Universitetet i Oslo
Institutt for Informatikk
Research group for Reliable Systems (PSY)
Silvia Lizeth Tapia Tarifa, Riccardo Sieve

INF 5170: Models of Concurrency

Fall 2026 — **Group Session 5** — 30.09.2026

Topic: Actors and Message Passing

Exercise 1 Implement an actor for a calculator that has two states (`DUAL` and `SNGL`). In state `DUAL`, it accepts messages of the following form.

- Message (`from`, `ADD`, `n`, `m`) should send back `n+m` to the sending actor `from`
- Message (`from`, `SNGL`) should switch to state `SNGL`

In state `SNGL`, it accepts messages of the following form.

- Message (`from`, `STORE`, `n`) should store value `n` in the memory of the actor
- Message (`from`, `INC`, `n`) should send back the sum of `n` and the stored value to the sending actor `from`
- Message (`from`, `DUAL`) should switch to state `DUAL`

Solution: See Listing 1.

Listing 1: Actor processes for calculators

```
1  loop1() −> receive
2    {from, ADD, n, m} −> from!{n+m}, loop1();
3    {from, SNGL} −> loop2(0);
4
5  loop2(n) −> receive
6    {from, STORE, m} −> loop2(m);
7    {from, INC, m} −> from!{n+m}, loop2(n);
8    {from, DUAL} −> loop1();
9
10 Calculator {
11   start () −> spawn ( fun() −> loop1()).
12 }
13 User {
14   start () −> Caclulator!{self, ADD, 1, 2};
15   receive {n} −> print("result: "+n);
16 }
```

Exercise 2 Implement an actor process for calculators as above in Go. You find the code skeleton in Listing 2.

Solution: See Listing 3.

Listing 2: A skeleton for an actor process for calculators in Go

```
package main

import "fmt"

type OP int

const (
        ADD   OP = 0
        INC   OP = 1
        STORE OP = 2
        DUAL  OP = 3
        SNGL  OP = 4
)

type Msg struct {
        op  OP
        p1  int
        p2  int
        ret  chan int
}

type State struct {
        /* define state here*/
}

func loop1(ch chan Msg, state State) {
        /* add code here*/
}

/* add functions here */

func main() {
        // simple test case ,write more
        input := make(chan Msg)
        go loop1(input, /* your starting state */)
        res := make(chan int)
        input <- Msg{STORE, 2, 0, res}
        input <- Msg{INC, 5, 0, res}
        fmt.Println(<-res) //should print 7
}
```

Listing 3: An actor process for calculators in Go

```
1  package main
2
3  import "fmt"
4
5  type OP int
6
7  const (
8          ADD   OP = 0
9          INC   OP = 1
10         STORE OP = 2
11         DUAL  OP = 3
12         SNGL  OP = 4
13 )
14
15 type Msg struct {
16         op  OP
17         p1  int
18         p2  int
19         ret chan<- int
20 }
21
22 type State struct {
23         stored int
24 }
25
26 func loop1(ch <-chan Msg, state State) {
27         switch msg := <-ch; msg.op {
28         case STORE:
29                 loop1(ch, State{msg.p1})
30         case INC:
31                 msg.ret <- msg.p1 + state.stored
32                 loop1(ch, state)
33         case DUAL:
34                 loop2(ch, state)
35         }
36 }
37
38 func loop2(ch <-chan Msg, state State) {
39         switch msg := <-ch; msg.op {
40         case ADD:
41                 msg.ret <- msg.p1 + msg.p2
42                 loop2(ch, state)
43         case SNGL:
44                 loop1(ch, state)
45         }
46 }
47
48 func main() {
49         input := make(chan Msg)
50         go loop1(input, State{0})
51         res := make(chan int)
52         input <- Msg{STORE, 2, 0, res}
53         input <- Msg{INC, 5, 0, res}
54         fmt.Println(<-res)
55 }
```

Listing 4: A skeleton for Exercise 3

```
package main

import "fmt"

func f(fut /* add type here*/, p1 int, p2 int) {
    /* add code here */
}

func main() {
    ch := make( /* add type here */ )
    go f(ch, 1, 2)

//should work for any number of reads, test with 2
    fmt.Println(<-ch)
    fmt.Println(<-ch)
}
```

Listing 5: A solution for Exercise 3

```
import "fmt"

/* typing is as for single-read future
func f(fut chan<- int, p1 int, p2 int) {
  var res int = p1 + p2
  for { fut <- res } //this does not terminate!
}

func main() {
    ch := make(chan int)
    go f(ch, 1, 2)
    fmt.Println(<-ch)
    fmt.Println(<-ch)
}
```

Exercise 3 We discussed how to encode single-read futures with channels in the lecture.

- Describe in words how to encode multi-read futures with channels, in particular how to type the channel and what effect your encoding has on termiation.
- Use your encoding in Go and write a small program that uses goroutine that takes two numbers, computes their sum and sends back the results using a multi-read future (according to your encoding).

You can use the skeleton in Listing 4.

Solution: As the number of reads is not known a priori, the process must end in a non-terminating loop that sends the same value over the channel. This, however, results in a goroutine that never terminates – a so-called goroutine leak, where a system does not fully terminate because some processes are still blocked.

Typing-wise, there is no difference to the single-read futures, and the same problems as with single-read futures can occur, e.g., sending different values, or some other process writing into it by accident. The solution is in Listing 5.