := true;
        while(enter1){
            if(turn = 1){
                enter2:= false;
                while(turn = 1) skip;
                enter2 := true;
            }
        }

        # CS

        # Exit protocol
        enter2:= false;
        turn := 1;

        # Non–CS
    }
    }
```

Check if the following properties are satisfied:

2.4.1 Part A: Mutal Exclusion

To exclude the other process the entry of the process has to be true and the turn has to be its own number. When this is the case the other process will enter the while loop with the condition of entering. The process will not break out of this loop until the entry is set to false AND the turn is set to the current processes turn. This is only possible after the other process has finished the execution of the statements, that the other process can enter.
Yes, there is mutual exclusion.

2.4.2 Part B: Absence of deadlock

Deadlocks happens at both processes being stuck on a while loop.
The first while loop is when both check the condition of the entry. If both sets this condition, then both enter the while loop and the turn will go to the process based on the turn variable. No deadlock here.
The second while loop for skipping when it is the other process turn. This turn variable can never be set to both. One process will be skipping, and then enter the critical section as the other one is exiting. Since they never can be stuck and turn is only modified as the last exit protocol statement, then there is no deadlock.
Can they be at the two different while loops? No, because then one will be waiting for their turn and the other would wait for their entry, and that is not possible. Before the process will wait for the turn, it will open entry for the other.
Yes, there is an absence of deadlock. (No deadlock)

2.4.3 Part C: Absence of unnecessary delay

Absence of unnecessary delay, means that if one process is in its non-critical section and the other is, then at least one will succeed.
Lets say that P1 is in the non-critical section. This can only happen if enter1 is false. When this is the case P2 will go straight to the critical section.
Yes, there is absence of unnecessary delay.

2.4.4 Part D: Eventual Entry

Eventual entry means that both will enter the critical section. When both compete for entry, then they will wait on the inner loop. This inner loop is dependent on the turn variable which can either be 1 or 2. Then one will enter and set the turn to the other process. This will give the other process presidents over the last process.
Every process will enter the critical section eventually.

3 Exercise #6

Topic is types for concurrency

3.1 Problem 1: Extend type system

Extend the following syntax for statements s such that the syntax also defines return statements, which enable to return the value defined by an expression. Given the syntax for types T, define a typing rule for the return statements. You can assume that the typing rules given from the lecture already exist

$$\begin{aligned} &\mathrm{s} ::= \mathrm{v} = \mathrm{e}; \mathrm{s} \mid \mathrm{Tv} = \mathrm{e}; \mathbf{s} \mid \text{ skip } \mid \text{ if } (\mathrm{e})\{\mathbf{s}\}\mathrm{s} \\ &\mathrm{e} ::= n \mid \text{ true } \mid \text{ false } \mid \mathrm{e} + \mathrm{e} \mid \mathrm{e} \wedge \mathrm{e} \mid \mathrm{e} \leq \mathrm{e} \mid \mathrm{v} \\ &\mathrm{T} ::= \text{ Int } \mid \text{Bool} \mid \text{Unit} \end{aligned} \tag{1}$$

Solution
To give this the correct syntax, we extend the statements s to be:

$$\mathrm{s} ::= \mathrm{v} = \mathrm{e}; \mathrm{s} \mid \mathrm{Tv} = \mathrm{e}; \mathbf{s} \mid \text{ skip } \mid \text{ if } (\mathrm{e})\{\mathbf{s}\}\mathrm{s} \mid \text{return e}$$

Because *return* is a statement, and we return an expression.

3.2 Problem 2: Function Call Type System

We extend the syntax given in Problem 1 such that we can perform function calls, where a function f maps an argument of type T_1 to type T_2, i.e., $f : \mathrm{T}_1 \mapsto \mathrm{T}_2$. An example of a function would be a successor function succ: Int $\mapsto$ Int that increases a given integer by one. Similar to variables, functions are also stored in the typing environment Γ. An example of Γ with a function would be $\Gamma = \{ \text{ succ} \mapsto \text{Int} \mapsto \text{Int } \}$ We update the syntax as follows:

$$\mathrm{e} ::= n \mid \text{ true } \mid \text{ false } \mid \mathrm{e} + \mathrm{e} \mid \mathrm{e} \wedge \mathrm{e} \mid \mathrm{e} \leq \mathrm{e} \mid \mathrm{v} \mid \mathrm{f(e)}$$

The type system of Example 1 is extended by the following typing rule:

$$\frac{\Gamma(\mathrm{f}) = \mathrm{T}_1 \mapsto \ \mathrm{T}_2 \quad \Gamma \vdash \mathrm{e} : \mathrm{T}_1' \quad \mathrm{T}_1' <: \mathrm{T}_1}{\Gamma \vdash \mathrm{f(e)} : \ \text{Unit}} \text{ function-call}$$

Does this extension of the type still ensure type soundness considering also the typing rules in the lecture? If not, can you give an example that shows the violation of type soundness.

Solution
This violates types soundness because a function-call is an expression and therefor not a statement. In the original typing rule it is typed as an statement. To change this, we use T_2 as the type of the function call, where T_2 is the return type. We can also validate this with

3.3 Problem 3: Annotate Go Program for different type systems

Consider the following Go code:

