Universitetet i Oslo
Institutt for Informatikk
Research group for Reliable Systems (PSY)

IN5170: Models of Concurrency

Fall 2023 28.11.2023

Issued: 28.11.2023

Merk oppgaven med at det er lov å tegne på papir!

1 General Questions

Exercise 1 (4p.) Consider the following program

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

1. Is the program interference-free? Explain your answer. (2p.)
2. Do the two assignments satisfy the AMO-property? Explain your answer. (2p.)

Exercise 2 (2p.) Explain the difference between weak and strong fairness.

Exercise 3 (3p.) Explain the difference between promises and futures.

Exercise 4 (2p.) Explain the difference between a future and a linear channel used for callbacks.

Exercise 5 (6p.) 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 variable(s) 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.)

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.

Exercise 6 (Barbershop (10p.)) Complete the code below 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.

```
int freeSeats = n;

process Customer {
  while (true) {


  }
}
process Barber {
  while (true) {


  }
}
```

Exercise 7 (Fairness of Barbershop (2p.)) Is your solution to Exercise **6** fair? Explain briefly.

Exercise 8 (Barbershop with priorities (10p.)) 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 Exercise **6** above. Provide a solution to the Barbershop with priorities by completing the code below. Briefly explain how your solution gives priority to express customers.

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

process RegularCustomer {
  while (true) {


  }
}


process PriorityCustomer {
  while (true) {


  }
}
```

```
process Barber {
  while (true) {


  }
}
```

Exercise 9 (Properties of barbershop with priorities (8p.)) Does your solution to Exercise **8** guarantee:

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

Explain each answer briefly.

3 Monitors

We consider monitors with the following operations:

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

Exercise 10 (Barbershop monitor with priorities (12p.)) 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.

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

  procedure barber(){
    while (true) {


    }
  }

  procedure regularCustomer() {
    while (true) {


    }
  }

  procedure priorityCustomer() {
    while (true) {


  }
}
```

Exercise 11 (Monitor invariant (2p.)) What could be a monitor invariant for the Barbershop monitor? Explain briefly why the monitor invariant holds for your monitor solution.

4 Asynchronous message passing

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

Exercise 12 (Actors (12p.)) Write two actors using the code skeleton below that implements the following behavior. For each state of the actor, use a different loop. You find a reminder about syntax on the next page.

> 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.

Exercise 13 (Actors (3p.)) Assume that the messages sent by the users arrive in the order they are sent.

1. Does your solution of Exc. 12 ensure that a user that sends a `(SET,n)` message followed by a `GET` message gets `n` as an answer (if no other user communicated with GUI or backend)? Describe an execution to explain an answer.

5 Syntax for actors

```
// Create a new actor that runs a method actorMethod
Actor {
  start() -> spawn (fun() -> actorMethod(val))
}

// Declaration of method that accepts a variable
actorMethod(val) -> // method body

// Declaration of method that does not accept a variable
actorMethod() -> // method body

// Receive messages
receive
  {message_type1} -> // do something ;
  {message_type2} -> // do something else ;
  _ -> // do if you receive other messages than the above

// Send message message_type to receiver
receiver!{message_type};
```

6 Types

Exercise 14 (Rust (9p.)) Consider the following two methods.

```
1  fn f(write_ref : &mut Vec<i32>){
2    write_ref[0] = 0;
3  }
4  fn g(read_ref : &Vec<i32>){
5    println!("{}", read_ref[0]);
6  }
```

1. Does type checking succeed? Annotate for each line of code below the current owner of the created vector.

```
1  fn main() {
2    let mut vec = vec![1,2,3];
3    f(&mut vec);
4    thread::spawn(move || g(&vec));
5  }
```

2. For which of the below versions of `main` ([1], [2], [3]) does type checking fail? For each version explain why it fails (if it does) or why the restrictions on ownership and references are satisfied (if is succeeds)

```
1   fn main() { //[1]
2     let mut vec = vec![1,2,3];
3     f(&mut vec);
4     thread::spawn(move || g(&vec));
5     g(&vec);
6   }
7   fn main() { //[2]
8     let mut vec = vec![1,2,3];
9     f(&mut vec);
10    g(&vec);
11    g(&vec);
12  }
13  fn main() { //[3]
14    let mut vec = vec![1,2,3];
15    f(&mut vec);
16    f(&mut vec);
17    g(&vec);
18  }
```

Exercise 15 (Linearity (7p.)) 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.

```
1   func main(){
2     c = make(chan<!1,?1> int)
3     d = make(chan<!1,?1> chan<!1,?1> int)
4     e = make(chan<!1,?1> int);
5     f = 0;                // c -> !1,?1, d -> !1,?1
6     res = 0;
7     ret = 0;
8     go func {
9       go func {
10        d <- e; e <- 1; skip;
11      }
12      res = <-d;
13      ret = 0;
14      if((<-res) < 0) {
15         ret = -1;
16      } else {
17         ret = 0;
18      }
19      c <- ret*(<-e); skip;
20    };
21    f = <-c; skip;
22  }
```

Exercise 16 (Binary Session Types (8p.)) Consider the following Go-like code.

```
func main(b bool, val1 int, val2 int){
  (c, c_dual) = make(chan T, chan T̄);
  go f(c_dual);
  if(b) {
    (r, r_dual) = make(chan S, chan S̄);
    c <- l1;
    c <- r;
    c <- val1;
    r_dual <- val2;
    if( <-r_dual ) println("success");
    else           println("failure");
  }
  else c <- abort;
}

func f(c chan T̄){
  switch <-c {
    case l1:
       ret = <- c;
       param1 = <-c;
       param2 = <- ret;
       ret <- param1 + param2 >= 0;
    case abort:
    }
}
```

1. Give the session types for `T` and `S` so the program is well-typed.
2. Give the duals of `T` and `S`.
3. Give a subtype of $\mathtt{T}$ and subtype of $\overline{\mathtt{T}}$.