8. Speicherverwaltung

Michael Schöttner
Betriebssysteme und Systemprogrammierung

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

8.0 Vorschau

- Grundlagen & Terminologie
- Anforderungen an die Speicherverwaltung.
- Partitionierung des Arbeitsspeichers
- Speicherverwaltung und Zuteilung.
- Automatische Freispeichersammlung

8.1 Grundlagen & Terminologie

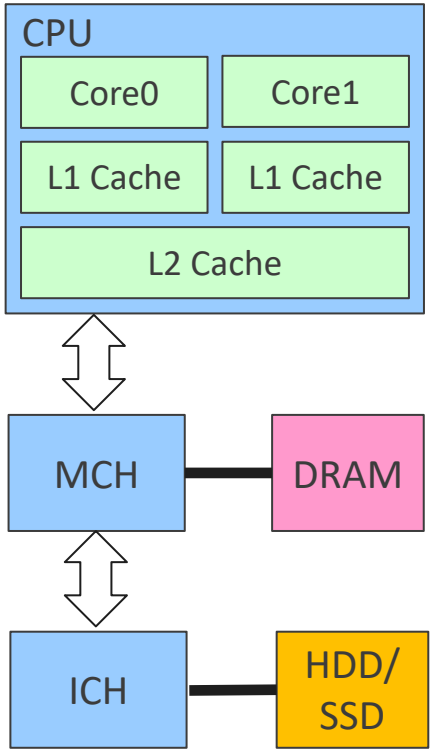
- Es gibt bisher keine universelle Speicherart, welche schnell und persistent ist
- Unterschiedliche Speichertechniken haben jeweils eigene Vor- und Nachteile
- Deswegen kommen in einem Computer i.d.R. verschiedene Speicherarten zum Einsatz, wodurch sich eine Speicherhierarchie ergibt

- Die Speicherverwaltung im Betriebssystem organisiert den Transfer zwischen den Ebenen der Speicherhierarchie.

- Zukünftig kann NVRAM dies revolutionieren.
  - NVRAM = Non-Volatile Random Access Memory
  - Derzeit sind diese Speicherbausteine jedoch noch nicht so schnell wie DRAM

Speicherhierarchie

- Caches
  - Sehr schneller wahlfreier Zugriff <1ns
  - Flüchtiger Inhalt, geringe Kapazitäten in Kilobyte, meist mehrstufig (Level 1 - 4)
- DRAM: Arbeits- oder Hauptspeicher
  - Schneller wahlfreier Zugriff, ~10ns
  - Flüchtiger Inhalt, Module bis je 256 GB
- SSD: Sekundärspeicher
  - Vergleichsweise langsamer Zugriff, Festplatte ~7-10ms, SSD ~30 – 500µs
  - Sequentieller Zugriff schneller, als wahlfreier
  - Große Kapazitäten, bis 16 TB



MCH = Memory Controller Hub
ICH = I/O Controller Hub

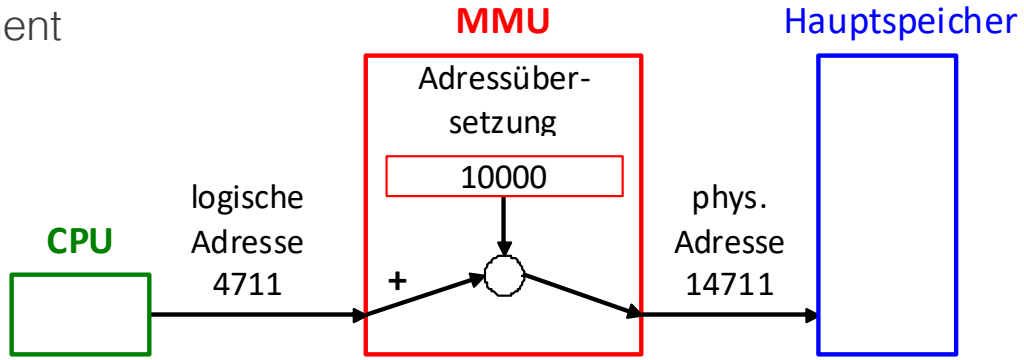
Begriffe

- **Speicherblock**: Menge von fortlaufenden logischen Speicheradressen.
- **Partition** = (größerer) Gesamtspeicherblock für ein Programm.
- **Swapping** = Aus- und Wiedereinlagern von ganzen Partitionen/Programmen auf den Sekundärspeicher.

- **Physikalische** (absolute) **Speicheradresse**: bezeichnet/zeigt in physisch vorhandenen Arbeitsspeicher.
- **Logische Speicheradresse**: Position im Arbeitsspeicher aus Sicht des Programms, unabhängig von der physikalischen Speicherorganisation.
- **Relative Speicheradresse**: Position relativ zu einem bekannten Punkt im Programm, i.d.R. Instruction-Pointer (Register EIP/RIP)

Binden von Speicheradressen

- Adressbindung zur **Übersetzungszeit:**
  - Durch den Compiler; für statische Bibliotheken
- Adressbindung zur **Ladezeit:**
  - Durch den Lader; für dynamische Bibliotheken
- Adressbindung zur **Laufzeit:**
  - erlaubt die Relozierung / Verschiebung des Programms zur Laufzeit,
  - benötigt Memory Management Unit (MMU) im Prozessor,
  - logische versus physikalische Adressen.



8.2 Anforderungen an die Speicherverwaltung

- Ausgangssituation: Mehrprogrammbetrieb
  - mehrere Programme/Prozesse teilen sich den Arbeitsspeicher,
  - vorab ist unklar, welche und wie viele Programme zu einem Zeitpunkt geladen sind.
- Zuteilung von Speicherblöcken:
  - schnell, und mit möglichst geringem Verschnitt.
- Freigabe von Speicherblöcken:
  - Aufräumen beim Terminieren eines Programms,
  - manuelles oder automatisches Einsammeln.

8.2 Anforderungen an die Speicherverwaltung

- Aus- und Einlagern von Programmen:
  - Ziel: bessere Ausnutzung des Arbeitsspeichers und CPU durch Auslagern von inaktiven Programmen
- Relozierung:
  - Wiedereinlagern eines Programms kann an anderer Adresse erfolgen, somit muss Programm relozierbar sein
  - Damit nicht alle Zeiger einzeln angepasst werden müssen, wird hierzu Hardware-Unterstützung benötigt (siehe virtuelle Speicherverw.),
  - Relozierung ist auch für die Kompaktifizierung wichtig.

