LES GOROUTINES DANS LE LANGAGE DE PROGRAMMATION GO

Concurrence et parallélisme

Avant d'aborder des goroutines il est essentiel de comprendre la différence entre concurrence et parallélisme.

Concurrence

La concurrence est la capacité de **traiter plusieurs de choses à la fois**, par exemple :

Un humain normal doit d'abord finir sa bouchée avant de pouvoir parler, une fois qu'il aura fini sa bouchée il pourra parler ensuite une fois qu'il aura fini de parler il pourra encore une fois reprendre une autre bouchée et reparler juste après.

Dans cet exemple la personne est capable capable de gérer plusieurs de choses (manger et parler) dans un intervalle de temps différent.

Parallélisme

Le parallélisme permet de **traiter beaucoup de choses en même temps**, je m'explique :

L'humain normal de l'exemple précèdent devient un humain mutant avec un bras et une bouche en plus. Cette fois-ci il est capable de manger et de parler en même temps.

Le fonctionnement d'un point de vue informatique

Maintenant que vous avez compris ce qu'est la concurrence et comment elle diffère du parallélisme en utilisant des exemples de la vie réelle, il est temps maintenant de comprendre qu'est ce ça donne d'un point de vue technique.

Imaginons que vous codez votre propre navigateur Web et qu'un utilisateur utilise votre navigateur pour visiter une page web afin de télécharger une vidéo. Il clique alors sur le bouton de téléchargement de la page et en attendant la fin de son téléchargement il visite d'autres pages du site.

Votre navigateur doit dans ce cas gérer deux taches de manière indépendante, une tâche pour le téléchargement et une autre tache pour le rendu des pages web que l'utilisateur va visiter. Cette tâche indépendante est ce qu'on appelle un **thread**.

Lorsque votre navigateur est exécuté dans un processeur avec un seul cœur (ou processeur multicœur), le processeur bascule entre les deux tâches (chaque tâche va s'exécuter un petit moment puis une autre, puis une autre, puis on revient à la première, etc) ceci est connu sous le nom de la **concurrence**. Dans ce cas, le téléchargement et le rendu commencent à différents moments et leurs exécutions se chevauchent.

Disons que le même navigateur est exécuté sur une autre machine avec un processeur multicœur et que la tâche de téléchargement et la tâche de rendu HTML s'exécute simultanément sans chevauchement dans des cœurs différents alors ceci est plus connu sous le nom de **parallélisme**.

