UiO : Department of Informatics
University of Oslo

IN5170 Models of Concurrency

Exam 2023

Author:
Kjetil K. Indrehus

November 25, 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 f̈ulfill ẗhe 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:

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

2 Semaphores

We consider a barbershop with one barber and a waiting room with n chairs for waiting customers (n may be 0). The following rules apply:

- If there are no customers, the barber falls asleep.
- A customer must wake the barber if he is asleep.
- If a customer arrives while the barber is working, the customer leaves if all chairs are occupied and sits in an empty chair if it's available.
- When the barber finishes a haircut, he inspects the waiting room to see if there are any waiting customers and falls asleep if there are none.

2.1 Problem 6: Barbershop

Complete the code to provide a solution based on semaphores that ensures the following requirements:

- the barber never sleeps while there are waiting customers and
- there is never more than n customers waiting in the waiting room.

Briefly explain why your solution satisfies these requirements.

Solution
The code below is a solution for the given problem. For the sleeping barber we make sure that if this waiting customer is the first one to enter the waiting room, that it wakes up the barber. The barber will be awake as long as there are waiting costumers. It will only go to sleep when the number of waiting customers are 0. In that case, the next costumer will wake up the barber.

For the customers, we make sure that there are only the given amount of customers in the waiting room by having a variable for waiting customers. We use a semaphore with initial value 1, such that it acts as a mutex to enter the waitingRoom. If there are no seats available, then the costumer leaves.

```
int freeChairs := n;

// Number of waiting costumers
int nwc := 0;

// Enter waiting room mutex
sem waitingRoom := 1;

// Signal when barber allow a costumer enter the chair
sem barber := 0;
```

```
// Signal when costumer can leave
sem chair := 0;

// Signaled to wake up barber
sem barberSleep := 0;

procedure Costumers{
    while(true){
        // Try to enter the shop
        P(waitingRoom)
        if (freeChairs > 0){
            // Take a chair
            freeChairs := freeChairs - 1;

            // Wake up the barber if you are the first waiting costumer
            if(nwc = 0){
                V(barberSleep);
            }

            // Increment waiting costumers;
            nwc++;
        }else{
            // No place in the waiting room, leave
            V(waitingRoom)
            continue;
        }
        V(waitingRoom)

        // Waiting for be called by the barber
        P(barber)

        // Allowing a new costumer to enter waiting room
        P(waitingRoom)
        freeChairs++;
        nwc--;
        V(waitingRoom)

        // Getting haircut logic

        // Wait until asked to leave the shop
        P(chair)
    }

}
```

```
procedure Barber{
    while(true){
        P(waitingRoom)
        if (nwc > 0){
            // Allow costumer to enter the waiting room
            V(waitingRoom);
            V(barber);
        }else{
            V(waitingRoom);

            // Go to sleep
            P(barberSleep)

            // Woken up, try again
            continue;
        }

        // Doing haircut logic

        // Done! Asking customer to leave
        V(chair);
    }

}
```

2.2 Problem 7: fairness

Is the solution in problem 6 fair? Explain briefly

Solution
The solution is not fair for the costumers. Since we use semaphores and not queues, then we can make a customer wait in the waiting room or never being allowed to enter the barbershop. Semaphores uses busy wait. This means there are no ordering of the first or last to busy wait. To allow this we must make sure that the order of which the customers enter the barber shop, is the order they are processed. If the semaphores was implemented fairly, then the solution would be first (i.e implemented with FIFO order).

2.3 Problem 8: barbershop with priority

The barber shop introduces an "express" category of customers, who should have priority over regular customers. Implement a solution such that priority customers can bypass regular customers, using the same semaphores as in Problem 6. Provide a solution to the Barbershop with priorities by completing the code below. Briefly explain how your solution gives priority to express customer

