"Allocator design and implementation has strong potential as the subject of career-long obsession ..."
(Jason Evans)

Dynamische Speicherverwaltung

Wie funktionieren malloc & Co.?

Florian Westphal

26. April 2007

`1024D/F260502D <fw@strlen.de>`
`1C81 1AD5 EA8F 3047 7555`
`E8EE 5E2F DA6C F260 502D`

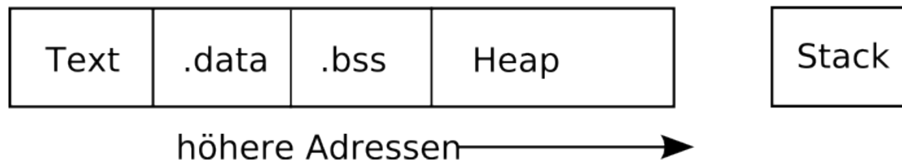
Themenübersicht

- Grundlagen: Speicherorganisation, etc.
- Funktionsweise: K&R malloc
- Aktuelle Implementationen:
  - GNU libc (ptmalloc2)
  - FreeBSD (phkmalloc, jemalloc)
- Automatische Speicherverwaltung ("Garbage Collection")
  - Funktionsprinzip
  - Verfahren / Algorithmen
  - Anwendung
    - Sun Java VM
    - Boehm GC

malloc/free vs. new/delete

- "beschaffen Speicher":
- `malloc(sizeof(bla))` vs. `new bla()` bzw.
  `malloc(42)` vs. `new char[42]`
- `malloc` und `new` sind aber nicht dasselbe (Konstruktoren, Überladung, etc.)
- `char *ptr = malloc(42) ... delete[] ptr` ist nicht erlaubt (können unterschiedlich implementiert sein, andere Heap-Bereiche verwalten, etc.)

Typische Speicherorganisation



- .data: Initialisierte Daten; .bss: Uninitialisierte Daten
- Größe d. Stacks wird bei Bedarf automatisch angepasst, wächst in Richtung d. niedrigeren Adressen
- Linux: `cat /proc/self/maps` zeigt (u.a.) von Prozess gemappte Speicher-Regionen

Heap

- Erweiterung des Datensegments
- Platz für den Heap muss "händisch" reserviert werden (*nix: mit `brk()`)
  - `brk(void *end_data_segment)` setzt das Ende des Datensegments auf bestimmten Wert
  - `sbrk(intptr_t increment)` erhöht Platz um `increment` Bytes (`brk`-Wrapper)
- Heap wächst "nach oben" (d.h. in Richtung der höheren Adressen)

Ein einfaches System

- Stets die nächsthöhere noch nicht vergebene Adresse liefern
- `brk()` wenn mehr Speicher gebraucht wird

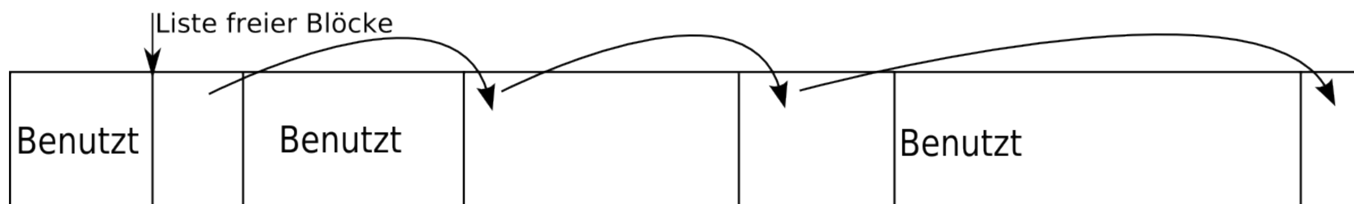
Vorteil: Schnell, sehr einfache Umsetzung, aber . . .

- Keine Unbegrenzte Ressource
- Speicher, den ein Programm gar nicht (mehr) benötigt, ist nicht mehr verfügbar

K&R malloc

Aus: Kerninghan, Ritchie: "The C Programming Language"

- Heap wird in Blöcke eingeteilt
- Block wird entweder verwendet, ist frei, oder gehört nicht zum Allokator



