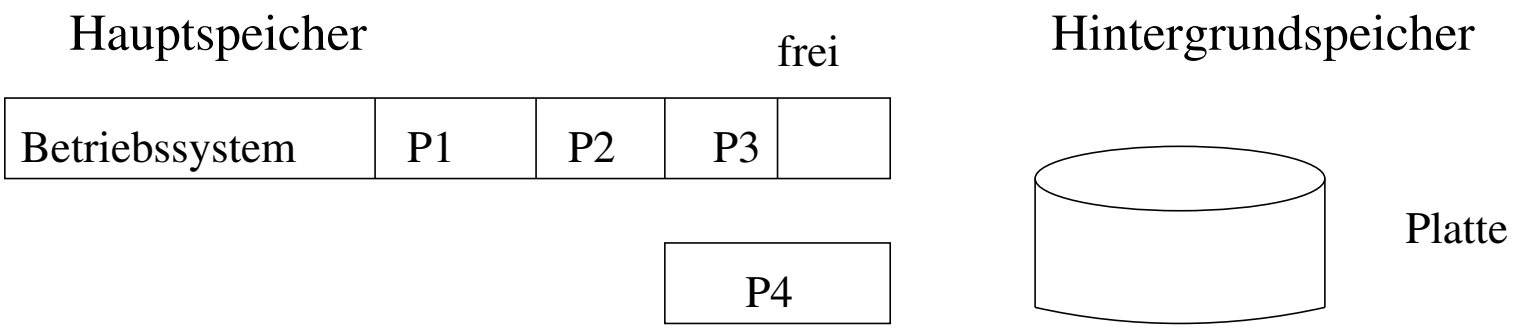
"Ideally one would desire an indefinitely large memory capacity such that any particular ... word would be immediately available ... We are ... forced to recognize the possibility of constructing a ***hierarchy of memories****, each of which has greater capacity than the preceding but which is less quickly accessible."*

A. W. Burks, H. H. Goldstine und J. von Neumann, 1946 (gefunden im Patterson/Hennessy)



Wiederholung: Speicherhierarchie

1. Register
2. Cache
3. Hauptspeicher/Primärspeicher:
   Flüchtiger Speicher (volatile memory): Wenn die Stromzufuhr unterbrochen ist, gehen alle Daten verloren.
   $\Longrightarrow$ geeignet für Programme in Ausführung

Random-Access Memory (RAM): is a form of computer memory that can be read and changed in any order, typically used to store working data and machine code. A random-access memory device allows data items to be read or written in almost the same amount of time irrespective of the physical location of data inside the memory.

Technologien:

1. SRAM (Static RAM):
   Behält seine Dateninformation, solange die Betriebsspannung anliegt.
2. DRAM (Dynamic RAM) (seit 1975): Im Gegensatz zu SRAM benötigt der DRAM ein periodisches (dynamisches) Auffrischen (Re-Writing) zur Vermeidung von Datenverlust in jeder Datenzelle
   Zugriffszeit DRAM: ca. 80 Nanosekunden $\Longrightarrow$ 100 000-mal schneller als Festplatte

4. Sekundärspeicher (Hintergrundspeicher):
   nichtflüchtiger (non-volatile) Speicher, der dauerhaftes Speichern von Daten ermöglicht

Beispiel:

- magnetische Festplatten (seit 1965):
  - \+ Kosten pro Megabyte geringer als DRAM
  - \- Zugriffszeiten aufgrund der mechanischen Komponenten deutlich länger als bei DRAM: Festplatte 5-15 ms
- eingebettete Systeme: FLASH-Speicher: nichtflüchtiger Halbleiterspeicher
  Speicherkarten für Digitalkameras, Mobiltelefone
  auch als Solid-State-Drives (SSD) als Alternative zu den bisherigen magnetischen Festplatten

Definition 23:

Unter Swapping versteht man das Verlagern von Prozessen zwischen Hauptspeicher und Hintergrundspeicher.