8.3 Partitionen im Arbeitsspeicher

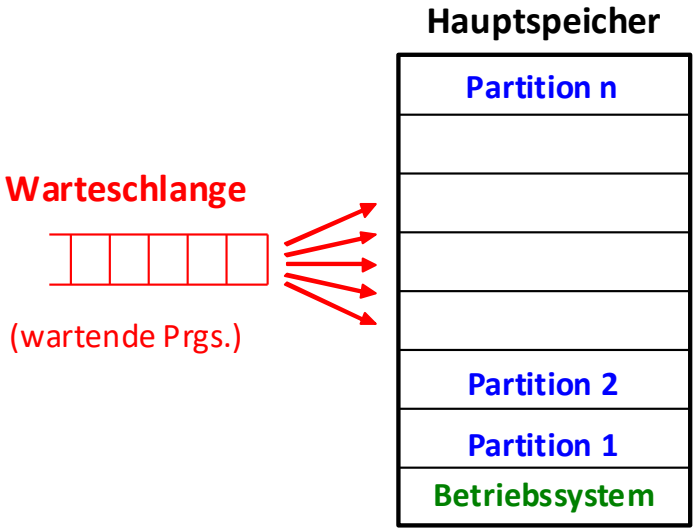
- Ziel: mehrere Programme gleichzeitig im Arbeitsspeicher ausführen
  → Multiprogramming (Vorläufer vom Multitasking)
- Ältere Betriebssysteme kannten keinen virtuellen Speicher, sondern verwendeten eine Aufteilung des Arbeitsspeichers in **Partitionen**.
- Je nach Betriebssystem ist die Partitionierung mit und ohne Auslagern (engl. swapping) realisiert.

8.3.1 Statische Partitionierung

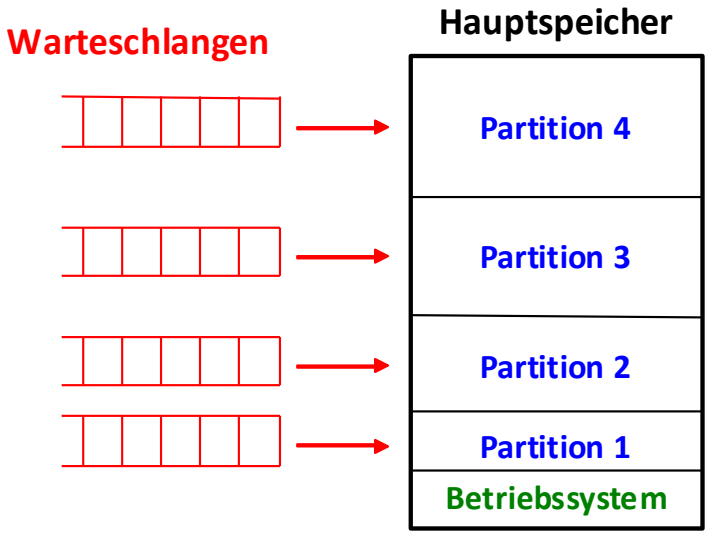
- Statische Unterteilung des Arbeitsspeichers in gleich große oder variabel große Partitionen.
- Partitionierung ist während Laufzeit nicht mehr änderbar.
- Jedes Programm erhält eine eigene Partition.
- Programm erhält kleinste Partition in das es hineinpasst.
- Sind alle Partitionen belegt, so warten die Programme in einer Zuteilungswarteschlange.

8.3.1 Statische Partitionierung

- Partitionen fester Größe mit einer Warteschlangen:



Variable Partitionsgrößen und mehreren Warteschlangen:



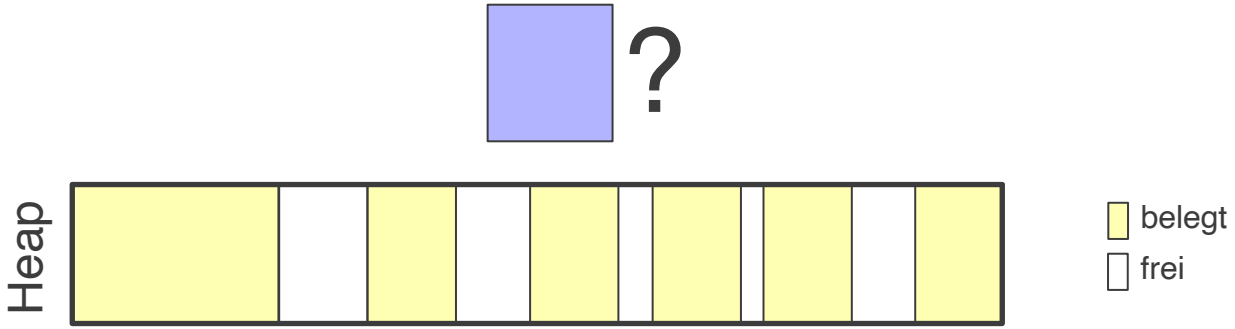
Bewertung

- Einfach implementierbar
- Aber die maximale Anzahl der Programme ist statisch festgelegt und evt. passt ein Programm in keine Partition.
- Speicherbedarf eines Programms muss vorab bekannt sein.

- Ungenutzter Speicherplatz in einer Partition geht verloren
  → interne Fragmentierung/Speicherverschnitt

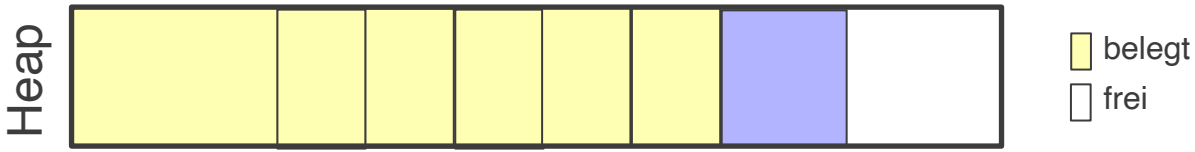
8.3.2 Dynamische Partitionierung

- Programm erhält genau so viel Speicher wie es benötigt und nicht mehr.
  → Länge, Anzahl & Anfangsadresse der Partitionen ändern sich dynamisch.
- Interne Fragmentierung innerhalb einer Partition wird verhindert,
  aber ein **neues Problem entsteht: externe Fragmentierung**:
  - Im Laufe der Zeit entstehen Löcher zwischen den Partitionen.
  - Ein neues Programm kann eventuell nicht geladen werden,
    obwohl genügend Speicher vorhanden ist, aber nicht am Stück



8.3.2 Dynamische Partitionierung

- Externe Fragmentierung kann durch eine Neuanordung der Partitionen behoben werden
  - Dazu müssen die Partitionen relozierbar sein
  - Dies ist aufwändig, da u.U. viele Partitionen verschoben werden müssen



- Bem.: Das Problem der externen Fragmentierung löst sich im nächsten Kapitel durch die virtuelle Speicherverwaltung auf.

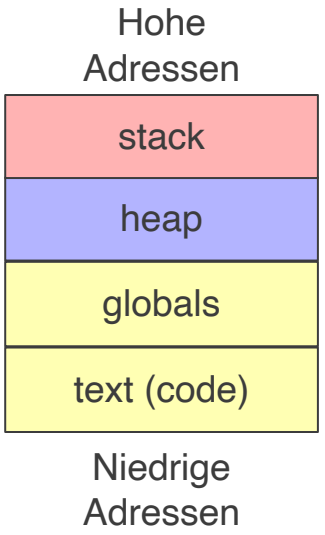
Beispiel: OS/360 von IBM

- Diese Varianten der Partitionen waren in OS/360 implementiert
  - Betriebssystem von IBM, 1964.
  - Stapelsystem für Mainframes.

- Partitionierung mit drei Varianten:
  - PCD = Primary Control Program: Einprogrammbetrieb.
  - MFT = Multiprogramming with a Fixed number of Tasks.
  - MVT = Multiprogramming with a Variable number of Tasks.

