Moderne Algorithmen für die Garbage Collection

Vorstellung moderner Algorithmen für die Garbage Collection am Beispiel von Go und Java

---

xnacly, hlxid

23. Mai 2024

DHBW

1. Garbage collection
2. Strategien
3. Programmiersprachen
4. Implementierungen
5. Performance
6. Borrow Checking
7. Fragen?

Garbage collection

Garbage collection

Garbage collection ist definiert als

- Erkennung von nicht mehr verwendeten Speicherbereichen
- automatisches Entfernen dieser

Und zielt darauf ab das obige:

- schnell
- effizient
- mit wenig Latenz
- mit wenig RAM und CPU Auslastung

umzusetzen.

Garbage collection - Nutzen

Befreit den Programmierer zumeist von:

- Manueller Speicherallokierung mit `malloc`, `calloc` und `realloc`
- Manueller Speicherfreigabe mit `free`
- Pointer arithmetik

Verhindert Fehler wie: [1]

- Seg faults (illegaler Speicherzugriff)
- Use after `free` (Speicherzugriff auf bereits aufgeräumte Speicherbereiche)
- Memory leaks (Speicher wird nach Verwendung nicht freigegeben)

[1] 70% der CVE's von Chromium sind Speicherzugriffbezogen, siehe: https://www.chromium.org/Home/chromium-security/memory-safety/

Garbage collection - Problematiken

- Peformance schlechter im Vergleich zu manuellem management und borrow checker
- Einfluss von GC-zyklen auf Program nicht vorhersehbar
- Manuelle Allokationen mit Speicherarena effizienter für große Datenmengen als jedes Objekt einzeln allokieren

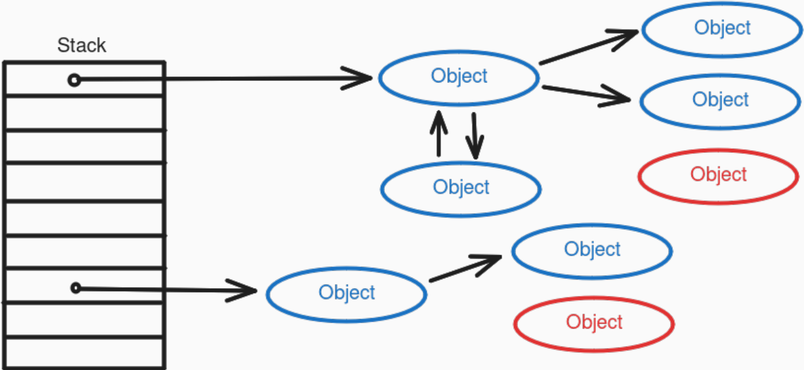
Strategien

Strategien

Strategien unterscheiden sich in ihrer:

- Erkennung von unerreichbaren Objekten
- Entfernung von unerreichbaren Objekten
- Latenz, RAM und CPU Verbrauch

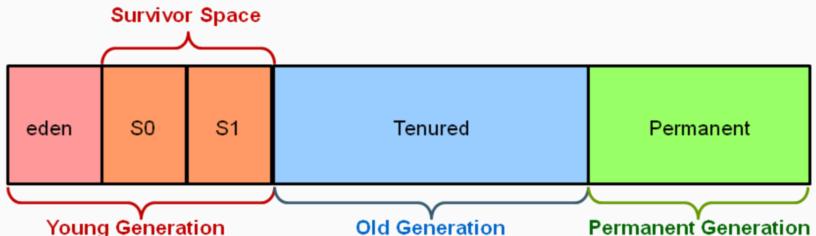
Strategien - Mark & Sweep



Strategien - Generational garbage collection

Optimierung basierend auf der Beobachtung, dass die meisten Objekte nur kurzlebig sind (Infant Mortality)
Aufteilung in drei Speicherbereiche aufgeteilt:

- Young, bestehend aus Eden und Survivor Space
- Old
- Permanent



Strategien - Generational garbage collection

- Eden
  - Alle neuen Objekte werden hier allokiert
  - Objekte welche einen GC Lauf überleben, werden in den Survivor Space verschoben
- Survivor
  - Hier sind Objekte welche eine gewisse Zeit überlebt haben
  - Objekte welche eine längere Zeit überlebt haben werden in den Old Space verschoben
- Old
  - Hier sind Objekte welche schon länger existieren und es unwahrscheinlich ist, dass sie bald gelöscht werden

Eden wird aufgrund von Infant Mortality häufig collected, ist aber klein da lang lebende Objekte nicht dort sind.