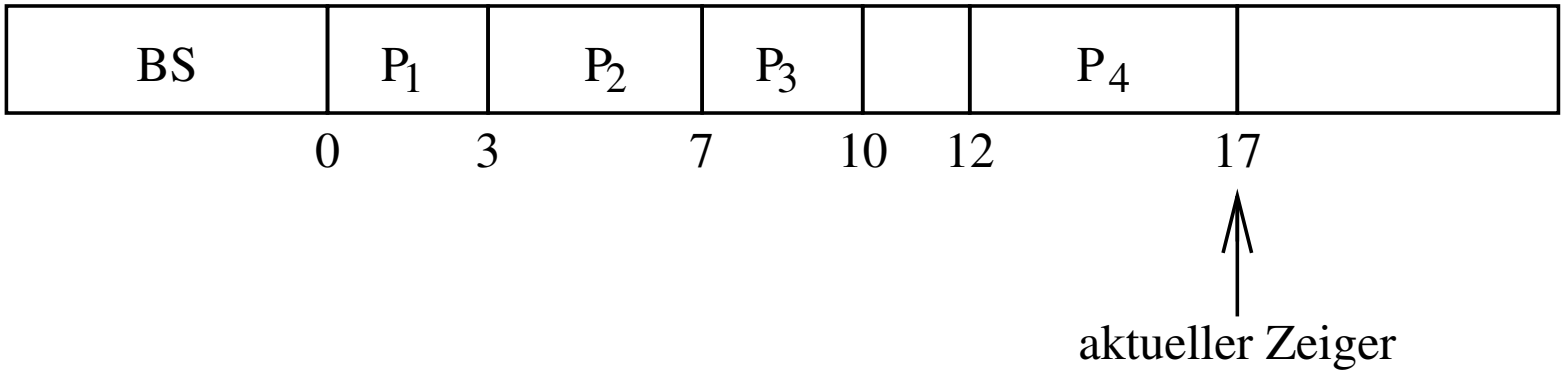
Idee:

- Auslagern (swap out) blockierter Prozesse auf den Hintergrundspeicher.
- Einlagern (swap in) nach Übergang in den Zustand „bereit".

Probleme:

- Aufwand für Ein- und Auslagern,
- Belegungsstrategie beim Einlagern.

Beispiel:



Ablauf:
Prozeß P_2 terminiert
Prozeß P_5 der Größe 1 einlagern

verkettete Liste mit Einträgen:

Belegungsstrategien

First Fit: Der Prozeß wird in die erste hinreichend große Lücke eingelagert (Liste von vorn durchlaufen).

Next Fit: Analog zu First Fit von der aktuellen Zeigerposition aus.

Best Fit: Der Prozeß wird in diejenige Lücke eingelagert, die den kleinsten Rest ergibt.

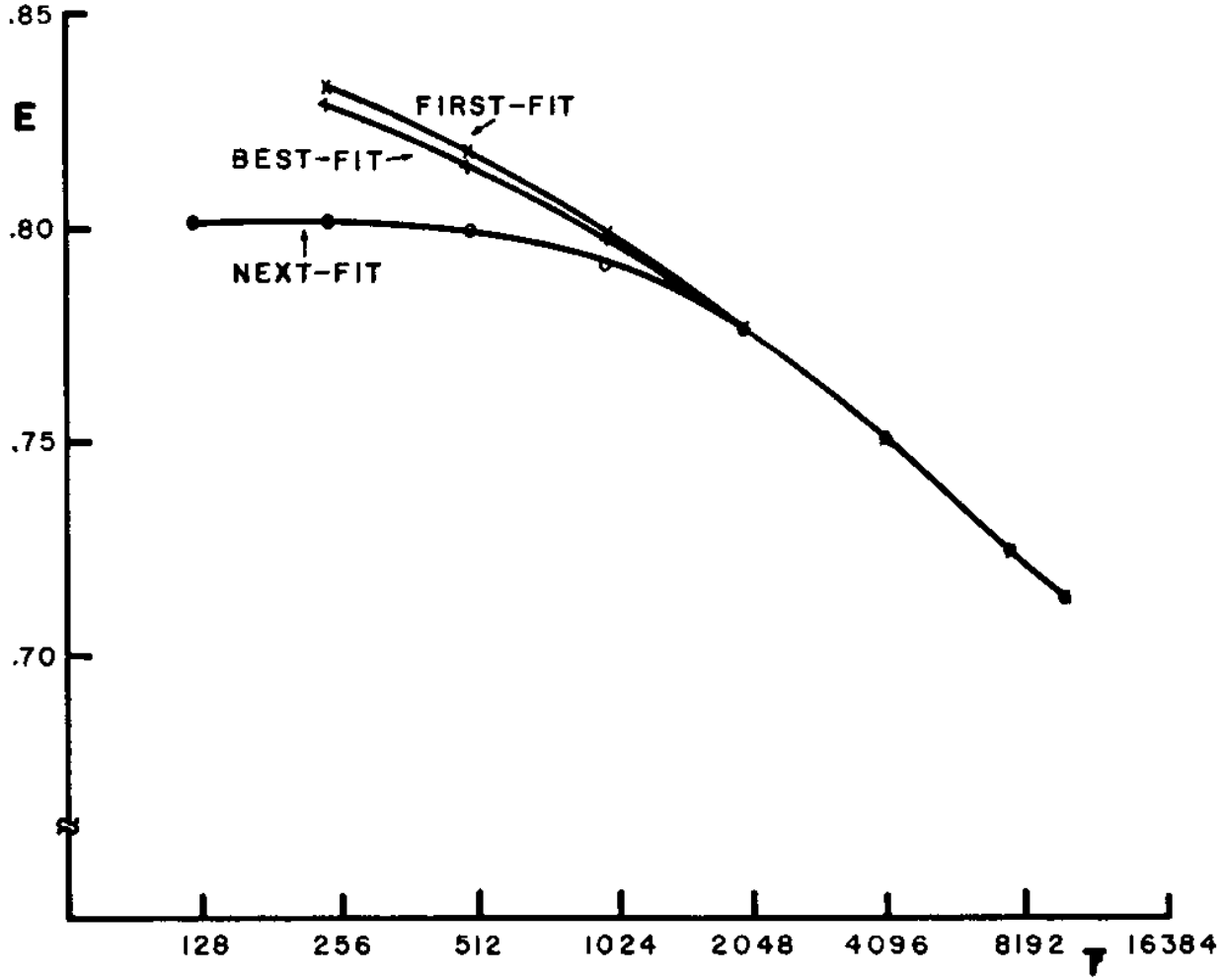
Worst Fit: Der Prozeß wird in die größte Lücke eingelagert.