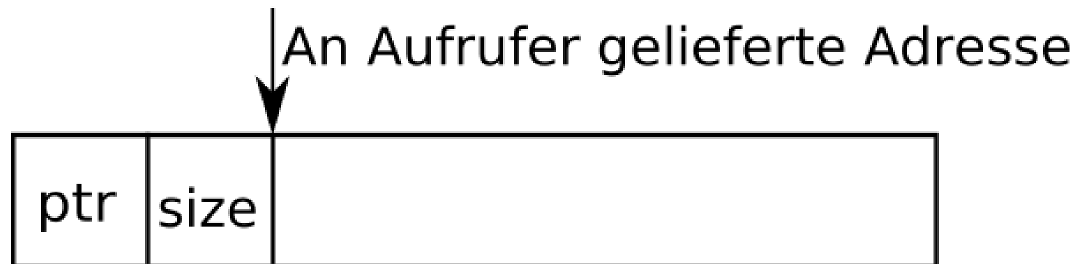
- Bei Speicheranfoderung (`malloc(len)`) wird Freelist durchlaufen bis Block $>=$ `len` gefunden wird
- Fall kein passender Block verfügbar: Datensegment vergrößern (`brk(2)`)

K&R malloc (forts.)

Header (vereinfacht, ohne Alignment):

```
struct header {
   struct header *ptr; /* naechster freier block */
   unsigned size;      /* groesse dieses blocks */
};
```

Header steht direkt vor den von z.B. `malloc` gelieferten Blöcken



Warum neue/andere Implementation?

1. Wie groß ist der Overhead? (Suchen von freien Blöcken, eigener Speicherverbrauch, etc.)

2. Qualität, z.B:

   - Fragmentation
   - Robustheit gegenüber fehlern (`free(a); ... free(a);`, Speicherüberschreiber, etc.)
   - Rückgabe von Speicher an das Betriebssystem, etc.

früher[tm]: Speicher Segmentiert (typisch: .text, .data/.bss/Heap, Stack)

- Prozess ist vollständig im Speicher, oder nicht ablauffähig

- Daher: Speicherverbrauch (Working Set) so klein wie möglich halten

Warum neue/andere Implementation (forts.) ?

- Heute: Speicher ist in "Pages" eingeteilt (typischer Wert (x86): 4k-Pages)
- nur gerade benötigte Pages müssen im Arbeitsspeicher sein
- trad. malloc: Meta-Informationen (malloc-Header) stehen vor jedem Block → sind über den ganzen Speicher verteilt, aufwendige Suche nach freien Blöcken
  - Caches werden schlecht genutzt
  - Falls ausgelagert wurde: Es müssen Pages auch dann wieder geladen werden, wenn die eigentl. Programmlogik sie gar nicht benötigt

→ (mehrere) getrennte Free-Listen für verschiedene Größen
→ Anzahl der Page-Zugriffe minimieren, wenn mögl. innerhalb von Pages arbeiten

mmap

- Blendet (Geräte-) Dateien oder "anonymen" Speicher in den Virtuellen Adressraum eines Prozesses ein
- Mapping kann unter Prozessen geteilt werden (`MAP_SHARED`,`MAP_PRIVATE`)
- Genaue Kontrolle über Zugriffsschutz (`PROT_EXEC`, `PROT_READ`, `PROT_WRITE`, `PROT_NONE`)
- Hinweise über Verwendung möglich (mit `madvise(2)`)
- `munmap()` gibt Speicher an den Kernel zurück

mmap (forts.)

- Vor/Nachteile:
  + kein "Locking" zwischen verschiedenen Blöcken möglich,
  + freigegebener Speicher steht dem System sofort wieder zu Verfügung
  - verschwendet Platz infolge Page-Alignment
  - Kernel muss Speicher erst "ausnullen"
- Bei "großen" Blöcken überwiegen die Vorteile (exakter Wert ist Systemabh.)
- Linux-Bonus: `mremap()`

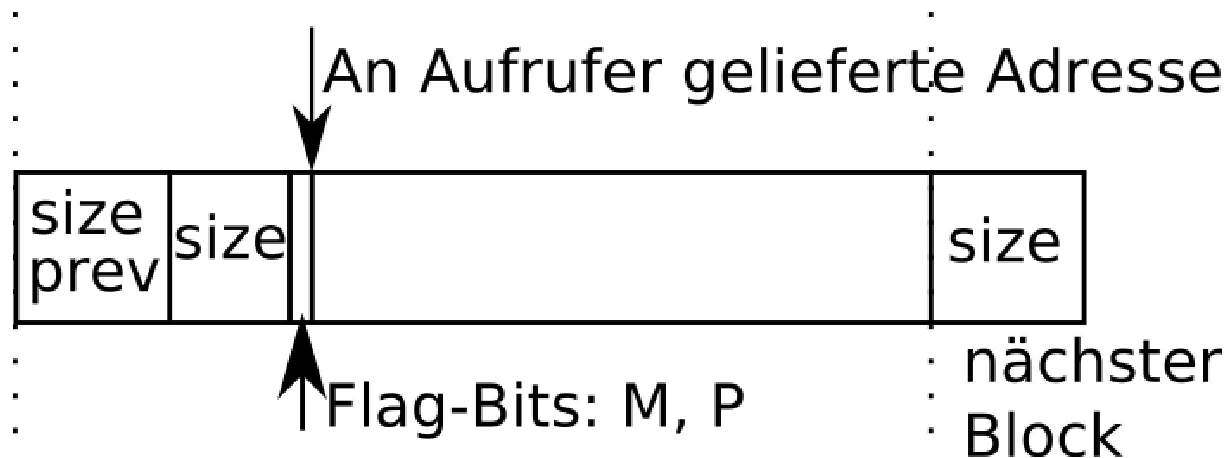
GNU libc (ptmalloc2)

