Esercitazione 3
Programmazione Concorrente nel linguaggio go

13 Novembre 2017

Concorrenza in go

creazione goroutine

- Sintassi :

```
go <invocazione funzione>
```

- Esempio

```
func IsReady(what string, minutes int64) {
      time.Sleep(minutes*60*1e9)// unità: nanosecondi
      fmt.Println(what, "is ready")
}

func main(){
  go IsReady("tea", 6)
  go IsReady("coffee", 2)
  fmt.Println("I'm waiting...")
  …
}
```

-> tre threads: main, Isready(“tea”..), Isready(“coffee”...))

Interazione: canali

"*Do not communicate by sharing memory. Instead, share memory by communicating.* "

L'interazione tra processi può essere espressa tramite comunicazione attraverso **canali**.

Il canale permette sia la comunicazione che la sincronizzazione tra goroutines.

I canali sono oggetti di prima classe:

```
var C chan int
C=<espressione>
Op(C)
```

Canale: caratteristiche

Proprietà del canale in go:

- **Simmetrico/asimmetrico**, permette la comunicazione:
  - 1-1,
  - 1-molti
  - molti-molti
  - molti-1
- Comunicazione **sincrona** e **asincrona**
- **bidirezionale**, **monodirezionale**
- Oggetto **tipato**

Definizione:

```
var ch chan <tipo>  // <tipo> dei messaggi
```

Canale: inizializzazione

- Una volta definito, ogni canale va inizializzato:

```
var C1, C2 chan bool
C1=make(chan bool) // canale non bufferizzato:
                   //send sincrone
C2=make(chan bool, 100) // canale bufferizzato:
                        //send asincrone
```

Oppure

```
C1:= make(chan bool)
C2:= make(chan bool, 100)
```

Il valore di un canale non inizializzato è la costante **`nil`**.

Canale: uso

L'operatore di comunicazione **<-** permette di esprimere sia send che receive:

Send:

<canale> **<-** <messaggio>

Esempio:

```
c := make(chan int) // c non bufferizzato
c<-1 //send sincrona del valore 1 in c (send sincrona!)
```

NB La freccia punta nella direzione del flusso dei messaggi.

Receive:

<variabile> = **<-** <canale>

Esempio:

```
v = <-c   // riceve un valore da c, da assegnare
             a v
<-c       // riceve un messaggio che viene
             scartato
i := <-c  // riceve un messaggio, il cui valore
             inizializza i
```

Semantica

A default (canali non bufferizzati), la comunicazione è sincrona. Quindi:

1) la **send blocca** il processo mittente in attesa che il destinatario esegua la receive

2) la **receive blocca** il processo destinatario in attesa che il mittente esegua la send

In questo caso la comunicazione è una forma di **sincronizzazione** tra goroutines concorrenti.

NB una receive da un canale non inizializzato (**`nil`**) è **bloccante**

Esempio

```
func partenza(ch chan<- int) {
      for i := 0; ; i++ { ch <- i } // invia
}
func arrivo(ch <-chan int) {
      for { fmt.Println(<-ch) } // ricevi e stampa
  }

...
ch1 := make(chan int)
go partenza(ch1)
go arrivo(ch1)
...
```

Sincronizzazione padre-figlio

Come imporre al padre l'attesa della terminazione di un figlio?

Uso un canale dedicato alla sincronizzazione:

```
...
var done=make(chan bool)
func figlio() {
...
done<-true
}

func main() {
go  figlio
<-done // attesa figlio
}
```

Funzioni & canali

Una funzione può restituire un canale:

```
func partenza() chan int {
      ch := make(chan int)
      go func() {
          for i := 0; ; i++ { ch <- i }
      }()
return ch }

stream := partenza()  // stream è un canale int
fmt.Println(<-stream)   // stampa il primo messaggio:0
```

Chiusura canale: close

Un canale può essere chiuso (dal sender) tramite close:

```
close(ch)
```

Il destinatario può verificare se il canale è chiuso nel modo seguente:

```
msg, ok := <-ch
```

se il canale è ancora aperto, ok è vero

altrimenti è falso (il sender lo ha chiuso).

Esempio:

```
for {
      v, ok := <-ch
      if ok {fmt.Println(v) }else {break}
}
```

range

La clausola

`range <canale>`

nel for ripete la receive dal canale specificato fino a che il canale non viene **chiuso**.