Solution
The solutions keeps track of the amount of waiting priority costumers in the waiting room. When a express customer is waiting, it will always be asked to enter before the regular customers. The barber dictates who is next to get a haircut.

```
int seatsInBarbershop := n;
int freeSeats := seatsInBarbershop;

// Number of waiting costumers
int nwc := 0;

// Number of waiting priority customers
int npc := 0;

// Enter waiting room mutex
sem waitingRoom := 1;

// Signal when barber allow a costumer enter the chair
sem regularCustomerEnter := 0;
sem expressCustomerEnter := 0;

// Signal when costumer can leave
sem chair := 0;

// Signaled to wake up barber
sem barberSleep := 0;

procedure RegularCostumers{
    while(true){
        // Try to enter the shop
        P(waitingRoom)
        if (freeChairs > 0){
            // Take a chair
            freeChairs := freeChairs - 1;

            // Wake up the barber if you are the first waiting costumer
            if(nwc = 0){
                V(barberSleep);
            }

            // Increment waiting costumers;
            nwc++;
        }else{
            // No place in the waiting room, leave
            V(waitingRoom)
```

```
            continue;
        }
        V(waitingRoom)

        // Waiting for be called by the barber
        P(regularCustomerEnter)

        // Allowing a new costumer to enter waiting room
        P(waitingRoom)
        freeChairs++;
        nwc--;
        V(waitingRoom)

        // Getting haircut logic

        // Wait until asked to leave the shop
        P(chair)
    }
}

procedure ExpressCustomers{
    while(true){
        // Try to enter the shop
        P(waitingRoom)
        if (freeChairs > 0){
            // Take a chair
            freeChairs := freeChairs - 1;

            // Wake up the barber if you are the first waiting costumer
            if(nwc = 0){
                V(barberSleep);
            }

            // Increment waiting costumers;
            nwc++;

            // Increment amount of waiting priority customers
            npc++;
        }else{
            // No place in the waiting room, leave
            V(waitingRoom)
            continue;
        }
        V(waitingRoom)

        // Waiting for be called by the barber
```

```
        P(expressCustomerEnter)

        // Allowing a new costumer to enter waiting room
        P(waitingRoom)
        freeChairs++;
        nwc--;
        npc--;
        V(waitingRoom)

        // Getting haircut logic

        // Wait until asked to leave the shop
        P(chair)
    }

}

procedure Barber{
    while(true){
        P(waitingRoom)
        if (nwc > 0){
            // Allow costumer to enter the waiting room
            // Check if there are priority costumers
            if(npc > 0){
                V(expressCustomerEnter)
            }else{
                V(regularCustomerEnter)
            }
            V(waitingRoom);
        }else{
            V(waitingRoom);

            // Go to sleep
            P(barberSleep)

            // Woken up, try again
            continue;
        }

        // Doing haircut logic

        // Done! Asking customer to leave
        V(chair);
    }

}
```

2.4 Problem 9: properties of the barbershop problem

Does the solution in problem 8 ensure:

1. mutual exclusion?
2. absence of deadlock?
3. absence of unnecessary delay?
4. fairness? If the solution is not fair, explain how you could make it fair.

Solution
Mutual exclusion is guaranteed in this solution because we use a semaphore as a mutex to make changes to the variables. The mutex ensures that only one process is allowed to change the shared variables: *freeSeats, npc, nwc*. We also make sure that only one is allowed into the barbershop by making the barber allowing only one customer to enter the shop at the time. This is due to the barber giving haircuts is a sequential order.

Absence of deadlock is also guaranteed. There's no circular wait condition because resources (semaphores) are acquired and released in a consistent order. Barber will always be woken up. When the barber is awake it will take customers in the waiting room. We also make sure to release the semaphore that acts like a mutex always. Because of the mutex being correctly used, we see that customers logic will also always progress. The program will always progress.

Absence of unnecessary delay is also correct, because we only use mutex for code that has race conditions, We also release the lock as soon as possible. The seats are also free as soon as possible, making it available for new costumers to enter the shop.

Fairness is again not guaranteed. This is because of the same reason of Problem 6. Even though it given more priority to the express customers, it will still do busy wait for allowing what customers to enter. Also, there is the problem of having continues express customers enter the shop. When this is the case, i.e there is always a express customer in the waiting room, then there is no chance for the regular customers to be handled. There is no known ration between regular customers and express customers. If this ratio is low, then the solution will work. We always will give priority to express customers, but there is no policy in how many express customers in a row can be handled before a regular customer must be used. Such a policy could guarantee that regular customers are also processed. For example after every x customer, always handle a regular customer. This policy should be implemented with knowledge of how many express customers there are.

3 Monitors

We consider monitors with the following operations:

```
cond cv;
wait(cv);
signal(cv);
signal_all(cv);
```

3.1 Problem 10: Barbershop monitor with priorities

Use the monitor operations listed above to make a monitor solution to the priority customer barber shop of Exercise 8. Provide your solution by completing the code below and explain briefly how your solution gives priority to express customers.

