UiO : Department of Informatics
University of Oslo

IN5170 Models of Concurrency

Exercise #1-7

Author:
Kjetil K. Indrehus

November 14, 2024

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 reader. We introduce two *done* variables to signal termination for both.

```
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 Reader1 { # read new lines
    string line2;
    while (true) {
        if (EOF) {done := true; break;}

        <await bufferEmpty {
            buffer := line2;
            bufferEmpty := false;
        }>
    }
}

process Reader2 { # read new lines
    string line3;
    while (true) {
```

```
            if (EOF) {done := true; break;}

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