

Programmation concurrente en Go

Programmation concurrente

- ▶ Un programme concurrent est un programme contenant plusieurs processus ou plusieurs fils d'exécution
 - Un processus est généralement défini comme une entité s'exécutant de façon indépendante et contrôlant sa propre mémoire
 - Un fil (*thread* ou *light-weight process*) est une instance s'exécutant dans un processus. Les différents fils d'un processus généralement partage la mémoire allouée par le processus.

Fil d'exécution

- ▶ Un fil d'exécution (thread) est une séquence d'exécution pouvant ou non interagir avec d'autres fils
- ▶ Les fils partagent souvent les même variables
- ▶ Les fils ont souvent (mais pas nécessairement) une durée de vie limitée
- ▶ Un fil peut être bloqué:
 - Si il désire utiliser des variables partagées avec d'autres fils
 - Si il doit attendre des résultats d'un autre fil
- ▶ Une application se subdivise en processus et un processus peut être composé de fils
- ▶ Les processus sont généralement créés lors de la conception de l'architecture alors que les fils sont créés lors de la phase de programmation

Programmation parallèle vs programmation concorrente

- ▶ Si les processus ou les fils s'exécutent sur des processeurs différents on parle de programmation parallèle
- ▶ La programmation concorrente peut aussi se faire sur un seul processeur
 - Les processus se partagent alors le temps d'execution
- ▶ Si le programme s'execute sur plusieurs machine, on parle de programmation distribuée

Langage de programmation concurrente

- ▶ Un langage de programmation concurrente doit permettre:
 - la création de processus et de fils d'exécution
 - la synchronisation de leurs opérations:
 - *Synchronisation coopérative*: lorsqu' un processus attend la fin de l'exécution d' un autre avant de poursuivre son exécution.
 - *Synchronisation compétitive*: lorsque plusieurs processus utilise la même ressource. Il faut alors disposer d' un mécanisme d'*exclusion mutuelle* afin d'éviter que les processus interfèrent entre eux
 - la communication des données entre processus: en utilisant des mécanismes de communication inter-processus définis par le système d'exploitation

Fils en Java

```
public class HelloRunnable implements Runnable {  
  
    public void run() {  
        System.out.println("Je marche sur un fil!");  
    }  
  
    public static void main(String args[]) {  
        (new Thread(new HelloRunnable())).start();  
    }  
  
}  
  
public class HelloThread extends Thread {  
  
    public void run() {  
        System.out.println("Je marche sur un fil!");  
    }  
  
    public static void main(String args[]) {  
        (new HelloThread()).start();  
    }  
  
}
```

Niveaux de concurrence

- ▶ *au niveau des énoncés:*
 - une série d' énoncés sont exécutés de façon concurrente, le processus principal suspend alors son exécution. Chacun des processus ainsi créés partagent le même ensemble de données (OpenMP)
- ▶ *au niveau des sous-programmes:*
 - un processus commande alors la création d'un autre processus dont la tâche consiste à exécuter un certain sous-programme. Une fois le processus lancé, le processus appelant continue son exécution. Un mécanisme de synchronisation doit toutefois être disponible.
- ▶ *au niveau des objets:*
 - chacune des instances d' une classe devient une entité concurrente; il y a alors exécution concurrente d' une de ses méthodes.
- ▶ *au niveau des programmes:*
 - ceci se produit lorsqu' un processus parent a la capacité de lancer un ou plusieurs processus enfant. Il doit toutefois exister un moyen de connaître l' identité d' un processus. Les données peuvent être partagées ou non.

Type de concurrence

- ▶ Physique:
 - lorsque plusieurs processeurs se partagent les différents processus
- ▶ Logique:
 - lorsque plusieurs processus se partagent le temps d'exécution sur un seul processeur.
- ▶ Distribué:
 - lorsque plusieurs machines constituées en réseau se partagent les processus.