Esempio:

```
for v := range ch { fmt.Println(v) }
```

equivale a:

```
for {
      v, ok := <-ch
      if ok {fmt.Println(v) }else {break}
}
```

Send asincrone

Creazione di un canale bufferizzato di capacità 50:

```
c := make(chan int, 50)
go func() {
      time.Sleep(60*1e9)
      x := <-c
      fmt.Println(”ricevuto", x)
    }()
  fmt.Println("sending", 10)
  c <- 10 // non è sospensiva!
  fmt.Println(”inviato", 10)
```

Output:

```
sending 10      (subito)
inviato 10      (subito)
ricevuto 10     (dopo 60 secondi)
```

Comandi con guardia: select

Select è un'istruzione di controllo ***analoga*** al comando con guardia alternativo.

Sintassi:

```
select{
   case <guardia1>:
      <sequenza istruzioni1>
   case <guardia2>:
      <sequenza istruzioni2>
   ...
   case <guardiaN>:
      <sequenza istruzioniN>
 }
```

Selezione **non deterministica** di un ramo con guardia valida, altrimenti attesa.

STRUTTURA della GUARDIA:

Nella select le guardie sono **receive** (o send): il linguaggio go **non prevede la guardia logica**. ➔ guardie **valide** o **ritardate**.

Esempio:

```
ci, cs := make(chan int), make(chan string)
select {
  case v := <-ci:
       fmt.Printf("ricevuto %d da ci\n", v)
  case v := <-cs:
       fmt.Printf("ricevuto %s da cs\n", v)
}
```

Possibilità di un ramo **default**, sempre valido.

Esempio pool con prio (poolprio.go) versione con send asincrone

```
package main
import (
   "fmt"
   "time" )
const MAXPROC = 10
const MAXRES = 3
const MAXBUFF = 20
var richiesta = make(chan int, MAXBUFF)
var rilascio = make(chan int, MAXBUFF)
var risorsa [MAXPROC]chan int
var done = make(chan int) //sincro padre-figli
var termina = make(chan int)  //term.server
```

```
func client(i int) {
      richiesta <- i
      r := <-risorsa[i]

// uso della risorsa r:
fmt.Printf("\n [client %d] uso ris. %d\n", i, r)
time.Sleep(time.Second * 2)

      rilascio <- r

      done <- i //terminaz. e sincron. col padre
}
```

```
func server() {
var disponibili int = MAXRES
var res, p, i int
var libera [MAXRES]bool
var sospesi = 0
var bloccato [MAXPROC]bool
// inizializzazioni:
for i := 0; i < MAXRES; i++ {
  libera[i] = true
}
for i := 0; i < MAXPROC; i++ {
     bloccato[i] = false
}
// continua..
```

```
for {
select {
case res = <-rilascio:
  if sospesi == 0 {
        disponibili++
        libera[res] = true
  } else { // prio ai processi con indice maggiore:
        for i = MAXPROC - 1; i >= 0 && !bloccato[i]; i-- { }
        bloccato[i] = false
        sospesi--
        risorsa[i] <- res
case p = <-richiesta:
  if disponibili > 0 {
        for i = 0; i < MAXRES && !libera[i]; i++ {}
        libera[i] = false
        disponibili--
        risorsa[p] <- i
  } else {
        sospesi++
        bloccato[p] = true     }
case <-termina: // quando tutti i clienti hanno finito
  done <- 1
  return                }       }}//fine server
```

```
func main() {
var cli int
fmt.Printf("\n quanti clienti (max %d)? ", MAXPROC)
fmt.Scanf("%d", &cli)
fmt.Println("clienti:", cli)

//inizializzazione canali clienti
for i := 0; i < cli; i++ {
        risorsa[i] = make(chan int, MAXBUFF)
}
for i := 0; i < cli; i++ {
        go client(i)
}
go server()

for i := 0; i < cli; i++ { <-done }
termina <- 1  // terminazione server
<-done //attesa terminazione server
}
```

Guardia logica

Nella select le guardie sono **receive** (o send): il linguaggio go **non prevede la guardia logica**.