- basiert auf malloc-Implementierung von Doug Lea ("dlmalloc")
- Thread-Support etc. von Wolfram Gloger
- sucht Kompromiss zwischen Speicherplatz-Effizienz und Geschwindigkeit: "good general-purpose allocator for malloc-intensive programs"

ptmalloc/dlmalloc Chunk-Struktur

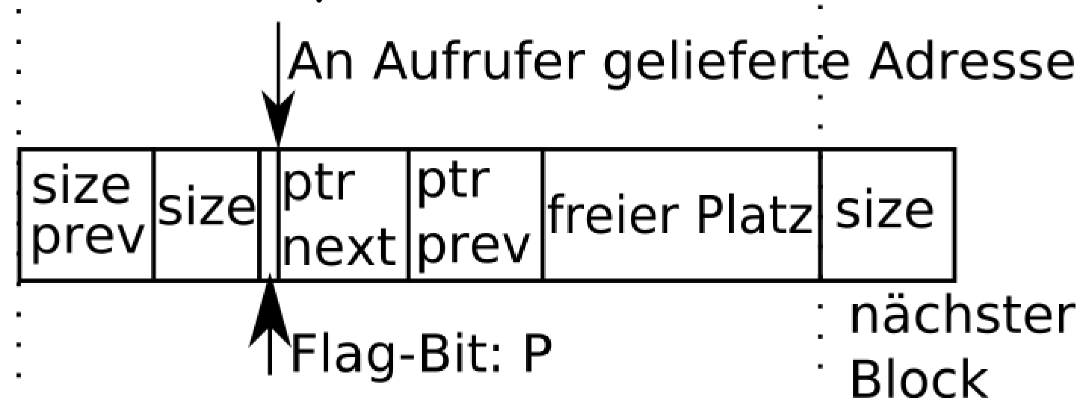
```
struct malloc_chunk {
  size_t prev_size; /* groesse d. vorherigen Blocks (falls frei)*/
  size_t size;      /*  Groesse in Bytes, incl. Overhead */
  struct malloc_chunk* fd; /* forward/backwards Zeiger, */
  struct malloc_chunk* bk; /* -- nur, wenn Block frei ist */ };
```

Header steht direkt vor den von z.B. `malloc` gelieferten Blöcken



P: vorheriger Block in Verwendung, M: `mmap()`?

ptmalloc/dlmalloc Chunk-Struktur – freier Chunk



- Chunks sind in Klassen ("bins", 128 doppelt verkettete Listen) eingeteilt:
  - ≥ 512 Byte: Nach Größe sortiert
  - *Smallbins*: 8, 16, 24, . . . 504 Byte; alle Chunks gleiche Größe
  - *Fastbins*: Array aus (einfach) verketteten Listen mit "aktuell" freigegebenen Chunks (≤ 64 Byte)
  - "unsorted": Reste aus Chunk-Splits, etc. fungiert als Warteschlange bis Chunks sortiert werden

Vorgehen bei Allozierung ("malloc(42)")

(glibc 2.5, malloc.c:3981 ff.)

1. Größe aufrunden (Alignment)

2. *Fastbins* (≤ 64, LIFO) bzw. *Smallbins* (< 512) durchsuchen

3. sonst ist Anfrage ≥ 512: $\rightarrow$ *Fastbins* zusammenfassen/auflösen

4. "unsorted"-Liste durchlaufen, in Bins einsortieren

   - Falls exakte Übereinstimmung mit angeforderter Größe: Ende
   - < 512, keine exakte Übereinstimmung und einziger größerer Chunk vorhanden: Split

5. $\geq 512 \rightarrow$ "bestfit"-Suche in nächstgrößten Bins

...und wenn das alles nichts half...