Principe de concurrence en Go

- ▶ Le grand problème en concurrence est le partage des données
- ▶ Ne pas communiquer en partageant des données
- ▶ Il faut plutôt partager des données en communiquant!
 - La communication est la clé d'une bonne synchronisation
- ▶ Paradigme CSP
 - Communicating Sequential Processes
 - Échange de messages

Les GoRoutines

- ▶ La goroutine est une entité qui tourne en concurrence
- ▶ Une goroutine peut correspondre à un ou plusieurs fils
- ▶ Les goroutines partagent le même espace mémoire
- ▶ Les goroutines ont été concues de façon à être très légères
 - Création peu coûteuse
- ▶ Une goroutine est une fonction ou une méthode
- ▶ Une goroutine est invoquée en utilisant le mot clé `go`

Appeler des goroutines

```
package main
import (
    "fmt"
    "runtime"
    "time"
)

func main() {
    runtime.GOMAXPROCS(3) // nombre max de fils

    debut := time.Now(); // chrono
    fmt.Println("Debut")

    // lancement de 2 goroutines
    go lettres()
    go nombres()

    fmt.Println("En attente")
    time.Sleep(2*time.Second) // afin d'attendre les goroutines
    fmt.Println("\nFin\n")

    fin := time.Now();
    fmt.Printf("Temps d'exécution: %s", fin.Sub(debut))
}
```

Les goroutines sont des fonctions

```
func nombres() {  
  
    for number := 1; number < 27; number++ {  
        // pause afin de ralentir la fonction  
        time.Sleep(10*time.Millisecond)  
        fmt.Printf("%d ", number)  
    }  
}  
  
func lettres() {  
  
    for char := 'a'; char < 'a'+26; char++ {  
        time.Sleep(10)  
        fmt.Printf("%c ", char)  
    }  
}
```

Temps d'execution concurrent

Avec les goroutines

Debut

En attente

```
1 a b 2 c 3 4 d e 5 f 6 7 g 8 h i 9 j 10 k 11 l 12 m 13 n 14  
o 15 p 16 17 q r 18 s 19 t 20 21 u v 22 23 w x 24 y 25 z 26
```

Fin

Temps d'execution: 2.0000278s

Sans les goroutines

Debut

```
a b c d e f g h i j k l m n o p q r s t u v w x y z  
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
```

En attente

Fin

Temps d'execution: 2.5369612s

Communication entre goroutines

- ▶ Les goroutines sont vraiment effectives lorsqu'elles peuvent communiquer entre elles
 - en s'échangeant de l'information
 - en se synchronisant
- ▶ En Go ceci se réalise via l'utilisation de canaux de communications
 - les *channels*

Le concept de *channel*

- ▶ Un *channel* agit comme un conduit pour un type de données
- ▶ Avec un *channel*, une seule goroutine a accès à une donnée à tout moment
- ▶ Le *channel* est une file de données (FIFO)
- ▶ La lecture et l'écriture dans un *channel* sont des énoncés bloquants
 - L'exécution ne sera pas bloquée si la capacité du *channel* n'est pas excédée

Déclarer un *channel*

- ▶ Déclarer un *channel* ne fait que créer une référence
 - Il faut utiliser *make* afin d'allouer l'espace pour le channel

```
var monCanal chan string // un channel de strings
monCanal = make(chan string)
```

- ▶ On peut aussi fermer un channel

```
close(monCanal)
x,ok := <- monCanal // ok est faux si le channel
                     // a été fermé
```

- ▶ Par défaut un channel est sans mémoire
 - Une écriture dans un tel channel sera bloquante
 - On déclare une mémoire tampon comme suit:

```
monCanal = make(chan string,2) // permet l'insertion de
                               // 2 instances avant bloquage
```

- ▶ Pour envoyer ou recevoir, il faut utiliser l'opérateur flèche

```
monCanal <- uneChaine
uneAutreChaine= <- monCanal
```

Communication entre goroutines

```
package main          func main() {  
  
import (  
    "fmt"  
    "time"  
)  
    ch := make(chan string)  
    go envoyer(ch)  
    go recevoir(ch)  
    time.Sleep(1*time.Second)  
}  
  
func envoyer(ch chan string) { // production de donnees  
  
    ch <- "Ottawa"  
    ch <- "Toronto"  
    ch <- "Gatineau"  
    ch <- "Casselman"  
}  
  
func recevoir(ch chan string) { // traitement des donnees  
  
    var ville string  
    for {  
        ville= <-ch  
        fmt.Printf("%s ", ville)  
    }  
}  
  
// Résultat: Ottawa Toronto Gatineau Casselman
```