8.3.3 Struktur einer Partition (Wdlg.)

- Jedes Programm erhält eine Partition
- Bestandteile einer Partition (abstrakt):
  - Stack: für Funktionsaufrufe (Parameter, lokale Variablen)
  - Heap: dynamische allozierte Daten („malloc“ und „free“);
  - Globals: globale Variablen
  - Text: Instruktionen des Programms



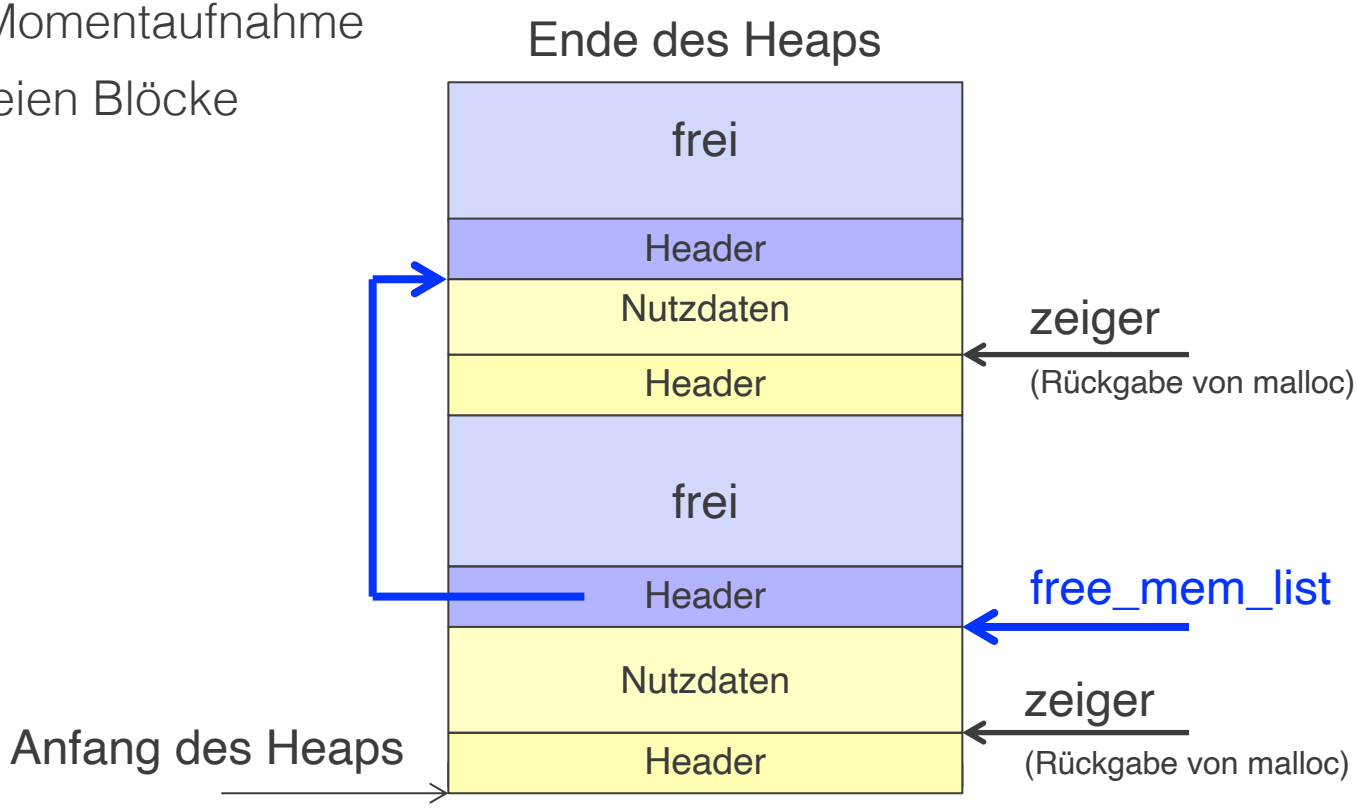
- Stack siehe Kapitel 3 und 4.

Format von Heap-Blöcken

- Ein Aufruf von **`malloc`** oder **`new`** (C++ & Java) liefert einen neuen Heapblock
- Header: enthält Informationen für Speicherverwaltung (~ 4-8 Byte)
- Header außerhalb des Nutzdatenblocks
- Header-Inhalt:
  - Längenfeld, nächster Heapblock, Anzahl Elemente bei Arrays, ...
  - Flags: Free, Used, Locked, Marked, …
  - Typ-Zeiger auf den Klassendeskriptor bei Instanzen in objekt-orientierten Sprachen

Beispiel: Heap-Belegung

- Beispiel einer Momentaufnahme
- Hier sind die freien Blöcke verkettet



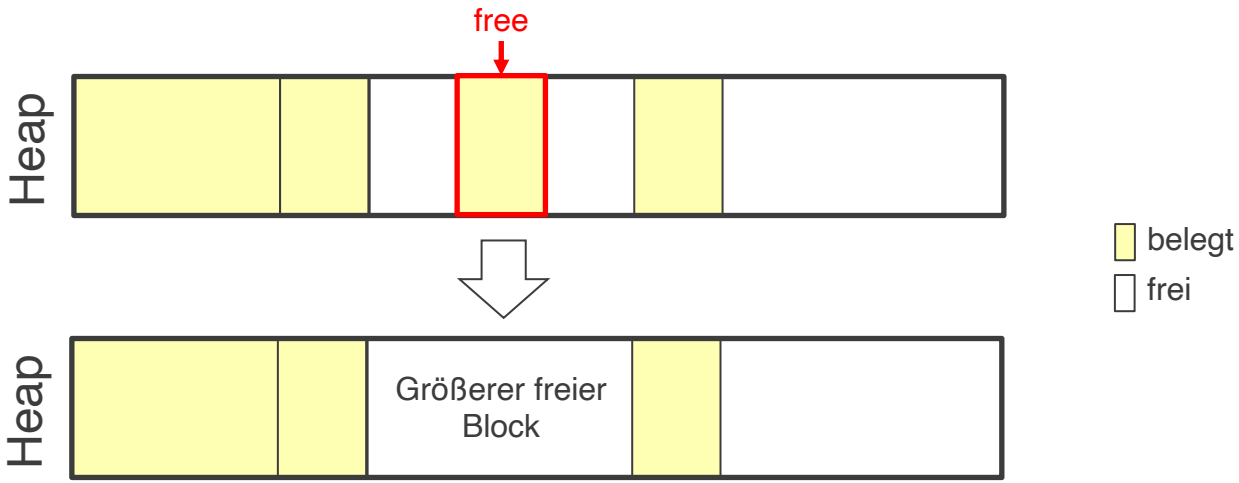
8.4 Speicherverwaltung

Aspekte

- Wiedereingliederung von unbenutzten Blöcken.
- Verwaltung des freien Speichers
- Granularität der Speicherblöcke.
- Verschnitt (interne und externe Fragmentierung).
- Auswahlstrategie für freie Stücke.