Ziel: Externe Fragmentierung (Externer Verschnitt), d.h. ungenutzten Hauptspeicherplatz, minimieren!

Literatur:

John E. Shore: *On the External Storage Fragmentation Produced by First-Fit and Best-Fit Allocation Strategies*, Communications of the ACM, Vol. 18, Number 8, 1975.

Fig. 1.



entnommen: Carter Bays: *A comparison of next-fit, first-fit, and best-fit*, Communications of the ACM, Volume 20 Issue 3, March 1977, pages 191 - 192.

Buddy-Verfahren (Knuth '73):

n Listen für Lücken der Länge $1, 2, 4, 8, \ldots, 2^{n-1}$

Belegen:

1. Runde die Größe der Anforderung auf die nächste Zweierpotenz 2^k auf
2. Suche die erste nicht-leere 2^m-Liste mit $m \geq k$
3. Falls $m = k$: Wähle den ersten Block, andernfalls unterteile den ersten Block aus der 2^m-Liste!

Freigeben (Merge *Best Buddies*):

1. Falls sich der freigegebene Block auf seiner 2^k-Liste verschmelzen läßt, tue dies! Ggf. auf der 2^{k+1}-Liste wiederholen, usw.

$\Rightarrow$ reduziert externen Verschnitt!

aber: Es entsteht interner Verschnitt (interne Fragmentierung), da Prozeßgrößen i.a. keine 2er-Potenz sind.

Nachteil vom Swapping:

1. Fragmentierung

   Abhilfe:

   - Buddy-Verfahren $\Longrightarrow$ interne Fragmentierung
   - Kompaktifizierung: Verlagern von Prozessen im Hauptspeicher, um den zusammenhängenden freien Adreßraum zu vergrößern.
     $\Longrightarrow$ hoher Kopieraufwand!

2. Die Prozeßgröße wird durch die Hauptspeichergröße beschränkt!

Lokalitätsprinzip:= Prozesse greifen während eines vorgegebenen Zeitintervalls nur auf eine Teilmenge ihres Adreßraums zu. $\Longrightarrow$ Diese Teilmenge ist durch eine höhere Zugriffswahrscheinlichkeit gekennzeichnet.

Ursachen:

- sequentielle Instruktionsfolgen
- iterative Anweisungen (Schleifen)
- Prozeduraufrufe
- modulares Programmieren

Fazit: Prozesse können laufen, ohne daß sich der komplette Adreßraum im Hauptspeicher befindet $\Longrightarrow$ Gesamteinlagerung von Prozessen ist überflüssig!

Idee: Programme benutzen virtuelle Adressen!

- Der virtuelle und der physikalische Adreßraum werden in Blöcke gleicher Größe aufgeteilt.
  Üblich: Blöcke von 512 bytes – 8 KByte
  Ein Block im virtuellen Adreßraum heißt Seite (page), ein Block im physikalischen Adreßraum heißt Kachel bzw. Seitenrahmen (page frame).
  ! Häufig werden auch physikalische Blöcke "pages" genannt !

- Zur Laufzeit muß die virtuelle Adresse dann auf die physikalische Adresse im Hauptspeicher abgebildet werden.
  Die Seitentabelle realisiert die Zuordnung:

$$\text{Seite} \mapsto \text{Seitenrahmen}$$

Beispiel: Der Prozeß P besteht aus 5 Blöcken $A_1, A_2, ..., A_5$ der Größe 4K mit den Seitennummern $1, 2, ..., 5$.

