D. Hauptspeicher

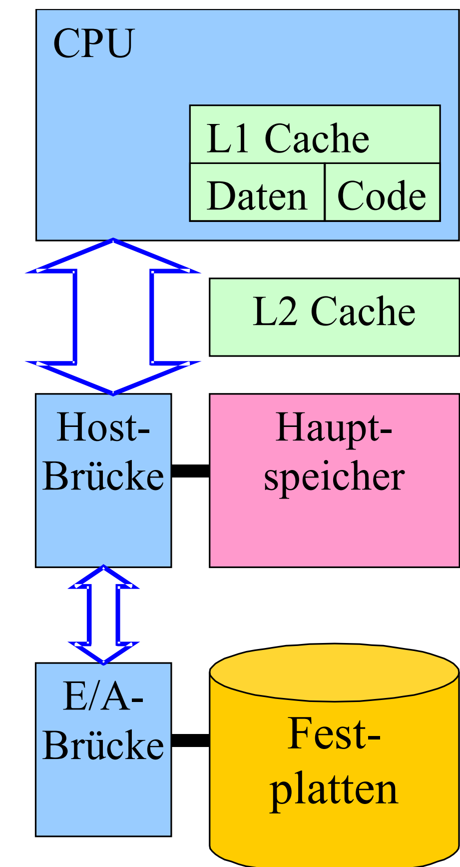
D.1 Einführung

D.1.1 Aufgaben der Speicherverwaltung.

- Partitionierung des Hauptspeichers:
  - statische und dynamische Unterteilung.
  - Struktur einer Partition.
- Speicherverwaltung und Zuteilung:
  - Aufgabeteilung zwischen OS und Laufzeitroutinen,
  - dynamische Datenstrukturen im Heap (Halde),
  - prozedurale Rekursion im Keller (Stack),
  - Kellerrahmen evtl. als OS Konvention.
- Auslagern von Programm(teil)en:
  - Overlay Technik & Swapping
  - Virtueller Speicher.
- Freispeichersammlung:
  - Mark & Sweep & Copy Collectoren,
  - im Netz, im Rechner, in der Partition ...

D.1.2 Speicherhierarchie

- **Cache**:
  - flüchtiger Inhalt, Zugriff On-Chip meist mit vollem CPU-Takt,
  - Intel 9550: 2*32 kB L1 Cache/CPU-Kern, 12 MB L2 Cache gesamt.
  - meist mehrstufig, unter Umständen L3 Cache off-Package .
- **Hauptspeicher**:
  - flüchtiger Inhalt, schneller Zugriff (z.B. SDRAM 5 ns).
  - DDR3 RAM theoretisch bis zu 20 GB/s.
  - höhere Kapazitäten, z.B. 4 GB.
- **Plattenspeicher**:
  - große Kapazitäten, z.B. 1 Terabyte.
  - kostengünstige persistente Speicherung.
  - Zugriffszeiten in Millisekunden (z.B. 7ms).
  - SATA Festplatten erreichen theoretisch bis zu 300 MBytes/s.



- Die Speicherverwaltung organisiert unter anderem den Transfer zwischen den Haupt- u. Plattenspeicher (den Transfer mit dem Cache besorgt die Hardware).

D.1.3 Begriffe

Speicherblock:

- Menge von fortlaufenden logischen Speicheradressen (Seite, Segment ...).

Partition:

- (größerer) Gesamtspeicherblock typisch für ein ganzes Programm.

Swapping:

- Aus- und Wiedereinlagern von ganzer Partitionen auf Disk.

Physikalische (absolute) **Speicheradresse**:

- bezeichnet/zeigt in physisch vorhandenen Hauptspeicher.

Logische Speicheradresse:

- Position im Hauptspeicher aus Sicht des Programms,
- unabhängig von der physikalischen Speicherorganisation,
- => Virtualisierung der Speicheradressen.

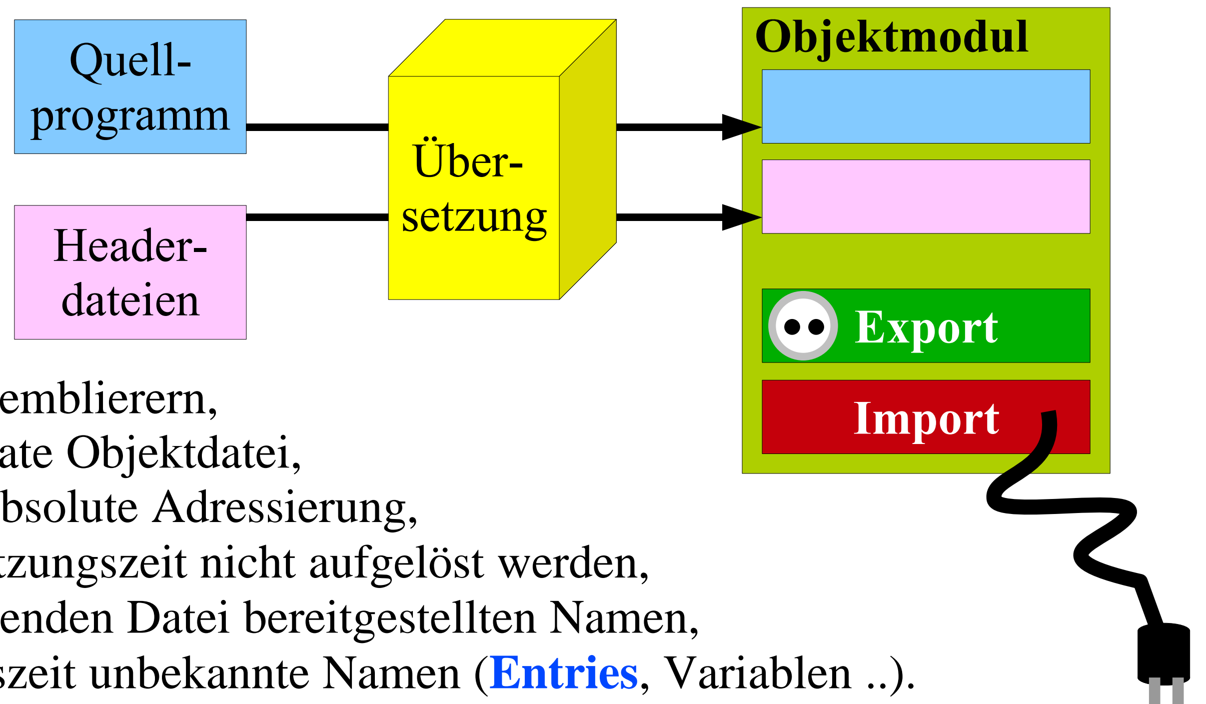
Relative Speicheradresse:

- Position relativ zu einem bekannten Punkt im Programm.
- meist relativ zum Programmzähler oder zum Programmbeginn,
- für Sprünge, Verzweigungen und Aufrufe.

D.2 Binden von Objektmodulen zu Lademodulen

D.2.1 Übersetzungsvorgang

- **Quellprogramme & -dateien (*.c, *.pas ...):**
  - werden mit einem Editor oder einem IDE erstellt (Integrated Development Environment).
- **Headerdateien (*.h):**
  - mit **#include** deklarierte Headerdateien werden hinzucompiliert,
  - zu diesem Zweck enthält der C-Compiler einen **Präprozessor**,
  - enthalten als Quelltext vorhandene Zusatzfunktionen,
  - etwa aus Verzeichnis **\include .**

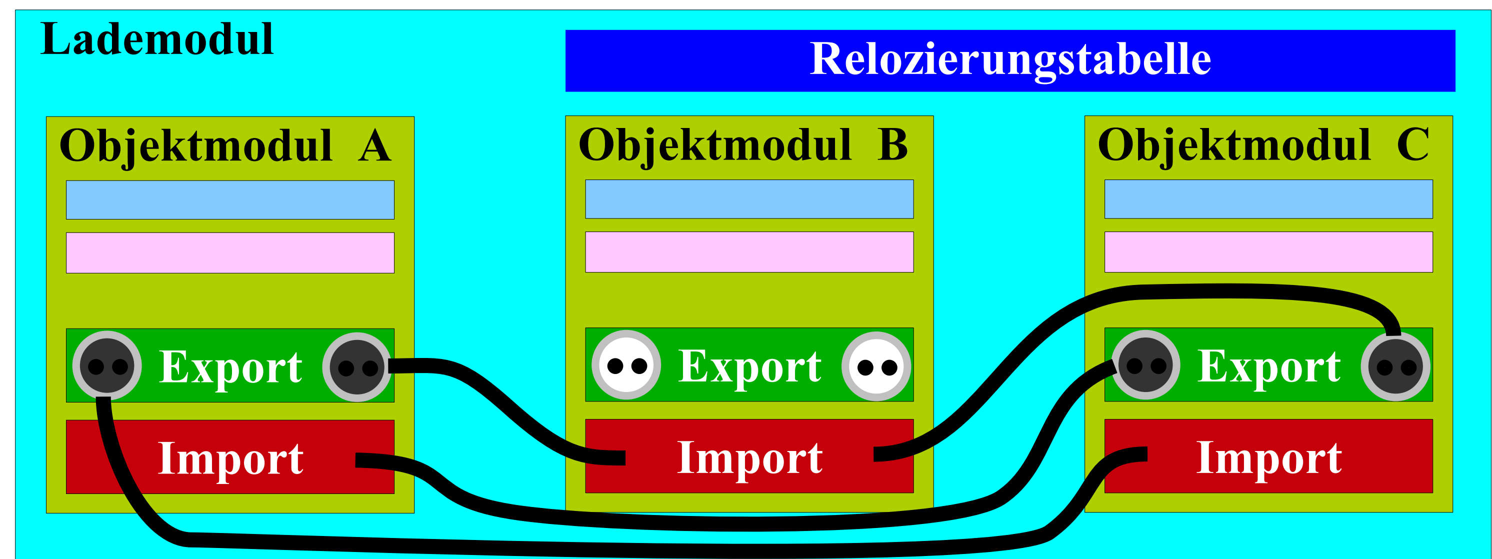


- **Objektdateien (*.obj):**
  - werden erzeugt von Compilern und Assemblierern,
  - meist pro Klasse oder Modul eine separate Objektdatei,
  - innerhalb der Objektdatei relative und absolute Adressierung,
  - **Bibliotheksaufrufe** können zur Übersetzungszeit nicht aufgelöst werden,
  - **Exporttabelle** enthält die in der vorliegenden Datei bereitgestellten Namen,
  - **Importtabelle** enthält zur Übersetzungszeit unbekannte Namen (**Entries**, Variablen ..).

D.2.2 Bindevorgang

- **Binder** (= Linker/Linkage Editor/ Mehrfachreferenzstellenverknüpfer **:)**
  - zum Betriebssystem gehöriges Hilfsprogramm,
  - bindet separat compilierte Objektdateien zu einem ausführbaren Programm,
  - löst Referenzen auf importierte Funktionen/Variablen/Klassen auf,
  - Erzeugt eine **Relozierungstabelle** zuhanden des Laders,
  - schreibt eine ausführbare Datei auf Disk z.B..

- **Lader**:
  - bringt das Lademodul zur Ausführung in den Hauptspeicher,
  - passende Ladeadresse ist erst zur Ausführungszeit bekannt,
  - Adressen in der Relozierungstabelle werden angepasst,
  - Evtl. verknüpfen mit Standarddateien (*.dll).



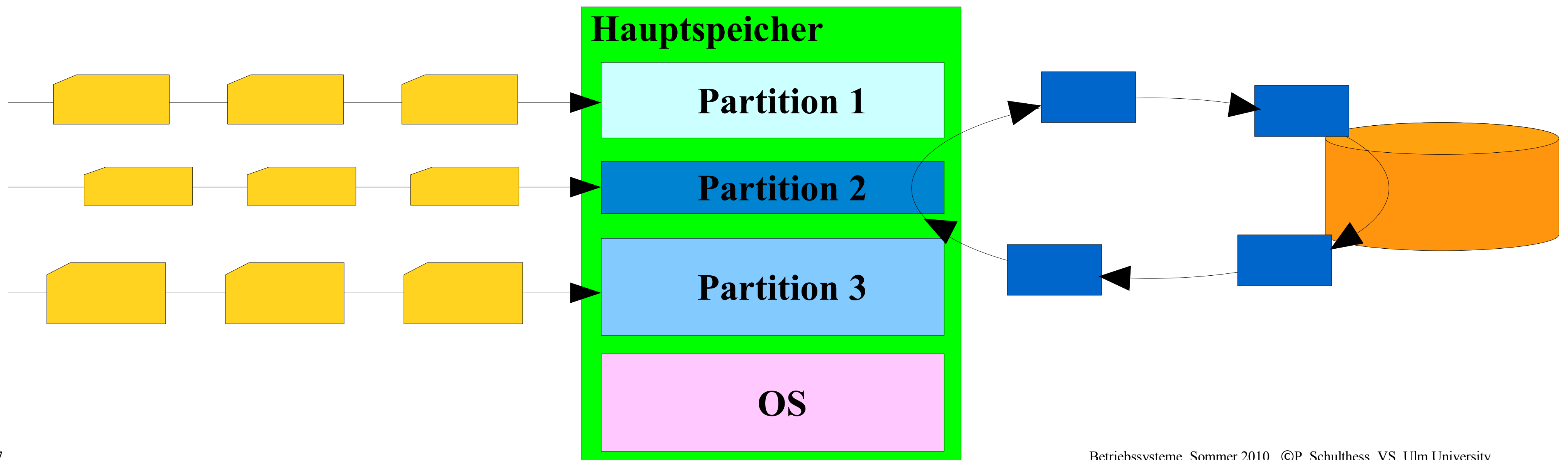
D.2.3 Late Binding und Early Binding

- Spätes Binden zum Ladezeitpunkt:
  - alle möglicherweise benötigten Module werden gesucht und verlinkt,
  - evtl. würden in diesem Programmlauf nicht alle wirklich benötigt,
  - Shared Libraries können genutzt werden (*.dll, *.so).
- Spätes Binden während der Programmausführung:
  - Das Programm entscheidet wann eine Bibliothek geladen und verlinkt werden soll,
  - 1. Schritt: Bilbliothek laden (LoadLibrary ...),
  - 2. Schritt: Einsprungpunkt bestimmen,
  - 3. Schritt aufrufen.
- Frühzeitige Adressbindung:
  - Innerhalb einer Compilierung können Adressen zur Übersetzungszeit aufgelöst werden,
  - Binder verknüpft separat übersetzte Objektmodule und Bibliotheken,
  - Im Quelltext vorhandene Hilfsfunktionen werden hinzucompiliert,
  - voll statisch gebundene Lademodule sind komplett,
  - benötigen mehr Platz im Speicher und auf Disk,
  - dafür keine fehlenden DLLs mehr,
  - Installation überflüssig !
- Spätes Binden in Java:
  - einzelne Klassen werden bei Bedarf automatisch geladen,
  - unter Umständen sogar erst fertig compiliert (JIT).