8.4.1 Freigabe und Wiedereingliederung

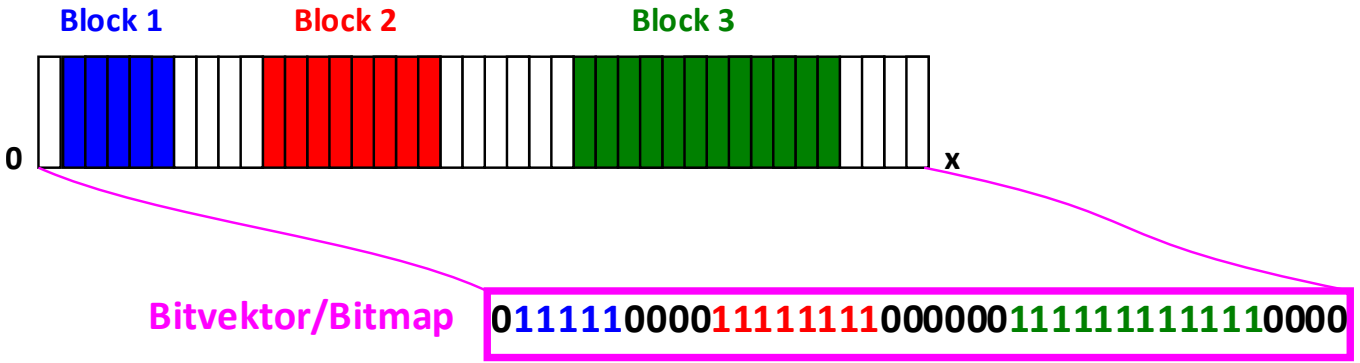
- Bei Freigabe eines Speicherblocks prüfen, ob Nachblöcke frei sind und gegebenenfalls zusammenfassen.
- Hiermit entstehen wieder größere freie Speicherblöcke.



8.4.2 Verwaltung des freien Speichers

Bitvektor/Bitmap

- Speicher unterteilen in Einheiten fester Länge (z.B. 512 B oder 4 KB).
  - Jeder Einheit wird ein Bit in einem Bitvektor (Bitmap) zugeordnet.
  - Je kleiner die Einheit, desto größer ist der resultierende Bitvektor.
  - Je größer die Einheit, desto mehr interne Fragmentierung tritt auf.



8.4.2 Verwaltung des freien Speichers

Bitvektor/Bitmap

- Kompakte Datenstruktur:
  - Beispiel: 128 MB in 512 Byte Blöcke unterteilt ergibt 32 KB Bitvektor.

- Aber die Allokation eines größeren Speicherblocks ist u.U. langsam, da der Bitvektors nach Nullbit-Folgen durchsucht werden muss.

Freispeichertabelle

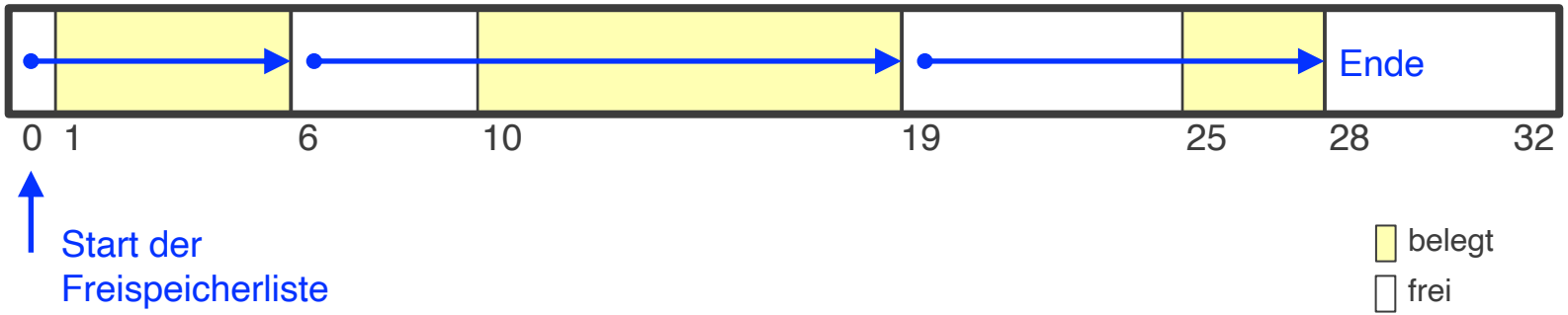
- Freie Speicherblöcke werden in einer Tabelle verwaltet.
- Zum Beispiel sortiert nach der Größe.
- Speicher muss nicht in Einheiten fester Länge unterteilt werden

Größe	Adresse
1	0
4	6
5	19
6	28

belegt
frei

Freispeicherliste

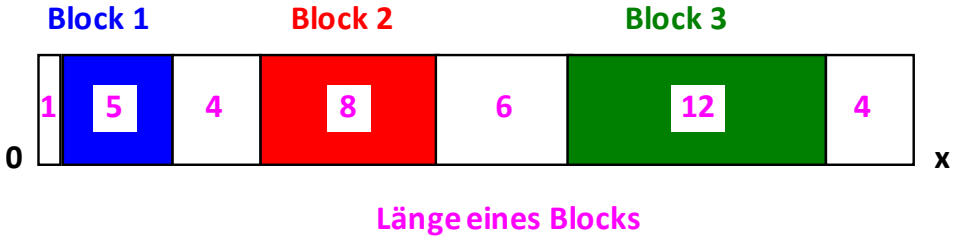
- Freie Heap-Blöcke mit Zeiger verketten
- Benötigt keinen zusätzlichen Speicher, da freier Speicher genutzt wird



- Eventuell mehrere Listen, z.B. um verschiedene Größenordnungen separat zu verketten → Suche wird beschleunigt

Linearer Heap

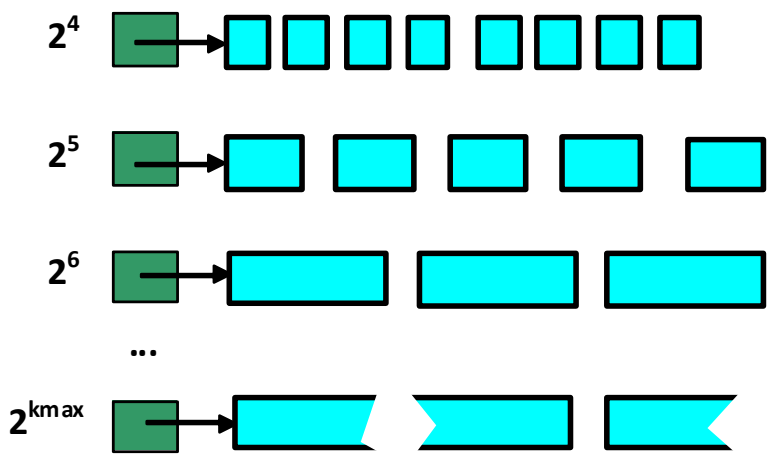
- Freie & belegte Blöcke sind bündig aneinander gereiht.
- „Verkettung“ der Blöcke erfolgt über das Längenfeld.
- Freie Blöcke sind durch ein Bit im Header gekennzeichnet.