Possiamo costruire le guardie logiche tramite una funzione che restituisce un canale:

```
func when(b bool, c chan int) chan int {
   if !b {
      return nil
   }
   return c
}
```

Quindi, **when(condizione, ch)** ritorna:

- il canale **ch** se la condizione è vera
- **nil** se la condizione è falsa*

* *ricordiamo che una receive da un canale nil è sospensiva*

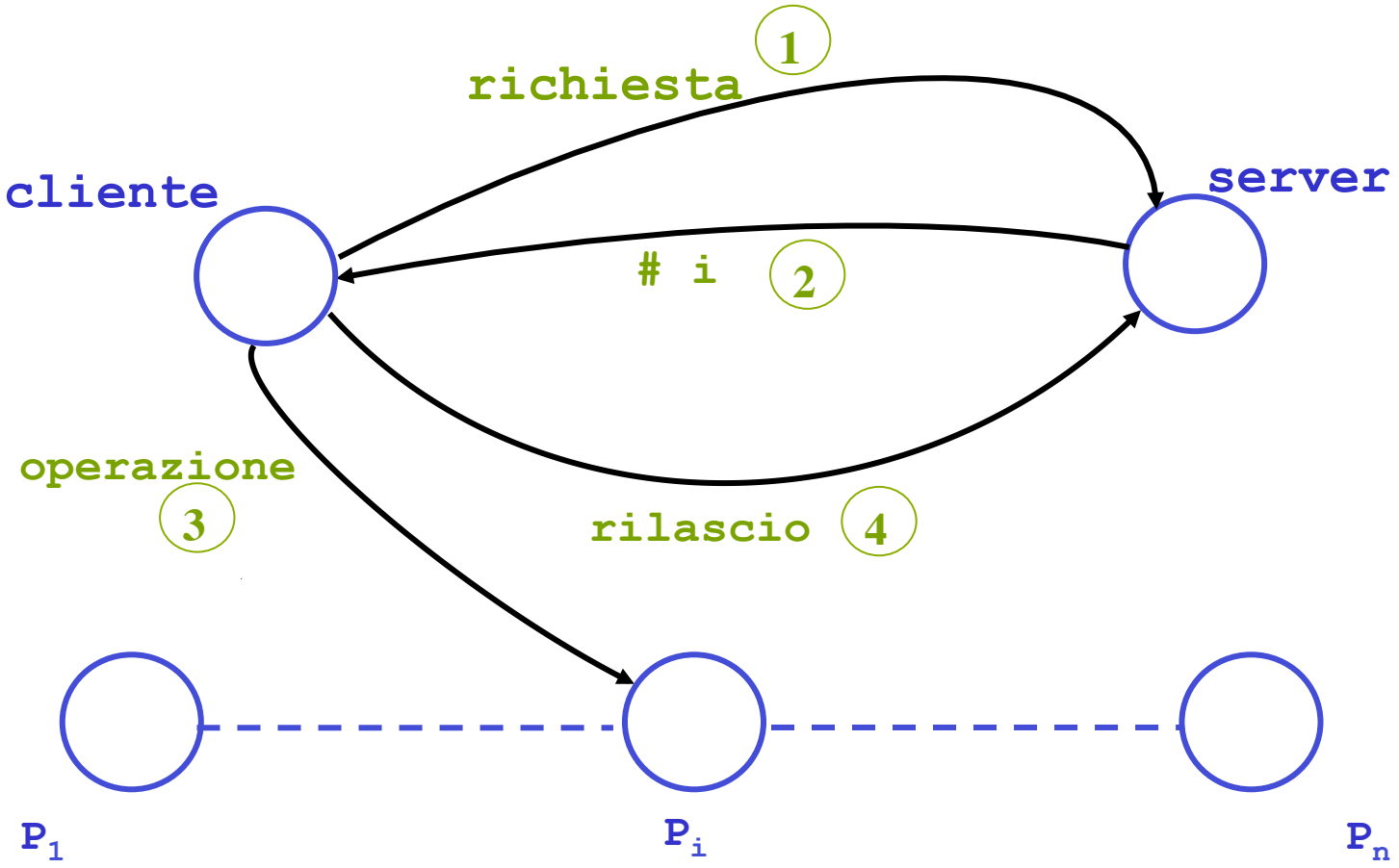
Uso di when

Esempio produttori/consumatori (v. procons_sync.go):

```
var pronto_prod = make(chan int)
var pronto_cons = make(chan int)
var dati = make(chan int)
var DATI_CONS [MAXPROC]chan int
var contatore int = 0
...
select {
case x:= <-when(contatore < N, pronto_prod):
       contatore++
       msg = <-dati  // ricezione messaggio da inserire
       <inserimento msg nel buffer >
case x := <-when(contatore > 0, pronto_cons):
       contatore--
       <estrazione msg dal buffer >
       DATI_CONS[x] <- msg //consegna messaggio a consumatore
..
}
```