D.3 Partitionen im Hauptspeicher

D.3.1 Ausgangssituation

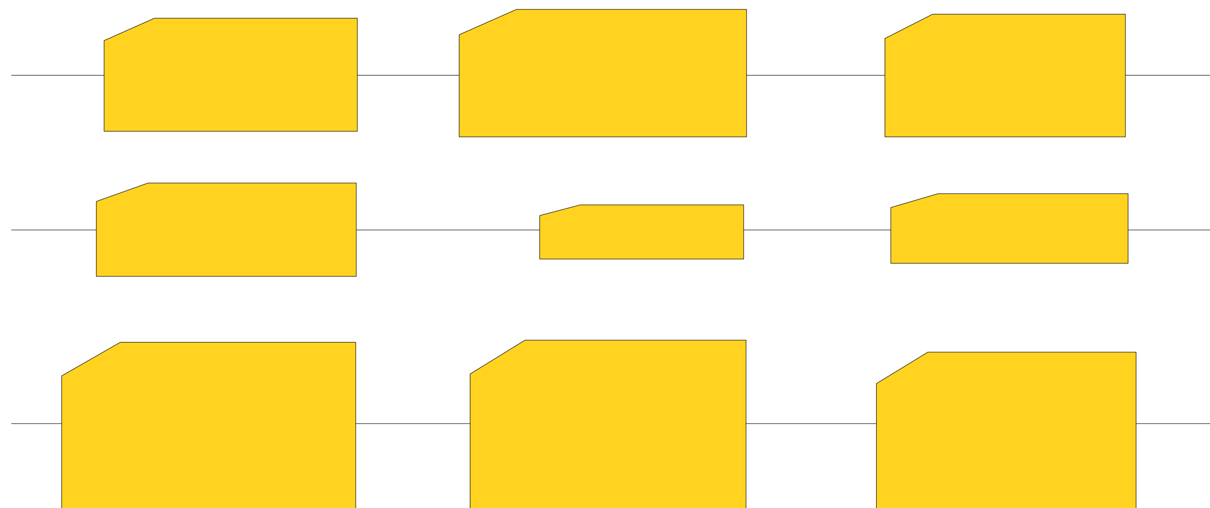
- Mehrprogrammbetrieb:
  - mehrere Programme teilen sich den vorhandenen Adressbereich,
  - Hauptspeicher ist eine knappe Resource und soll nicht brach liegen,
  - gleichzeitig geladene Programme müssen vonenander isoliert werden,
  - vorab ist unklar, welche Programme zu einem Zeitpunkt geladen sind.
- Swapping/Auslagerung:
  - Eine zeitweise inaktive Partition oder eine Segment kann als Ganzes ausgelagert werden,
  - Neuere Betriebssysteme verwenden Virtuellen Speicher anstelle von Partitionen (=> später).

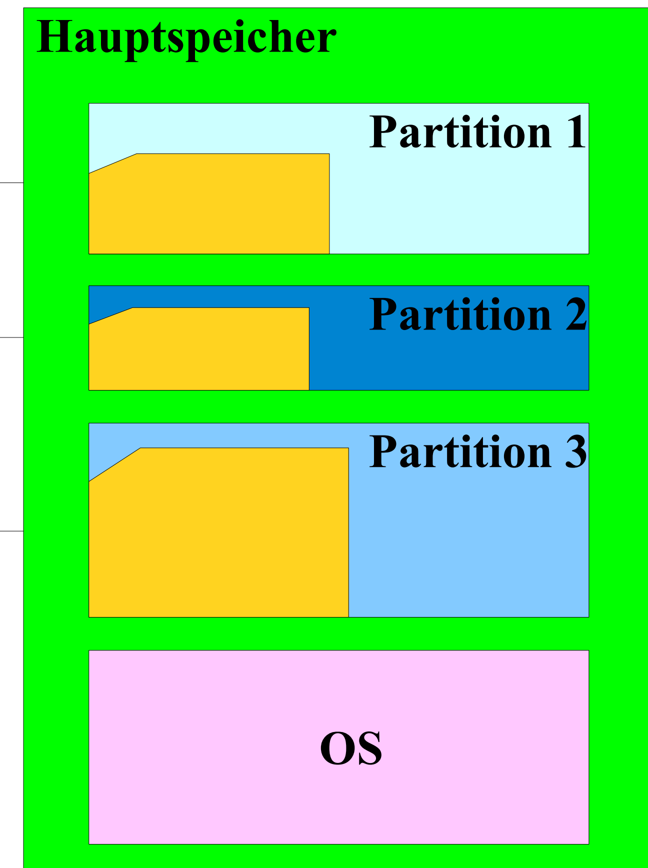


D.3.2 Interne versus externe Speicherfragmentierung

- Statische Partitionierung des Hauptspeichers:
  - Partitionierung ist während dem Betrieb nicht mehr änderbar.
  - Jedes Programm erhält eine eigene Partition,
  - Programm erhält kleinste Partition in die es hineinpasst.
  - Sind alle Partitionen belegt, so warten die Programme in einer Zuteilungsschlange.

- **=> Interne Speicherfragmentierung:**
  - Speicherverschnitt entsteht innerhalb eine Partition.

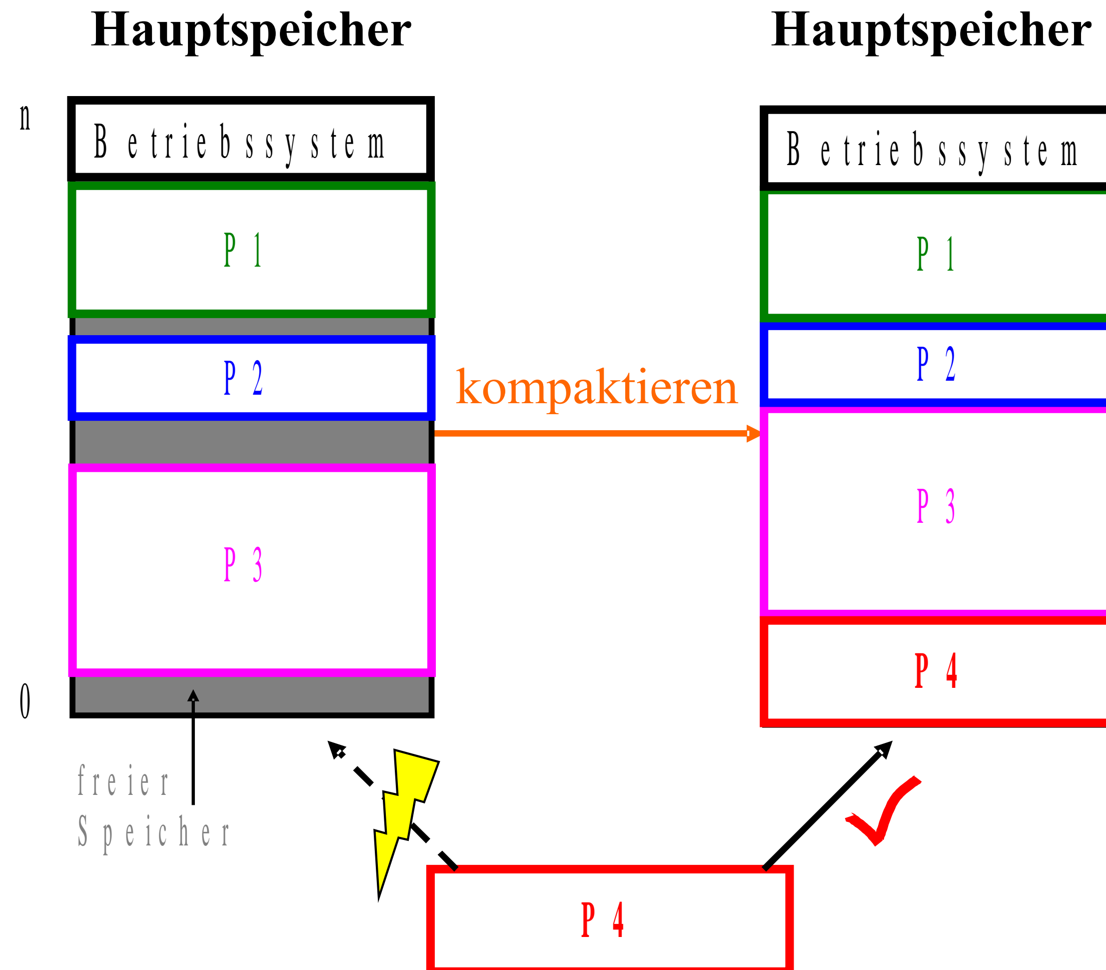




- Dynamische Partitionierung
  - Länge, Anzahl & Anfangsadresse d. Partitionen ändern sich dynamisch,
  - Programm erhält genau so viel Speicher wie es benötigt und nicht mehr,
  - Interne Fragmentierung innerhalb einer Partition wird verhindert.
- Aber: **Externe Fragmentierung**:
  - im Laufe der Zeit entstehen Löcher zwischen den Partitionen.
  - ein neues Programm kann eventuell nicht geladen werden,
  - obschon insgesamt genügend Speicher vorhanden ist.
- Lösung: **Heap Kompaktierung**
  - Im Prinzip möglich, wenn die Partitionen relozierbar sind,
  - unter Umständen Partitionen relozieren und Adressen neu binden,
  - evtl. HW Unterstützung (Segmentdeskr., Basisregister, Virtueller Sp.).
- Historisches Beispiel: OS/360 Varianten.
  - 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.

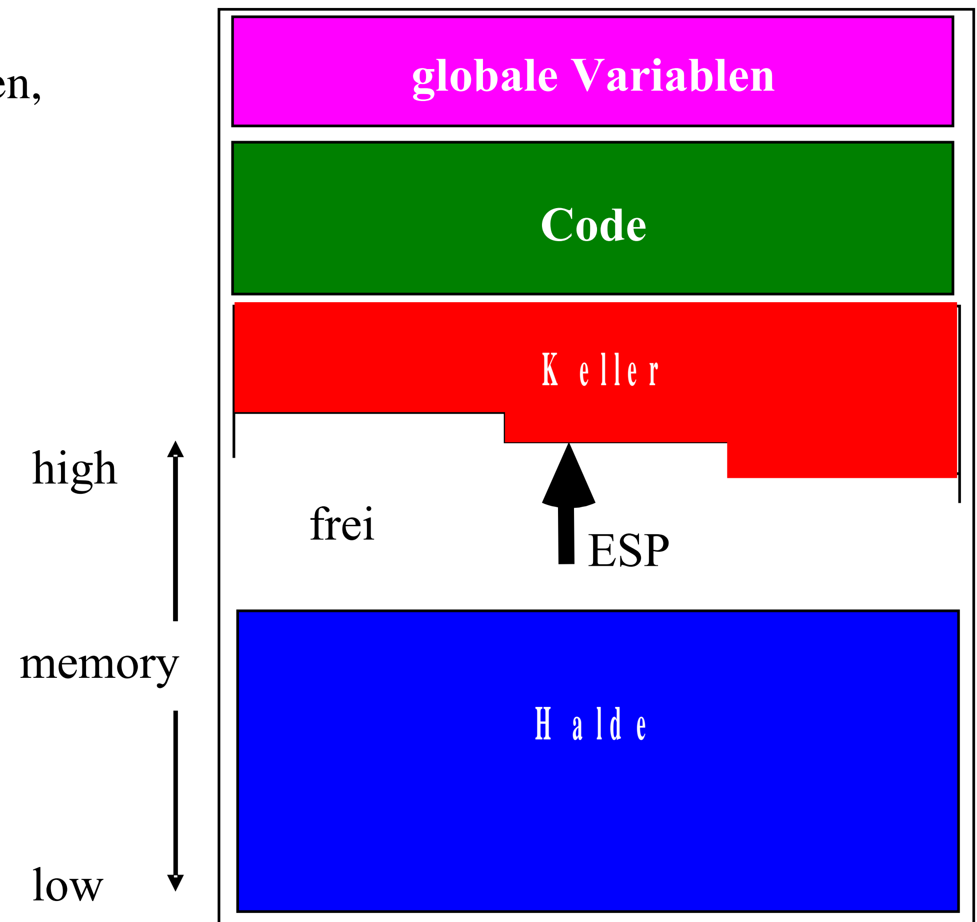
D.3.3 Beispiel für externe Fragmentierung:

- Programm P4 kann erst geladen werden, nachdem P2 und P3 verschoben wurden.



D.4 Interne Organisation einer Partition

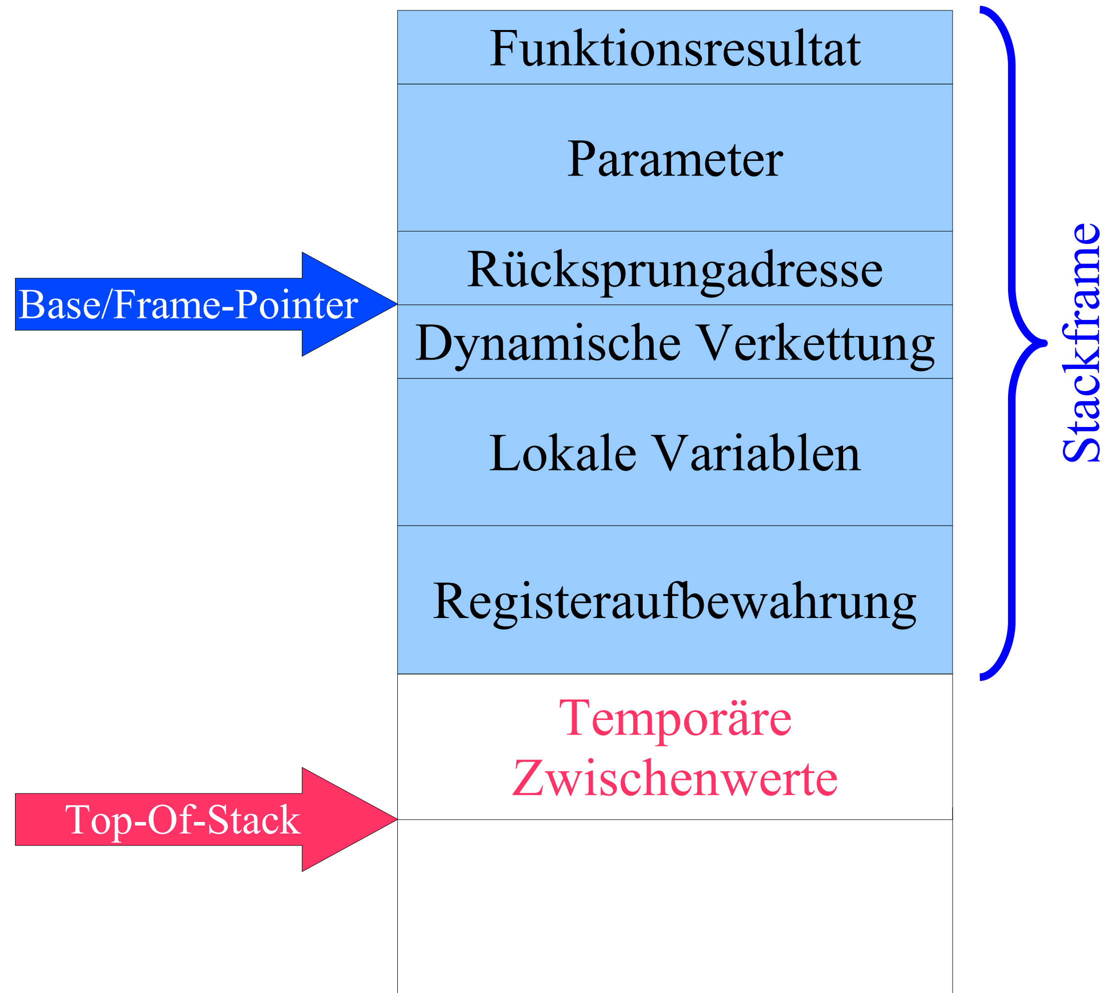
- **Halde (Heap)**:
  - explizite Allozierung zur Laufzeit,
  - für dynamische Datenstrukturen,
  - und Ressourcen (z.B. Puffer).
- **Keller (Stack)**:
  - Parameter & lokale Variablen, Rückkehradressen,
  - Stackzeiger (Intel: ESP),
  - Hinweis: schwer identifizierbare Fehler, falls Stack und Heap sich überschneiden.
- **Code-Abschnitt**:
  - eine oder mehrere Methoden,
  - meist schreibgeschützt,
  - manchmal Code auch in Heap, ...
- **Globale Variablen**:
  - modulübergreifende Daten.