Hauptspeicherbelegung:

Seitentabelle pro Prozeß:

Seitennr.	Kachelnr.
1	2
2	–
3	–
4	8
5	1

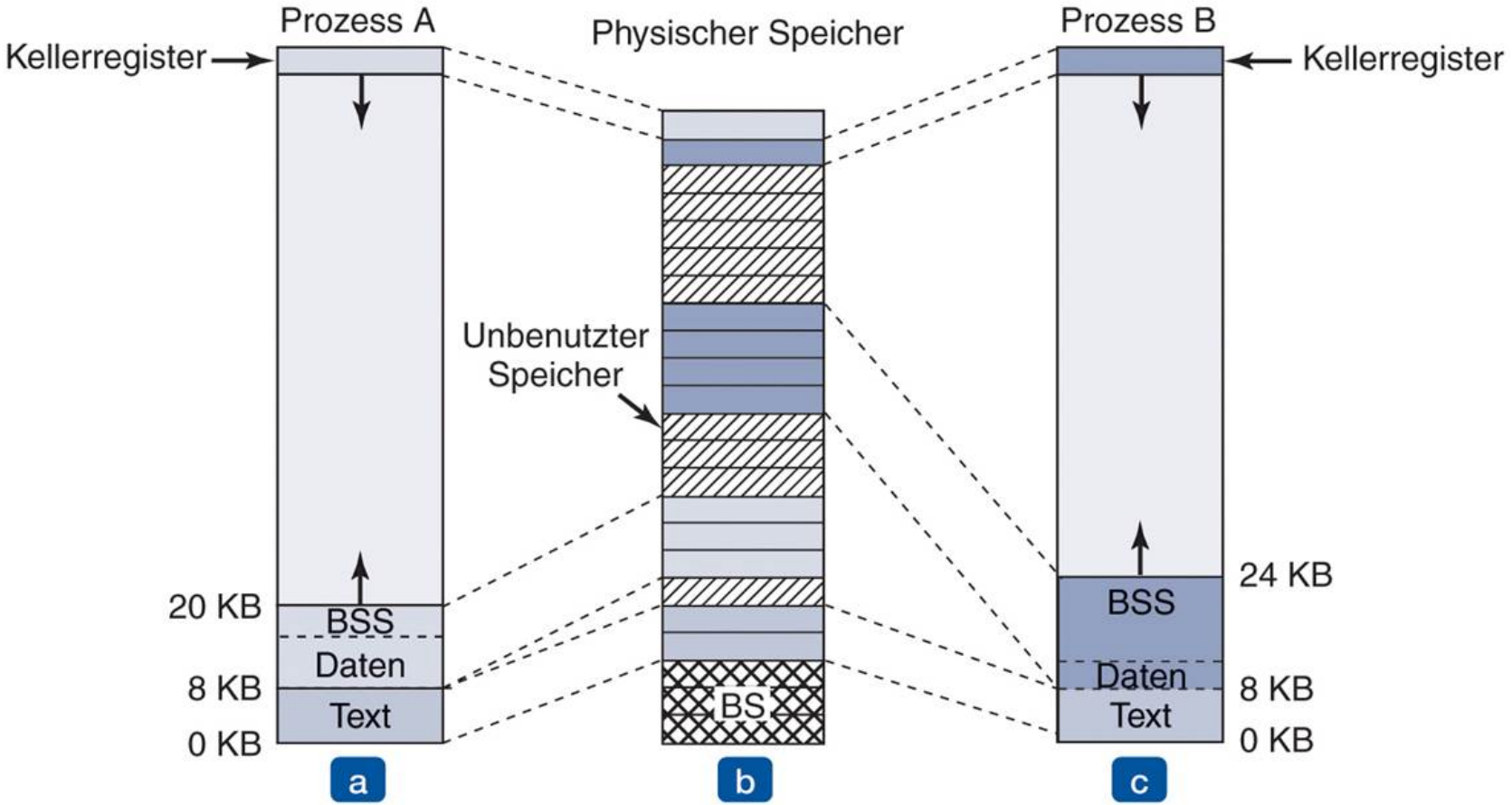


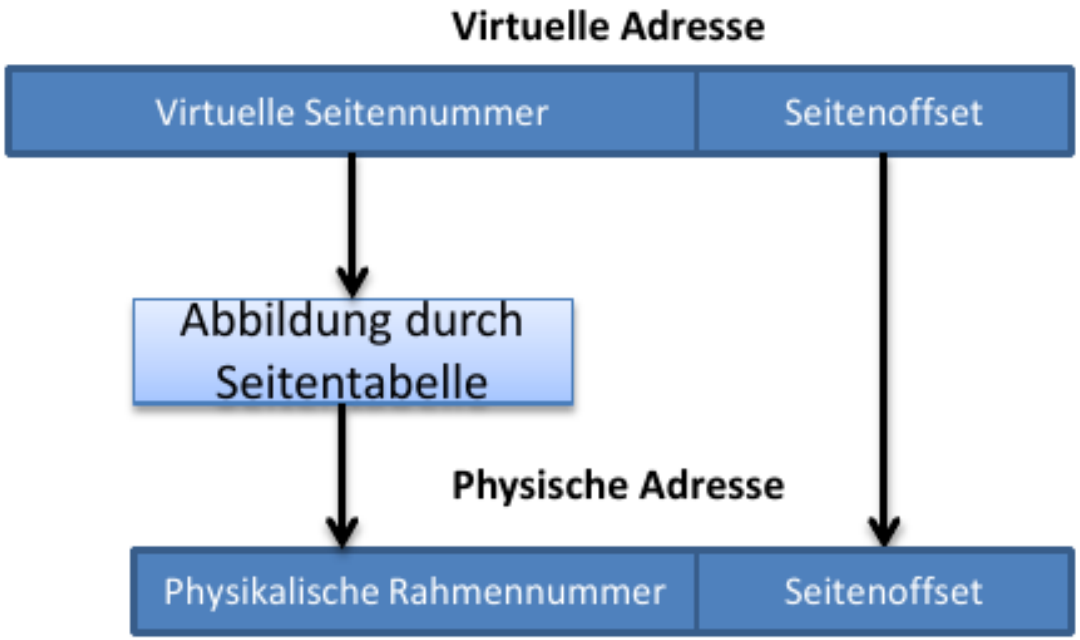
Abbildung 10.12: (a) Virtueller Adressraum von Prozess *A* (b) Physischer Speicher (c) Virtueller Adressraum von Prozess *B*

Quelle: Andrew Tanenbaum: Moderne Betriebssysteme

