UiO : Department of Informatics
University of Oslo

IN5170 Models of Concurrency

Exam 2023

Author:
Kjetil K. Indrehus

November 23, 2024

Contents

1 General Questions

1.1 Problem 1: Interference and AMO

Consider the following program:

```
co
    <x:= x + y> // P1
||
    <y:= y + x> // P2
oc
```

Is the program Interference free?

To check it we can write the v and w variables:

$$v_{P1} = \{x, y\}, w_{P1} = \{x\}$$

$$v_{P2} = \{x, y\}, w_{P2} = \{y\}$$

Checking if the read and write interfere:

$$v_{P1} \wedge w_{P2} = \{y\} \neq \emptyset$$

$$v_{P2} \wedge w_{P1} = \{x\} \neq \emptyset$$

We see there are interference in the program, since neither union of read and write variables are empty set.

Does the two assignment satisfy AMO-property?

Since two assignments does not satisfy the AMO property since. We can verify this by using the AMO rule for assignments. It assigns a critical reference and it is also referenced in the other process. This means that the order of operations would lead to different results. It does not fulfills the AMO property.

1.2 Problem 2: fairness

Explain the difference between weak and strong fairness.

Solution

Fairness ensures that enabled statements should not be systematically be neglected by the scheduling strategy. We use conditions that are enabled or disabled. Both has to be *unconditional fair*. A scheduling strategy is unconditional fair if each enabled unconditional atomic action will eventually be chosen.
Weak fairness is when a condition gets enabled and stays enabled, then it will execute. Under *strong fairness* if a condition is infinitely enabled, then it will execute.

1.3 Problem 3: promise and future

Explain the difference between promises and futures.

Solution

A **future** is an abstraction used to represent the result of an asynchronous computation. It acts as a read-only handle for the caller, allowing it to access the result of the computation once it is available, and synchronize with the callee by waiting for the result or registering callbacks for when the result is ready.
In simpler terms, a future can be thought of as a mailbox that holds the return value of a task. Once the task is completed, the future contains the result. Futures can typically be read multiple times. Also allow the caller to wait (block) or register a callback to handle the result asynchronously.

A **promise**, on the other hand, is a writable counterpart to a future.

1. It is used to manually set or ̈fulfill ̈the value of a future.
2. Promises allow the producer (whoever completes the task) to signal that the computation is complete and provide the result.

Key Differences:

1. A future is the read-only handle used by the caller to access the result.
2. A promise is the write-access handle used by the producer (or callee) to fulfill the result.

Properties:

1. Futures can be read multiple times but cannot be written to directly.
2. Promises are write-once: they must be completed exactly once, either with a value or an error, and are often used to produce a future.

For example:

- The caller creates a promise and obtains a future associated with it.
- The callee completes the promise, thereby fulfilling the future.

This separation ensures that the caller and callee can operate independently, decoupling the production and consumption of the computation result.

1.4 Problem 4: future and linear channels

Explain the difference between a future and a linear channel used for callbacks.

Solution
A future can be read multiple times, but a channel can only be read once. This is the main difference between the two.

1.5 Problem 5: channels

Consider a channel for integers that is read by one thread and written by another.

1. Give a declaration for this channel as a linear type, a usage type and a session type. (1.5p.)
2. Each of these systems will split the type environment at the start of a new thread. Give
   (a) a type environment containing only the variables related to the declaration from above and
   (b) the split into the two split environments when a new thread is started for each type system. The started thread will perform the read. (4.5p.)

Solution
For task 1, we can declare each type like this:

Linear type is declared by showing how many operations are allowed on each thread:

$$cl = make(chan <!.1, ?.1 > int)$$

Usage type shows the order of allowed operations for each thread:

$$cu = make(chan <!.0+?.0 > int)$$

Session type makes a channel than must held the duality between each receive and send:

$$(c, d) = make(chan <!int.0 >, chan <?int, 0 >)$$

For task 2, we create the two splits like this:

$$\begin{aligned}\Gamma_1 = \{&cl \to chan <!.1, ?.1 > int,\\ &cu \to chan <!.0+?.0 > int,\\ &(c, s) \to chan <!int.0+?.int.0 >\}\end{aligned}$$

The new environments gets split into the following:

$$\Gamma_1 = \Gamma_2 + \Gamma_3$$

Write environment:

$$\begin{aligned}\Gamma_2 = \{&cl \to chan <!.1 > int,\\ &cu \to chan <!.0 > int,\\ &c \to chan <!int.0 >\}\end{aligned}$$

Read environment:

$$\begin{aligned}\Gamma_3 = \{&cl \to chan <?.1 > int,\\ &cu \to chan <?.0 > int,\\ &d \to chan <?.int.0 >\}\end{aligned}$$