Solution
The solution uses Signal and Continue (SC). We make sure to give priority to express customers by signaling them before regular customers. The barber will always prioritize express customers over regular customers.

```
monitor Barbershop {
    int seatsInBarbershop = n;
    int nr = 0; // number of regular customers
    int np = 0; // number of priority customers

    // Queue of regular customer
    cond queuer;

    // Queueu of express customer
    cond queuep;

    // Signal to wake up barber
    cond customerReady;

    procedure barber(){
        while (true) {
            // Check if waiting customer
            while(nr + np) == 0 {
                wait(customerReady)
            }

            // There is a customer
            // Signal express customer before regular.
            if (np > 0) {
                signal(queuep);
            } else {
```

```
                signal(queuer);
            }
        }
    }
    procedure regularCustomer() {
        while (true) {
            if (nr + np) < seatsInBarbershop {
                nr := nr + 1;
                signal(customerReady);
                wait(queuer);
                nr := nr - 1;
            }
        }
    }
    procedure priorityCustomer() {
        while (true) {
            if (nr + np) < seatsInBarbershop {
                np := np + 1;
                signal(customerReady);
                wait(queuep);
                np := np - 1;
            }
        }
    }
}
```

3.2 Problem 11: Monitor invariant

What could be a monitor invariant for the Barbershop monitor? Explain briefly why the monitor invariant holds for your monitor solution.

Solution
The monitor must hold the following requirements:

1. The number of total customers must be less than seats available.
2. There must be 0 or more priority customers. The same must hold for regular customers.

It can be expressed as the following invariant:

$$(\text{np} + \text{nr} \leq \text{freeSeats}) \wedge (\text{np} \geq 0) \wedge (\text{nr} \geq 0)$$

4 Async Message Passing

In the following we assume that messages are never lost, but can arrive in a different order than they are sent.

4.1 Problem 12: Actor with Erlang Style Code

Write two actors using the code skeleton below that implements the following behavior. For each state of the actor, use a different loop:

> You are designing a web application with two components: GUI and Backend. The backend stores a single value that can be set and retrieved through the GUI. The GUI is either WAITING or RESPONSIVE. If it is RESPONSIVE, it accepts messages of the form (GET, user) and (SET, n). In the first case, it changes its state to WAITING, stores the user and sends a message to the backend to retrieve the stored value. In the latter case, it sends a message to the backend to store the value n. If it is WAITING, it only accepts messages of the form (VALUE,n), send the value n to the stored user. The backend accepts (SET, n) messages to update its stored value and answers (GET) messages by sending the stored value to the GUI.

Solution

```
handleGUIRequest(state, user) ->
    receive
        {from, GET, user} -> {

            handleGUIRequest(WAITING, user)
        },
        {from, SET, n} -> {

        }


handleBackendRequest(value) ->
    receive
        {}


% Create a new GUI Actor
GUI {
    start() -> spawn (fun() -> handleGUIRequest())
}

%  Create a new Backend Actor with initial value of 0
```

```
Backend{
    start() -> spawn (fun() -> handleBackendRequest(0))
}
```

4.2 Problem 13: Actor message flow

5 Types

5.1 Problem 14: Rust type checking

Consider the following program

```
fn f(write_ref: &mut Vec<i32>){
    write_ref[0] = 0;
}

fn g(read_ref : &Vec<i32>){
    println!("{}", read_ref[0])
}
```

5.1.1 Part A: Annotate owner

Does the following program type check? Annotate for each line the current owner of the created vector.

```
fn main(){
    let mut vec = vec![1,2,3];
    f(&mut vec);
    thread::spawn(move || g(&vec));
}
```

Solution
The program does typecheck! The mutable reference is used before we move the closure. The owner is always *vec* until the spawn of the new thread. Then we cannot use vec anymore due to the mov keyword.

```
fn main(){
    let mut vec = vec![1,2,3]; // Owner: vec
    f(&mut vec); // Owner: vec
    thread::spawn(move || g(&vec)); // Owner: thread owns the vec now
}
```

5.1.2 Part B: Ownership typechecking

```
// #1
fn main(){
    let mut vec = vec![1,2,3];
    f(&mut vec);
    thread::spawn(move || g(&vec));
    g(&vec);
}

// #2
```

```
fn main(){
    let mut vec = vec![1,2,3];
    f(&mut vec);
    g(&vec);
    g(&vec);
}

// #3
fn main(){
    let mut vec = vec![1,2,3];
    f(&mut vec);
    g(&vec);
    g(&vec);
}
```

Solution
For program *1*, it does not typecheck. Because the ownership of the *vec* to the thread. This means that after that line the vector cannot be used since it is moved out of scope.