- Bewertung:
  - Vorteil: zusätzliche Zeiger entfallen.
  - Nachteil: evt. muss Heap linear nach passendem Block durchsucht werden. (Abmilderung durch mehrere Einstiegspunkte)

Buddy-System

- Zwei gleichgroße benachbarte Blöcke nennt man Buddys („Kumpels“)
- Speicher besteht idealerweise aus 2^{kmax} Einheiten
- Speichervergabe in Blockgrößen von 2^k
- Jeweils Liste für Blöcke der Größe 2^k



- Bem.: verwendet in Linux-Kern für die physikalische Speicherverwaltung

Buddy-System

Ablauf einer Allokation

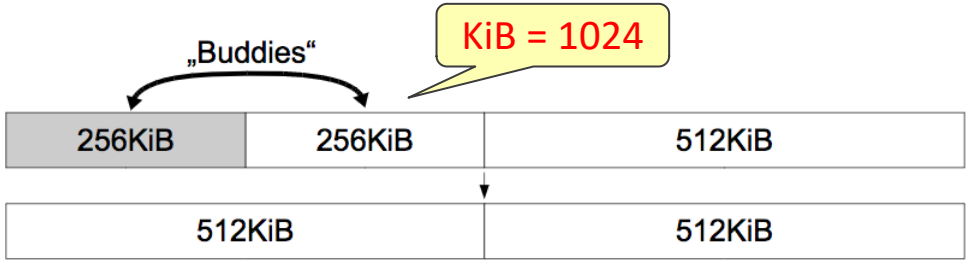
- Aufrunden auf die nächste Zweierpotenz 2^i
- Zugriff auf erstes freies Stück der Liste 2^i
- Falls Liste 2^i leer (rekursiv):
  - Zugriff auf die Liste der nächsten Größe 2^{i+1}
  - Stück entfernen und halbieren
  - Vordere Hälfte zuteilen, die Hintere (=Buddy) in zugehörige Liste 2^i einhängen

- Kleinere Stücke entstehen aus (rekursiver) Halbierung größerer Stücke

Buddy-System

Ablauf einer Freigabe

- 1. Buddy bestimmen

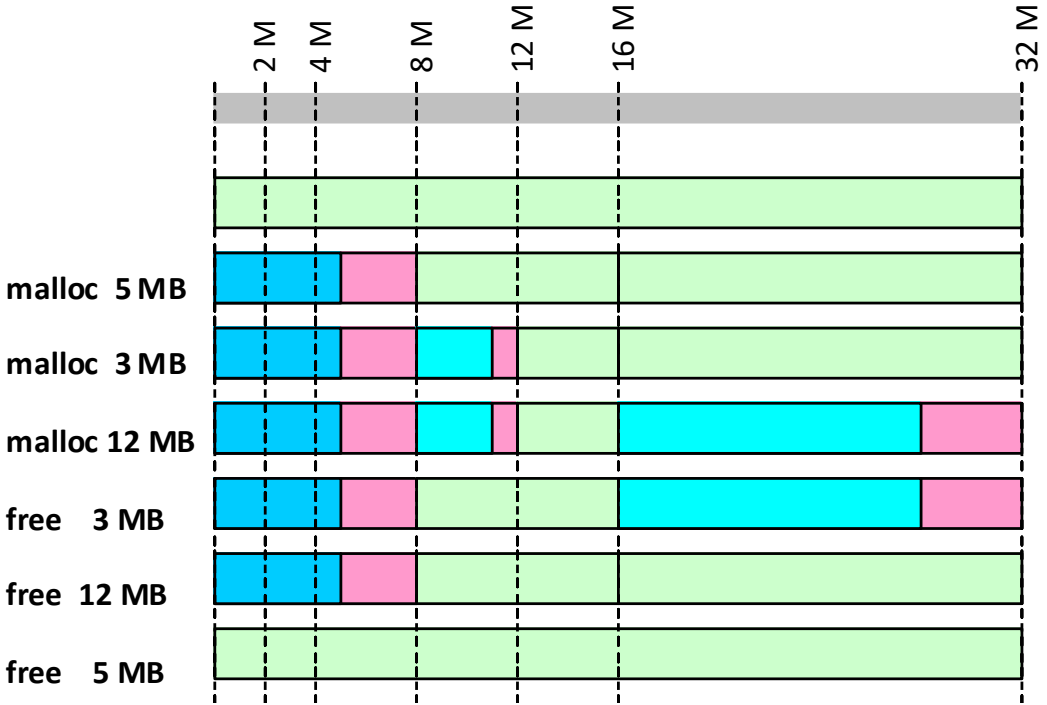


- 2. Falls Buddy belegt, freigewordenes Stück in die zugehörige Liste einhängen, stoppen und zum Aufrufer zurückkehren

- 3. Falls Buddy frei → Vereinigung
  - Dadurch wird das freie Stück doppelt so groß
  - Gehe zu 1.

Buddy-System

- Beispiel: Nutzdaten und interner Verschnitt



Buddy-System

Bewertung

- Vorteil: schnelles Verschmelzen freiwerdender Blöcke möglich

- Nachteil: sowohl interne als auch externe Fragmentierung vorhanden

8.4.3 Auswahlstrategien

- Wie wird ein passender freier Speicherblock ausgewählt?
- Kriterien: Fragmentierung und Geschwindigkeit
- Informal: „Gute Strategie kommt mit einem kleinem Heap aus.“

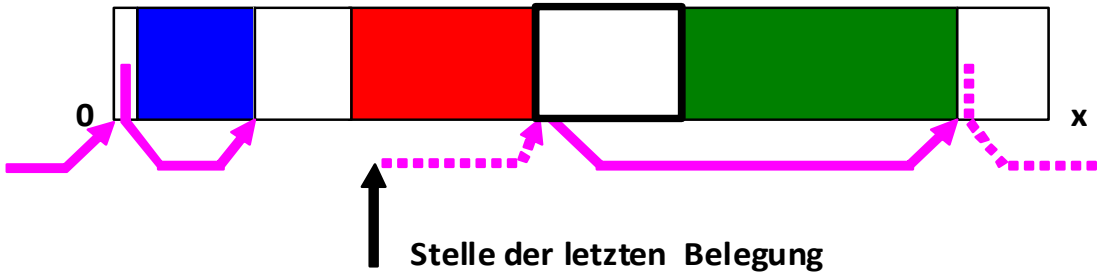
First-Fit

- Durchsucht die Liste der freien Speicherblöcke ausgehend vom Anfang und nimmt den ersten freien Block der groß genug ist.
- Zu großen Block eventuell teilen, um unbenötigten Platz zu sparen:
  - ohne Teilen → interne Fragmentierung
  - mit Teilen → externe Fragmentierung

- Vorteil: sehr schnelle Speicherzuteilung.
- Nachteil: Konzentration belegter Stücke am Anfang.

Next-Fit

- Freispeicherliste wird zyklisch durchlaufen.
- Suche beginnt dort, wo letzte Belegung stattgefunden hat.
- Eigenschaften wie bei „First Fit“, vermeidet aber die Konzentration von belegten Blöcken am Anfang.



Best-Fit

- Sucht den Block, der am wenigsten Speicherverschnitt verursacht.
- Freispeicherliste muss ein Mal komplett durchlaufen werden.
- Verbesserung: verwende nach Größe sortierte Freispeicherliste

- Vorteil: Zerschneidung großer Stücke i.d.R. unnötig.
- Nachteil: langsam; neigt bei Zerschneiden dazu sehr kleine unbrauchbare Stücke zu erzeugen.

Worst-Fit

- Nimmt größten freien Block, damit nach dem Zerschneiden noch brauchbare Stücke übrig bleiben.