Beispiel für ein Speicherlayout, wenn zwei Prozesse ein gemeinsames Textsegment haben
(Zwei Benutzer starten zum Beispiel denselben Texteditor.)

Adreßumsetzung

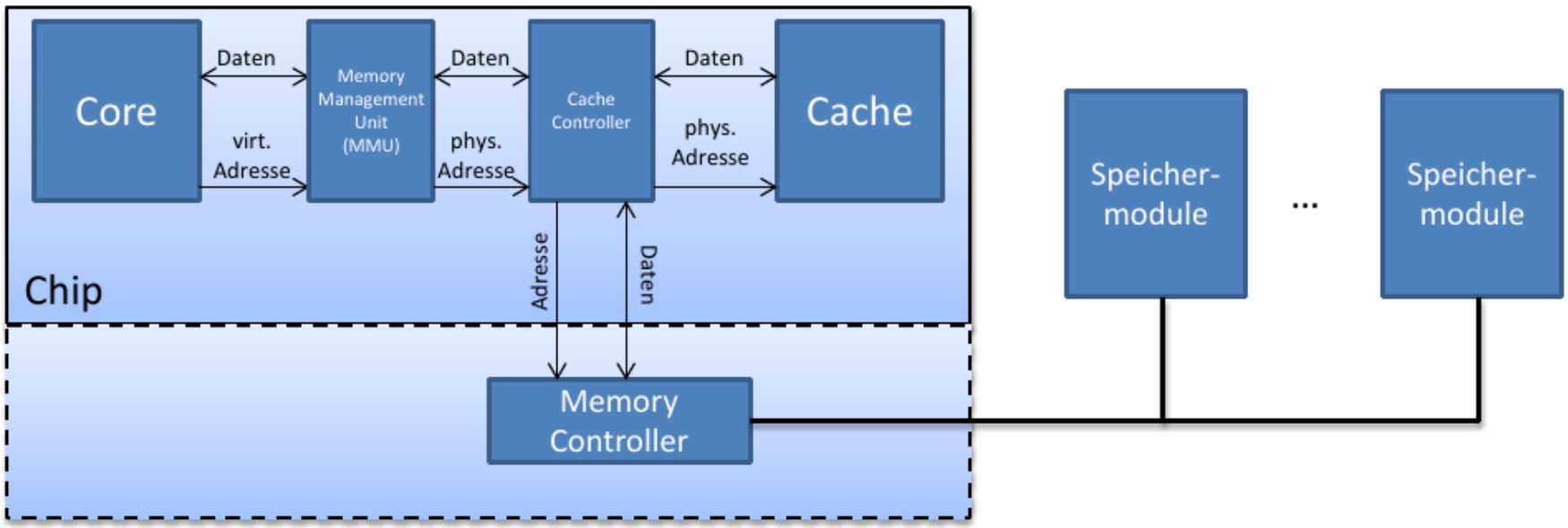
virtuelle Adresse: (Seitennummer, Offset innerhalb der Seite)



Quelle: VL-Skript Mario Schölzel

Realisierung einer Seitentabelle

Die Memory Management Unit (MMU) übersetzt die virtuellen Adressen in physikalische Adressen.