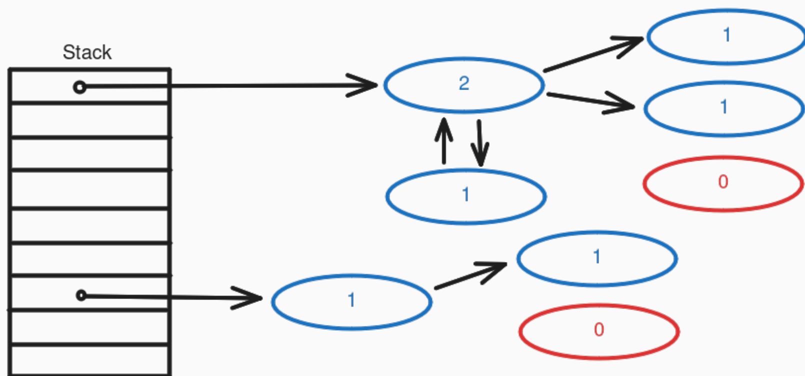
Strategien - Reference counting

- Jedes Objekt hat einen Zähler, welcher die Anzahl der Referenzen auf das Objekt zählt.
- Beim Erstellen eines Pointers auf das Objekt wird der Zähler inkrementiert, beim Löschen dekrementiert.
- Wird der Zähler 0, so wird das Objekt deallokiert.

Probleme bei Referenzzyklen: Zähler wird nie 0, Objekte werden nicht gelöscht.

Strategien - Reference counting

```
struct PyObject {
    // ...
    uint32_t ob_ref_local;       // local reference count
    // ...
};
```



Programmiersprachen

Programmiersprachen

- Viele Sprachen mit und ohne GC
- Algorithmen, Implementierungen und Performance sehr unterschiedlich
- High level sprachen eher mit GC, low level eher ohne

Programmiersprachen - Ohne gc

- C
  - manuelles memory managment
- C++
  - manuelles memory managment
  - reference counting on demand
- Rust
  - borrow checker
  - reference counting on demand

Programmiersprachen - Mit gc

- Go
  - escape analysis
  - mark & sweep
  - Compiliert in Maschinen Code(AOT), GC in generierter Binary
- Java
  - mehrere garbage collectoren
  - generational standardmäßig
  - Compiliert in Bytecode(AOT), Ausführung mit Bytecode vm (JVM)
- Python
  - Reference counting
  - Erkennung von Referenzzyklen
- JavaScript
  - generational
  - JIT, Bytecode vm (V8)

Implementierungen

Implementierungen - Go

Go verwendet einen Mark & Sweep garbage collector.

- Concurrent
- Tri-color (objekte werden eingefärbt)
- Minimale Latenz durch kurzes stop the world [2]
- Konfigurierbar über:
  - `GOGC`: Größe heap relativ zur Größe aller erreichbaren Objekte

[2] siehe: https://www.youtube.com/watch?v=aiv1JOfMjm0

GO - Tri-color

Basierend auf Dijkstra (1978)[3]

- Objekte: Weiß, Grau oder Schwarz
- zu Begin des GC-Zyklus alle Objekte Weiß
- GC besucht alle Roots[4] markiert als Grau
- Ein Objekt wird ausgewählt und als Schwarz markiert, GC sucht ab hier nach Referenzen zu anderen Objekten
- Wird ein weißes Objekt gefunden $\rightarrow$ als grau markiert.
- Prozess wird wiederholt bis keine weißen Objekte mehr
- Verbleibende weiße Objekte sind nicht erreichbar $\Rightarrow$ Deallokieren

[3] siehe: https://dl.acm.org/doi/10.1145/359642.359655
[4] stack & statische variablen, etc...