Bemerkungen

- Die beste Zuteilungsstrategie zu finden, ist auch nach dem Programmende ein schwer lösbares Problem.
- Speicheranfragen werden i.d.R. aufgerundet:
  - auf 32/64 Bit Grenzen (aus Geschwindigkeitsgründen)
  - oder falls verbleibende Reststücke zu klein sind:
    - evt. zu klein für sinnvolle Nutzung
    - Verwaltungsaufwand vermeiden
- Aber für manche Anwendungen zählt jedes Byte
  - Sehr große Graphen, z.B. soziale Netzwerke
  - Hier müssen Billionen von sehr kleinen Speicherblöcken, meist <64 Byte, verwaltet werden

8.5 Automatische Freispeichersammlung

- Explizite Rückgabe durch Programmierer ist fehleranfällig & mühsam:
  - Abbau komplexer Strukturen oft schwierig
  - Wird vergessen Speicher freizugeben,
    so entstehen Speicherlecks (engl. **memory leaks**)
  - Wird ein Speicherblock zu früh freigegeben,
    so entstehen ungültige Zeiger (engl. **dangling pointers**)

- Lösung: automatische Freispeichersammlung (engl. garbage collection)
  - Nicht mehr adressierbare Blöcke automatisch identifizieren und freigeben
  - Entweder für ein einzelnes Programm oder systemweit
  - Beispiele: Java, .NET, ...

8.5.1 Grundprinzip der Freispeichersammlung

- GC-Phasen:
  - 1. Phase: Garbage Detection:
    Erkennung von referenzierten und nicht mehr referenzierten Blöcken
  - 2. Phase: Garbage Reclamation
    Freigabe des Speichers von nicht mehr referenzierbaren Blöcken

- Garbage: nicht mehr referenzierbare Blöcke zum Zeitpunkt des GC-Aufrufs
- Collector: sammelt Garbage
- Mutator: alle Programme, welche den Heap ändern (mutieren)

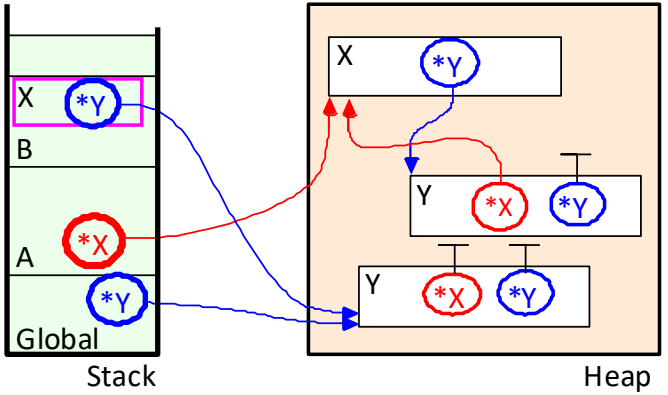
8.5.2 Voraussetzungen

- Referenzen müssen identifizierbar sein
- Typsichere Sprache ist notwendig
  - C ist keine typsichere Sprache.
  - Deswegen gibt es für C keine Garbage Collection
- Der GC muss den Aufbau eines Speicherblocks und der Stackframes kennen
  - Wo sind Zeiger?
  - Wo sind andere Variablen (nicht-Zeiger)?
  - Diese Informationen speichert der Compiler in der sogenannten Symboltabelle
    - Diese wird auch von Debuggern verwendet

Prinzip einer Symboltabelle

- Offsets von Variablen in Instanzen, Structs und auch in den Stackframes (Parameter und lokale Variablen)

- Bemerkungen:
  - Java: in .class-Dateien enthalten
  - C/C++: Debug-Versionen



Symboltabelle

Globale Var.	------*Y--
Mth.-A Frame	--*X------
Mth.-B Frame	-*Y-------
Instanz-X	----*Y----
Instanz-Y	-*X----*Y--
...	

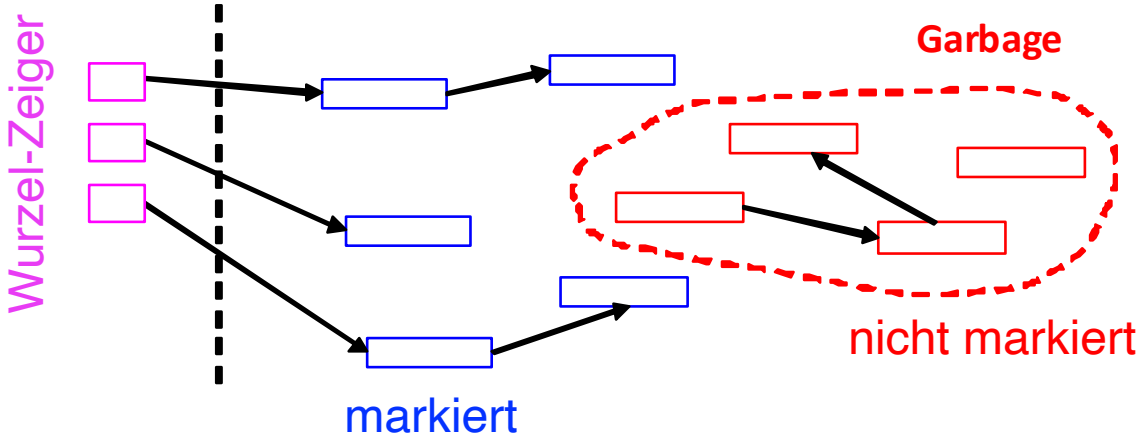
Garbage

- = nicht mehr referenzierbare Blöcke zum Zeitpunkt des GC-Aufrufs
- Problem: wo soll die GC anfangen?
- Lösung: bei den **Wurzel-Zeigern (engl. root set)**
  - Menge aller gültigen Zeigervariablen
  - Zeiger in globalen Variablen (Klassenvariablen in Java)
  - Alle Zeiger im Stack (alle Stackframes betrachten)
  - Auch Zeiger in Registern des Prozessors

- **Garbage: Es existiert kein Pfad zwischen dem betrachteten Speicherblock und einem Wurzelzeiger.**

8.5.3 Mark & Sweep Algorithmus

- Algorithmus markiert alle noch erreichbaren Blöcke im Heap
- Ausgehend von den Wurzelzeigern werden transitiv alle erreichbaren Blöcke besucht und markiert.
- Nicht markierte Blöcke sind Garbage und können freigegeben werden



8.5.3 Mark & Sweep Algorithmus

- Algorithmus:
  - Mark

```
Für jeden Wurzelzeiger z:
     Markiere(z);

Markiere(block):
     wenn block.mark = 1 dann beende Prozedur
     block.mark := 1;
     für jeden von block referenzierten Block b:
       Markiere(b)
```

  - Sweep

```
Für jeden Block b, für den gilt b.mark = 0;
     Speicherfreigabe(b)
```

- Markierungsphase muss in einem Stück zu Ende laufen.
  - U.U. würde sonst ein Zeiger in einem bereits abgearbeiteten Block verändert,
  - Evt. würden dann noch benutzte/neue Blöcke fälschlicherweise eingesammelt.

8.5.3 Mark & Sweep Algorithmus

- Vorteile:
  - Zyklen werden erkannt.
  - Einfach zu implementieren.

