

Tema 10 - Recolección de basura

Felipe Ortega, Enrique Soriano, Gorka Guardiola
GSyC, ETSIT. URJC.

Sistemas Distribuidos (SD)

18 de enero, 2021





(cc) 2015- Grupo de Sistemas y Comunicaciones.,
Algunos derechos reservados. Este trabajo se entrega bajo la licencia
Creative Commons Reconocimiento - NoComercial - SinObraDerivada
(by-nc-nd). Para obtener la licencia completa, véase
<https://creativecommons.org/licenses/by-nc-nd/3.0/es/>.

Contenidos

10.1 Concepto

10.2 Reference counting

10.3 Mark-and-sweep

10.4 Tri-color

10.5 Aplicaciones

Referencias

10.1 Concepto

¿Qué es la recolección de basura?

- ▶ Gestión de memoria automática.
- ▶ El GC encarga de liberar la memoria no usada por nosotros.
- ▶ Evita problemas de doble-free, leaks, liberar memoria que está todavía referenciada, etc.
- ▶ Desventajas: menos eficiente (gasto de memoria), pausas para recolectar.

¿Qué es la recolección de basura?

Hay distintos tipos de recolectores:

- ▶ Reference Counting.
- ▶ Mark-and-Sweep.
- ▶ Tri-color.
- ▶ y muchos otros: semi-space, generacionales, etc...

10.2 Reference counting

Reference counting

- ▶ Cada objeto tiene asociado un contador de referencias.
- ▶ Cada vez que se ejecuta una asignación o se sale de ámbito, se actualiza la cuenta de referencias de los objetos involucrados.
- ▶ Cuando un objeto tiene cero referencias, se puede recolectar.

Reference counting

Siendo p y q referencias, la siguiente asignación

$$p = q$$

se puede traducir en el siguiente código:

```
if p != q {  
    if p != nil {  
        p.refCount--  
        if p.refCount == 0 {  
            free(p)  
        }  
    }  
    p = q  
    if p != nil {  
        p.refCount++  
    }  
}
```

Reference counting

- ▶ Ventaja: cada objeto se puede recolectar justo en el momento en el que se queda sin referencias.
- ▶ ¿Si tenemos objetos que referencian a otros objetos?

$$A \rightarrow B \rightarrow C \rightarrow D$$

- ▶ Funciona bien: si se libera A, se decrementa el contador de referencias de B, y si llega a cero, se libera B. Lo mismo pasaría para C y para D.

Reference counting

- Problema: ¿qué pasa con los ciclos?

$$A \rightarrow B \rightarrow C \rightarrow A$$

- No funciona.
- En ese caso, hay que trazar las referencias.
- *Reference tracing* (alcanzabilidad).

10.3 Mark-and-sweep

Mark-and-sweep

- ▶ Es la base de la mayoría de GC actuales.
- ▶ Los objetos no se recolectan inmediatamente.
- ▶ Se lanza el GC cuando hay poca memoria libre, la memoria está muy fragmentada, cuando se reserva mucha memoria de golpe, etc.
- ▶ Tiene dos fases:
 1. Mark: se marcan los objetos como vivos o no vivos.
 2. Sweep: se liberan los objetos no vivos.

Mark-and-sweep

- ▶ Se denominan **root**:
 - ▶ Los objetos referenciados por variables globales/estáticas.
 - ▶ Los objetos referenciados por variables en la pila.
 - ▶ Los objetos referenciados por registros de la CPU.
- ▶ Se trazan las referencias desde los root.

Mark-and-sweep

Mark: se recorre los objetos accesibles desde los *root* y los marca.

Esta operación es recursiva:

```
func mark(p Object) {  
    if !p.marked {  
        p.marked = true  
        for q := range objectsReferencedBy(p) {  
            mark(q)  
        }  
    }  
}
```

Mark-and-sweep

Sweep: se escanea el *heap* en busca de objetos no accesibles y se eliminan.