Synchronisation

```
func worker(done chan bool) {
    fmt.Println("traitement en cours...")
    time.Sleep(time.Second)
    fmt.Println("fini")
    done <- true // signal de fin
}

func main() {

    // canal de synchronisation
    done := make(chan bool, 1)
    // lancement de la goroutine
    go worker(done)

    //ici on continue le main

    // point de synchronisation (rendez-vous)
    <-done
}
```

Boucle parallèle

```
package main

import "fmt"
import "time"

func main() {

    x := []int{3, 1, 4, 1, 5, 9, 2, 6}
    var y [8]int

    // boucle parallèle
    for i, v := range x {

        go func (i int, v int) {
            y[i] = calcul(v)
        }(i,v) // appel à la goroutine
    }
    // ajouter synchronisation
    time.Sleep(1*time.Second)
}

func calcul(v int) (int) {

    return 2*v*v*v+v*v
}
```

Boucle parallèle (slices)

```
package main

import "fmt"
import "time"

func main() {

    x := []int{3, 1, 4, 1, 5, 9, 2, 6}
    var y [8]int

    // boucle parallelisée en 2 slices
    go calcul2(x[:4],y[:4])
    go calcul2(x[4:],y[4:])

    time.Sleep(time.Second)
}

func calcul2(in []int, out []int) {

    for i, v:= range in {
        out[i]= 2*v*v*v+v*v
    }
}
```

Sémaphore

- ▶ Une séaphore est un mécanisme qui permet la synchronisation et le partage de ressource entre processus
- ▶ Il n'y a pas de semaphore en Go
 - Elles sont concues à l'aide de channels
 - La capacité du channel correspond au nombre de ressources à synchroniser
 - Le nombre d'éléments dans le channel correspond aux nombres de ressources couramment utilisées

```
semaphore := make(chan bool, N)
```

Sémaphore en Go

```
package main

import "fmt"
import "time"

type semaphore chan bool

// afin d'acquerir n resources
func (s semaphore) acquiere(n int) {

    for i:=0; i<n; i++ {
        s <- true
    }
}

// afin de liberer n resources
func (s semaphore) libere(n int) {

    for i:=0; i<n; i++ {
        <- s
    }
}
```

Mutex

- ▶ Un mutex (*mutual exclusion*) permet de verrouiller une ressource pendant qu'un fil y accède
- ▶ Un mutex est donc associé à une ressource
- ▶ Un mutex verrou est implementés en utilisant des sémaphore binaires
- ▶ Un mutex à n resource utilise le patron signal et attente (*signal and wait*)

Mutex en Go

```
package main

import "fmt"
import "time"

// MUTEX
// verouiller
func (s semaphore) Lock() {
    s.acquiere(1)
}

// deverouiller
func (s semaphore) Unlock() {
    s.libere(1)
}

// attendre
func (s semaphore) Wait(n int) {
    s.libere(n)
}

// signaler
func (s semaphore) Signal() {
    s.acquiere(1)
}
```

Mutex et verrou

```
func main() {  
  
    x := []int{3, 1, 4, 1, 5, 9, 2, 6}  
    var y [8]int  
    // creation d'un verrou  
    verrou:= make(semaphore,1)  
  
    go modif(x,2,verrou)  
    go modif(x,5,verrou)  
  
    time.Sleep(time.Second)  
}  
  
// fonction modifiant les valeurs d'un tableau  
func modif(tableau []int, inc int, sem semaphore) {  
  
    // verouille l'accès au tableau  
    sem.Lock()  
    for i, v:= range tableau {  
        tableau[i]= v+inc  
    }  
    // libere l'accès au tableau  
    sem.Unlock()  
}
```

Mutex en attente

```
func main() {

    x := []int{3, 1, 4, 1, 5, 9, 2, 6}
    var y [8]int
    // creation d'un mutex de capacite 2
    mutex := make(semaphore, 2)

    go calcul3(x[:4],y[:4],mutex)
    go calcul3(x[4:],y[4:],mutex)

    // attendre 2 signaux
    mutex.Wait(2)
}

func calcul3(in []int, out []int, sem semaphore) {

    for i, v:= range in {
        out[i] = 2*v*v*v+v*v
    }

    // lance un signal
    sem.Signal()
}
```

