Plan

Go language

Principes de conception

- Syntaxe légère inspirée du C
- Typage statique fort avec inférence
- Interfaces avec extension et polymorphisme (typage structurel / duck typing à la Smalltalk)
- Ramasse-miettes

Concepts pour la concurrence

- Descendant de CSP (Hoare 1978), cousin d'Erlang
- Goroutine $\sim$ activité/thread
  - une fonction s'exécutant indépendant (avec sa pile)
  - très léger (plusieurs milliers sans problème)
  - gérée par le noyau Go qui alloue les ressources processeurs
- Canaux pour la communication et la synchronisation

Go – canaux

Canaux

- Création : `make(chan type)` ou `make(chan type, 10)` (synchrone / asynchrone avec capacité)
- Envoi d'une valeur sur le canal `chan` : `chan <- valeur`
- Réception d'une valeur depuis `chan` : `<- chan`
- Canal transmissible en paramètre ou dans un canal : `chan chan int` est un canal qui transporte des canaux (transportant des entiers)

Go – canaux

Alternative en réception et émission

```
select {
 case v1 := <- chan1:
        fmt.Printf("received %v from chan1\n", v1)
 case v2 := <- chan2:
        fmt.Printf("received %v from chan2\n", v2)
 case chan3 <- 42:
        fmt.Printf("sent %v to chan3\n", 42)
 default:
        fmt.Printf("no one ready to communicate\n")
}
```

Exemple élémentaire

```
func boring(msg string, c chan string) {
    for i := 0; ; i++ {
        c <- fmt.Sprintf("%s %d", msg, i)
        time.Sleep(time.Duration(rand.Intn(4)) * time.Second)
    }
}
```

```
func main() {
    c := make(chan string)
    go boring("boring!", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("You say: %q\n", <- c)
    }
    fmt.Println("You're boring; I'm leaving.")
}
```

Moteur de recherche

Objectif : agrégation de la recherche dans plusieurs bases

func Web(query **string**) Result
func Image(query **string**) Result
func Video(query **string**) Result

Moteur séquentiel

```
func Google(query string) ( results []Result) {
    results = append(results, Web(query))
    results = append(results, Image(query))
    results = append(results, Video(query))
    return
}
```

exemple tiré de `https://talks.golang.org/2012/concurrency.slide`

Recherche concurrente

Moteur concurrent

```
func Google(query string) ( results  []Result ) {
    c := make(chan Result)
    go func() { c <- Web(query) } ()
    go func() { c <- Image(query) } ()
    go func() { c <- Video(query) } ()

    for i := 0; i < 3; i++ {
        result := <- c
        results = append(results, result )
    }
    return
}
```

Le temps sans interruption

Crée un canal sur lequel un message sera envoyé après la durée spécifiée.

time.After

```
func After(d time.Duration) <-chan bool {
    // Returns a receive-only channel
    // A message will be sent on it after the duration
    c := make(chan bool)
    go func() {
        time.Sleep(d)
        c <- true
    }()
  return c
}
```

Recherche concurrente en temps borné

Moteur concurrent avec timeout

```
c := make(chan Result)
go func() { c <- Web(query) } ()
go func() { c <- Image(query) } ()
go func() { c <- Video(query) } ()

timeout := time.After(80 * time.Millisecond)
for i := 0; i < 3; i++ {
    select {
        case result := <-c:
            results = append(results, result)
        case <-timeout:
            fmt.Println("timed out")
            return
    }
}
return
```

Recherche répliquée

Utiliser plusieurs serveurs répliqués et garder la réponse du premier qui répond.

Recherche en parallèle

```
func First(query string, replicas ...Search) Result {
    c := make(chan Result)
    searchReplica := func(i int) { c <- replicas[i](query) }
    for i := range replicas {
        go searchReplica(i)
    }
    return <-c
}
```

Recherche répliquée

Moteur concurrent répliqué avec timeout

```
c := make(chan Result)
go func() { c <- First(query, Web1, Web2, Web3) } ()
go func() { c <- First(query, Image1, Image2) } ()
go func() { c <- First(query, Video1, Video2) } ()
timeout := time.After(80 * time.Millisecond)
for i := 0; i < 3; i++ {
    select {
        case result := <-c:
            results = append(results, result)
        case <-timeout:
            fmt.Println("timed out")
            return
    }
}
return
```