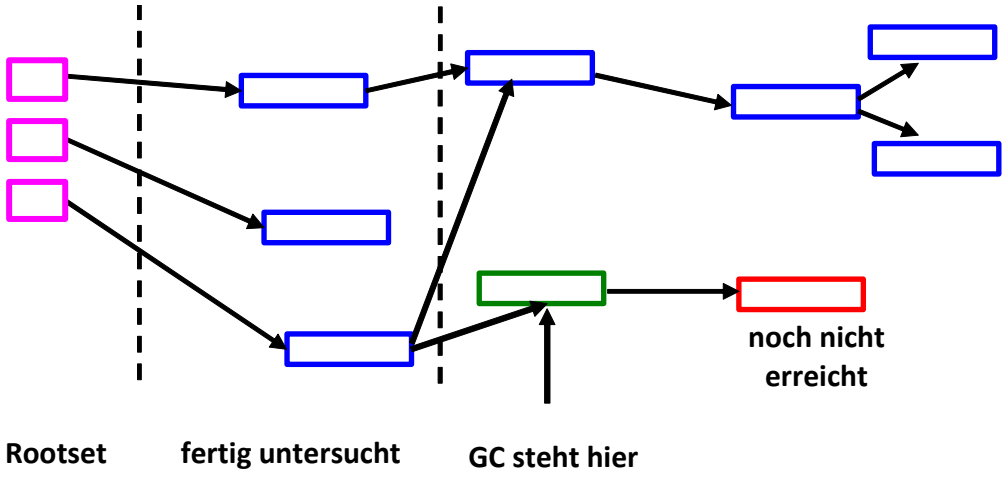
- Nachteile:
  - Funktion zum Markieren beinhaltet unter Umständen tiefe Rekursion → evt. viel Speicherplatz im Keller notwendig.
  - Heap wird durch GC nicht kompaktiert.

8.5.4 Inkrementeller Mark & Sweep Algorithmus

- Nebenläufiges Mark & Sweep nach einer Idee von E. W. Dijkstra, 1978.
  - Blöcke werden mit drei Farben markiert:
    - blau: Block wurde komplett untersucht
    - rot: Block wurde noch nicht inspiziert
    - grün: Block wurde bereits besucht, aber noch nicht alle seine Nachfolger
  - Alle bereits besuchten Blöcke werden blau markiert und alle von hier aus erreichbaren Blöcke grün
  - Der Algorithmus terminiert, wenn keine grünen Blöcke mehr existieren.
  - Dann werden alle roten Blöcke gelöscht, da diese nicht mehr erreichbar sind,

8.5.4 Inkrementeller Mark & Sweep Algorithmus

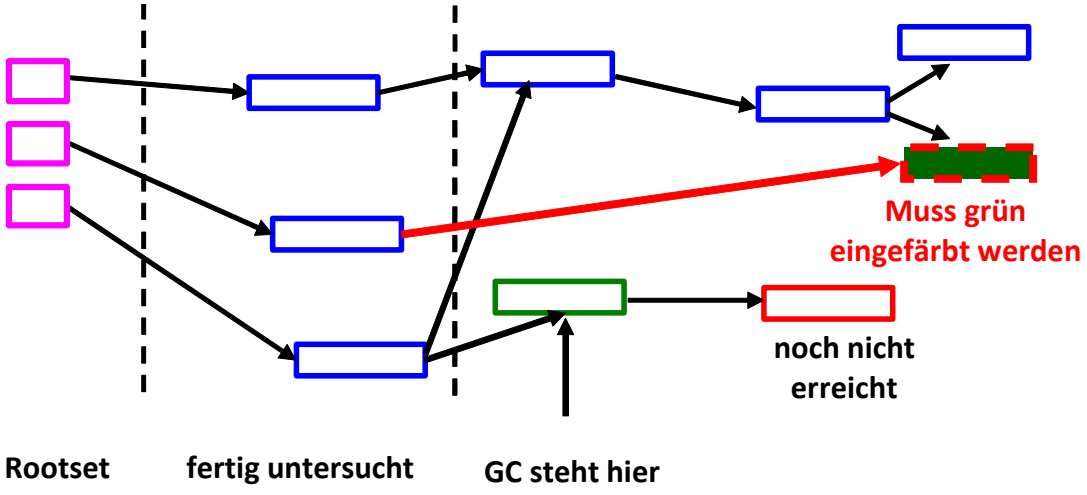
- Der Collector schiebt eine Front grüner Blöcke vor sich her:



- Es werden u.U. nicht alle Garbage-Blöcke in einem Durchlauf eingesammelt.

8.5.4 Inkrementeller Mark & Sweep Algorithmus

- Bedingung: bereits komplett untersuchte Blöcke, dürfen keine Zeiger auf noch nicht untersuchte Blöcke beinhalten.
- Erfolgt eine Zuweisung einer Referenz von einem blauen auf einen roten Block, so muss der rote Block grün eingefärbt werden.

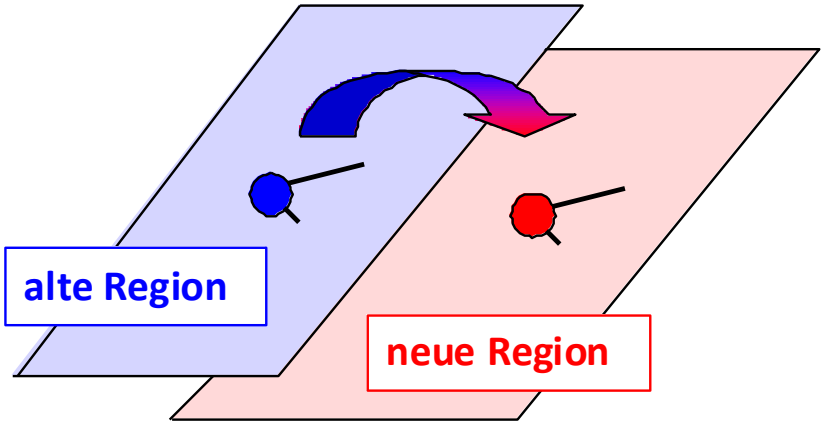


8.5.4 Inkrementeller Mark & Sweep Algorithmus

- Erfordert eine Überwachung von Zeigerzuweisungen
- Realisierung indem der Compiler bei einer Zeigerzuweisung einen Aufruf an eine Funktion des GCs generiert.
- Somit kann der GC entsprechend reagieren

8.5.5 Kopierende Freispeichersammlung

- Erste Implementierung Marvin Minsky, 1963.
- Halde in zwei Regionen alt & neu unterteilt.
- Alle von den Wurzel-Zeigern aus erreichbaren Blöcke werden rekursiv in die neue Region kopiert.
- Garbage verbleibt in alter Region.
- Beim nächsten GC-Aufruf tauschen die alte und neue Region ihre Rollen.