Betriebssysteme, Sommer 2010, ©P. Schulthess, VS, Ulm University

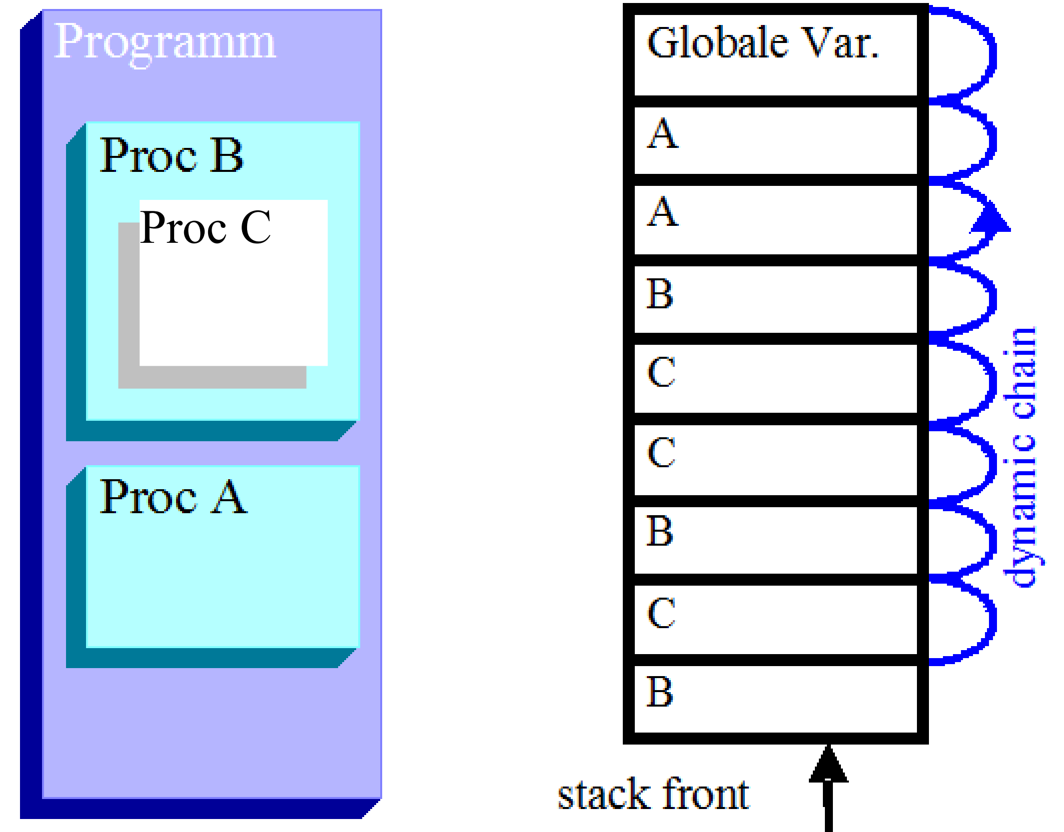
D.5.1 Kellerrahmen (Stackframe)

- Welche Informationen werden **für einen Prozeduraufruf** im Keller festgehalten?
- Aufgaben des Kellers:
  - Funktionsprinzip: Last in First Out .
  - Variablen im Keller, aber kein Code,
  - Funktionsresultat nur bei Funktionen.
  - Übergabe von Parametern (Register?),
  - Rücksprungadresse nach Abschluss,
  - Verkettungen & Register Save Area,
  - Speichern von lokalen Variablen,
  - Zwischenresultate von Berechnungen.
- Bei objektorientierten Sprachen:
  - zusätzlich aktuelle Klasse,
  - zusätzlich aktuelle Instanz.
- Alternativ: Parameter und Rückgabewerte in Registern.



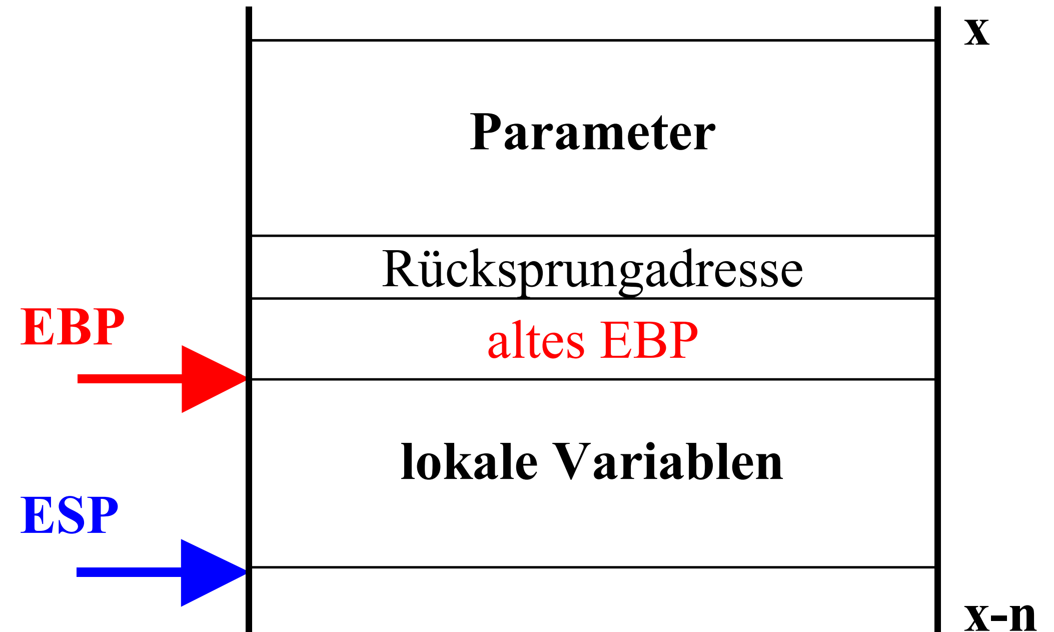
D.5.2 Dynamische Verkettung

- Bei Rücksprung aus einer Prozedur muss der alte Kellerrahmen wieder gefunden werden.
- → **dynamische Verkettung:**
  - alten Stackzeiger sichern,
  - alten Basezeiger sichern.
- Verschachtelte Prozeduren benötigen zusätzlich eine Statische Verkettung.
- Statisch # dynamisch!



D.5.3 Kellerrahmen bei Intel x86

- Keller wächst nach unten:
  - Ein Eintrag belegt 32-Bit.
  - 16-Bit mögl., aber nicht empfohlen.
- EBP = **E**xtended **B**ase **P**ointer:
  - 32 Bit Register,
  - zeigt auf den akt. Kellerrahmen.
- ESP = **E**xtended **S**tack **P**ointer:
  - 32 Bit Register,
  - zeigt auf aktuellen Stack-Eintrag.
- Adressierung:
  - Parameter mit positivem Offset relativ zu EBP,
  - lokale Variablen mit negativem Offest relativ zu EBP.
- Rückgabewert in EAX.



Betriebssysteme, Sommer 2010, ©P. Schulthess, VS, Ulm University

D.5.4 Aufrufkonventionen für Prozeduren

- Die Aufrufkonvention ist abhängig von der Programmiersprache.
- Beispiel: Programmiersprache C:
  - Parameter werden von rechts nach links übergeben.
  - Die aufrufende Prozedur räumt den Keller auf.
  - Funktionsresultate werden in Registern zurückgegeben.

C	Assembler

`int Calc(int x, int y)`
`{`
`int result;`
 	
`result = x + y;`
 	
`return result;`
`}`
 	
`...`
 	
`Calc(5, 7)`
	`push EBP`
	`mov EBP,ESP`
	`sub ESP,4`
	`mov EAX,[EBP+8]`
	`add EAX,[EBP+12]`
	`mov [EBP-4],EAX`
	`mov EAX,[EBP-4]`
	`mov ESP,EBP`
	`pop EBP`
	`ret`
	
	`push 7`
	`push 5`
	`call Calc`
	`add ESP,8`

D.6 Heapspeicher-Verwaltung

- Zum Vergleich: Speicherverwaltung für einen Keller bzw. Stack:
  - Automatische **Vergabe(=Allozierung)** eines Kellerrahmens beim Prozeduraufruf,
  - entsprechend der dynamischen Verschachtelung der Methoden.
  - einfachste Rückgewinnung der Allozierung im Stack,
  - streng Last-In/First-out.
- Speicherverwaltung in einem Heap (deutsch Halde):
  - Explizite Allozierung mit **new() oder malloc()** zum Aufbau dynamischer Datenstrukturen,
  - Fortbestand der Datenstrukturen unabhängig von der Methodenverschachtelung,
  - Ein Heapspeicher hat **nichts** mit dem Heapsort-Verfahren zu tun.
- Freispeichersammlung:
  - Speicherblöcke dürfen erst wieder verwendet werden, wenn es darauf keine Referenzen mehr gibt, Entweder explizite Freigabe von nicht mehr benötigtem Speicher (**free()**),
  - Explizite Freigabe erhöht das Risiko von Programmierfehlern,
  - Automatische Freispeichersammlung ist sicherer, erfordert jedoch ausgefeilte GC-Techniken.
- Fehlersituationen:
  - Speicherleaks Heapspeicher wird nicht mehr eingesammelt und geht damit verloren,
  - Dangling References übriggebliebene Zeiger auf schon zurückgegebenen Speicher.

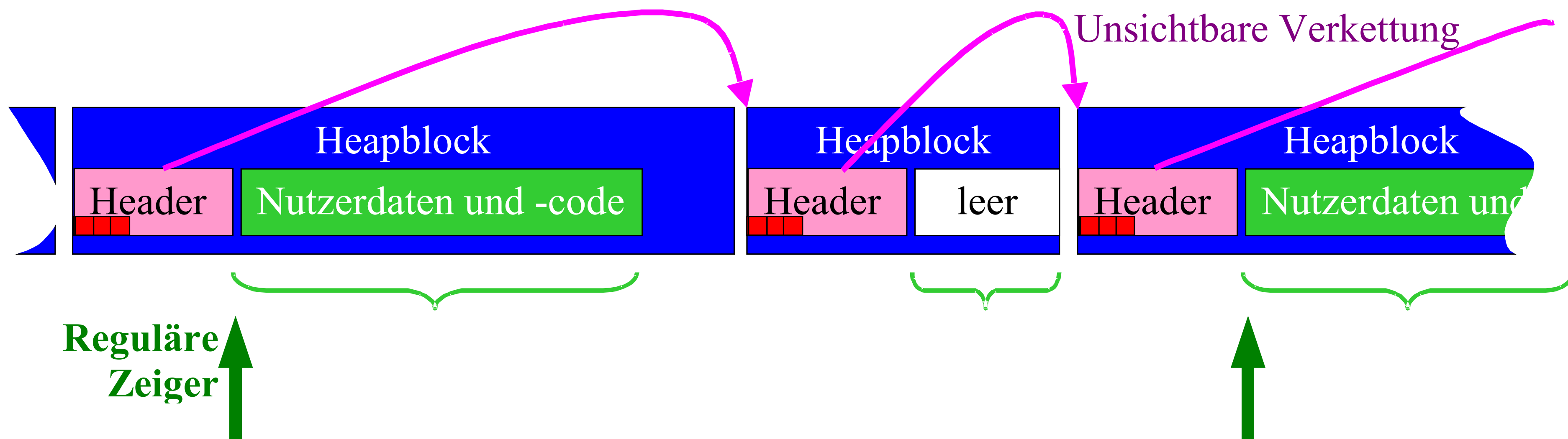
D.6.1 Speicherleak in Java - Beispiel

```
public class BuggyStack{
   private Object[] stack = new Object[10];
   private int count = 0;

   private void ensureCapacity(int top, Object[] oldStack){
      if (top > stack.length){        // reallocate if needed
         stack = new Object[top*2];   // here always top > 10
         while (top-- >= 0) stack[top] = oldStack[top];
      }
   }
   public Object pop(){   // should clear element reference
      if (count>0) return stack[ --count ];
      throw new IllegalStateException("Empty Stack!");
   }
   public void push(Object element){
      ensureCapacity(count, stack);
      stack[count++] = element;
   }
}
```

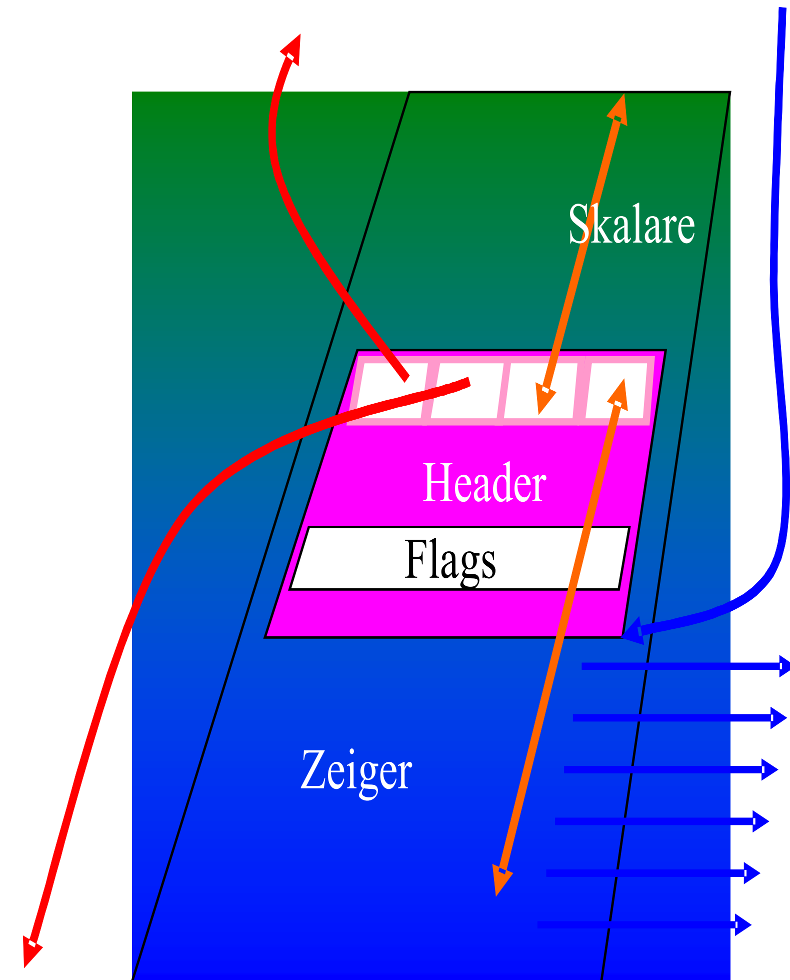
D.6.2 Format von Heapblöcken:

- Ein Aufruf von *new()* oder *malloc()* liefert einen neuen Heapblock.
- **Header**: enthält Informationen für Speicherverwaltung (~ 4-8 Byte)
  - Längenfelder, **nächster Heapblock**, Containergrösse, Anzahl Elemente,
  - Header normalerweise außerhalb des **Nutzdatenblocks**,
  - Typ: z.B. Zeiger auf den Klassendeskriptor,
  - Flags: Locked, Read-Only, Free, Marked &
- Heapverkettung:
  - Alle Heapblöcke sind verkettet, und können sequentiell geprüft werden.
  - Auch ein nicht belegter Block braucht deshalb einen Header,
  - besonders wichtig für die Freispeichersammlung.