- "top" (liegt an der oberen Grenze des verfügbaren Speichers) hat genug Reserve?: Split
- Falls nein und es sind noch Fastbins vorhanden: diese Zusammenfassen und "alles von vorn" (siehe vorherige Folie)
- . . . und wenn alles nichts hilft: Heap vergrößern bzw. `mmap()`
  - Falls Anforderung Größer als `mmap_threshold`: `mmap()`
  - sonst `(s)brk()`
- `return NULL`, falls `(s)brk()` fehlschlägt

ptmalloc: free()

1. `chunk_is_mmapped`? $\rightarrow$ `munmap()`
2. $\leq 64 \rightarrow \textit{Fastbin}$
3. Anonsten: evtl. freie Nachbarn mit Block zusammenlegen
4. In "unsorted" bin ablegen
5. Ausnahme: Chunk grenzt an "top" $\rightarrow$ mit "top" zusammenlegen
6. Heap wird verkleinert, wenn "top" $\geq$ `DEFAULT_TRIM_THRESHOLD` (ca. 128KB)

ptmalloc: Tuning

- Umgebungsvariablen:
  - `MALLOC_CHECK_`: 0: Fehler ignorieren; 1: Fehlermeldung auf `stderr`; 2: `abort()`
  - `MALLOC_`...`MMAP_MAX_` , `TRIM_THRESHOLD_` , `MMAP_THRESHOLD_`, ... etc.

- Kann auch von Applikation selbst eingestellt werden: `mallopt()` bzw. `mallinfo()` (info mallopt bzw. info mallinfo)

- Linux: "Memory Overcommittment" (`/proc/sys/vm/overcommit_memory`)

ptmalloc: Tuning (forts.)

- `mtrace()` :
  - Schreibt Debug-Informationen in Datei (`export MALLOC_TRACE=filename`)
  - `/usr/bin/mtrace` kann diese auswerten
  - . . . aber nicht so mächtig wie z.B. "`valgrind --leak-check=full --tool=memcheck app`"
- Debugging-Hooks:
  `void *(*__malloc_hook) (size_t size, const void *caller)`, etc.

FreeBSD (phkmalloc)

- malloc-Implementierung von Poul-Henning Kamp (seit FreeBSD 2.2 / 1995): teilt Heap in Seiten fester Größe ein
- Haupt-Datenstruktur: Page Directory, enthält Zeiger auf Speicherseiten: "`struct pginfo **page_dir`" – wird via `mmap()` alloziert. Werte:
  - `MALLOC_NOT_MINE`: Seite gehört nicht zum Allokator
  - `MALLOC_FREE`: Seite ist frei
  - `MALLOC_FIRST` bzw. `MALLOC_FOLLOW`: Erste Seite bzw. Folgeseiten einer Allokation über mehrere Seiten
  - Seite ist in kleinere Chunks aufgeteilt: Zeiger auf `struct pginfo`

phkmalloc: Free-List

- Verzeichnet freie Seiten(-bereiche)
- die Freelist-Strukturen sind selbst mit `malloc()` alloziiert
- die freien Seiten selbst werden nicht referenziert

```
struct pgfree {
  struct pgfree *next,*prev; /* Freelist ist doppelt verkettet */
  void          *page;       /* naechste freie Seite(n) */
  void          *end;        /* Ende des freien Bereiches */
  size_t        size;        /* Anzahl freier Bytes */
};
```

phkmalloc: pginfo-Struktur

- Seite ist in Chunks gleicher Größe eingeteilt
- Bitmap zeigt noch freie Chunks innerhalb dieser Page an
- Pages mit denselben Chunk-Grösen bilden eine Liste, deren Köpfe in Slots am Beginn des pagedirs verzeichnet sind

```
struct pginfo {
 struct pginfo *next;  /* naechste freie page */
 void          *page;  /* Zeiger auf d. Speicherseite */
 u_short       size;   /* Groesse d. Chunks dieser Seite */
 u_short       shift;  /* shift-faktor f"ur die Groesse */
 u_short       total,free;/* Wieviel Chunks insgesamt/sind frei? */
 u_int         bits[1];/* Bitmap: Welche Chunks sind frei? */};
```

Vorgehen bei Allozierung ("malloc(42)")

(FreeBSD 6.2, src/lib/libc/stdlib/malloc.c:724 ff.)