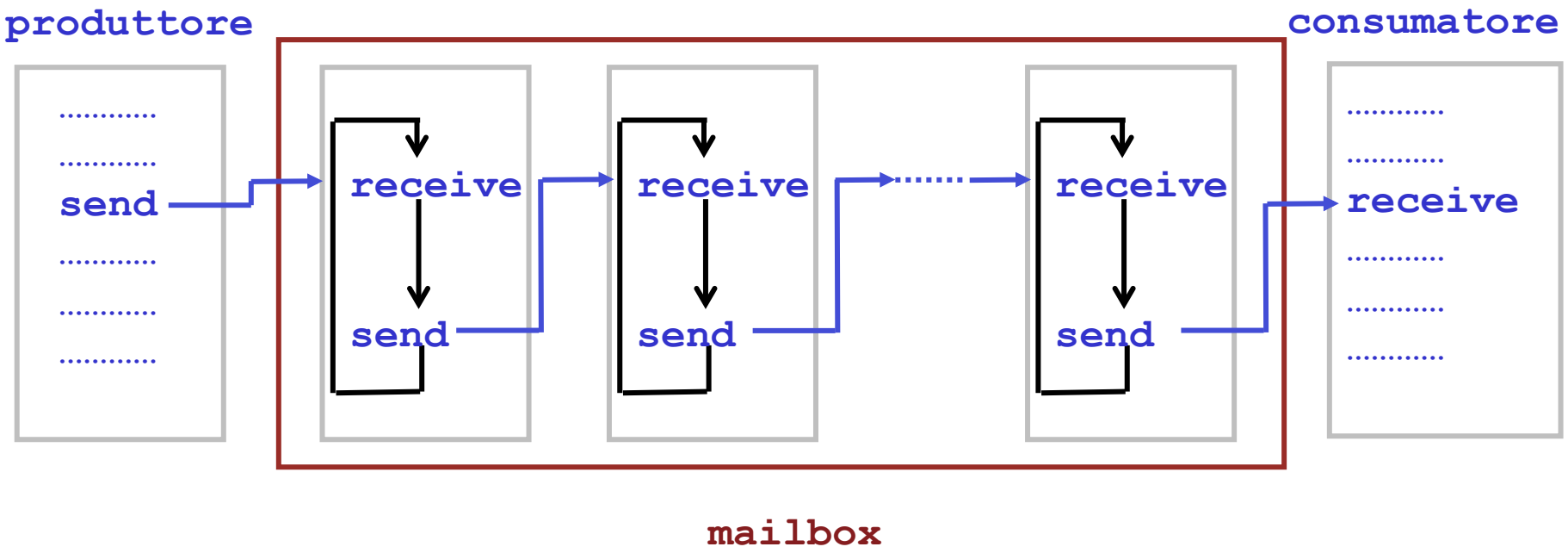
Esercizio 1 – uso delle guardie

A partire dalla soluzione della gestione di un pool di risorse equivalenti con priorità (poolprio.go), realizzare la gestione di un pool di risorse equivalenti senza priorità.



Esercizio 2 – send sincrone in go

Implementare una mailbox concorrente in go usando send sincrone (cioè canali non bufferizzati).



La soluzione dovrà permettere a più produttori e più consumatori l'uso della mailbox.

Impostazione mailbox concorrente

- La mailbox è caratterizzata da una capacità DIM.