8.5.5 Kopierende Freispeichersammlung

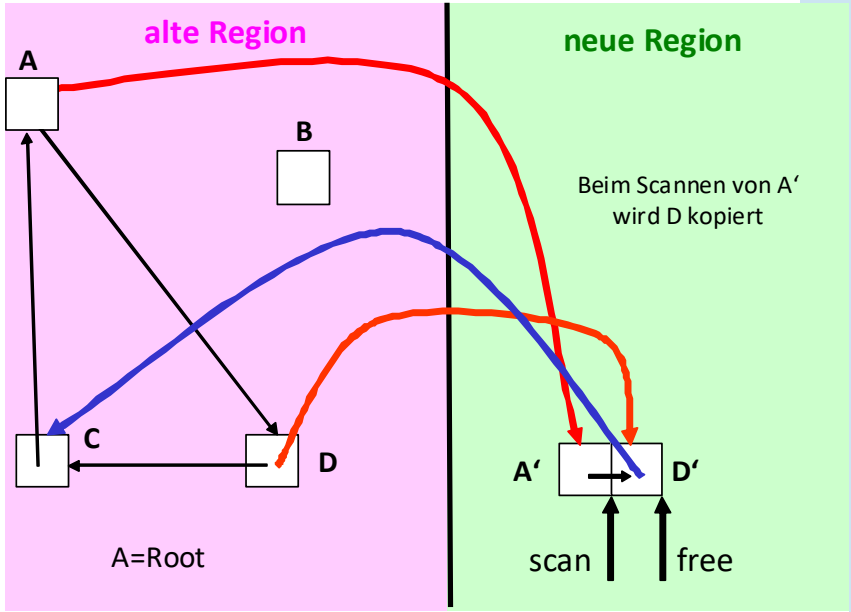
- Vorteile:
  - Heap wird automatisch kompaktifiziert
  - Speicherallokation ist einfach, da freier Speicher immer ein großer Block ist
  - Zyklen werden eliminiert.

- Nachteile:
  - Es ist zeitaufwändig, viele kleine Blöcke zu kopieren
  - Der logische Adressraum wird halbiert
  - GC muss atomar komplett durchlaufen

8.5.6 Inkrementeller Copying-Collector

- Pro Aufruf der GC eine vorgegebene Anzahl von Blöcken kopieren (nicht für längere Zeit das Programm anhalten).
- Iterative Lösung nach Cheney, 1970:
  - Neue Region wird durch Umkopieren fortlaufend gefüllt
  - **scan-Zeiger**: Blöcke bis hier sind komplett abgearbeitet.
  - **free-Zeiger**: Blöcke zwischen scan- und free-Zeiger sind kopiert, haben aber noch Zeiger in die alte Region.
  - Kopierte alte Blöcke verweisen auf Ihre Kopie

Momentaufnahme; A ist Wurzelzeiger



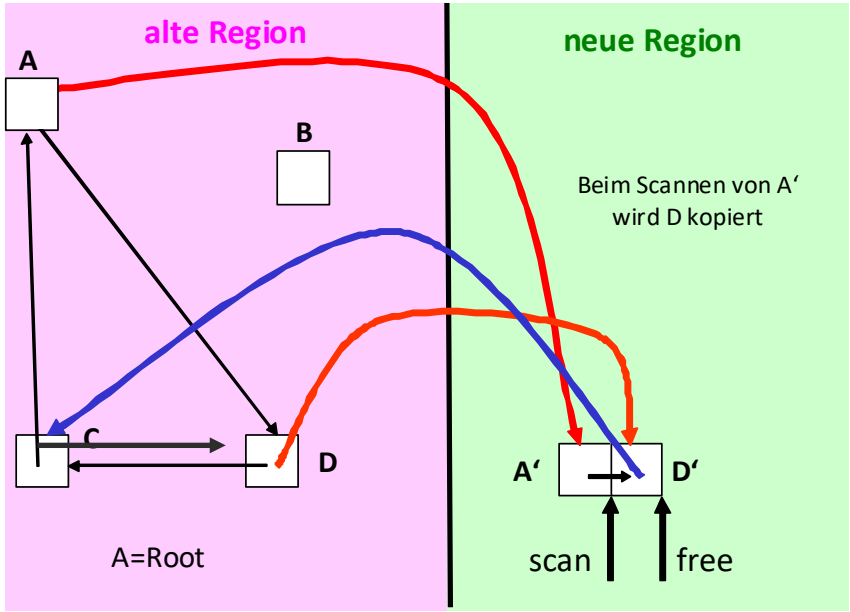
8.5.6 Inkrementeller Copying-Collector

- Algorithmus terminiert, wenn scan-Zeiger auf free-Zeiger trifft

- Bedingung: Komplett abgearbeitete Blöcke, dürfen nicht auf Blöcke in alter Region zeigen
  - Erfolgt eine derartige Zuweisung, so muss der referenzierte Block sofort kopiert werden
  - Zeigerzuweisungen müssen auch hier überwacht werden.

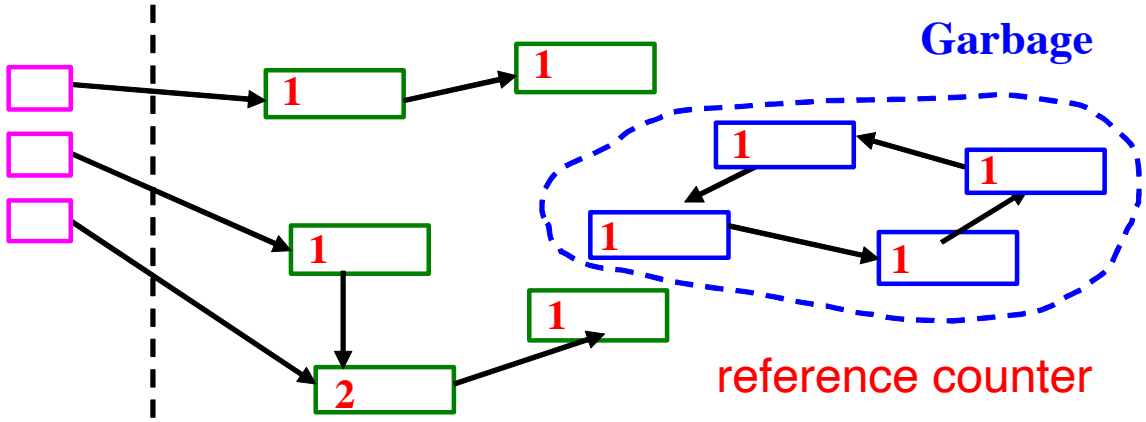
8.5.6 Inkrementeller Copying-Collector

- Nachteile:
  - Überwachung von Zeigerzuweisungen ist teuer.
  - Datenzugriffe auf bereits kopierte Blöcke (von noch nicht kopierten Blöcken aus) müssen erkannt und synchronisiert werden
    - Beispiel: C zeigt nun auch auf D
    - Nun könnten sowohl Zugriffe auf D über C erfolgen, als auch von A auf D‘
    - Dadurch könnten Inkonsistenzen entstehen -> muss verhindert werden



8.5.7 Reference Counting Algorithmus

- Jeder Speicherblock wird durch einen versteckten Referenzzähler erweitert und speichert die Anzahl der Referenzen auf sich
- Ein Block ist Garbage, wenn der Referenzzähler null ist.
- Zeigerzuweisung über Laufzeitfunktion:
  - In der Laufzeitroutine erfolgt Zeigerzuweisung und Inkrementierung des Referenzzählers.
  - Bei Zuweisung von „null“ wird der Referenz-zähler erniedrigt.



8.5.7 Reference Counting Algorithmus

- Vorteile:
  - inkrementelle GC möglich,
  - Garbage wird sofort freigegeben.
  - einfach implementierbar.
- Nachteile:
  - Zyklen werden nicht erkannt.
  - Zeigerverwaltung erfordert den Aufruf einer Laufzeitroutine