- Allokator kennt 2 Strategien:
  1. Anforderung ist $>$ einer halben Page: ganze Seite(n) suchen
  2. Anforderung ist $\leq$ einer halben Page: Teile einer Seite liefern

- im zweiten Fall:
  - auf nächstes Vielfache von 2 Aufrunden
  - sucht passende Position im pagedir: Ergibt sich aus der Bitposition der "1" $\rightarrow$ schnell, da nur Bitshifts nötig
  - Wenn keine vorhanden: neue Page anfordern (nächste Folie) & partitionieren
  - (`pginfo[idx]->free--`), wenn 0 erreicht: Pagedir-Eintrag auf `pginfo[idx]->next` setzen

malloc_pages() : Anfordern ein oder mehrerer Seiten

- Runden auf `pagesize`-Grenze (4096 Byte auf x86)
- Free-Page Liste durchlaufen, wenn ausreichend freier Platz gefunden: nötige Seitenzahl abspalten
- Wenn kein passender Platz: `brk()`
- Pagedir-Zeiger wird auf `MALLOC_FIRST` gesetzt
- falls mehr als eine Page angefordert wurde: Rest auf `MALLOC_FOLLOW` setzen
- Pagedir-Eintrag ergibt sich durch Maskierung der Adresse der ersten Seite $\rightarrow$ keine Suche im Pagedir nötig

phkmalloc: free()

Da zwei Allokationsarten muss auch beim freigeben von Speicher unterschieden werden. Vorgehen:

- Pagedir-Eintrag finden (maskieren d. Adresse). Falls pagedir-Eintrag kein Magic-Wert (d.h. `MALLOC_FIRST`): Die Anforderung war $\leq$ einer halben Seite
- Sonst handelt es sich (mindestens) um eine ganze Seite $\rightarrow$ `free_pages()`
  - pagedir-Eintrag und alle folgenden auf `MALLOC_FREE` setzen (bis `pagedir[i] != MALLOC_FOLLOW`)
  - In Free-Liste eintragen (nach Adresse sortiert), direkt anliegende freie Pages werden zusammengelegt
  - Falls letzte Page: Heap verkleinern

phkmalloc: free_bytes()

Falls ein Pageinfo-Struktur vorhanden ist:

- Chunk-Nummer innerhalb der Page ermitteln: (via `pagedir[idx]->shift`) und Bitmap freier Chunks (`pginfo[idx]->bits[]`) aktualisieren
- Falls erster frei gewordene Chunk: Page wieder in `pginfo->next` des richtigen Pagedir-Eintrags einhängen
- Falls kein einziger Chunk mehr belegt ist: `free_pages()`

phkmalloc: Tuning

- . . . erfolgt über `/etc/malloc.conf`, Umgebungsvariable `MALLOC_OPTIONS` oder '`extern char *_malloc_options`'

- `/etc/malloc.conf`: 'Dangling symlink', "Datei" ist der Option-String, z.B. `ln -s AR /etc/malloc.conf`

  A : `abort()` , wenn der Allokator Fehler erkennt (z.B. double-free)
  H : `madvise()`- Hinweise über nicht verwendete Seiten an Kernel (verbessert Performance wenn Speicher ausgelagert werden muss)
  R : `realloc()` gibt Speicher wieder frei (sonst no-op wenn neue_grösse $\leq$ alte_grösse)

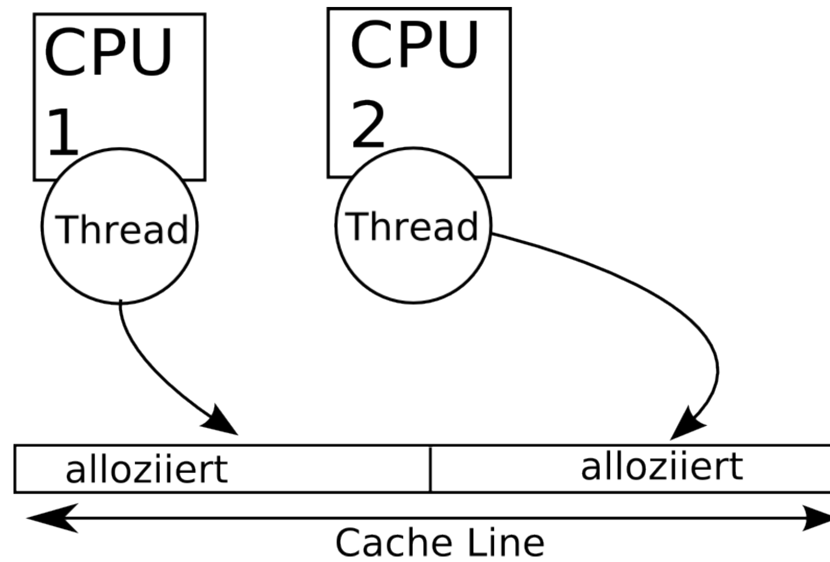
- Vielzahl weiterer (Debug-)Optionen, siehe "man malloc"