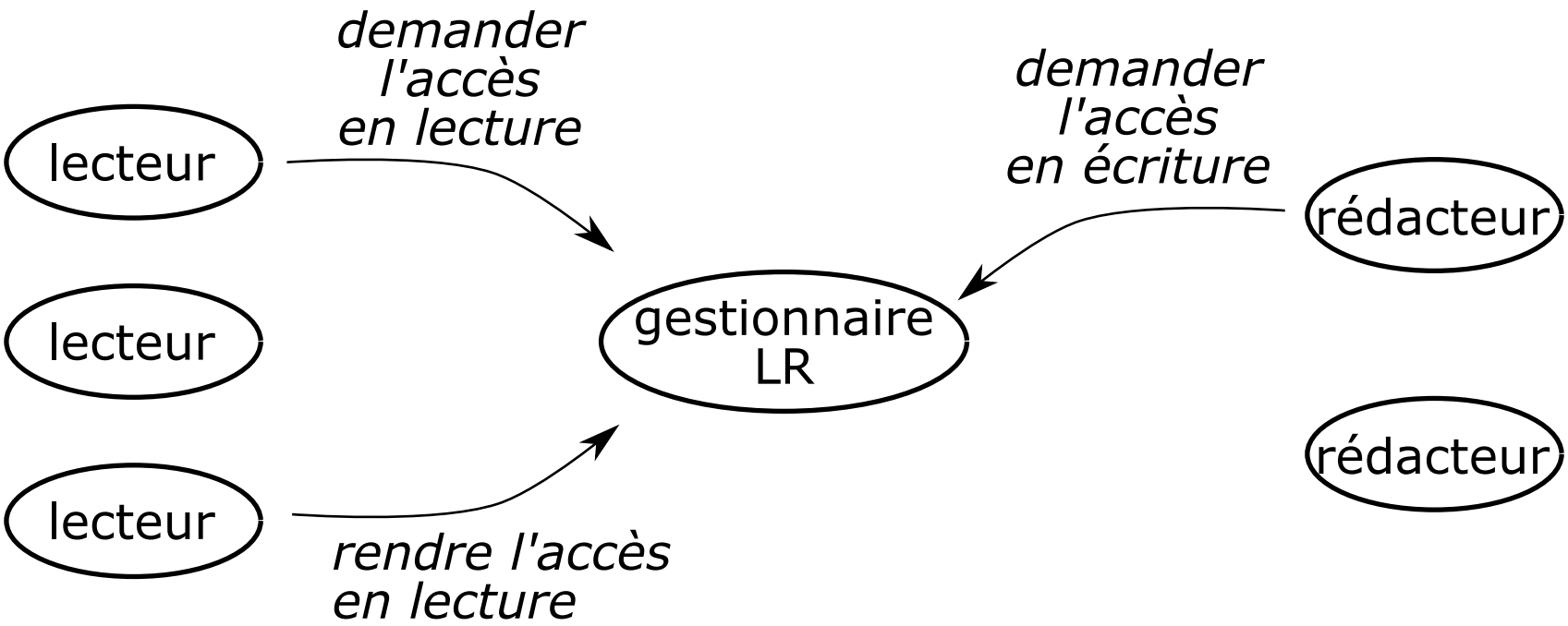
Bilan

- Création ultra-légère de goroutine : penser concurrent
- Pas besoin de variables partagées
  $\Rightarrow$ Pas de verrous
- Pas besoin de variable condition / sémaphore pour synchroniser
- Pas besoin de callback ou d'interruption

> Don't communicate by sharing memory, share memory by communicating.

(la bibliothèque Go contient *aussi* les objets usuels de synchronisation pour travailler en mémoire partagée : verrous, sémaphores, moniteur...)

Lecteurs/rédacteurs



- Un canal pour chaque type de requête : DL, TL, DE, TE
- Émission bloquante ⇒ accepter un message (une requête) uniquement si l'état l'autorise