For program *2* and *3* does typecheck. The core rule of rust borrowing is:

> At any give time, you can either have:
>
> 1. One mutable reference
> 2. Any number of immutable references.

But, since we to a temporary borrow here, the ownership is returned at the end of the function call. It would be different if we created variables and not dropped them before calling new references. Then the borrow would stay active until the end of the program. But instead we return the borrowed memory after using it.

5.2 Problem 15: Linearity

Consider the following Go-like code. Does it type-check? If no, give the line of the statement where type-checking fails, the reason and the line where the misused channel is declared. If yes, annotate for each declared channel (declared in the variables c,d,e) the line where it is read and where it is written.

```
func main(){
    c = make(chan<!1,?1> int)
    d = make(chan<!1,?1> chan<!1,?1> int)
    e = make(chan<!1,?1> int);

    f = 0;
    res = 0;
    ret = 0;
```

```
        go func{
            go func{
                d <-e; e <- 1; skip;
            }

            res = <-d;
            ret = 0;
            if((<-res) < 0){
                ret = -1;
            }else{
                ret = 0;
            }
            c <- ret*(<-e); skip;
        }

        f = <-c; skip;
    }
```

Solution
It fails on line 19. It reads E twice! We read the channel e into d which is read once (on line 12), and then we read e again on line 19. This will make the program not type checked.

```
    func main(){
        c = make(chan<!1,?1> int)
        d = make(chan<!1,?1> chan<!1,?1> int)
        e = make(chan<!1,?1> int);

        f = 0; // c -> !1,?1, d -> !1,?1, e -> !1,?1
        res = 0; // c -> !1,?1, d -> !1,?1, e -> !1,?1
        ret = 0; // c -> !1,?1, d -> !1,?1, e -> !1,?1

        go func{ // c -> !1,?1, d -> !1,?1, e -> !1,?1
            go func{ // c -> !1,?1, d -> !1,?1, e -> !1,?1
                d <-e; e <- 1; skip; // c -> !1,?1, d -> !0,?1, e -> !0,?1
}

            res = <-d; // c -> !1,?1, d -> !0,?0, e -> !0,?1
            ret = 0; // c -> !1,?1, d -> !0,?0, e -> !0,?1
            if((<-res) < 0){ // c -> !1,?1, d -> !0,?0, e -> !0,?0
                // Does not enter
                ret = -1;
            }else{ // c -> !1,?1, d -> !0,?0, e -> !0,?0
                ret = 0; // c -> !1,?1, d -> !0,?0, e -> !0,?0
            }
            // Error here! We are reading from channel e, but it has used up all
```

```
        c <- ret*(<-e); skip;

    }

    f = <-c; skip;
}
```

5.3 Problem 16: Binary Session types

Consider the following Go-like code

```
func main(b bool, val1 int, val2 int){
    (c, c_dual) = make(chan T, chan T_)

    go f(c_dual);

    if (b){
        (r, r_dual) = make(chan S, chan S_)
        c <- l_1;
        c <-r;
        c <-val1;
        r_dual <-val2;

        if (<-r_dual){
            println("success);
        }else{
            println("failure");

        }else{
            c <- abort;
        }
    }
}
```

5.3.1 Part A: sessiontype T and S

Give the session type for T and S so the program is well typed.

Solution

$$T = \& \begin{cases} l_1 : & !chan < S > .!int.0 \\ abort : & 0 \end{cases}$$

$$S = ?int.!bool.0$$

5.3.2 Part B: Duals

Give the duals of T and S .

Solution

$$\hat{T} = \oplus \begin{cases} l_1 : & ?chan < S > .?int.0 \\ abort : & 0 \end{cases}$$

$$\hat{S} = !int.?bool.0$$

5.3.3 Part C: subtype

Give the subtype of T and $\hat{T}$

Solution
We know the following:

$$T = \& \begin{cases} l_1 : & !chan < S > .!int.0 \\ abort : & 0 \end{cases}$$

$$\hat{T} = \oplus \begin{cases} l_1 : & ?chan < S > .?int.0 \\ abort : & 0 \end{cases}$$

The subtypes are expressed as followed:

$$T' <: T$$

$$\hat{T'} <: \hat{T}$$

The subtypes are expressed like this:

$$T' = \& \begin{cases} l_1 : & !chan < S > .!int.0 \\ abort : & 0 \\ l : & 0 \end{cases}$$

$$\hat{T'} = \oplus \begin{cases} l_1 : & ?chan < S > .?int.0 \end{cases}$$