FreeBSD 7 (jemalloc)

- malloc-Implementierung von Jason Evans
- löst phkmalloc in FreeBSD 7 ab
- Hauptgründe:
  - Skalierung in Multithreading-Umgebungen
  - RAM ist billiger & nicht mehr so knapp
- Malloc-Optionen sind kompatibel zu phkmalloc

Threading-Probleme auf SMP-Systemen

- Lock Contention: (phkmalloc: nur ein malloc/free gleichzeitig!)



- false Cachline-Sharing

Arena-Mechanismus (z.B. ptmalloc, jemalloc)

"per-Thread-Heap": Heap wird in unabhängige Einheiten (Arena) geteilt
jede Arena kann unabhängig von anderen Arenen mit Locks versehen werden

- ptmalloc:
  - Thread versucht zuerst exklusiven Zugriff auf die zuletzt durch diesen Thread verwendete Arena zu erhalten
    $\rightarrow$ Thread blockiert (normalerweise) nicht
  - Falls keine Arena verfügbar: Neue Arena alloziieren (mit `mmap()`)

- jemalloc:
  - Anzahl Arenen: Anzahl CPUs $*4$ (TLS), im nicht-TLS Fall wird nächsthöhere Primzahl verwendet
  - Auswahl der Arena via Round-Robin oder TID Hashing (`_pthread_self() % narenas;`)

jemalloc: Datenstrukturen

- Speicher wird in Chunks (Voreinstellung 2 MB) verwaltet
- Arenen teilen diese ggf. auf (binärer Buddy-Algorithmus)
- Lookup freier Blöcke mittels Bitmaps
- Verwaltung von Chunks in RB-Tree
- verwendet `mmap`, auf einigen Plattformen (arm, i386, ppc) zusätzlich `brk`

Vorgehen bei malloc/free

(FreeBSD 7, malloc.c,v 1.146 i, Zeile 2160 ff.)

- bis 1M: je nach angeforderter Größe werden Tiny/Small/Large etc. Bins einer Arena abgesucht
  - Small
    - Tiny (2,4,8 Byte)
    - Quantum-Spaced (16 . . . 512 Byte)
    - Sub-Page (1k, 2k)
  - Large (4,8,16 . . . 1024 kb)
- "Huge" ($>$ 1 MB): besteht aus einem oder mehreren Chunks

Und wer noch nicht genug hat . . .

- Es existiert bis heute kein 'perfekter' Allokator
- verschiedene gängige Implementationen haben alle verschiedenste Ziele/Vor- und Nachteile
  - Geschwindigkeit, Speicherverschnitt, Portabilität, etc.
- Grosse Anzahl an Allokatoren, v.a. für Threaded-Anwendungen
- z.B: Hoard, tcmalloc, . . .

Garbage Collection ("Automatische Speicherbereinigung")

Der aktive Speicher erkennt und beseitigt automatisch WinXX-Codefragmente, bevor sie Schaden anrichten können.
(Holger Veit, FdI #329)

- J. McCarty, LISP, ca. 1959
  Heutige Verwendung: Java, C#, (fast) alle Scriptsprachen
- GC-Bibliotheken für z.B. C/C++ verfügbar

GC hat zwei Aufgaben:

1. Erkennen, wann Objekte nicht mehr erreichbar sind
2. Von solchen Objekten belegten Speicher wieder zur Verfügung stellen
   (hängt maßgeblich vom "Garbage"-Detektionsverfahren ab)

GC-Verfahren

- Reference-Counting
- Mark & Sweep
- Mark & Compact
- Copy-Kollektoren

Reference Counting

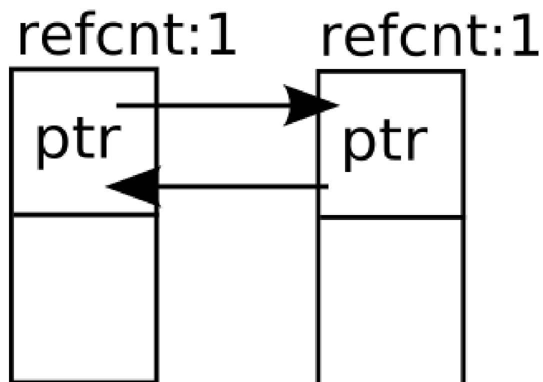
- Der Allokator verwaltet einen Referenz-Zähler pro Objekt