- **Quali e quante goroutines ?**
  - La mailbox viene realizzata da tanti thread quanti sono gli elementi (stage) che la compongono: dovremo quindi creare DIM goroutine per la gestione della mailbox, ognuna delle quali esegue la funzione **server**:

```
func server(stage int, ch_in chan int, ch_out chan int) {
   …
}
```

  - Dovremo inoltre definire e creare un numero arbitrario di **produttori** e un numero arbitrario di **consumatori**

Impostazione

- Quali e quanti canali?
  - Un canale di ingresso alla pipeline di server:

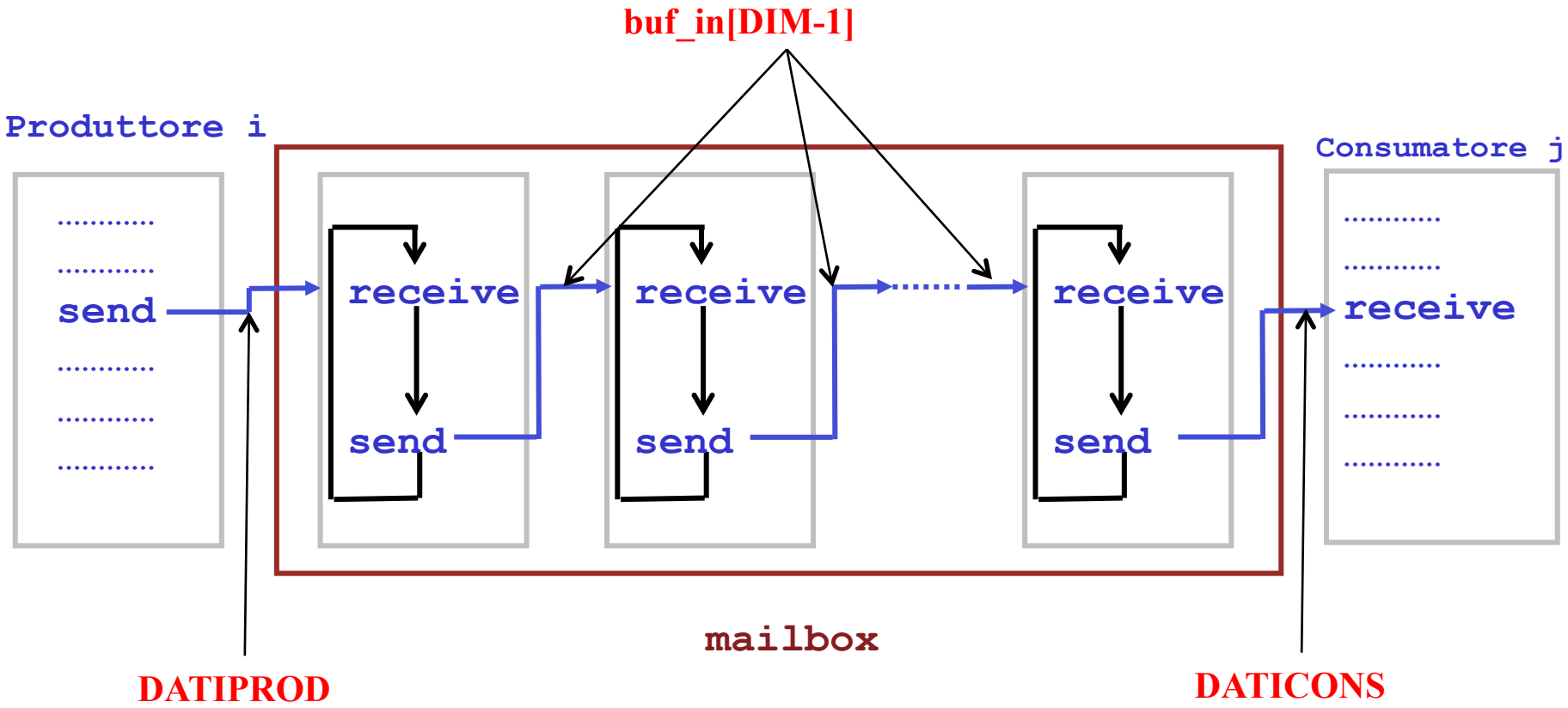
```
var DATIPROD chan int
```

  - Un canale di uscita della pipeline (I canali non sono porte! Su un canale possono essere eseguite receive da più processi)

```
var DATICONS chan int
```

  - I canali di comunicazione tra gli stage della mailbox:

```
var buf_in[DIM-1]
```


buf_in[DIM-1]
Produttore i
Consumatore j
send
receive
send
receive
send
receive
send
receive
mailbox
DATIPROD
DATICONS