Quelle: VL-Skript Mario Schölzel

Ziel:

1. schneller Zugriff,
2. möglichst wenig Speicherplatzbedarf.

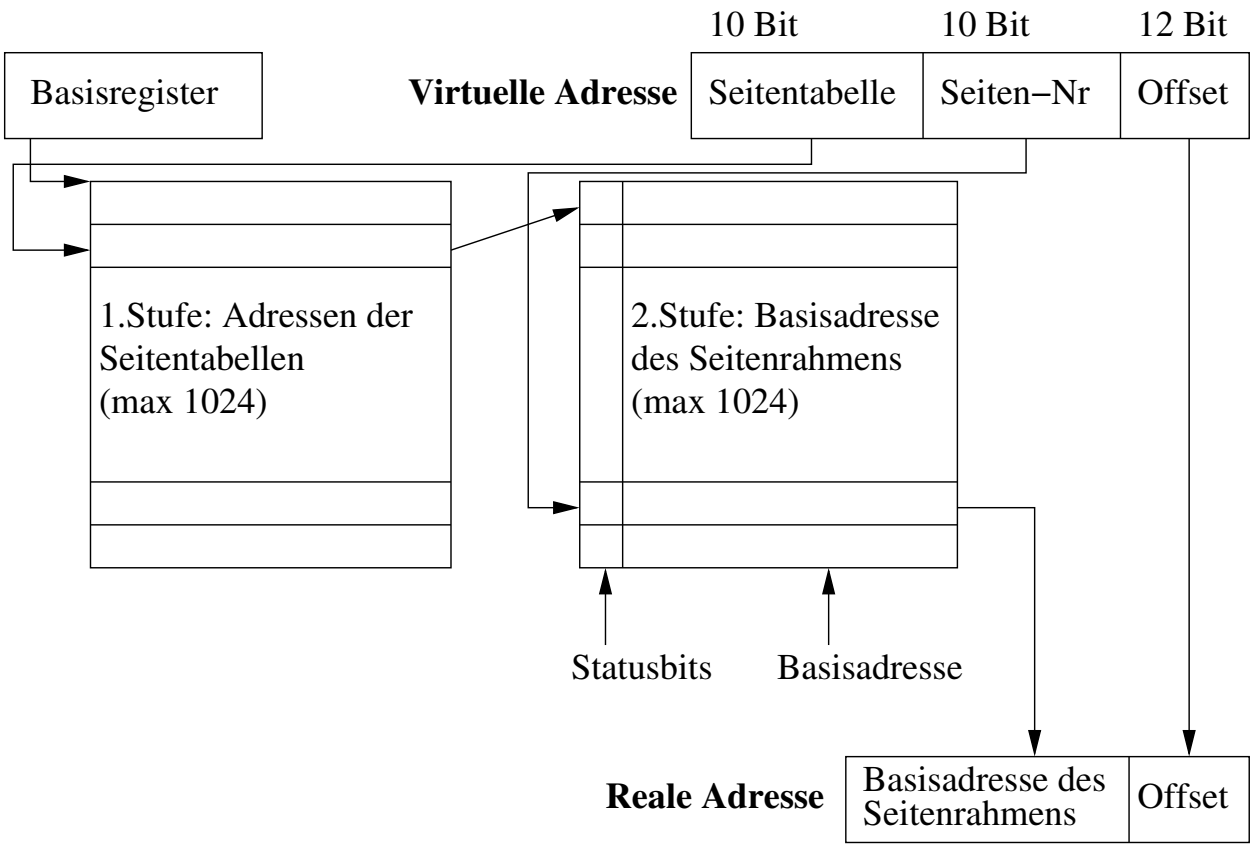
zu 1.: Translation Lookaside Buffer (TLB): schneller assoziativer Zwischenspeicher in der MMU, der
– Anfangsadresse der Seitentabelle und
– ca. 32-1024 „aktuell benötigte" Einträge der Seitentabelle enthält. Die Verwaltung von Treffern findet vollständig in Hardware statt. Trefferrate typischerweise > 99 %.

zu 2. Notwendiger Kompromiß:
kleine Seitengröße $\Rightarrow$ Seitentabelle wächst!
große Seitengröße $\Rightarrow$ interner Verschnitt wächst!