- Compiler bzw. Interpreter verwaltet diesen
  - Zähler inkrementieren, wenn ein Zeiger auf das Objekt kopiert wird
  - Wenn Referenz auf ein Objekt entfernt wird: Zähler dekrementieren

- Allokator kontrolliert Zählstand: Wenn 0 wird Objekt nicht (mehr) benötigt

Reference Counting

2 Probleme: Effizienz und Effektivität [WIL92]:



Wird auch dann nicht freigegeben wenn keines der beiden Objekte mehr erreichbar ist

Wenig Effektiv: Aufwand ist proportional zu durchgeführten Aktionen (z.B. Zuweisungen)

GC-Verfahren: Mark & Sweep

- Durchläuft alle Referenzen
  - Aufwand hängt von Größe des Heaps bzw. Anzahl Referenzen ab
  - Ermittelt alle (noch) erreichbaren Objekte
  - . . . damit stehen auch alle nicht (mehr) erreichbaren Objekte fest

- Entspricht der Erreichbarkeit in Graphen → Breiten- / Tiefensuche

Ein wesentliches Problem – die Fragmentation des Speichers – wird aber nicht gelöst

GC-Verfahren: Mark & Compact

- Mark & Compact sehr ähnlich

- nach der Mark-Phase werden alle noch "lebenden" Objekte in einen zusammenhängenden Block kopiert
  $\rightarrow$ damit ist auch der freie Speicher zusammenhängend.

- Diese *Compact-Phase* hat Nachteile:

  – Kopieraufwand
  – Aktualisierung aller Referenzen
  – Je nach Implementierung muss Heap mehrfach durchsucht werden (Berechnen der nötigen Größen und Zieladressen, Aktualisierung der Zeiger, etc.)

- . . . eigentlich gar kein Garbage-, sondern ein "Lebend-Kollektor" ;)

Kopierende Kollektoren

- Ähnlich Mark & Sweep, Mark Phase entfällt jedoch
- stattdessen werden alle "gefundenen" Objekte direkt in einen neuen Speicherbereich kopiert, der Restspeicher wird dann als frei markiert
- Einfache Variante ('stop & copy'): Teilt Speicher in 'to' und 'from'-Hälften
- Aufwand ist proportional zur Anzahl noch referenzierten Objekte
- i.A. reduziert sich der Gesamtaufwand bei einer geringeren GC-Frequenz (bei höherem Speicherverbrauch)

Java5 Hotspot VM

- Implementiert verschiedene Garbage-Kollektoren

- 'Generational Garbage Collection' (*young*, *tenured*, *permanent*)
  Beobachtung: Objekte haben überwiegend nur eine sehr kurze Lebenszeit
  $\rightarrow$ bei einem GC-Lauf sind die meisten Objekte nicht mehr referenziert

- *Young Generation*: Kopierender Kollektor
  - 3 Segmente: *Tospace*, *Fromspace*, *Eden*; *Eden ist größer*
  - Objekte werden aus Eden alloziiert
  - wenn kein Speicher mehr frei ist werden noch lebende Objekte (aus *Eden* und *Fromspace*) in den *Tospace* kopiert, dieser wird neuer *Fromspace*
  - Falls *Tospace* nicht alle Objekte aufnehmen kann, wird der Heap der "Tenured"-Generation verwendet

Java5 Hotspot VM: Gabage-Kollektoren (Auswahl)

Auswahl erfolgt automatisch durch die VM oder über Argumente

- Serial-Kollektor, Anwendung wird unterbrochen ("*stop-the-world*")
- Throughput-Kollektor
  - Minor-Collection läuft mit mehreren Threads ab
  - Tenured-Generation wie bei Serial
  - Richtwert für maximale GC-Unterbrechungsdauer
- concurrent low pause
  - Mark&Sweep Kollektor für *Tenured*-Generation
  - Garbage-Collection wird kurzzeitig unterbrochen → Längere Gesamtdauer für GC-Durchlauf, aber bessere Interaktivität der Anwendung

Boehm GC

Boehm-Demers-Weiser-Kollektor

- *Konservativer* GC für C / C++
- Mark & Sweep - Algorithmus
- eigener Allokator
- Wird u.a. von `gcj` (GCC Java) verwendet

```
#include "gc.h"
void * GC_malloc(size_t size);
void * GC_realloc(void *ptr, size_t size);
cc ... gc.a
```