Moniteur

- ▶ Un moniteur est une abstraction qui contient les données partagées ainsi que les procédures qui accèdent à ces données
 - La synchronisation est alors réalisée implicitement en n'autorisant qu'un accès à la fois au moniteur
 - Lorsque le moniteur est occupé, tous les processus en attente pour accès sont placés dans une file
- ▶ En réalisant un appel à une opération du moniteur, un processus obtient des droits exclusifs d'accès aux ressources du moniteur.
 - Cette exclusivité demeure jusqu'à ce que l'opération soit complétée ou jusqu'à ce que ce processus se place en attente

Moniteur en Go

```
type Employe struct {
    moniteur semaphore
    nom      string
    numero   int
    salaire  float32
}

func (e *Employe) augmentation(pourcentage float32) {
    e.moniteur.Lock()
    fmt.Println("Calcul du nouveau salaire...")
    time.Sleep(time.Second)
    e.salaire *= 1.0 + pourcentage
    fmt.Println("Fin du calcul salaire...")
    e.moniteur.Unlock()
}

func (e Employe) bonus(pourcentage float32) float32 {
    e.moniteur.Lock()
    fmt.Println("Calcul du bonus...")
    time.Sleep(time.Second)
    bonus := e.salaire * pourcentage
    fmt.Println("Fin du calcul bonus...")
    e.moniteur.Unlock()
    return bonus
}

func (e Employe) String() string {
    return e.nom + fmt.Sprintf(": $%6.2f", e.salaire)
}
```

Moniteur en Go

```
func main() {  
  
    jean := Employe{make(semaphore,1), "Jean", 12345, 22000}  
    pierre := Employe{make(semaphore,1), "Pierre", 57683, 67000}  
  
    go jean.augmentation(0.2)  
    go pierre.augmentation(0.1)  
    go jean.bonus(0.01)  
  
    time.Sleep(3*time.Second)  
  
    fmt.Println(jean)  
    fmt.Println(pierre)  
}
```

Calcul du nouveau salaire...
Calcul du nouveau salaire...
Fin du calcul salaire...
Fin du calcul salaire...
Calcul du bonus...
Fin du calcul bonus...
Jean: \$26400.00
Pierre: \$73700.00

Java et moniteurs

- ▶ En Java, tous les objets ont un moniteur intrinsèque associé
 - Méthodes synchronisées
 - Blocs synchronisés
 - avec l'énoncé synchronized(object)
- ▶ La file d'attente est gérée par la machine virtuelle Java
- ▶ Jusqu'à 50 fois plus lente que les méthodes non-synchronisées

Le producteur et le consommateur

▶ La production des données

```
func main() {
    jobs := make(chan int, 5)
    mutex := make(semaphore, 1)

    go consomme(jobs,mutex) // démarre le consommateur

    // producteur
    for j := 1; j <= 10; j++ {
        jobs <- j
        time.Sleep(time.Second) // je créé
        fmt.Println("Travail envoyé", j)
    }
    close(jobs)
    fmt.Println("Plus de travail à produire")

    mutex.Wait(1)
}
```

Le producteur et le consommateur

▶ La consommation des données

```
func consomme(jobs chan int, done semaphore) {
    for {
        j, more := <-jobs

        if more {
            fmt.Println("je consomme: ", j*j)
            time.Sleep(4*time.Second) // je perd du temps

        } else {
            fmt.Println("C'est fini!")
            done.Signal()
            return
        }
    }
}
```

Le producteur et le consommateur

```
je consomme: 1
Travail envoyé 1
Travail envoyé 2
Travail envoyé 3
je consomme: 4
Travail envoyé 4
Travail envoyé 5
Travail envoyé 6
Travail envoyé 7
je consomme: 9
Travail envoyé 8
je consomme: 16
Travail envoyé 9
je consomme: 25
Travail envoyé 10
Plus de travail à produire
je consomme: 36
je consomme: 49
je consomme: 64
je consomme: 81
je consomme: 100
C'est fini!
```