Tiene que dejar preparados los objetos para la siguiente iteración del GC:

```
func sweep() {  
    for o := range objectsInTheHeap() {  
        if p.marked {  
            p.marked = false  
        } else {  
            free(p)  
        }  
    }  
}
```


Mark-and-sweep

- ▶ Hasta que no termina la fase de mark no sabremos si un objeto puede ser liberado o no.
- ▶ La fase *mark* no es concurrente con la aplicación → ***Stop the world!*** (STW).
- ▶ Mete parones muy largos porque hay que escanear todas las pilas.
- ▶ Hay muchas variantes de este algoritmo, algunas concurrentes.

10.4 Tri-color

Tri-color

- ▶ Dijkstra et al., 1978.
- ▶ Concurrente: el GC ejecuta concurrentemente con la aplicación (que se denomina *mutator*).
- ▶ El GC es un bucle haciendo *mark* y *sweep*.
- ▶ GC de Go (al menos desde 1.5) implementa una versión de este algoritmo, cambian detalles de la barrera.

Tri-color

Se crean tres grupos de objetos.

- ▶ **Blancos:** son los condenados a la recolección por ahora. Se inicializa con todos los objetos que no son referenciables desde los root.
- ▶ **Negros:** son objetos referenciables (vivos) para los que hay certeza de que no hacen referencia a ningún objeto Blanco.
- ▶ **Grisés:** son objetos referenciables (vivos) que no han sido escaneados todavía para ver si tienen alguna referencia a objetos actualmente en el grupo de Blancos. El grupo se inicializa con todos los objetos referenciables por los root.

Tri-color

Mark: colorea los objetos. Para cuando no hay objetos Grises.

```
func mark() {  
    for ! GreySet.Empty() {  
        o := GreySet.Next()  
        o.setBlack()  
        for q := range objectsReferencedBy(o) {  
            q.setGrey()  
        }  
    }  
}
```

Tri-color

Sweep: libera los objetos Blancos.

```
func sweep() {  
    for ! WhiteSet.Empty() {  
        o := WhiteSet.Next()  
        free(o)  
    }  
}
```

Tri-color

- ▶ En todo momento se debe conservar la siguiente invariante:

“Ningún objeto Negro hace referencia directa a un objeto Blanco”.

- ▶ Cuando no hay objetos Grises, sabemos que los Blancos ya no pueden ser referenciados por nadie → se pueden liberar.

Tri-color

- ▶ El *mutator* tiene que conservar la invariante.
- ▶ Para ello, cada vez que el *mutator* cambia una referencia al heap, se se ejecuta la *Write Barrier* (WB).
- ▶ La *Write Barrier* (WB) es una pequeña función que colorea el objeto que se acaba de referenciar a Gris si este era Blanco.
- ▶ Esto asegura que ese objeto será escaneado en un futuro.

10.5 Aplicaciones

GC en Go 1.5

GC Algorithm Phases

Off		GC disabled Pointer writes are just memory writes: *slot = ptr
Stack scan	WB on	Collect pointers from globals and goroutine stacks Stacks scanned at preemption points
Mark		Mark objects and follow pointers until pointer queue is empty Write barrier tracks pointer changes by mutator
Mark termination		Rescan globals/changed stacks, finish marking, shrink stacks, ... Literature contains non-STW algorithms: keeping it simple for now
Sweep	STW	Reclaim unmarked objects as needed Adjust GC pacing for next cycle
Off		Rinse and repeat

<https://talks.golang.org/2015/go-gc.pdf>

Elegir un algoritmo de GC/saber cuál es mejor

- ▶ Va de la mano del *allocator*.
- ▶ Cómo se comporta con diferentes cargas de trabajo (fragmentación, a lo largo del tiempo...), cuanta basura no recoge.
- ▶ Latencia de una pasada y cada cuánto hay que ejecutarlo.
- ▶ Eficiencia (cuanto tiempo para el programa, ¿lo para?, uso CPU y memoria extra.
- ▶ Concurrencia.
- ▶ Escalabilidad (cuando el *heap* crece).
- ▶ Configuración (¿hay que configurarlo?, ¿es complicado?, ¿es configurable?).
- ▶ Portabilidad (depende del hardware).
- ▶ Muchos compromisos, no hay uno perfecto.

Bibliografía I



[van Steen & Tanenbaum, 2017] van Steen, M., Tanenbaum, A. S.
Distributed Systems.
Third Edition, version 01. 2017.