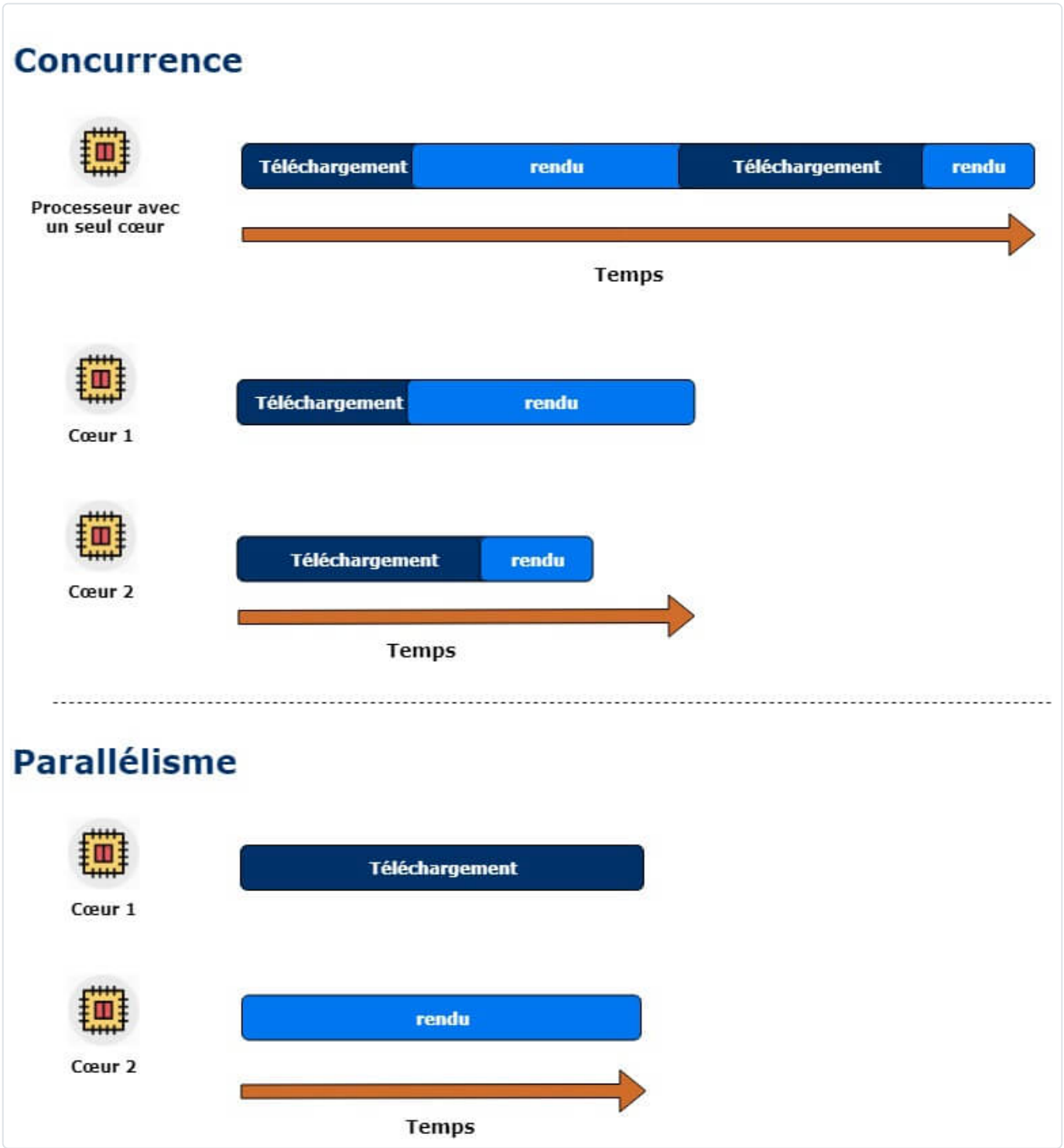

Schéma sur le fonctionnement de la concurrence et le parallélisme sur un Processeur avec un seul cœur et deux cœurs.

Vos tâches peuvent parfois avoir besoin de communiquer entre eux. Par exemple dans votre navigateur dès que l'utilisateur aura fini son téléchargement une popup indiquant à l'utilisateur que le téléchargement c'est bien déroulé apparaîtra au-dessus du rendu si et seulement si la page courante n'est pas en plein écran. Dans ce cas on parle de concurrence et l'avantage de cette dernière c'est que les différentes tâches peuvent potentiellement accéder à des **données partagées** cependant il peut y avoir un risque de décohérence entre les deux.

Les goroutines

Pourquoi les goroutines ?

L'un des aspects les plus intéressants dans Go est son modèle de concurrence, il rend la création de programmes multi-threads simples.

Go est capable d'effectuer plusieurs opérations simultanément. C'est particulièrement important sur les processeurs **multicœurs** actuels. Les programmes n'utilisant qu'un seul cœur laisse une grande partie de la puissance de traitement perdue, coup de chance car Go nous permet d'utiliser pleinement les cœurs de notre processeur grâce aux **goroutines**.

Pratiquons un peu !

Pour mieux comprendre les gains de performances avec les goroutines, laissez moi vous présenter un programme standard sans goroutines qui attend 3 secondes pour chaque appel de la fonction `run()`.

```
package main
```

```
import (
    "fmt"
    "time"
)

func run(name string) {
    for i := 0; i < 3; i++ {
        time.Sleep(1 * time.Second) // attendre 1 seconde
        fmt.Println(name, " : ", i)
    }
}

func main() {
    debut := time.Now()
    run("Hatim")
    run("Robert")
    run("Alex")
    fin := time.Now()
    fmt.Println(fin.Sub(debut))

}
```

Résultat :

```
Hatim  :  0
Hatim  :  1
Hatim  :  2
Robert  :  0
Robert  :  1
Robert  :  2
Alex  :  0
Alex  :  1
Alex  :  2
9.0095154s
```

Sans grande surprise le temps d'exécution est d'un peu après 9 secondes. Maintenant essayons d'améliorer les performances de notre programme en utilisant les goroutines.

Pour créer une goroutine il faut placer le mot clé `go` avant un appel de fonction, exemple :

```
package main
```

```
import (
    "fmt"
    "time"
)

func run(name string) {
    for i := 0; i < 3; i++ {
        time.Sleep(1 * time.Second)
        fmt.Println(name, " : ", i)
    }
}

func main() {
    debut := time.Now()
    go run("Hatim")
    go run("Robert")
    run("Alex")
    fin := time.Now()
    fmt.Println(fin.Sub(debut))

}
```

Avertissement

J'ai volontairement pas placé le mot clé go avant la ligne `run("Alex")`, vous allez comprendre pourquoi plus tard.