Implementierungen - Java

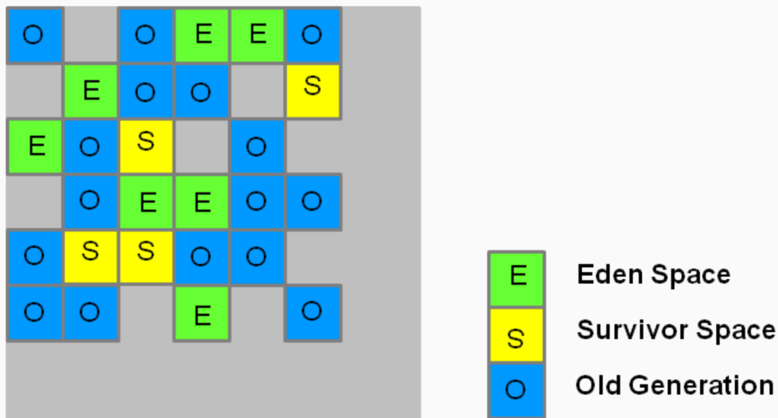
Bei Java gibt es verschiedene GC-Implementierungen, welche via CLI Flag ausgewählt werden können.

- Serial GC
- Parallel GC
- Concurrent Mark and Sweep (CMS)
- Garbage First (G1) GC
- Z GC

G1 ist Standard und wird deshalb hier vorgestellt.

Java - G1 GC Speicherstruktur

G1 ist ein generational GC. G1 allokiert Speicherblöcke, welche jeweils einer Generation zugeordnet sind.



Java - G1 GC Funktionsweise

- Wird eine Generation zu voll, so wird Garbage Collection gezielt für eine/mehrere Speicherregionen der Generation ausgeführt.
- Überlebende Objekte werden in einen neuen Bereich der gleichen oder darüberliegenden Generation kopiert
  $\Rightarrow$ Kompaktierung
- Dannach ist die Region leer und kann neu allokiert werden.

Performance

Performance

Diverse Kriterien möglich, hier beschränkt auf:

- Speicherverbrauch: stärke RAM-Intensivität
- Latenz: Umfang Stoppzeiten des Programs
- Sicherheit: Speicherzugriffsicherheit
- Nutzbarkeit: Komplexität der Strategien

Borrow Checking

Borrow Checking - Theorie

`free` Calls können automatisch eingefügt werden, wenn der Compiler weiß wielange ein Objekt verwendet wird (*Lifetime*)
$\Rightarrow$ automatisches Speichermanagement ohne GC
Um dies möglich zu machen werden die Konzepte des *Ownership* und *Borrowing* eingeführt.

Borrow Checking - Ownership & Borrowing

Ownership:

- Eine Variable hat einen Owner in Form einer Funktion
- Ownership kann von einer Funktion an eine andere übergeben werden (pass by value/return via moves)

Um auf Variablen an mehreren Stellen zugreifen zu können, kann diese geborrowed (verliehen) werden:

- Der Owner einer Variable kann diese an andere Strukturen verleihen.
- Borrowing ist nur für einen bestimmten Zeitraum möglich (Lifetime)
- Borrows dürfen nicht über die Lifetime des Owners hinaus bestehen, sichergestellt durch den Borrow Checker

Borrow Checking - Beispiel in Rust

```rust
use std::io::stdin;

fn main() {
    println!("Eingabe: ");

    let line = stdin().lines().next();
    if let Some(line) = line {
        let line = line.expect("Fehler beim Lesen");
        println!("Gelesene Eingabe: \"{line}\"");
    } else {
        println!("Keine Eingabe");
    }
}
```

Fragen?