Quicksort en parallèle

```
func qsort_pass(arr []int, done chan int) []int{
    if len(arr) < 2 {
        done <- len(arr)
        return arr
    }
    pivot := arr[0]
    i, j := 1, len(arr)-1
    for i != j {
        for arr[i] < pivot && i!=j{
            i++
        }
        for arr[j] >= pivot && i!=j{
            j--
        }
        if arr[i] > arr[j] {
            arr[i], arr[j] = arr[j], arr[i]
        }
    }
    if arr[j] >= pivot {
        j--
    }
    arr[0], arr[j] = arr[j], arr[0]
    done <- 1;

    go qsort_pass(arr[:j], done)
    go qsort_pass(arr[j+1:], done)
    return arr
}

func qsort(arr []int) []int {
    done := make(chan int)
    defer func() {
        close(done)
    }()
    go qsort_pass(arr[:], done)

    rslt := len(arr)
    for rslt > 0 {
        rslt -= <-done;
    }
    return arr
}
```

Un serveur de requêtes

▶ Les types de données

```
package main

import (
    "fmt"
    "strings"
    "time"
)

type Reply struct {
    nombre int
}

type Request struct {
    Mot      string
    Lettre   string
    Reponse chan *Reply
}
```

Le serveur

```
// fonction service pour chaque requête
func lettreCompteur(r *Request) {

    // effectuer la tâche demandée
    n := strings.Count(r.Mot, r.Lettre)

    // construire la réponse
    rep := &Reply{n}
    // insertion de la réponse
    // possiblement plusieurs
    r.Reponse <- rep
}

// serveur de requêtes
func server(service chan *Request) {

    for {
        // arrivée d'une requête
        req := <-service
        // lancement d'une routine de service
        go lettreCompteur(req)
    }
}

// Démarrage du server
func startServer() chan *Request {

    r := make(chan *Request)
    go server(r)
    return r
}
```

Le côté client

```
// simulateur de client
func client(nom string, lettre string, vitesse time.Duration, accept
chan *Request) {

    bidon := "test"
    var req *Request
    var rep *Reply

    for i := 0; i < 10; i++ {
        // Création d'une requête
        req = &Request{bidon, lettre, make(chan *Reply)}
        time.Sleep(vitesse*time.Millisecond)
        // Envoie de la requête
        accept <- req
        // Attente de la réponse
        rep = <-req.Reponse
        fmt.Printf("%s: %d\n", nom, rep.nombre)
        close(req.Reponse)
        // Modification de la chaîne (test)
        bidon += bidon
    }
}
```

Simulation client–serveur

```
func main() {  
  
    accept := startServer()  
  
    go client("User ABC", "t", 200, accept)  
    go client("User XYZ", "e", 300, accept)  
    go client("User WOW", "s", 100, accept)  
    go client("User NON", "x", 700, accept)  
  
    // devrait inclure un signal de terminaison...  
    time.Sleep(5 * time.Second)  
  
}  
  
User WOW: 1  
User ABC: 2  
User WOW: 2  
User XYZ: 1  
User WOW: 4  
User ABC: 4  
User WOW: 8  
User WOW: 16  
User XYZ: 2  
User WOW: 32  
User ABC: 8  
User NON: 0  
User WOW: 64  
User ABC: 16  
User WOW: 128  
User XYZ: 4  
User WOW: 256  
User ABC: 32  
User WOW: 512  
User XYZ: 8  
User ABC: 64  
User NON: 0  
User ABC: 128  
User XYZ: 16  
User ABC: 256  
...  
}
```

Patron de programmation concurrente

- ▶ Parallélisme de données
 - On se partage les données à traiter
- ▶ Parallélisme de contrôle
 - On se partage les tâches
- ▶ Parallélisme de flux
 - Chaine de montage

Exemple:

$$a + bx + cx^2 + dx^3$$

Calcul d'un polynome pour N variables

▶ Parallélisme de données

- Chacune des goroutines fait le calcul pour un sous-ensemble de données

▶ Parallélisme de flux

- Une goroutine calcule $r1 = (dx + c)$
- Une goroutine calcule $r2 = r1 * x + b$
- Une goroutine calcule $r2 * x + a$

▶ Parallélisme de contrôle

- Une goroutine calcule $a + bx$
- Une goroutine calcule cx^2
- Une goroutine calcule dx^3