

4. Avalie experimentalmente quão leve goroutines em Go são em relação à threads em Java. Em particular, você deve avaliar o consumo de memória de programas equivalentes nas respectivas linguagens.

(Para esta questão fizemos o uso de métodos já prontos como o `totalMemory()` em java, E o `Alloc` de GO para descobrir quanto de memória tinha sido alocado para cada procedimento.)

Inicialmente, foi pensado na ideia de criarmos inúmeras threads para o programa em java e verificarmos, através do método pronto `totalMemory()`, quanto que a criação e execução dessas threads consumia de memória. A ideia que pensamos foi: criarmos um `while(True)` e deixar essas threads sempre sendo criadas ou colocarmos as threads para dormirem para que assim a execução não fosse tão rápida e conseguíssemos pegar o tanto de memória utilizada já que essa não muda se for muito rápido. **Problema:** Mesmo implementando das duas formas, não conseguimos fazer com que o total de memória consumida pelas threads fosse mudado. Conseguimos mudar os valores para o código em go, ou seja, cada momento que criamos uma nova goroutine, aumentamos o número de memória alocada para a execução do programa. A seguir segue o print da execução dos códigos:

1. Criação e execução de uma goroutine:



The screenshot shows a Go code editor on the left and a terminal window on the right. The code in `main.go` defines a `main` function that creates a slice `overall` of type `[]int`, appends three slices `a`, `b`, and `c` (each containing 9,999,999 integers) to it, sleeps for one second, sets `overall` to `nil`, prints memory usage, and calls `runtime.GC()`. The terminal output shows the Go version as `go1.9.4 linux/amd64` and the allocated memory as `Alloc = 7 MiB`.

```
main.go  saved
8
9 func main() {
10
11     var overall []int
12     a := make([]int, 0, 9999999)
13     //b := make([]int, 0, 9999999)
14     //c := make([]int, 0, 9999999)
15     overall = append(overall, a)
16     //overall = append(overall, b)
17     //overall = append(overall, c)
18     time.Sleep(time.Second)
19
20
21     overall = nil
22     PrintMemUsage()
23
24     runtime.GC()
25
26 }
```

```
go version go1.9.4 linux/amd64
Alloc = 7 MiB
```

2. Criação e execução de duas goroutines:

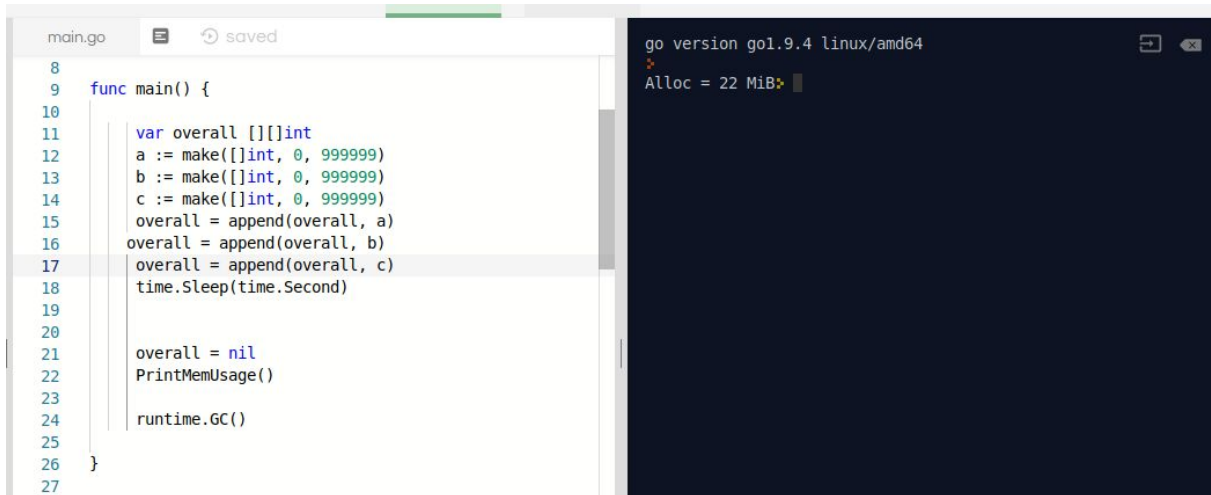


The screenshot shows a Go code editor on the left and a terminal window on the right. The code in `main.go` defines a `main` function that creates a slice `overall` of type `[]int`, appends two slices `a` and `b` (each containing 9,999,999 integers) to it, sleeps for one second, sets `overall` to `nil`, prints memory usage, and calls `runtime.GC()`. The terminal output shows the Go version as `go1.9.4 linux/amd64` and the allocated memory as `Alloc = 15 MiB`.

```
main.go  saved
8
9 func main() {
10
11     var overall []int
12     a := make([]int, 0, 9999999)
13     b := make([]int, 0, 9999999)
14     //c := make([]int, 0, 9999999)
15     overall = append(overall, a)
16     overall = append(overall, b)
17     //overall = append(overall, c)
18     time.Sleep(time.Second)
19
20
21     overall = nil
22     PrintMemUsage()
23
24     runtime.GC()
25
26 }
```

```
go version go1.9.4 linux/amd64
Alloc = 15 MiB
```

