



UiO : **Department of Informatics**  
University of Oslo

IN5170 MODELS OF CONCURRENCY

## Exercise #1-7

*Author:*  
Kjetil K. Indrehus

November 14, 2024

## Contents

<b>1</b>	<b>Exercise #1</b>	<b>2</b>
1.1	Problem 1 . . . . .	2
1.2	Problem 2 . . . . .	2
	1.2.1 Part A . . . . .	3
	1.2.2 Part B . . . . .	4
1.3	Problem 3 . . . . .	5
1.4	Problem 4 . . . . .	6
	1.4.1 Part A . . . . .	7
	1.4.2 Part B . . . . .	7
1.5	Problem 5 . . . . .	8
	1.5.1 Part A . . . . .	8
	1.5.2 Part B . . . . .	9

# 1 Exercise #1

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

## 1.1 Problem 1

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

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 readers. We introduce two *done* variables to signal termination for both. The two readers read one file each, and then put 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 them to both write to 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

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

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.  $S1, S2, P1, P2 \rightarrow x:= 5, y:=-5$
2.  $S1, P1, S2, P2 \rightarrow x:= 5, y:=-5$
3.  $S1, P1, P2, S2 \rightarrow x:= 5, y:=-3$
4.  $P1, S1, P2, S2 \rightarrow x:= 5, y:=-3$
5.  $P1, S1, S2, P2 \rightarrow x:= 5, y:=-5$
6.  $P1, P2, S1, S2 \rightarrow x:= 5, y:=-2$

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 there are 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 has 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 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

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  $\rightarrow$  x:= 5, y:=15
2. S2, S1  $\rightarrow$  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  $\rightarrow$   $x := 5, y := 15$
2. R1, R2, W1, W2  $\rightarrow$   $x := 5, y := 6$
3. R1, R2, W2, W1  $\rightarrow$   $x := 5, y := 6$
4. R2, R1, W1, W2  $\rightarrow$   $x := 5, y := 6$
5. R2, R1, W2, W1  $\rightarrow$   $x := 5, y := 6$
6. R2, W2, R1, W1  $\rightarrow$   $x := 8, y := 6$

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:  
 $(x := 5 \wedge (y := 15 \vee y := 6)) \vee (x := 8 \wedge y := 6)$