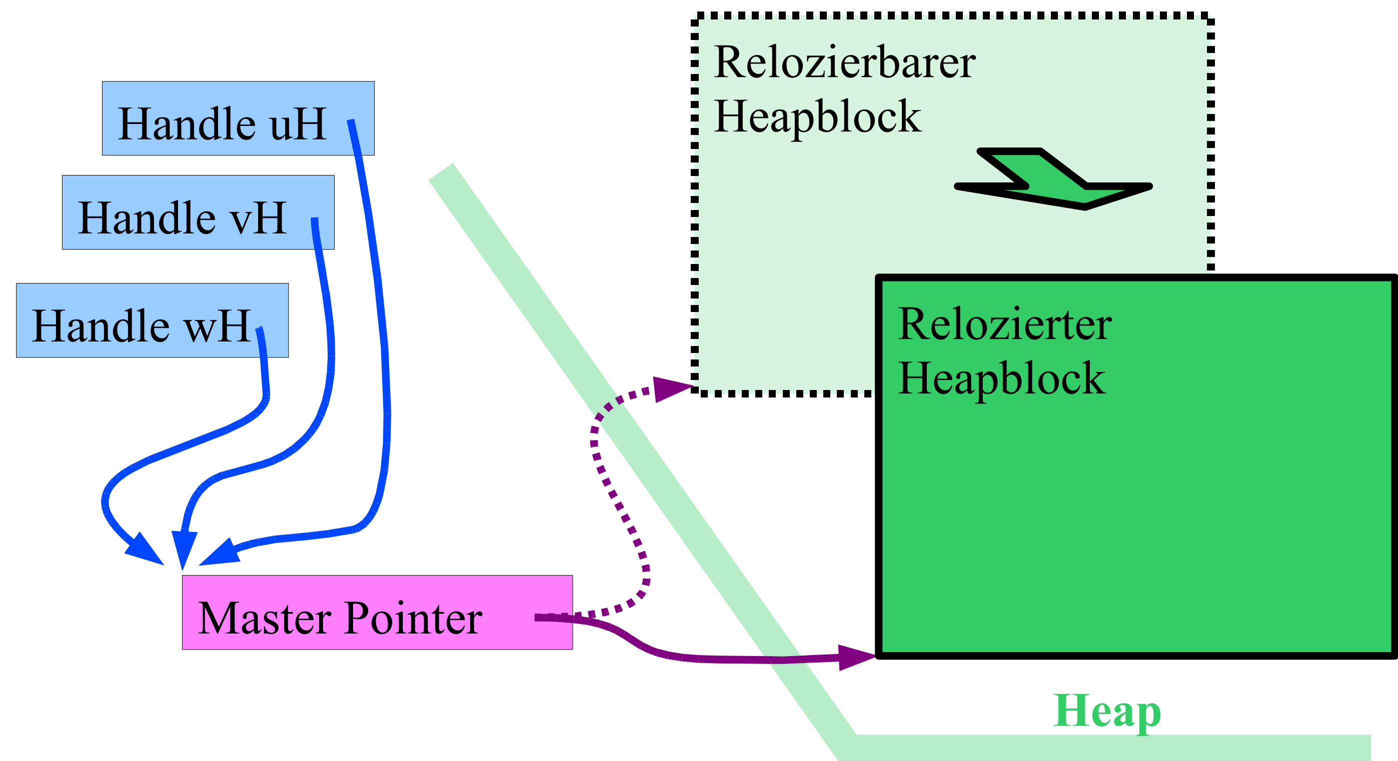
D.6.3 Doppelköpfige Heapblöcke im Rainbow OS

- Prototypisches OS des Institutes.
- Spezialität: Doppelköpfiges Format:
  - Trennung von **Zeigern** und **Skalaren**,
  - Referenzen zeigen immer auf Header,
  - vereinfacht das Heap-Management.
  - **Header** in der Mitte des Blocks,
  - Flags enthalten den Zustand,
  - **zweifache Verkettung**,
  - **zwei Längenfelder**.



D.6.4 Handles & Masterzeiger:

- Indirektions-Prinzip:
  - Nur der Masterzeiger referenziert die Daten im Heap.
  - Handles zeigen auf einen Masterzeiger und nur indrekt auf das Objekt.
  - Eventuell verweisen viele (kurze) Handles auf einen Masterzeiger.
- Relozierungs-Vorteil:
  - Ein Objekt bzw. ein Speicherblock kann reloziert werden, ohne alle Handles kennen zu müssen.
  - Wird ein Objekt temporär nicht benötigt, so teilt dies die Anwendung dem Betriebssystem mit,
  - das Objekt bzw. der Heapblock kann dann verschoben werden.
- Verwendet in 16 Bit Windows 3.x und älteren MacOS Versionen.
- Vor dem Dereferenzieren von Handles bitte Speicherblöcke fixieren.



D.6.5 Aspekte einer Speicherverwaltung

- Granularität der Speicherblöcke:
  - wie gross ist das Vergabequantum?

- Belegungsdarstellung:
  - wie wird Buch geführt über belegte und freie Speicherzellen?

- Verschnitt:
  - wie effektiv arbeitet die Vergabe hinsichtlich Fragmentierung,
  - wie leicht entsteht überhaupt eine Speicherfragmentierung,
  - externe und interne Fragmentierung betrachten.

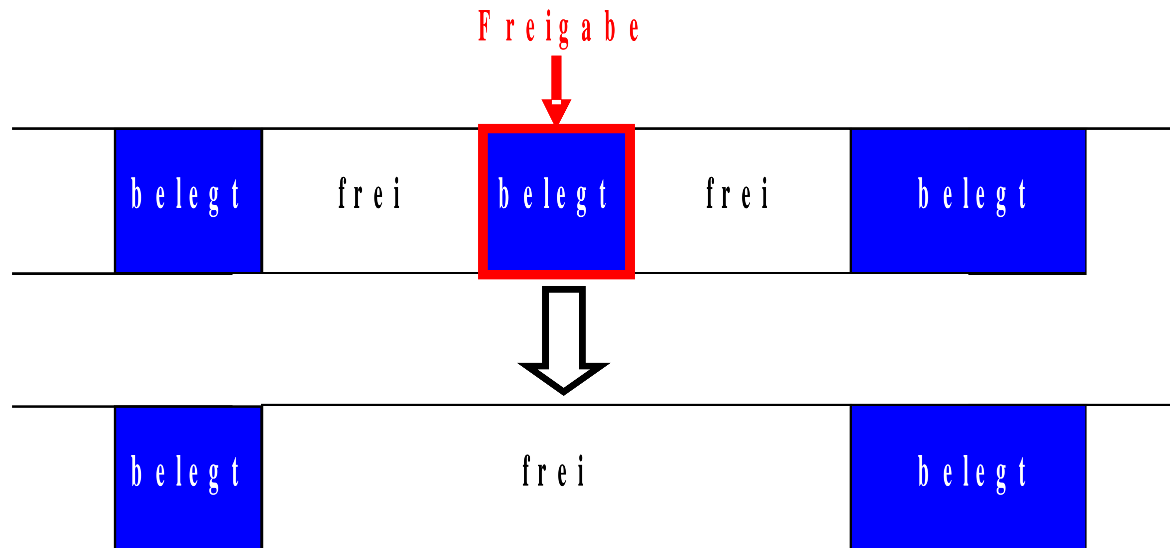
- Auswahlstrategie für freie Stücke:
  - wo soll der freie Speicher vorzugsweise gewählt werden,
  - wie lange dauert die Suche nach einem passenden Block.

- Wiedereingliederung von unbenutzten Blöcken:
  - wie können freigewordene Blöcke wieder verfügbar gemacht werden,
  - Auffinden und verschmelzen von leeren Blöcken.

D.6.6 Wiedereingliederung

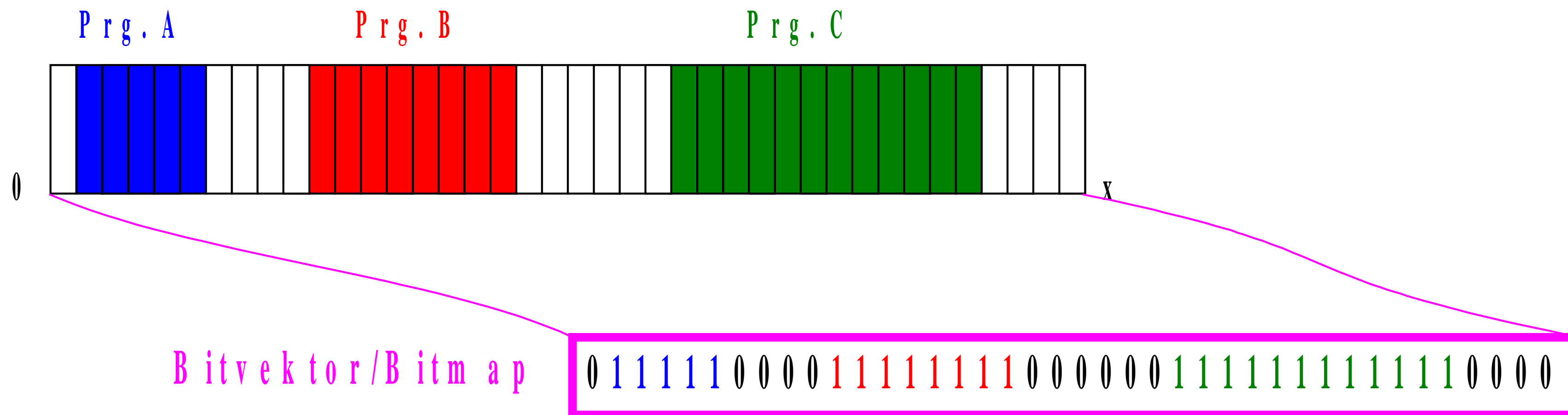
- Bei Freigabe eines Speicherblocks prüfen, ob Nachbarblöcke frei sind und gegebenenfalls zusammenfassen.

- Hiermit entstehen wieder größere Blöcke.



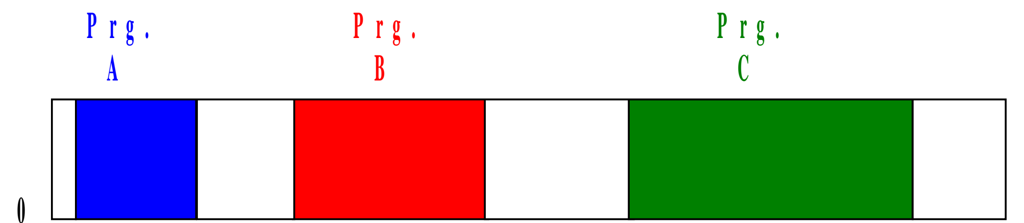
D.7.1 Belegung mit Bitvektor:

- 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.
- Beispiel: 128 MB in 512 Byte Blöcke unterteilt ergibt 32 KB Bitvektor.
- Belegung eines Speicherbereichs erfordert das Durchsuchen des Bitvektors nach Nullbit-Folgen (aufwendig).
- Beispiel:



D.7.2 Freispeichertabelle:

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



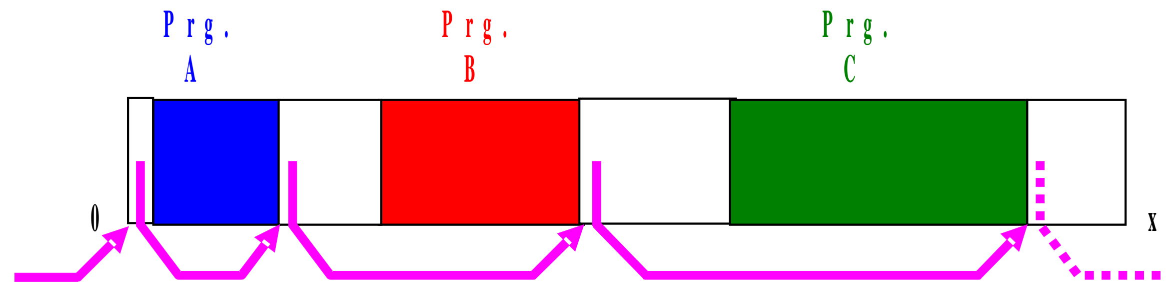
Adr.	Länge
0	1
6	4
19	6
39	4

Länge	Adr.
1	0
4	6
4	19
6	39

Betriebssysteme, Sommer 2010, ©P. Schulthess, VS, Ulm University

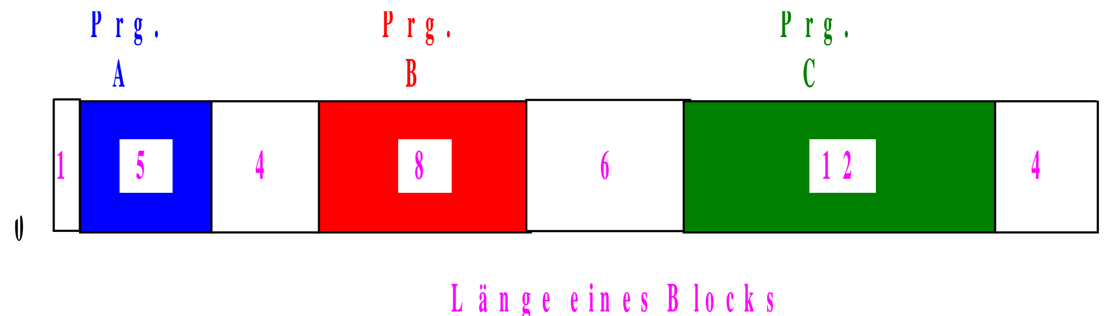
D.7.3 Freispeicherliste:

- Freie Speicherblöcke mit **Zeiger verketten**.
- Eventuell mehrere Listen:
  - verschied. Größen separat verketten,
  - pro Programm oder systemweit ...
- Optimierung:
  - Binärbaum für Zugriff,
  - Blockgröße als Schlüssel.



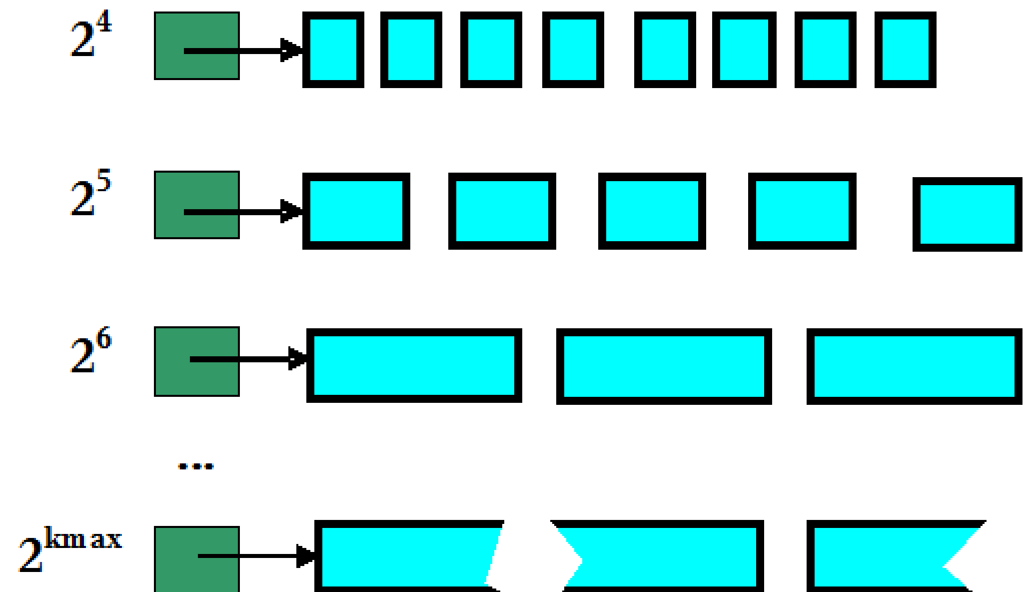
D.7.4 Linearer Heap:

- Freie & belegte Blöcke sind bündig aneinander gereiht.
- Verkettung der Blöcke erfolgt über das Längenfeld.
- Freie Blöcke sind durch ein Bit gekennzeichnet.
- Optimale Ordnung der Blöcke ist schwierig.
- Interne Zeiger sind überflüssig.
- Z.B. Mac OS & Plurix.



D.7.5 Buddy-System:

- Grundprinzip:
  - Zwei gleichgroße benachbarte Blöcke sind Buddies (Kumpels).
  - Speicher besteht anfänglich aus Einheiten der Grösse 2^{kmax} .
  - Speichervergabe in Blockgrößen von 2^k.
  - Jeweils Liste für Blöcke der Größe 2^k.
  - Variante in Linux-Kern verwendet.