```
func add(inp chan int, fin chan bool, res chan int){
    sum := 0
    for{
        select{
            case num:= <-inp:{
                sum = sum + num;
            }

            case <-fin:{
                res <- sum
            }
        }
    }
}

func provide(inp chan int, fin chan bool){
    inp <- 9
    inp <- 3
    fin <- true
}


func main(){
    input := make(chan int)
    finish := make(chan bool)
    res := make(chan int)

    go add(input, finish, res)
    go provide(input, finish)
    <- res
}
```

Annotate the channel types such that they support the following:

3.3.1 Modes

Modes describe what channels are allowed to do. A channel can either send, receive or both. The send operations is denoted with *!* and receive operations are denoted with *?*. For the program we can annotate it the following way:

```
func add(inp chan? int, fin chan? bool, res chan! int){...}

func provide(inp chan! int, fin chan! bool){...}
func main(){
    input := make(chan?! int)
    finish := make(chan?! bool)
    res := make(chan?! int)
    ....
}
```

3.3.2 Linear Types

Linear usage types keep track of how many times channel can do operations. With linear types we specify how many times an operation is allowed of each mode. *w* denotes more than two times and are noted as arbitrary amount of time. We denote this with the following:

```
func add(inp chan<?.w> int, fin chan<?.1> bool, res chan<!.1> int){...}

func provide(inp chan<!.w> int, fin chan<!.1> bool){...}

func main(){
     input := make(chan<?.w.!.w> int)
     finish := make(chan<?.1.!.1> bool)
     res := make(chan<?.1.!.1> int)
     ....
}
```

3.3.3 Usage Types

Usage types are denoted with how many times and the order of operations (the order of operation is very powerful thing to denote). The orders of operations are noted, and at the end there is 0 to make the channel not usable. We use the + operator when we create a channel to denote that the different types used in a channel. It shows the types of the channel when it is used by more than one thread. Unlike linear types, we do not use the *w* notation for arbitrary amount of operations.

```
func add(inp chan<?.?.0> int, fin chan<?.0> bool, res chan<!.0> int){...}

func provide(inp chan<!.!.0> int, fin chan<!.0> bool){...}

func main(){
     input := make(chan<?.?.0 + !.!.0> int)
     finish := make(chan<?.0 + !.0> bool)
     res := make(chan<?.0 + !.0> int)
     ....
}
```

3.4 Problem 4: Typing Three

For this problem, we were given a small go program to type check with typing three See solution written in notebook.
Key takeaways:

1. Follow the given for the environment rules, and split the environment
2. An unrestricted typing environment is one where are channels have used up their capabilities.
3. There should only be one place in the typing three where there is an error, if it does not type check.