Boehm GC: Allokator

- Blöcke fester Größe (einstellbar zur Compile-Zeit, 512 Byte bis 16K)
- Kleine Chunks aus partitionierten Blöcken
- getrennte Freelists für Objekte bis einer bestimmten Größe (Plattformabhängig, ca. 1 Kb)
- Falls kein Speicher verfügbar: Heap vergrößern oder GC-Zyklus
- Allokator vergibt verschiedene Objekttypen, die beiden wichtigsten:
  - `NORMAL`
  - `PTRFREE`

Boehm GC: GC-Zyklus

1. Mark-Bit eines jeden Objekts löschen (damit gelten alle Objekte als "nicht erreichbar")

2. Mark-Phase

3. Sweep-Phase: Unmarkierte Objekte in Freelists eintragen

4. Finalization-Phase

Boehm GC: Mark-Phase

- keine (zusätzlichen) Informationen über die Position von Zeigern; Absuchen von *root*-Segmenten (Register, Stack, .data/.bss)
- Die Adressen/Größen der Segmente wird abhängig von der Plattform ermittelt (unter Linux: Parsen von `/proc/self/{maps,stat}`, etc.)
- Marker verwaltet eigenen Stack mit Informationen über Speicherregionen, die Erreichbar sind aber noch nicht durchsucht wurden
- Zu Beginn der Phase werden dort alle Rootsegmente abgelegt $\rightarrow$ folgt allen Bitmustern die Adressen auf vom Kollektor verwaltete Heap-Bereiche sein *könnten*
- So gefundene Bereiche werden ebenfalls durchsucht

Boehm GC: Sweep-Phase

- Alle Block-Header werden durchsucht
- Unmarkierte Objekte werden in Free-List einsortiert
- Im Fall von partitionierten Seiten wird kontrolliert, ob alle Partitionen frei sind
  - wenn ja, wird d. ganze Seite in Freelist eintragen
  - sonst wird gewartet, bis ein `GC_malloc`-Aufruf auf eine leere Freeliste für diesen Typ stöst

Blacklisting von Adressen

- Da nach möglichen Adressen gesucht wird, entstehen mehrere Probleme:
  - Es muss u.u. eine große Datenmenge durchsucht werden
  - Datenblöcke, welche Bitmuster enthalten die wie eine gültige Adresse aussehen, führen dazu, dass der referenzierte Speicher nicht freigegeben werden kann

- Lösung:
  - Wenn während eines GC-Laufs *ungültige* Adressen gefunden werden, die in absehbarer Zeit (durch neue Allokationen) gültig werden *könnten* wird die zugehörige Seite in einer Blacklist verzeichnet
  - Aus solchen Seiten werden höchstens sehr keine Objekte alloziiert

Fazit

- Hohe Anzahl an verfügbaren Allokatoren
- Es gibt keine gängige Methode, Allokatoren zu beurteilen
- Durch neue HW entstehen neue Anforderungen/Probleme
- GC ja/nein hängt von den Anforderungen (Speicherbedarf, Laufzeitverhalten, . . .) ab

Lektüre

K&R Brian W. Kernighan, Dennis M. Ritchie." The C Programming Language", ISBN 978-0131103306

PHK98 Poul-Henning Kamp. "malloc(3) revisited." `http://phk.freebsd.dk/pubs/malloc.pdf`

WIL92 Paul R. Wilson. "Uniprocessor Garbage Collection Techniques" `ftp://ftp.cs.utexas.edu/pub/garbage/gcsurvey.ps`

JE06 Jason Evans. "A Scalable Concurrent malloc(3) Implemenation for FreeBSD" `people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf`

- Doug Lea. "dlmalloc" `ftp://gee.cs.oswego.edu/pub/misc/malloc.c`

- Wolfram Gloger. "ptmalloc". http://malloc.de
- Boehm-Demers-Weiser GC für C/C++
  http://www.hpl.hp.com/personal/Hans_Boehm/gc/
- H. Boehm, A. Demers, S. Shenker. "Mostly Parallel Garbage Collection"
  http://www.hpl.hp.com/personal/Hans_Boehm/gc/papers/pldi91.ps.Z
- H. Boehm. "Space Efficient Conservative Garbage Collection"
  http://www.hpl.hp.com/personal/Hans_Boehm/gc/papers/pldi93.ps.Z
- David Chase, GC-FAQ
  http://www.iecc.com/gclist/GC-faq.html
- "Tuning Garbage Collection with the 5.0 Java™ Virtual Machine"
  http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html