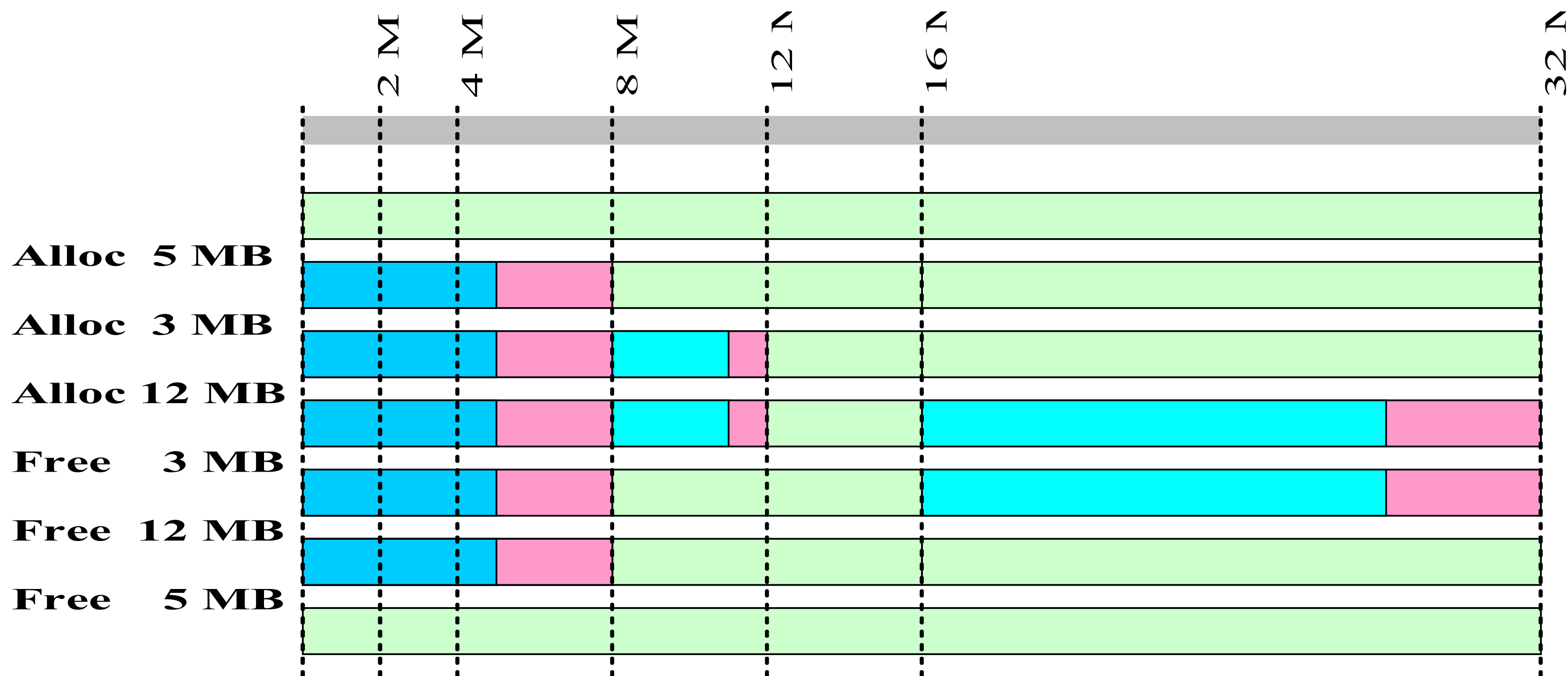
- Ablauf einer Anforderung:
  - Aufrunden auf nächste Zweierpotenz,
  - Zugriff auf erstes freies Stück der Liste.

- Falls Liste leer (rekursiv):
  - Zugriff auf Liste der nächsten Größe.
  - Stück entfernen & halbieren.
  - Hintere Hälfte in zugehörige Liste einfügen.

- Kleinere Stücke entstehen aus (rekursiver) Halbierung größerer Stücke.

Betriebssysteme, Sommer 2010, ©P. Schulthess, VS, Ulm University

- Ablauf der Speicherfreigabe im Buddy Vergabesystem:
  - Buddy bestimmen,
  - falls Buddy frei → Rekombination,
  - falls Buddy belegt, freigewordenes Stück in die Liste einhängen,
  - Vorgang iterieren, bis Buddy belegt oder bei der max. Größe angekommen.
- Vorteil: schnelles Verschmelzen freiwerdender Blöcke.
- Nachteil: sowohl interne als auch externe Fragmentierung.
- **Beispiel:** Nutzdaten **und interner Verschnitt**



Betriebssysteme, Sommer 2010, ©P. Schulthess, VS, Ulm University

D.8 Auswahlstrategien

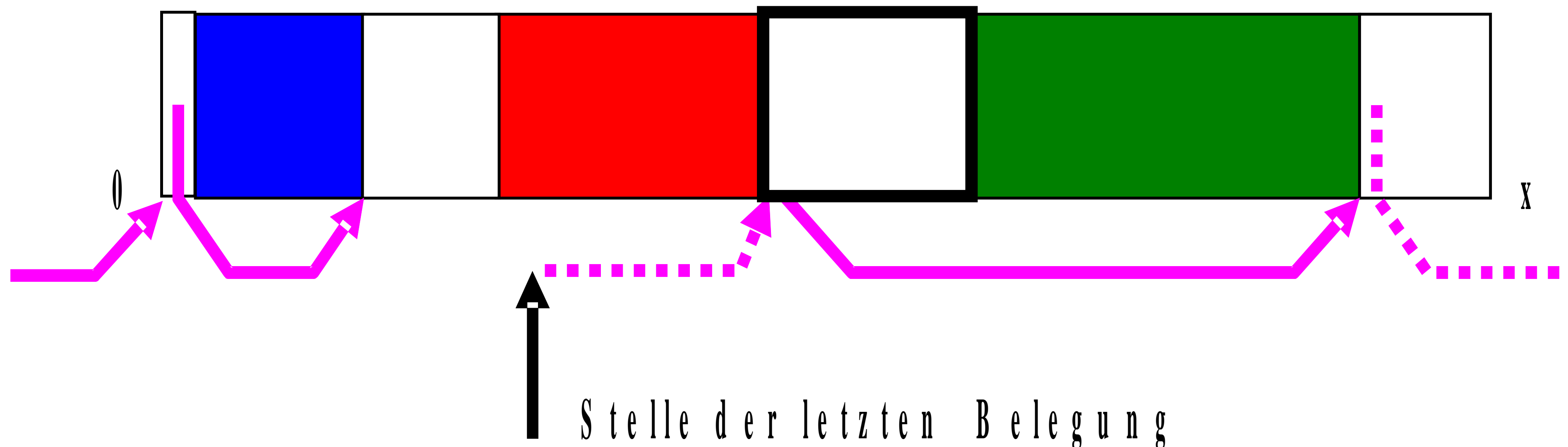
- Gütekriterien:
  - Die beste Strategie zu finden, ist auch bei bekannten Objektgrössen ein schwer lösbares Problem.
  - wird wenig Speicherverschnitt erzeugt, so kann der Heapbereich kleiner gewählt werden,
  - schnelle Allozierung bringt hohe Ausführungsgeschwindigkeit des Programmes,
  - suboptimale Allozierung erhöht den Aufwand der Speicherrückgewinnung,
  - Speicherrückgewinnung kann in Zeiten niedriger Last stattfinden.

D.8.1 First Fit :

- Grundprinzip:
  - Nimmt ersten freien Block der groß genug ist,
  - durchsuchen der Belegungstabelle immer vom selben Anfangspunkt her,
  - anwendbar für alle Belegungsdarstellungen.
- Zu großen Block eventuell teilen, um ungebrauchten Platz zu sparen:
  - ohne Teilen → interne Fragmentierung,
  - mit Teilen → externe Fragmentierung.
- Vorteil: sehr schnelle Speicherzuteilung.
- Nachteil: Konzentration belegter Stücke am Anfang.

D.8.2 Next Fit :

- Freispeicherliste (sortiert nach Adresse) 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.



Betriebssysteme, Sommer 2010, ©P. Schulthess, VS, Ulm University

D.8.1 Best Fit :

- Sucht den Block, der am wenigsten Speicherverschnitt verursacht.
- Empfohlene Belegungsdarstellung:
  - Unter Umständen nach Größe sortierte Freispeicherliste.
  - Evtl. Binärbaum mit Größe als Schlüssel für Zugriff,
  - nach Möglichkeit lineare Suche vermeiden.
- Vorteil: Zerschneiden großer Stücke in der Regel unnötig.
- Nachteil: langsam; neigt bei Zerschneiden dazu sehr kleine unbrauchbare Stücke zu erzeugen.

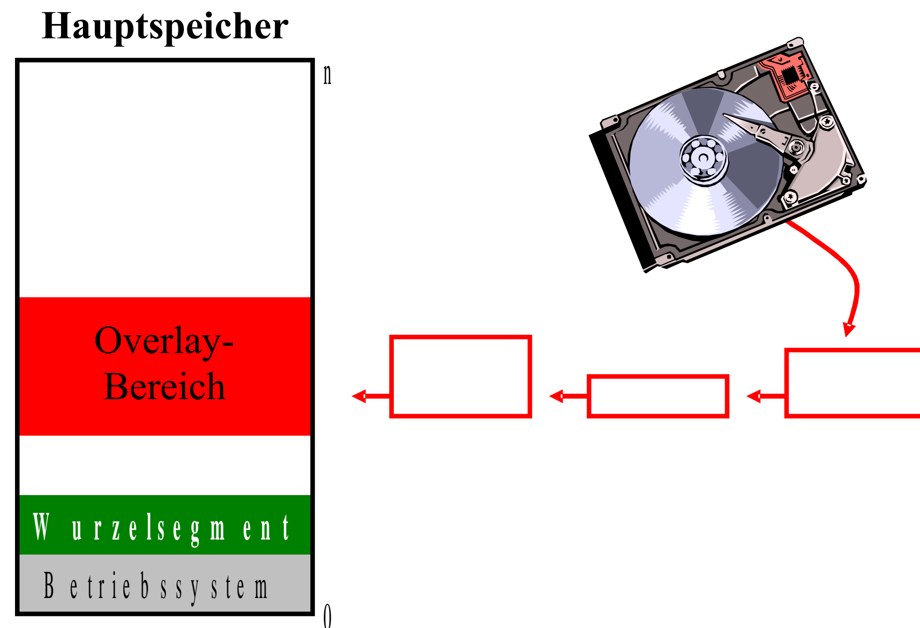
D.8.2 Worst Fit :

- Nimmt größten freien Block, damit nach dem Zerschneiden noch brauchbare Stücke übrig bleiben.
- Speicheranfragen werden i.d.R. aufgerundet:
  - aus Geschwindigkeitsgründen auf 32/64 Bit Grenzen,
  - oder falls verbleibende Reststücke zu klein sind.

D.9 Auslagern von Speicher

D.9.1 Overlay Technik

- Auslagern ist notwendig, wenn Programm größer als Hauptspeicher ist.
  - Idee: Nicht benötigte **Programmteile** werden durch andere überlagert.
  - Realisierung: ein Wurzelsegment muss immer im Hauptspeicher sein.
  - Overlays werden vor dem Binden durch Programmierer festgelegt.
- Unterstützung von Programm-Overlays durch:
  - Betriebssystem (z.B. MSDOS),
  - Compiler (z.B. Turbo Pascal → Units).
- Problem: Der HS-Bedarf für die Daten ist schwer abschätzbar.
- In Systemen mit virtuellem Speicher ist Overlay-technik überflüssig.



D.9.2 Swapping

- Swapping = Aus- und Wiedereinlagern auf Disk von **ganzen Programmen** oder **Partitionen**.
- Notwendig im Multiprogrammbetrieb, falls nicht genügend Hauptspeicher.
- Zeitaufwendig, da immer eine ganze Partition aus- und eingelagert wird.
- Eventuell erfolgt später die Wiedereinlagerung an anderer Adresse.
  → Zeigeranpassung benötigt HW-Unterstützung.
- Strategien:
  - Auslagern nicht rechenbereiter Programme.
  - Prioritäten berücksichtigen.
  - Wurde in Windows 3.x eingesetzt.
- Früher hat Unix auch Swapping eingesetzt:
  - zusätzlich zum Paging (virtueller Speicher),
  - wenn kein Proz. mehr genügend Speicher hat.
  - also nur in extremen Wettbewebsituationen.

D.10 Automatische Freispeichersammlung

- = **Garbage Collection** (GC)
- Explizite Rückgabe durch Programmierer ist *fehleranfällig & mühsam*:
  - Abbau komplexer Datenstrukturen oft schwierig und nur in mehreren Schritten.
  - Destruktoren in OO Sprachen (z.B. C++) löschen u. U. mehr als ein Objekt pro Aufruf.
  - wird vergessen Speicher freizugeben => **Memory Leaks** (Speicher Leck).
  - wird ein Objekt zu früh freigegeben => **Dangling Pointers** (baumelnde /ungültige Zeiger).
- Lösung: **automatische Freispeichersammlung**:
  - nicht mehr adressierbare Blöcke automatisch identifizieren und freigeben.
  - Entweder für ein einzelnes Programm oder systemweit,
  - Beispiele: Java, .NET, Oberon, ...

- Voraussetzungen:
  - sämtliche Referenzen auf einen Speicherblock müssen auffindbar sein.
  - typsichere Sprache dringend empfohlen.
- Aufrufen der GC:
  - implizit durch das BS,
  - explizit durch den Programmierer,
  - oder bei Bedarf, wenn der Speicher knapp wird.

D.10.1 Grundprinzip der Freispeichersammlung

Collector: sammelt Garbage.

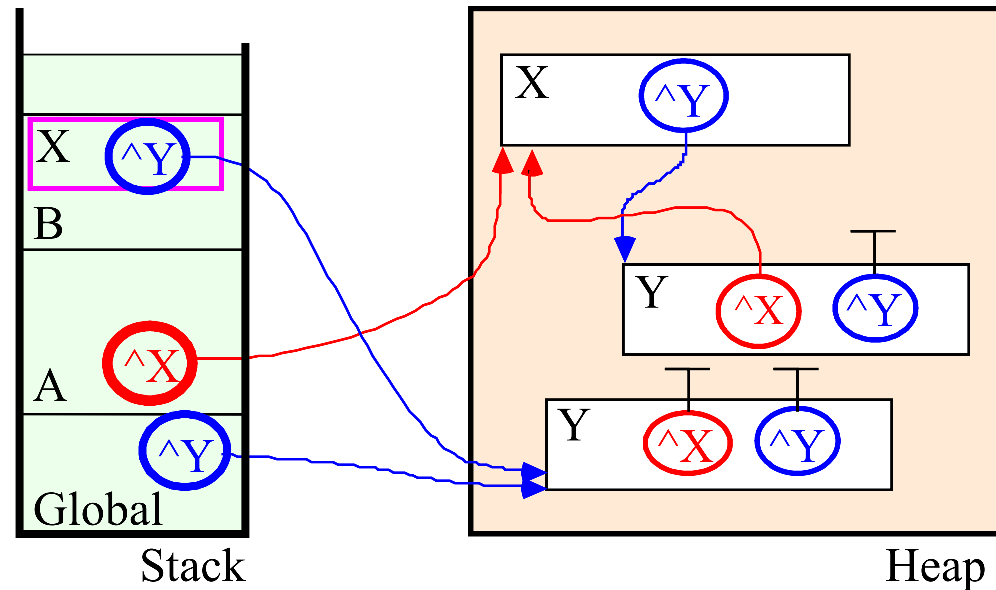
Mutator: alle Programme, welche den Heap ändern (mutieren).

- *1. Phase: Garbage Detection*
  - Erkennung von referenzierten und nicht mehr referenzierten Objekten.
- *2. Phase: Garbage Reclamation*
  - Freigabe des Speichers von nicht mehr referenzierbaren Objekten.
- Nicht mehr referenzierbare Objekte:
  - Es existiert kein Pfad zwischen dem Objekt und der Menge der Wurzelzeiger/-objekte.
- **Wurzelzeiger** (Root-Set):
  - globale Variablen (Klassenvariablen in Java),
  - lokale Variablen im Keller.
- Zirkulärer Garbage ist unangenehm:
  - jedes Objekt wird noch referenziert,
  - aber nicht vom Root-Set aus erreichbar.
  - Zyklus muss erkannt & aufgebrochen werden.

