

Théorie et Pratique de la Concurrency

TP n°1 — Routines (Go) et Threads (Java)

Jean-Baptiste Yunès
Jean-Baptiste.Yunes@u-paris.fr

28 septembre 2026

Attention : il est interdit d'utiliser une intelligence artificielle pour répondre aux questions.

Objectifs

- comprendre la différence d'architecture entre un thread OS (plate-forme) et une unité d'exécution légère (go-routine / green thread).
- observer l'empreinte mémoire et les limites de passage à l'échelle des deux modèles.
- manipuler les primitives de lancement et d'attente basiques en Go (`go`, `sync.WaitGroup`) et en Java (`Thread`, `join`).

Go

Écrire un programme en Go qui lance simultanément N goroutines. Chaque goroutine effectue une pause de 2 secondes (avec `Sleep` du package `time`) puis se termine.

Gérer l'attente de toutes les goroutines à l'aide d'un `WaitGroup` du package `sync`.

1. Tester le programme avec $N = 100, 1.000, 10.000, 100.000$.
2. Observer l'empreinte mémoire du processus pendant le `Sleep` (via la commande `top` / `htop` ou n'importe quel équivalent).
3. Mesurer le temps total d'exécution du programme. Pourquoi le temps reste-t-il proche de 2 secondes malgré le nombre massif de goroutines ?

Java

Écrire un programme équivalent en Java en utilisant des Threads natifs (plate-forme) (classe `Thread`).

Chaque thread effectue également un `Thread.sleep(2000)`. L'attente de la fin des threads se fera soit :

- avec une boucle d'appels à `.join()`,
- soit avec un `CountDownLatch` (package `java.util.concurrent`).

1. Tester le programme avec $N = 100, 1.000$, puis 10.000 threads.
2. Que se passe-t-il lorsque l'on tente de lancer 100.000 threads natifs ?
3. Quelle exception est levée et pourquoi ?
4. En vous basant sur la taille par défaut de la pile d'un thread Java (*stack size*, généralement 1 Mo), calculez la mémoire théorique nécessaire pour 10.000 threads.

Utiliser des *virtual threads* de Java 21 (`Thread.ofVirtual().start()`) et refaire les tests.

Du travail de calcul...

Modifier les codes de sorte que le travail des routines ou des Threads corresponde au calcul du 30^e terme de la suite de Fibonacci (attention on utilisera la définition récursive «bête» pour faire durer suffisamment le calcul).

Quels sont les temps obtenus avec 1, 2, 3, 4, 5,..., 20, 30, 40, 50..., 100 routines ou Threads ?

Que peut-on en conclure ?

Du travail réseau

On utilisera comme serveur (qui s'exécutera en local) le code Go suivant :

```
package main

import (
    "fmt"
    "net/http"
    "time"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        // On simule une latence d'E/S de 100ms (lecture fichier)
        time.Sleep(100 * time.Millisecond)
        fmt.Fprintln(w, "OK")
    })

    fmt.Println("Serveur de TP prêt sur http://localhost:8080")
    http.ListenAndServe(":8080", nil)
}
```

Modifier les codes de sorte que le travail des routines ou des Threads corresponde à l'envoi d'une requête au serveur et à la lecture de la réponse :

- Go : utiliser `http.Get("http://localhost:8080")` puis assurer la fermeture du *body* par `reponse.Body.Close()` ;
- Java : Utilisation de `HttpClient.newHttpClient().send(...)`.