Lecteurs/rédacteurs

Utilisateur

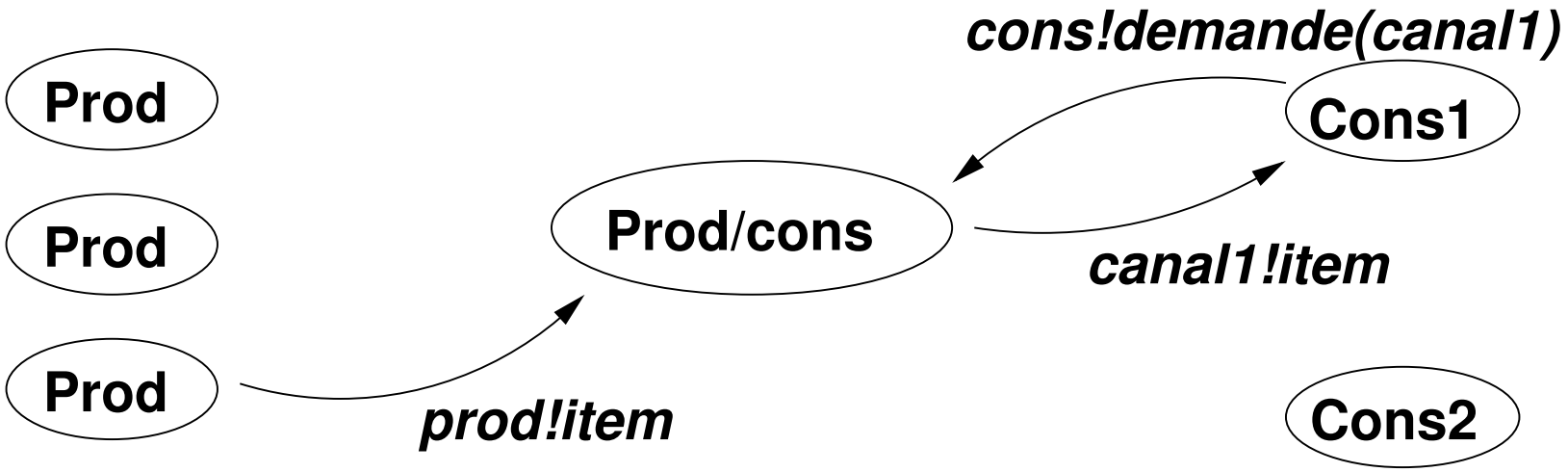
```
func Utilisateur () {
  nothing := struct{}{}
  for {
    DL <- nothing; // demander lecture
     ...
    TL <- nothing; // terminer  lecture
     ...
    DE <- nothing; // demander écriture
     ...
    TE <- nothing; // terminer  é criture
  }
}
```

Goroutine de synchronisation

```
func when(b bool, c chan struct{}) chan struct{} {
    if b { return c } else { return nil }
}

func SynchroLR() {
  nblec := 0;
  ecr := false;
  for {
    select {
      case <- when(nblec == 0 && !ecr, DE):
            ecr := true;
      case <- when(!ecr, DL):
            nblec++;
      case <- TE:
            ecr := false;
      case <- TL:
            nblec--;
    }
}}
```

Producteurs/consommateurs : architecture ♪♪♪



- Un canal pour les demandes de dépôt
- Un canal pour les demandes de retrait
- Un canal par activité demandant le retrait (pour la réponse à celle-ci)

(exercice futile : `make(chan T, N)` est déjà un tampon borné = un prod/cons de taille N)

Producteurs/consommateurs

Programme principal

```
func main() {
    prod := make(chan int)      // un canal portant des entiers
    cons := make(chan chan int) // un canal portant des canaux
    go prodcons(prod, cons)
    for i := 1; i < 10; i++ {
        go producteur(prod)
    }
    for i := 1; i < 5; i++ {
        go consommateur(cons)
    }
    time.Sleep (20*time.Second)
    fmt.Println ("DONE.")
}
```

Producteurs/consommateurs

Producteur

```
func producteur(prod chan int) {
  for {
    ...
    item := ...
    prod <- item
  }}
```

Consommateur

```
func consommateur(cons chan chan int) {
  moi := make(chan int)
  for {
    ...
    cons <- moi
    item := <- moi
    // utiliser item
  }}
```

Producteurs/consommateurs

Goroutine de synchronisation

```
func prodcons(prod chan int, cons chan chan int) {
  nbocc := 0;
  queue := make([]int, 0)
  for {
    if nbocc == 0 {
      m := <- prod; nbocc++; queue = append(queue, m)
    } else if nbocc == N {
      c := <- cons; c <- queue[0]; nbocc--; queue = queue[1:]
    } else {
      select {
        case m := <- prod: nbocc++; queue = append(queue, m)
        case c := <- cons:
              c <- queue[0]; nbocc--; queue = queue[1:]
      }
    }
  }
}
```