Betriebssysteme, Sommer 2010, ©P. Schulthess, VS, Ulm University

D.10.2 Compilerunterstützung

- Normalerweise weiss nur der Compiler, wo Zeigervariablen zu finden und nachzuverfolgen sind.
- Referenzen in einem Block müssen identifizierbar sein.
- Benötigt wird:
  - Offset und Typ der globalen Zeigervariablen,
  - Offset und Typ der lokalen Zeigervariablen,
  - Identifikation von Kellerrahmen,
  - Zeigerfelder in dynamisch allozierten Records bzw. Instanzen.
  - Evt. müssen auch die Registerinhalte beachtet werden.
- Vereinfachung durch doppelköpfiges Layout für Speicherblöcke in Plurix oder Rainbow.

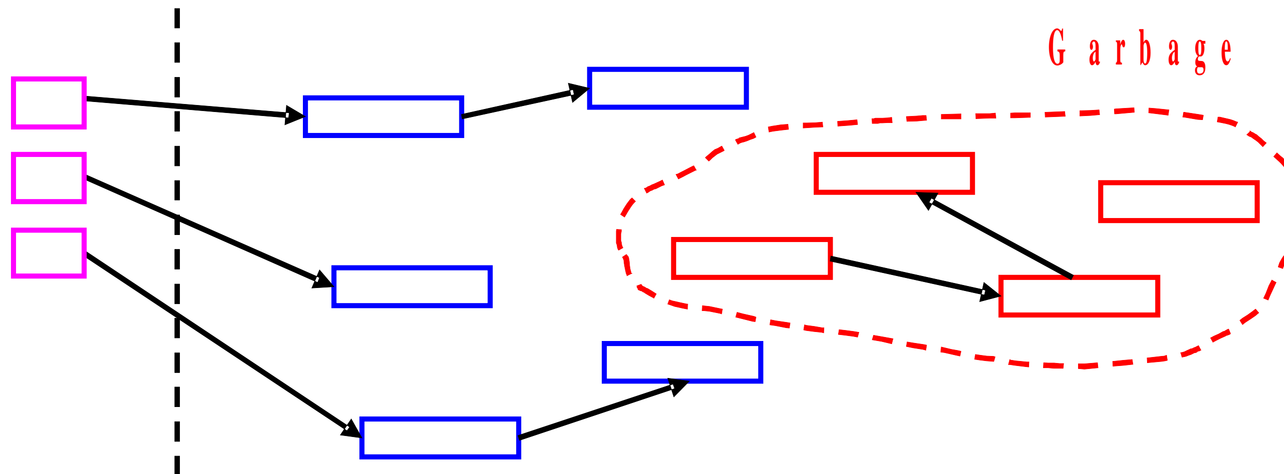


Symboltabelle

Global Frame	– – – – – –^Y – –
Proc-A Frame	– –^X – – – – – –
Proc-B Frame	–^Y – – – – – – –
Record-X Type	– – – –^Y – – – –
Record-Y Type	–^X – – – –^Y – –
…	

D.10.3 Mark & Sweep

- Algorithmus markiert alle noch erreichbaren Blöcke im Heap:
- Ausgehend von einer Menge von **Wurzelzeigern (Root-Set)** werden alle noch aktiven (live) Zeiger und deren Objekte gefunden & markiert.
- Nicht markierte Blöcke sind dann frei (bzw. Garbage) und können eingesammelt werden.
- Erforderliche Symbol- und Typentabelle wird vom Compiler erzeugt.



Pseudo-Code für Mark & Sweep Algorithmus:

```
für jeden Wurzelzeiger zgr:
    Markiere( zgr );

für jedes Objekt obj, für das gilt  obj.mark == 0:
    Speicherfreigabe( obj )

Markiere(parm):
    wenn parm.mark == 1 dann beende Prozedur
    parm.mark := 1;
    für jedes von parm referenzierte Objekt obj:
     Markiere( obj )
```

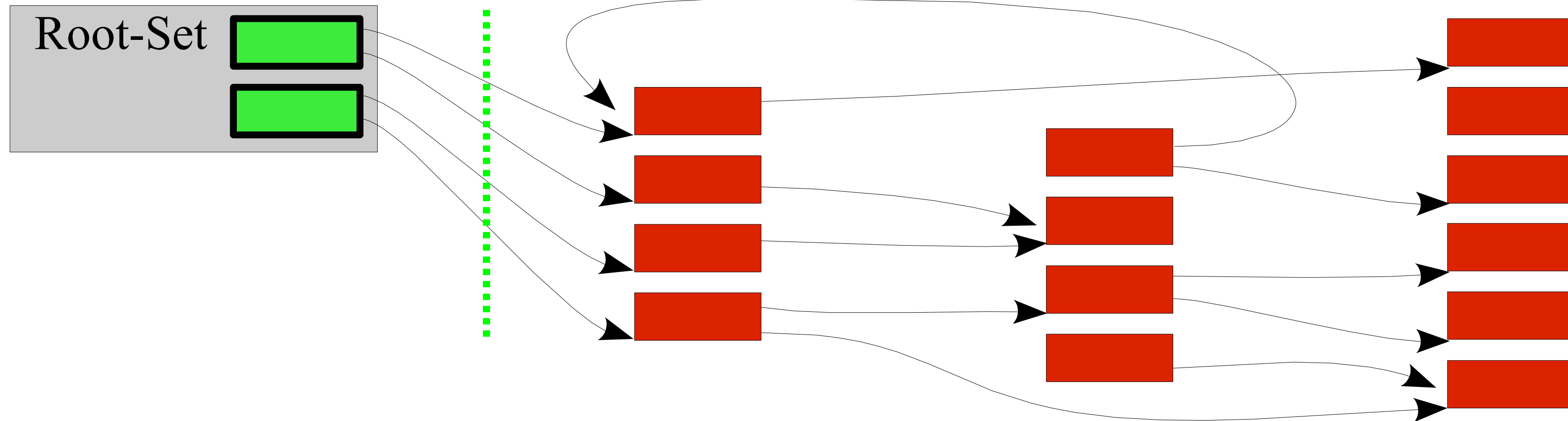
- Vorteil von Mark & Sweep: Garbage-Zyklen werden erkannt.
- Nachteil 1: Markierungsphase muss in einem Stück zu Ende laufen.
  - Unter Umständen würde sonst ein Zeiger in einem bereits abgearbeitetem Objekt verändert,
  - Eventuell würden dann noch benutzte/neue Objekte fälschlicherweise eingesammelt.
- Nachteil 2:
  - Heap wird durch Mark & Sweep Freispeichersammlung nicht kompaktiert.
  - Funktion zum Markieren beinhaltet unter Umständen tiefe Rekursion
    → unter Umständen ist viel Speicherplatz im Keller notwendig.

D.10.4 Inkrementelles Mark & Sweep

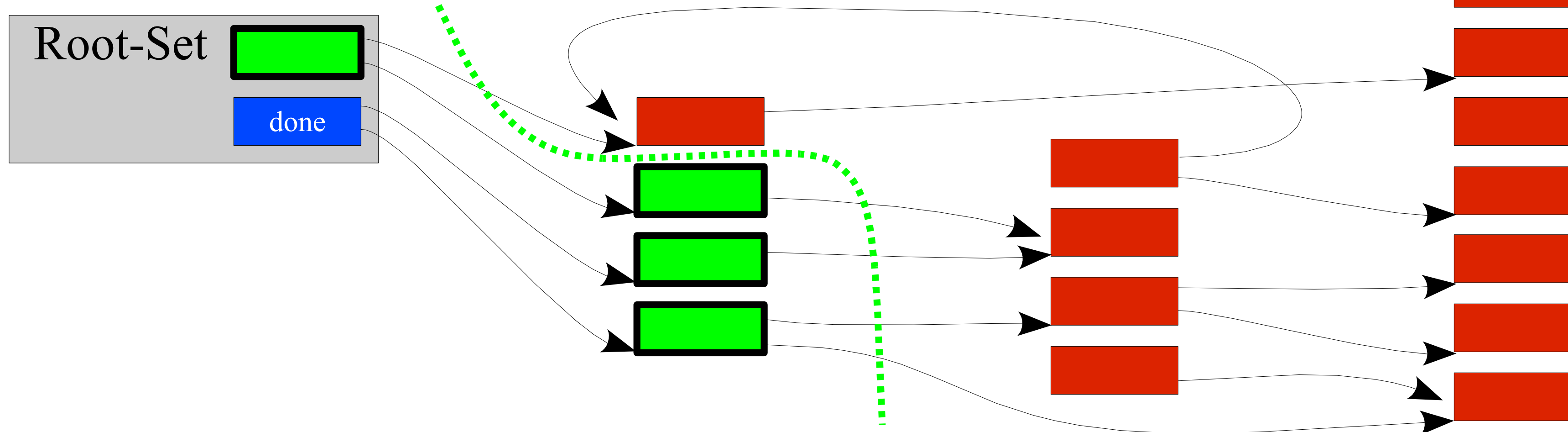
- Nebenläufiges Mark & Sweep nach einer Idee von E. W. Dijkstra, 1978.
- Objekte werden mit drei Farben markiert:
  rot: Objekt wurde noch nicht besucht (weiss).
  blau: Objekt wurde schon komplett besucht (schwarz).
  grün: Objekt wurde bereits besucht, aber noch nicht alle seine Nachfolger (pendent, grau).
- Ablauf:
  - Alle Objekte werden **rot** markiert,
  - Die Objekte im Root-Set werden **grün** markiert,
  - Alle **grünen Objekte** werden z.B. in einer Liste verlinkt,
  - Durch ein **grünes Objekt** referenzierte **rote Objekte** werden grün gefärbt,
  - Nachdem alle Referenzen eines **grünen Objektes** verfolgt wurden, wird es **blau** gefärbt,
  - sobald die **Menge der grünen Objekte** leer ist, terminiert das Verfahren,
  - alle **roten Objekte** sind jetzt Garbage.
- Integritätsbedingung:

bereits untersuchte Objekte, dürfen keine Zeiger auf noch **nicht untersuchte Objekte** beinhalten.

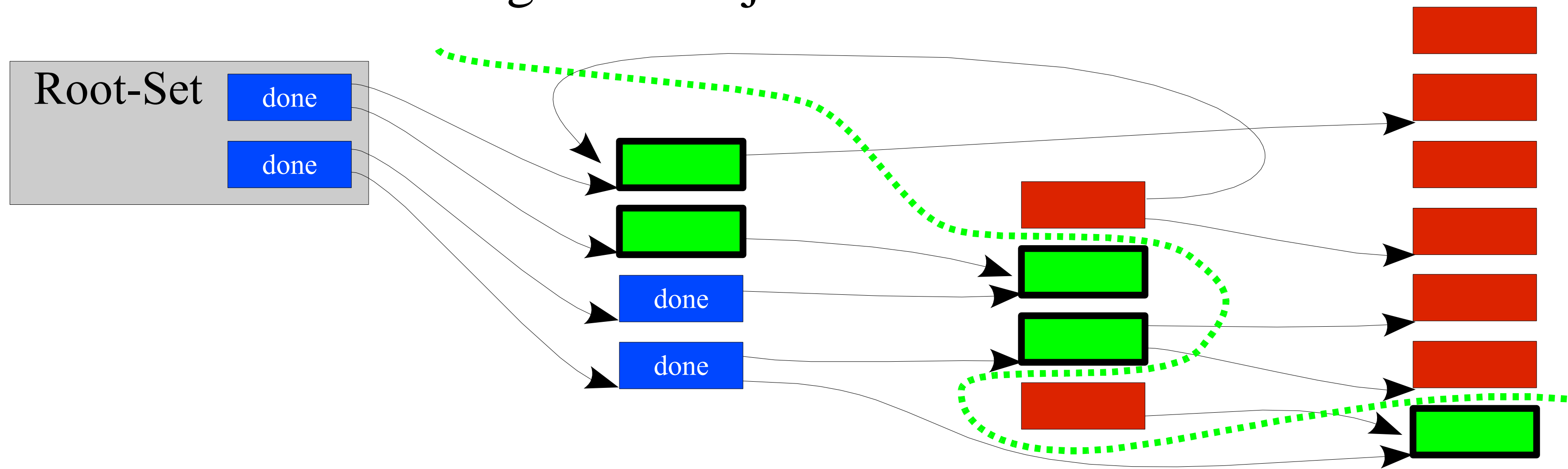
- Alle Objekte unbesucht. Nur Root/Set pendent:



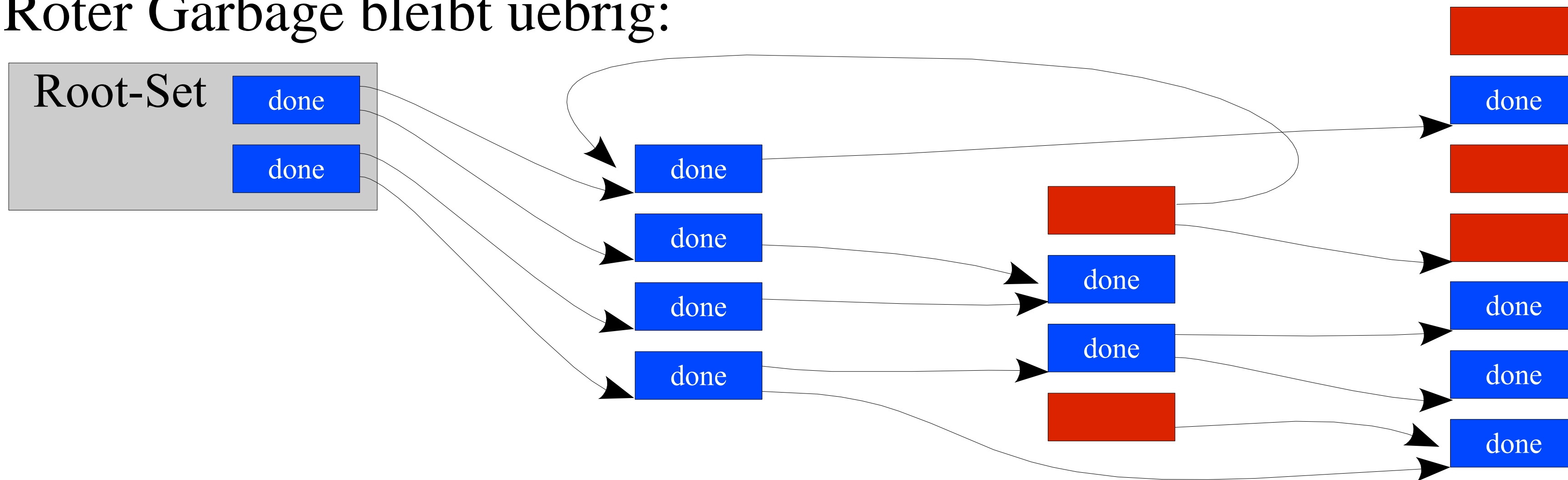
- Collector schiebt eine Front grüner Objekte vor sich her:



- Noch mehr blaue und gruene Objekte:

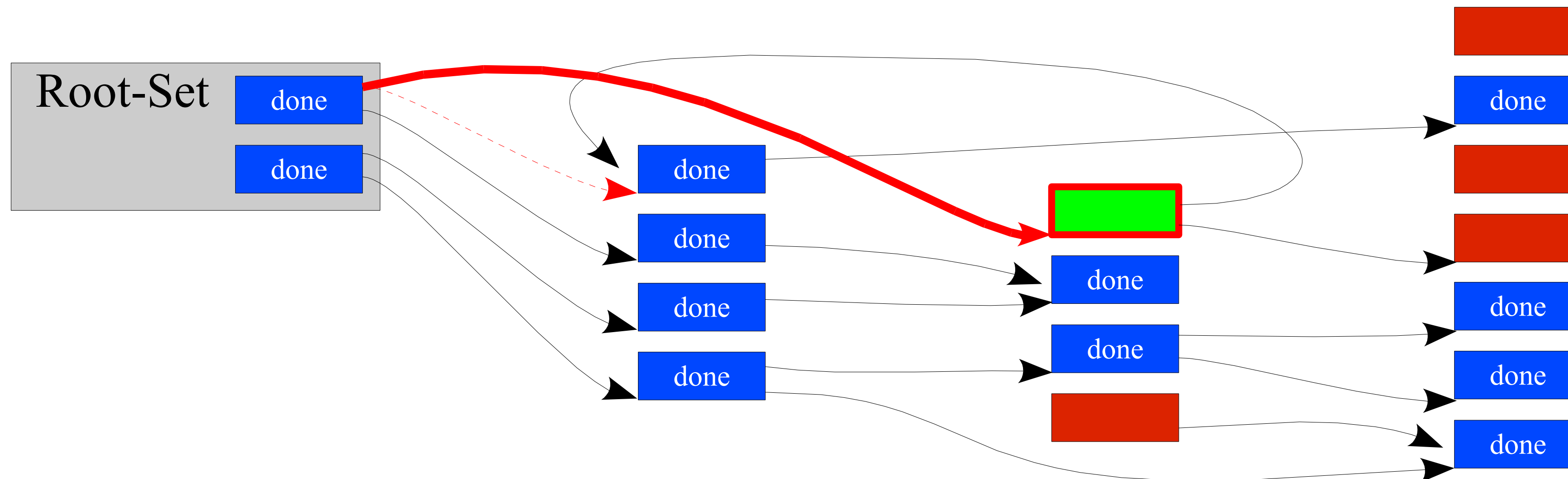


- Roter Garbage bleibt uebrig:



Betriebssysteme, Sommer 2010, ©P. Schulthess, VS, Ulm University

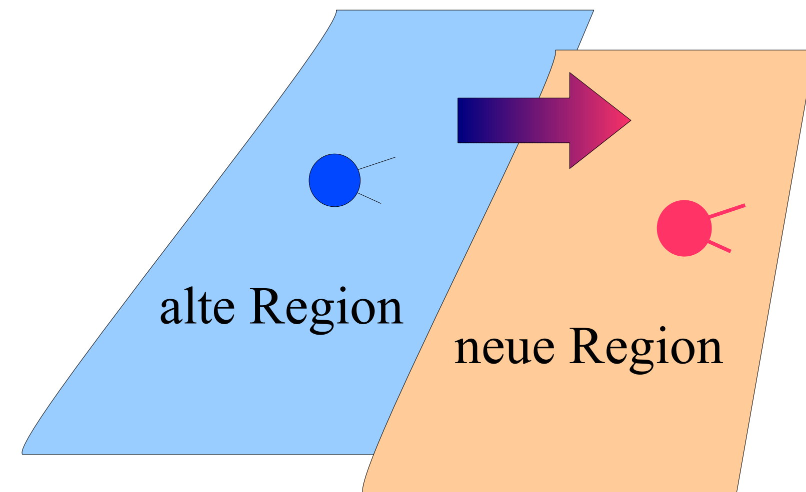
- Zeigerzuweisung durch einen Mutator wird ueberwacht:
  - ein vorher rotes Element wird anschliessend gruen gefaerbt,
  - Die Überwachung von Zeigerzuweisungen zur Laufzeit ist teuer,
  - Evtl. fuegt der Compiler bei Zeigerzuweisungen den Aufruf einer Laufzeitroutine ein.



- Geht eine Referenz auf ein grünes Objekt verloren, so verbleibt das Objekt in der grünen Liste.
- Erzeugen die Mutatoren schneller grüne Objekte, als der Collektor diese wieder blau einfärbt, so wird das Verfahren nie fertig.

D.10.5 Kopierende Freispeichersammlung

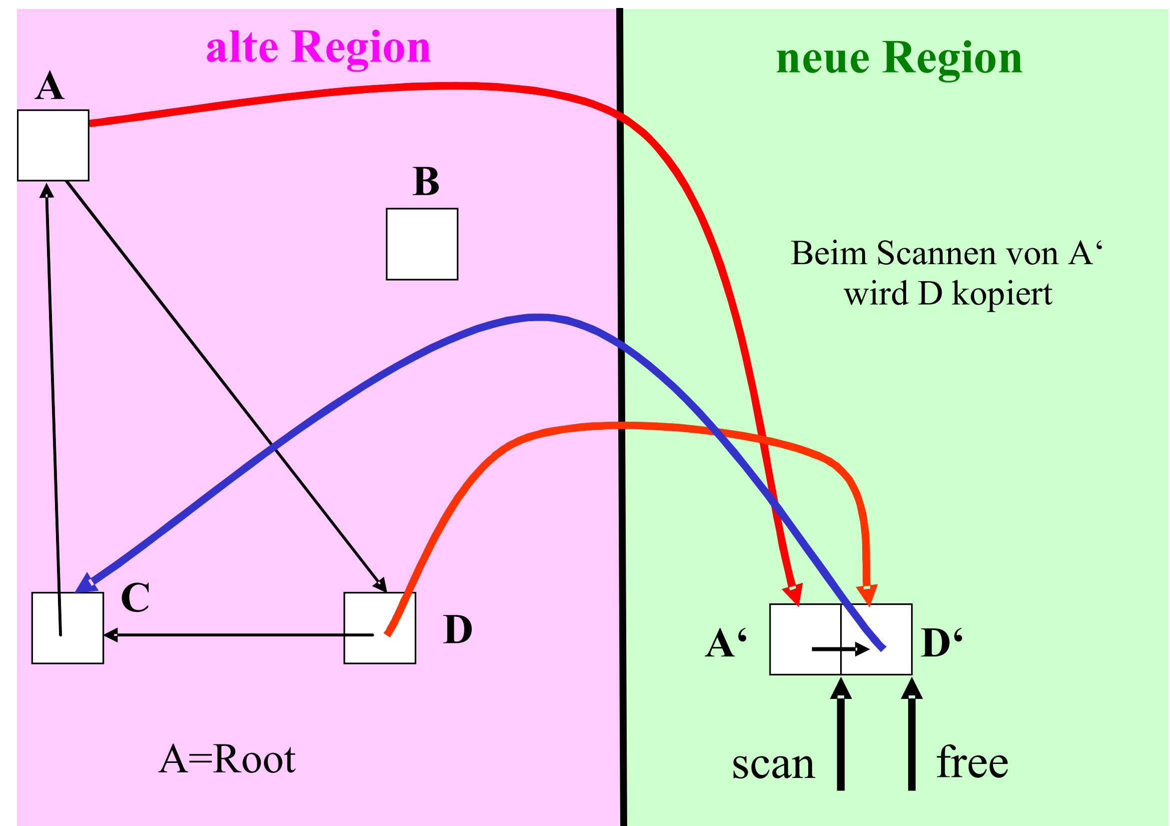
- Erste Implementierung Marvin Minsky, (1963, LISP).
- Halde in zwei Regionen alt & neu unterteilt.
- Alle vom Rootset aus erreichbaren Objekte werden rekursiv in die **neue Region** kopiert.
- Garbage verbleibt in **alter Region**.
- Beim nächsten GC-Aufruf tauschen die alte und neue Region ihre Rollen.
- Vorteil:
  - Heap wird automatisch kompaktifiziert.
  - Zyklen werden eliminiert.
- Nachteile:
  - Es ist teuer, viele kleine Objekte zu kopieren.
  - logischer Adressraum wird halbiert.
  - Kopieren muss atomar erfolgen.
    → nicht inkrementell



Betriebssysteme, Sommer 2010, ©P. Schulthess, VS, Ulm University

D.10.6 Inkrementell kopierender Freispeichersammler

- Pro Aufruf der GC wird nur eine vorgegebene Anzahl von Objekten kopiert.
- Es muss nicht für längere Zeit die ganze Verarbeitung gestoppt werden.
- Iterative Lösung nach Cheney:
  - Neue Region wird fortlaufend gefüllt (~ Queue).
  - **scan**-Zeiger: Objekte bis hier sind komplett abgearbeitet.
  - **free**-Zeiger: Objekte zwischen scan- und free-Zeiger sind kopiert, haben aber noch **Zeiger in die alte Region**.
  - Kopierte alte Obj. **verweisen auf ihre Kopie** (z.B. A = Root-Variable).
- Beispiel:
  - Momentaufnahme =>
  - A = Root-Variable

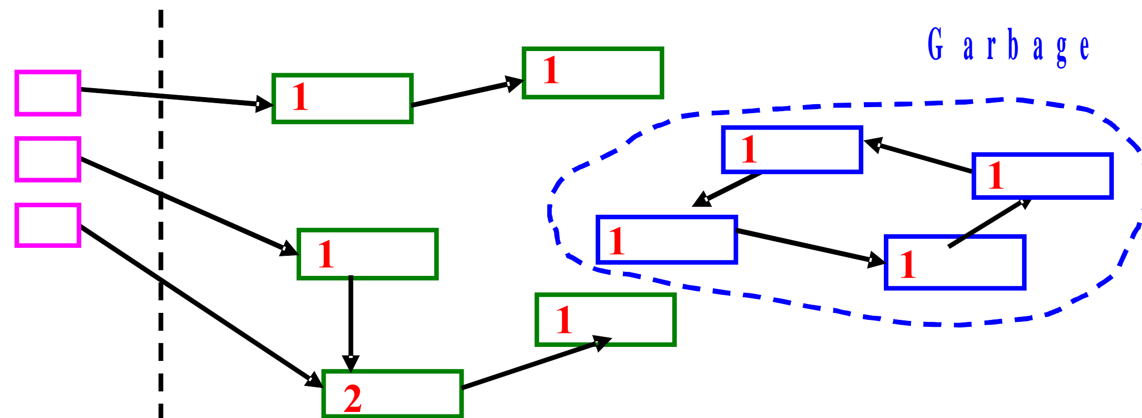


- Integritätsbedingung:
  - Komplett abgearbeitete Objekte, dürfen nicht mehr auf Objekte in alter Region verweisen.
- Terminierung einer Kopieroperation:
  - Ein Kopierschritt terminiert, wenn der scan-Zeiger auf den free-Zeiger trifft.
- Zeigerüberwachung:
  - Der Mutator darf keine alten Zeigerwerte in ein fertig kopiertes Objekt zuweisen,
  - Wird eine derartige Zuweisung versucht, so muss das referenzierte Objekt sofort kopiert werden
  - → Zeigerzuweisungen müssen also geeignet überwacht werden,
  - z.B. fügt der Compiler für jede Zeigerzuweisung den Aufruf einer Laufzeitroutine ein.
  - Die Überwachung von Zeigerzuweisungen ist teuer.

Betriebssysteme, Sommer 2010, ©P. Schulthess, VS, Ulm University

D.10.7 Reference Counting

- Jeder Speicherblock wird durch einen versteckten Referenzzähler erweitert und speichert die Anzahl der Referenzen auf ein Objekt
- Ein Objekt 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 Referenzzähler erniedrigt.
- Vorteile:
  - inkrementelle Freispeichersammlung möglich,
  - Garbage wird sofort freigegeben.
  - einfach implementierbar.
- Nachteile:
  - Zyklen werden nicht erkannt.
  - Zeigerverwaltung erfordert den Aufruf einer Laufzeitroutine



D.10.8 Freispeichersammlung in Plurix und Rainbow

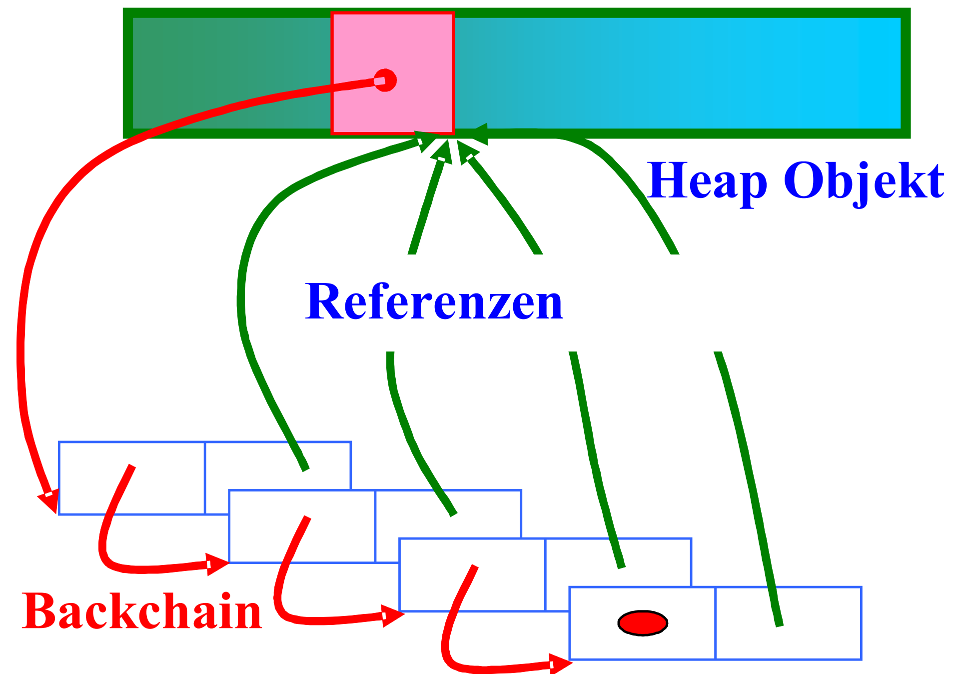
- Gemeinsame Charakteristiken:
  - Sprachbasierter Ansatz in Java.
  - Schnelles Cluster-Betriebssystem für PCs,
  - Freispeichersammlung arbeitet auf verteiltem gemeinsamen Speicher.
  - Objekte sind verschiebbar, da jederzeit alle Referenzen auf ein Objekt bekannt sind.
- Freispeichersammlung im verteilten Speicher DSM):
  - Markieren von Objekten durch den Kollektor erzeugt Invaliderungen,
  - Eine Referenz auf ein Objekt zu verfolgen bedeutet Netzwerk-kommunikation.
- Buchführungstechniken:
  - Backchain / Backlinks).
- Plurix:
  - Verteiltes 32-Bit Betriebssystem (Uni Ulm).
- Rainbow:
  - Abgeschwächtes Konsistenzmodell für Anwenderprogramme,
  - Verteiltes 64-Bit Betriebssystem,
  - Anwendungsfall Wissenheim.

Betriebssysteme, Sommer 2010, ©P. Schulthess, VS, Ulm University

D.10.9 Backchain

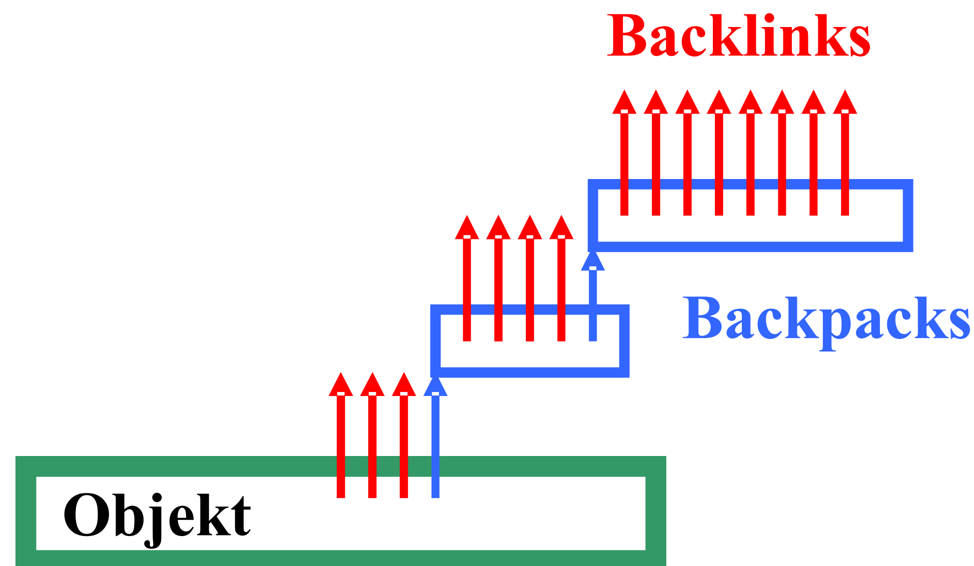
- Ursprünglich in Plurix verwendet.
- Jeder Speicherblock führt eine **Backchain**, eine Liste der Referenzen, welche auf ihn zeigen.
  - Im Falle einer Allozierung (new) wird der zugeordnete Zeiger in die Liste eingetragen.
  - Wird eine Zeigervariable freigegeben, wird sie aus der Backchain entfernt.
  - Wird ein gültiger Zeigerinhalt einer anderen Referenz zugewiesen (assigned), so wird auch dieser Zeiger eingetragen.