Um Platz für die Seitentabellen im Hauptspeicher zu sparen
$\Longrightarrow$ Mehrstufige Seitentabelle

Mehrstufige Seitentabellen

Beispiel: 2-stufige Seitentabelle



Definition 24:

Ein Seitenfehler tritt auf, falls während der Programmausführung ein Zugriffswunsch auf eine Seite erfolgt, die sich nicht im Hauptspeicher befindet.

$\Longrightarrow$ Demand-Paging := Seitenwechsel auf Anforderung

Gefahr: Seitenflattern := Das Betriebssystem beschäftigt sich nur noch mit dem Ein-/Auslagern von Seiten.

Demand-Paging-Verfahren

Definition 25:

Ein Programmablauf wird durch eine Seitenzugriffsfolge, auch Referenzstring genannt,

$$W = s[1]s[2]\ldots s[T]$$

beschrieben, wobei $s[t] \in \mathbf{N}$ die Seitennummer der Seite, auf die zum Zeitpunkt t zugegriffen wird, bezeichnet.

Demand-Paging-Verfahren:

- $s[t]$ zum Zeitpunkt t nicht im Hauptspeicher $\Rightarrow$ einlagern!
- Hauptspeicher belegt $\Rightarrow$ Es muß eine Seite $\tilde{s}$ ausgewählt werden, die ausgelagert wird.
- Falls $\tilde{s}$ modifiziert ist $\Rightarrow$ $\tilde{s}$ auf den Hintergrundspeicher zurückschreiben. Andernfalls kann $\tilde{s}$ überschrieben werden.

Ersetzungsstrategien (Page Replacement Algorithms)

Optimierungsziel: möglichst wenig Seitenfehler

FIFO: „älteste" Seite wird ausgelagert.

- einfach zu implementieren mit einer einfach verketteten Liste

  Beispiel mit m=3

  $\hat{=}$ zufällige Auswahl

Least Recently Used (LRU): Die am längsten nicht benutzte Seite wird ausgelagert.

Aufwand: Bei jedem Seitenzugriff muß die Verwaltungstruktur (z.B. doppelt-verkettete Liste) aktualisiert werden!

Belady's optimaler Algorithmus:

Die Seite mit dem größten Vorwärtsabstand wird ausgelagert.

Vorwärtsabstand der Seite x zum Zeitpunkt t eines Programms mit Laufzeit T:

$$d(x,t) := \begin{cases} k, & \text{falls } x \notin \{s[t+1], \ldots, s[t+k-1]\} \text{ und } s[t+k] = x \\ \infty, & \text{falls } x \notin \{s[t+1], \ldots, s[T]\} \end{cases}$$

- Optimaler Algorithmus,
- nur für die Beurteilung anderer Verfahren interessant, da $d(x,t)$ nicht a priori bekannt ist.

Definition 26:

$F(A, m, W)$ bezeichne die Anzahl Seitenfehler eines Paging-Algorithmus A zu gegebenem Referenzstring W und Hauptspeichergröße m.
Ein Paging-Algorithmus heißt gutartig, falls

$$\forall W \in IN^{IN} \quad \forall m \in IN : \; F(A, m+1, W) \leq F(A, m, W).$$

Definition 27:

$S(A, m, W, t)$ bezeichne die Menge der zum Zeitpunkt t gemäß dem Paging-Algorithmus A und dem Referenzstring W in den Hauptspeicher der Größe m eingelagerten Seiten.
Ein Algorithmus heißt Stack-Algorithmus, falls

$$\forall m \in IN \quad \forall t \in IN \quad \forall W \in IN^{IN} : S(A, m, W, t) \subseteq S(A, m+1, W, t).$$

Bem.: Stack-Algorithmen sind gutartig.

Bem.:

1. FIFO ist nicht gutartig (Belady's Anomalie).

   im Beispiel $m = 4$

W	0	1	2	3	0	1	4	0	1	2	3	4
jüngste Seite	0	1	2	3	3	3	4	0	1	2	3	4
	-	0	1	2	2	2	3	4	0	1	2	3
	-	-	0	1	1	1	2	3	4	0	1	2
älteste Seite	-	-	-	0	0	0	1	2	3	4	0	1
Seitenfehler	$*$	$*$	$*$	$*$			$*$	$*$	$*$	$*$	$*$	$*$

   $\Longrightarrow$ 10 Seitenfehler!