Résultat :

```
Robert  :  0
Hatim  :  0
Alex  :  0
Hatim  :  1
Robert  :  1
Alex  :  1
Robert  :  2
Hatim  :  2
Alex  :  2
3.0022266s
```

Les problèmes des goroutines

Hé hé, vous voyez c'est un sacré gain de temps d'exécution . Il ne faut pas se réjouir trop vite car maintenant je vais rajouter une goroutine à la ligne `run("Alex")`.

```
package main

import (
    "fmt"
    "time"
)

func run(name string) {
    for i := 0; i < 3; i++ {
        time.Sleep(1 * time.Second)
        fmt.Println(name, " : ", i)
    }
}

func main() {
    debut := time.Now()
    go run("Hatim")
    go run("Robert")
    go run("Alex")
    fin := time.Now()
    fmt.Println(fin.Sub(debut))

}
```

Résultat :

```
0s
```

Oulala , 0 seconde et aucune fonctio `run()` qui s'exécute ?! Mais que s'est-il passé ?

Pour répondre à ces questions répondrez d'abord à celle la : "Selon vous combien de goroutines se sont exécutés ?"

La réponse est 4 !

Il y a certes les 3 goroutines de la fonction `run()` mais aussi une **goroutine principale** , je m'explique. Lorsqu'un programme Go démarre, une goroutine commence à

s'exécuter immédiatement, c'est la goroutine principale de votre programme, c'est celle qui est exécutée lorsque votre programme commence à s'exécuter. C'est la goroutine à partir duquel d'autres **goroutines enfants** seront générées (ici les goroutines enfants sont ceux de la fonction `run()`).

Lorsque l'exécution de la goroutine principale est terminée, les goroutines enfants sont abandonnées aussi

Ce qui s'est passé c'est que le thread principal s'est alors terminé après l'exécution de la fonction `fmt.Println(fin.Sub(debut))` et c'est qui a tué les goroutines enfants.

La Solution

Pour éviter ce type problème il est possible d'utiliser la fonction `time.Sleep(temps d'exécution de vos goroutines)` avant la fin d'exécution de votre goroutine principale, mais le problème avec cette technique c'est qu'il faut connaître à l'avance connaître le temps d'exécution de la totalité de toutes vos goroutines.

Il existe une façon beaucoup plus simple pour synchroniser nos threads en utilisant la structure `WaitGroup` de bibliothèque `sync`. Je vous dévoile d'abord le code et ensuite je vous l'explique.

```
package main

import (
    "fmt"
    "sync"
    "time"
)

var wg sync.WaitGroup // instanciation de notre structure WaitGroup

func run(name string) {
    defer wg.Done()
```

```
    for i := 0; i < 3; i++ {
        time.Sleep(1 * time.Second)
        fmt.Println(name, " : ", i)
    }
}

func main() {
    debut := time.Now()

    wg.Add(1)
    go run("Hatim")
    wg.Add(1)
    go run("Robert")
    wg.Add(1)
    go run("Alex")

    wg.Wait()
    fin := time.Now()
    fmt.Println(fin.Sub(debut))
}
```

Résultat :

```
Hatim  :  0
Alex  :  0
Robert  :  0
Alex  :  1
Robert  :  1
Hatim  :  1
Alex  :  2
Robert  :  2
Hatim  :  2
3.0108647s
```

La structure WaitGroup vous permet d'attendre la fin d'exécution d'une collection de goroutines. La méthode `Add()` permet de définir le nombre de goroutines à attendre (on l'incrémente de 1 à chaque création de goroutine). Puis chacune des goroutines s'exécute et appelle la méthode `Done()` lorsque la goroutine a terminé de s'exécuter. Dans le même temps, la méthode `Wait()` est utilisée pour empêcher l'exécution d'autres lignes de code jusqu'à ce que toutes les goroutines soient terminées.

Ici le mot-clé `defer` est extrêmement important ! La fonction qui est placée après le mot-clé `defer` s'exécutera à chaque fois qu'on quittera notre fonction même en cas de panique (plantage) de la fonction ! Le mot-clé `defer` nous garantit alors l'exécution de la méthode `Done()`. Si vous supprimez le mot-clé et que votre programme panique (plante) alors la méthode `Done()` ne sera jamais exécutée et votre programme tournera en boucle.

Information

Pour information vous pouvez utilisez la méthode `Wait()` autant de fois que vous voulez.