- Freispeichersammlung sammelt Blöcke mit leerer Backchain ein.
- Im Prinzip eine Abwandlung der Reference Counting Technik.



D.10.10 Backlinks:

- Weiterentwicklung der Backchain.
- **Backlinks:**
  - Normalerweise genügen 2 Backlinks direkt im Objekt drin,
  - Mehrzahl der Backlinks werden in-line untergebracht,
- **Backpacks:**
  - Weitere Backlinks liegen in separaten Heap-Blöcken sog. Backpacks,
  - **Progressiv wachsende Containergrösse für die Backpacks.**
- Objekte ohne Backlink sind Garbage.



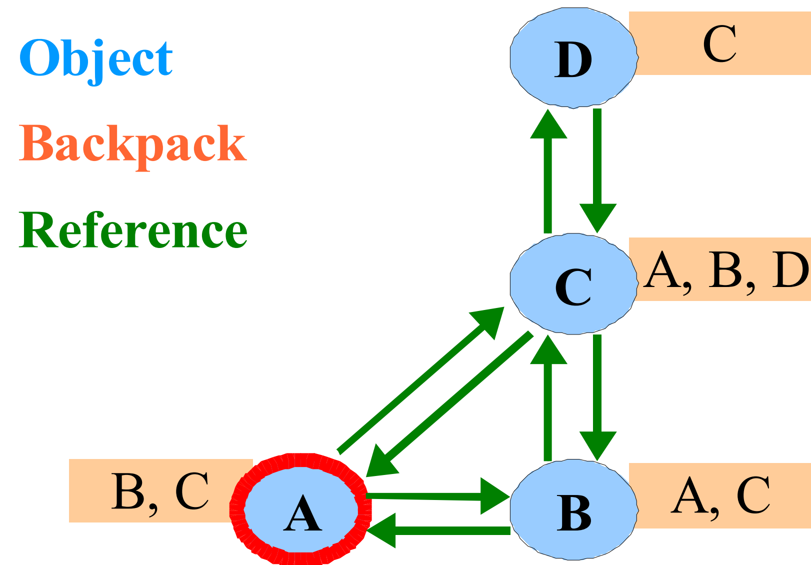
D.10.11 Zyklenerkennung mit Hilfe der Backlinks:

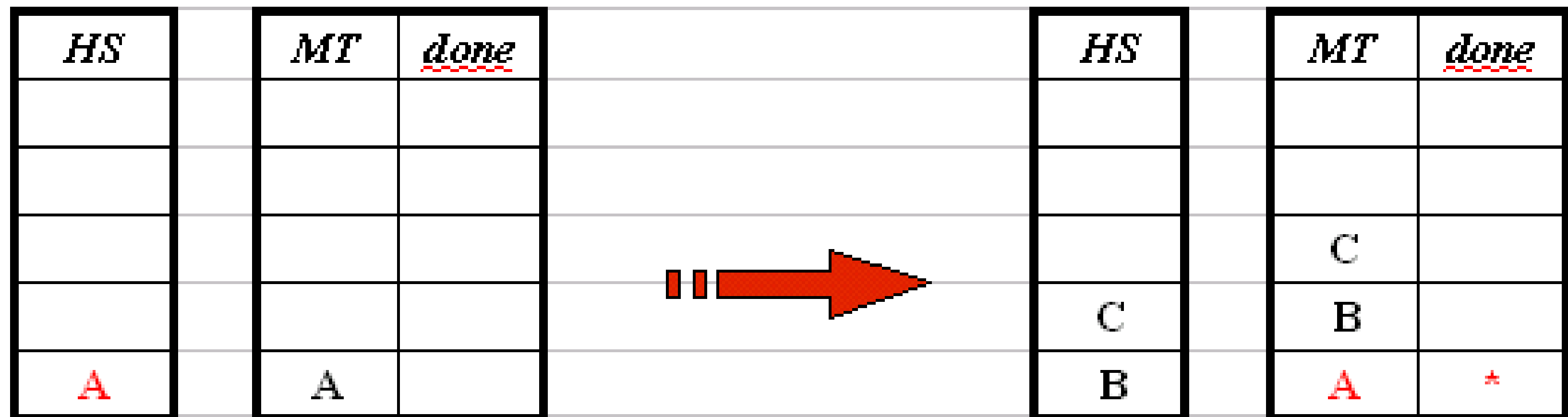
- Alternativer Markierungsansatz:
  - Alle Objekte, die im Namensdienst registriert sind, sind Root-Objekte (durch entsprechendes Bit im Header gekennzeichnet).
  - **Nicht** ausgehend vom Root-Set alle erreichbaren Objekte identifizieren.
  - Beginnend bei einem Objekt mit Hilfe der Backlinks prüfen, ob ein Root-Objekt erreichbar ist (Hüllenbildung über Backlinks).

- Zwei Hilfstabellen:
  - Je nach Größe erlauben die Tabellen grössere oder kleinere Zyklen zu erkennen.
  - Sind die Tabellen zu klein, so können sie dynamischvergrössert werden.
- Marking Table (MT):
  - speichert bereits besuchte Objekte → Schleifen vermeiden
- Handle Stack (HS):
  - enthält noch zu prüfende Backlinks

Beispiel: A wird untersucht

- A liegt oben auf Handle Stack
- A vom Handle Stack holen
- Falls A Root-Objekt:
  - kein Garbage → fertig
- Backlinks von A, die nicht in MT sind, auf Handle Stack ablegen
- A in MT als behandelt markieren.
- Falls HS leer ist → Zyklus erkannt

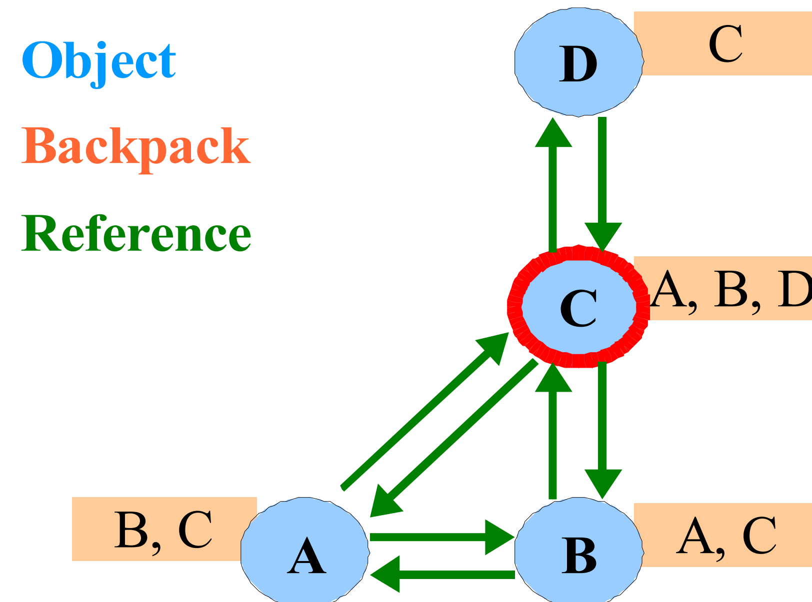




Betriebssysteme, Sommer 2010, ©P. Schulthess, VS, Ulm University

Beispiel: C wird untersucht

- C vom Handle Stack holen
- Falls C Root-Objekt:
- kein Garbage → fertig
- Backlinks von C, die nicht in MT sind, auf Handle Stack ablegen
- C in MT als behandelt markieren.
- Falls HS leer ist → Zyklus erkannt



Select whole table

HS
C
B

MT	done
C	
B	
A	*

→

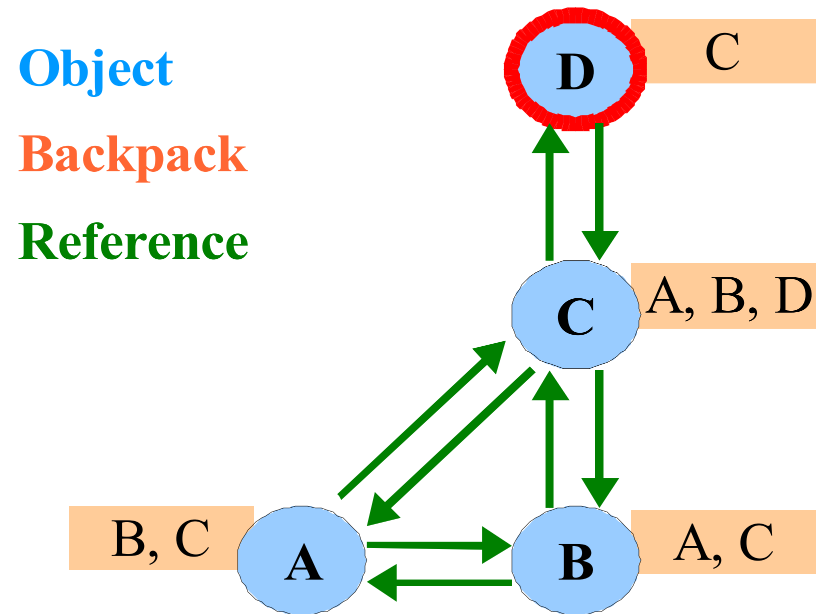
HS
D
B

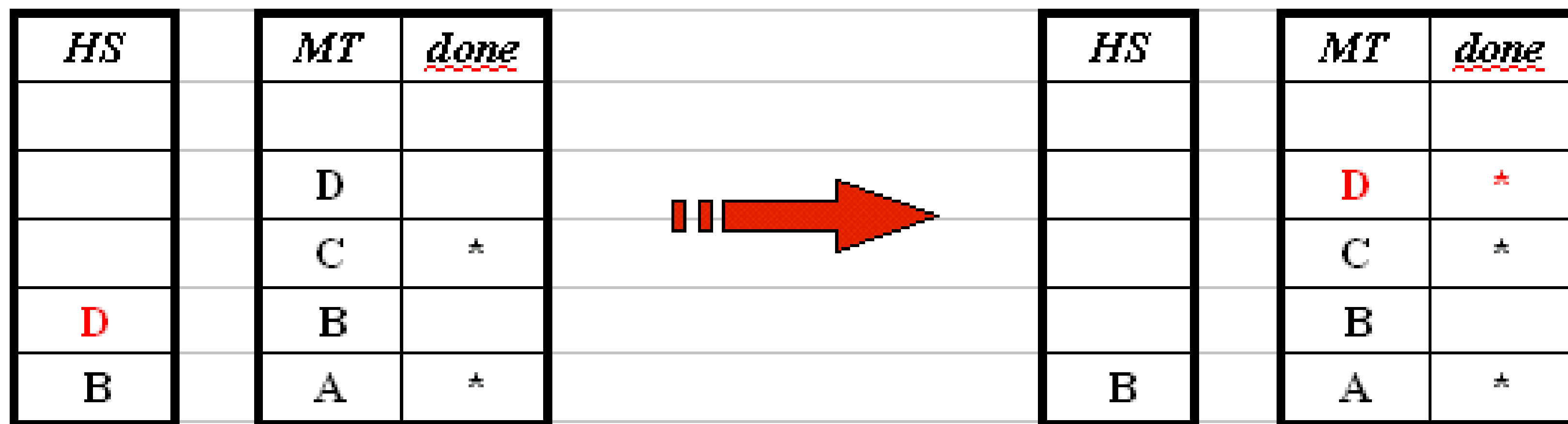
MT	done
D	
C	*
B	
A	*

Betriebssysteme, Sommer 2010, ©P. Schulthess, VS, Ulm University

Beispiel: D wird untersucht

- D vom Handle Stack holen
- Falls D Root-Objekt:
- kein Garbage → fertig
- Backlinks von D, die nicht in MT sind, auf Handle Stack ablegen
- D in MT als behandelt markieren.
- Falls HS leer ist → Zyklus erkannt

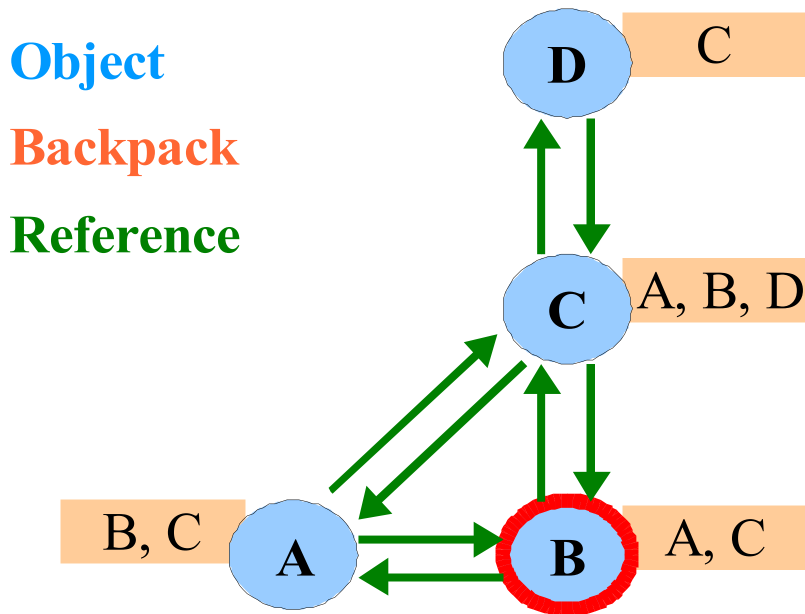




Betriebssysteme, Sommer 2010, ©P. Schulthess, VS, Ulm University

Beispiel: B wird untersucht

- B vom Handle Stack holen
- Falls B Root-Objekt:
- kein Garbage → fertig
- Backlinks von B, die nicht in MT sind, auf Handle Stack ablegen
- B in MT als behandelt markieren.
- **HS ist leer und kein Root-Objekt gefunden → Zyklus erkannt**



HS
D
B

MT	*done*
D	
C	*
B	
A	*

→

HS
B

MT	*done*
D	*
C	*
B	
A	*

Betriebssysteme, Sommer 2010, ©P. Schulthess, VS, Ulm University

- Nicht zyklischer Garbage ist bequem inkrementell behandelbar.
- Zyklischer Garbage:
  - Keine verteilte Bestimmung aller Wurzelzeiger vorab notwendig,
  - Sondern Rückwärtsverweise (Backlinks) prüfen, ob eine Root-Variable erreichbar ist,
  - Markierung in extra Tabelle → Objekte werden nicht invalidiert.
- Optimierung für verteilten Fall:
  - Zunächst keine Referenzen auf Objekte auf andere Rechner verfolgen
  - Sondern nur lokal arbeiten.

Betriebssysteme, Sommer 2010, ©P. Schulthess, VS, Ulm University

D.11 Zusammenfassung zur Speicherverwaltung

- Partitionierung des Hauptspeichers für Mehrprogrammbetrieb:
  - statisch: interne Fragmentierung,
  - dynamisch: externe Fragmentierung.
- Inhalt einer Partition: Heap, Stack, Code & globale Variablen
- Auslagern v. Speicher: Overlay und Swapping, Virtueller Speicher(später).
- Belegungsdarstellung: Bitmap, Verkettung, lin. Heap, Buddy-System.
- Auswahlstrategien: First-, Best-, Worst-Fit, Buddy ...
- Automatische Freispeichersammlung
  - typsichere Sprache empfohlen, Zeiger müssen gefunden werden (Symboltabelle),
  - Compiler-Unterstützung zur Überwachung von Referenzzuweisungen,
  - verschiedene Verfahren: blockierend vs. inkrementell, markierend vs. Kopierend
  - Typischer Ablauf:
    - 1. Phase: Suche nach erreichbaren Objekten, ausgehend vom Root-Set
    - 2. Phase: Freigabe nicht mehr referenzierter Objekte