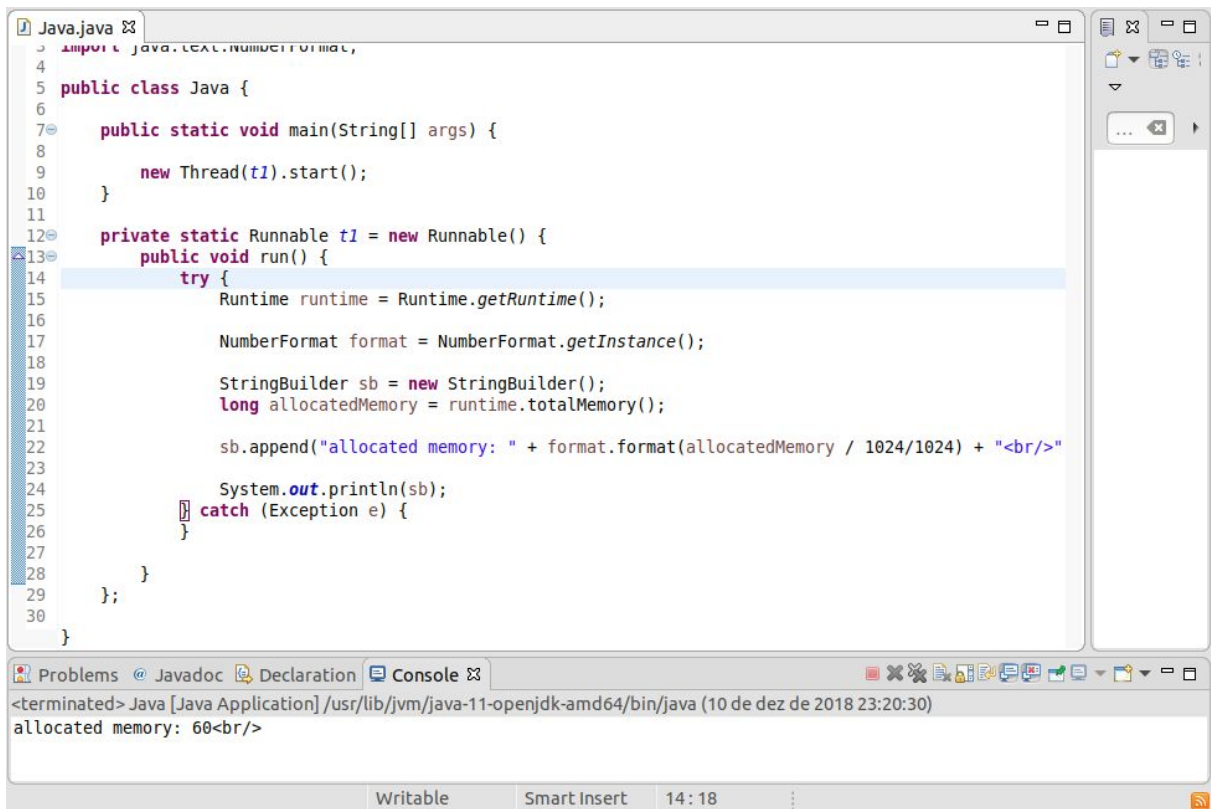
3. Criação e execução de três goroutines:



```
main.go
8
9 func main() {
10
11     var overall [][]int
12     a := make([]int, 0, 999999)
13     b := make([]int, 0, 999999)
14     c := make([]int, 0, 999999)
15     overall = append(overall, a)
16     overall = append(overall, b)
17     overall = append(overall, c)
18     time.Sleep(time.Second)
19
20
21     overall = nil
22     PrintMemUsage()
23
24     runtime.GC()
25
26 }
27
```

```
go version go1.9.4 linux/amd64
Alloc = 22 MiB
```

4. Criação e execução de uma thread:



```
Java.java
1 import java.text.NumberFormat;
2
3
4
5 public class Java {
6
7     public static void main(String[] args) {
8
9         new Thread(t1).start();
10    }
11
12    private static Runnable t1 = new Runnable() {
13        public void run() {
14            try {
15                Runtime runtime = Runtime.getRuntime();
16
17                NumberFormat format = NumberFormat.getInstance();
18
19                StringBuilder sb = new StringBuilder();
20                long allocatedMemory = runtime.totalMemory();
21
22                sb.append("allocated memory: " + format.format(allocatedMemory / 1024/1024) + "<br/>");
23
24                System.out.println(sb);
25            } catch (Exception e) {
26            }
27        }
28    };
29
30 }
```

```
<terminated> Java [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (10 de dez de 2018 23:20:30)
allocated memory: 60<br/>
```

Com isto podemos notar que a memória alocada com as threads em java são bem mais custosas que as goroutines, pois para uma gouroutine tivemos a alocação de 7 Mib que equivalem a 0,00734003 gigaBytes, e a criação de pelo menos uma thread deu bem superior: igual a 60.