Garbage Collection

Was ist Garbage Collection?

Wieso bricht folgendes nicht mit `OutOfMemoryError` ab?

```
Integer i;
Random r = new Random();
while (true) {
  i = new Integer(r.nextInt());
}
```

Objekte belegen Speicherplatz im Heap

JVM verwaltet Heap im Hintergrund

- neu erzeugte Objekte werden in freiem Speicher angelegt
- *garbage collection*: Speicherplatz nicht mehr referenzierter Objekte wird wieder freigegeben

  beachte: unreferenziertes Objekt kann (eigentich) nicht mehr referenziert werden; in Java kein Zugriff auf Heap möglich

Vor- und Nachteile

in einigen Sprachen wird Heapverwaltung dem Programmierer überlassen (z.B. C, C++)

Garbage Collection vermeidet zwei häufige Programmierfehler:

- Objekt wird freigegeben und danach verwendet
- Objekt wird *nicht* freigegeben und *nicht mehr* verwendet

Nachteile:

- keine Kontrolle über den Zeitpunkt, zu dem Objekte freigegeben werden; Garbage Collection findert zur Laufzeit statt
- wie findet man unreferenzierte Objekte?

  Garbage Collector muss alle verwendeten Objekte explizit in Tabellen halten

Reference Counting

eine Möglichkeit zum Erkennen unreferenzierter Objekte

einfachstes Verfahren, in modernen Compilern nicht mehr benutzt

Objekte werden annotiert mit Anzahl der Verweisen auf sie; Buchführung bei

- Objekterzeugung
- Zuweisung
- Parameterübergabe (Achtung! Java benutzt call-by-value, aber Objekte werden generell über Referenzen angesprochen)
- Beendigung einer Methode
- Freigabe eines Objekts

Vorteil: direkte Freigabe möglich; Nachteile: Overhead in Platz und Zeit, Probleme bei zyklischen Datenstrukturen

Mark-and-Sweep

zwei Phasen

1. Breitensuche von Programmvariablen (Wurzelzeigern) aus markiert alle noch referenzierten Objekte
2. alle unmarkierten Speicherbereiche werden freigegeben

Vorteile:

- zyklische Strukturen unproblematisch
- kein Zeit-Overhead bei Zuweisungen, etc.
- sehr geringer Platz-Overhead

Nachteil:

- Programmlauf muss unterbrochen werden
- Laufzeit hängt von Größe des Heaps ab
- Breitensuche selbst braucht Speicherplatz! ($\rightsquigarrow$ Deutsch-Schorr-Waite-Algorithmus)

Problem: Fragmentierung

nach gewisser Zeit können belegte und freie Speicherbereiche weit verstreut sein

im Extremfall wäre noch genügend freier Speicher vorhanden, Objekt kann aber nicht angelegt werden, da kein genügend großer freier Speicherbereich zur Verfügung steht

Abhilfe: *mark-and-compact*, zu mark-and-sweep zusätzlich dritter Durchlauf, in dem belegte Speicherbereiche verschoben werden

offensichtliche Vor- und Nachteile, zusätzlich:

- entweder Rückwärtsreferenzen oder intelligente Berechnung der neuen Speicherstellen nötig
- evtl. keine einfache Verschiebung von Objekten möglich

Stop-and-Copy

unterteile Heap in zwei Hälften, eine davon bleibt immer frei

ist andere Hälfte voll, dann kopiere referenzierte Objekte in freie Hälfte; vertausche danach Funktion der beiden Hälften

Vorteile:

- Kompaktierung geschieht automatisch
- nur ein Durchlauf statt 3 nötig

Nachteile:

- nur halber Heap-Space steht zur Verfügung
- Objektkopieren ist aufwändig; langlebige Objekte werden hin und her verschoben

Generationen

zur Vermeidung von Kopieren langlebiger Objekte werden *Generationen* eingeführt

Bsp.: 2 Generationen *young* and *old*

- neue Objekte kommen in die Generation young
- haben Objekte eine gewisse Zahl an Kopiervorgängen in der Generation young überlebt, dann werden sie in die Generation old verschoben
- führt zu 4-Teilung des Heaps
- Garbage Collection in jeder Generation separat

im Prinzip beliebig viele Generationen vorstellbar

Verbesserungen

verschiedene Verbesserungen möglich

zielen i.A. darauf ab, den Programmfluss durch GC möglichst wenig zu unterbrechen

- paralleles GC

  nutzt mehrere CPUs aus; (Achtung! nicht "parallel zum eigentlichen Programmablauf")

- inkrementelles GC

  vermeidet längeres Anhalten des Programms; GC wird z.B. nur auf kleinen Stücken des Heaps ausgeführt (z.B. *train* Algorithmus)

- nebenläufiges GC

  GC wird in eigenen Threads ohne Anhalten des Programms (außer durch den Scheduler) ausgeführt

GC in der J2SE 5.0 Hotspot JVM

- 3 Generationen: *young*, *old* und *permanent*
- Generation young unterteilt in zwei *survivor spaces*: *fromSpace* und *toSpace* für stop-and-copy-Verfahren
- Generationen *old* und *permanent* werden per mark-and-compact verwaltet
- serielles GC ist Standard, aber parallele und nebenläufige Verfahren können per Kommandozeile gewählt werden
- Garbage Collector hat Logging-Mechanismus
- Objekte haben Finalisierungsflag
  - noch nicht finalisierte Objekte werden von GC in Queue zum Finalisieren gesteckt
  - bereits finalisierte Objekte werden direkt entfernt

Die Methode `finalize`

analog zu Konstruktoren gibt es in Java auch Destruktoren:
Methode `protected void finalize()` in Klasse `Object`

wird vom Garbage Collector vor Freigabe des Objekts aufgerufen

gedacht z.B. zur Freigabe von Systemressourcen, die von Objekt gehalten werden

zu beachten:

- GC ruft `finalize` auf jedem Objekt höchstens einmal auf
- keine rekursiven Aufrufe auf Objekten in Instanzvariablen nötig
- Exceptions in `finalize` unterbrechen Methode, werden ansonsten aber ignoriert

`finalize` sollte nicht benutzt werden (Rereferenzierung möglich, `OutOfMemoryError` kann auftreten, ...)

Beispiel

```
public class GC {
    public static void main(String[] args) {
        while (true) {
            new Muell();
        }
    }
}

class Muell {
    protected void finalize() {
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {};
    }
}
```

führt ultimativ zu `OutOfMemoryError`, aber wieso?