2. LRU ist gutartig.

Working–Set–Modell (Denning 1970)

Definition 28:

Gegeben sei der Referenzstring $s[1]\, s[2]\, \ldots\, s[t]$ eines Programms zum Zeitpunkt t.
Dann heißt

$$W(t, T) := \{i \in \mathbb{N} \mid i \in \{s[t-T],\, s[t-T+1],\, \ldots,\, s[t-1]\}\}$$

Working-Set (Arbeitsmenge) des Programms zum Zeitpunkt t.
$T :=$ heißt Working-Set-Parameter.

Ein Working-Set-Algorithmus basiert auf folgender Eigenschaft:
Die Seiten des Working-Set werden im Hauptspeicher gehalten.
Jede beliebige Seite, die nicht zum Working-Set $W(t, T)$ gehört, kann ausgelagert werden.

Wahl von T: $0 < T < m$

Beispiel: Hauptspeicher mit m=4 Seitenrahmen
OBDA: Unter den Seiten, die nicht zum Working-Set gehören, wähle diejenige mit der kleinsten Seitennummer aus

$T = 1$

m	5	2	6	4	3	2	1	3	2	1	
1	5	5	5	5							
2		2	2	2							
3			6	6							
4				4							
SF	*	*	*	*							10 SF

$T = 2$

m	5	2	6	4	3	2	1	3	2	1	
1	5	5	5	5							
2		2	2	2							
3			6	6							
4				4							
SF	*	*	*	*							7 SF

$T = 3$

m	5	2	6	4	3	2	1	3	2	1	
1	5	5	5	5							
2		2	2	2							
3			6	6							
4				4							
SF	*	*	*	*							6 SF

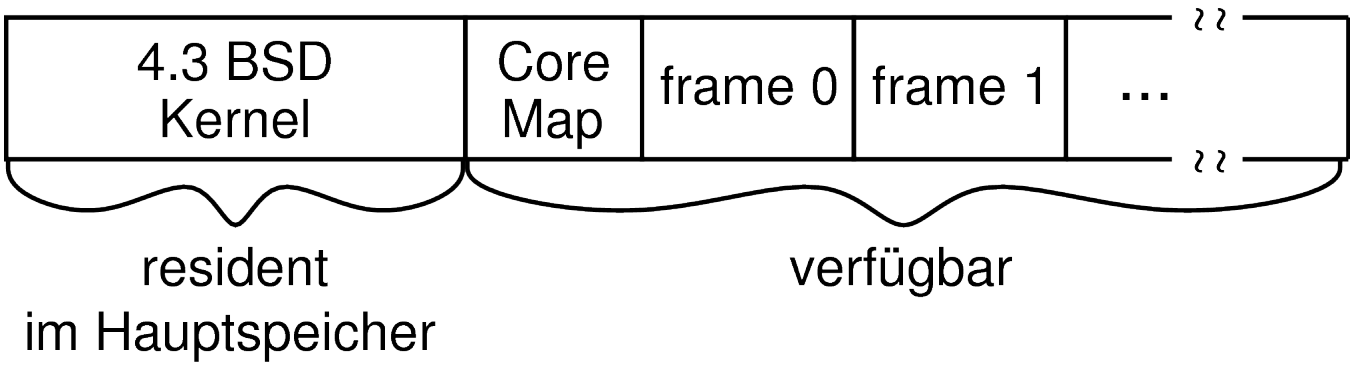
Statische/Dynamische Seitenersetzungsverfahren

Bei statischen (lokalen) Verfahren darf jeder Prozeß gleich viel Seitenrahmen bis zu einer vorgegebenen Maximalzahl im Hauptspeicher belegen.
Im Gegensatz dazu verändert sich bei dynamischen (globalen) Verfahren die Anzahl der Seitenrahmen, die ein Prozeß zur Laufzeit belegt. Es existiert keine maximale Obergrenze.

4.4 Beispiel: UNIX-Speicherverwaltung

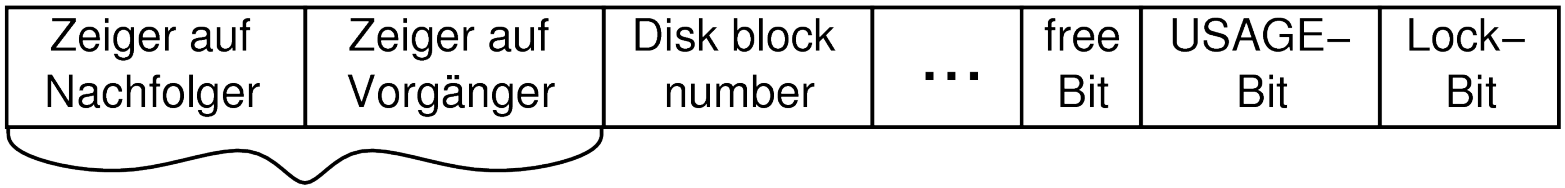
- erste Versionen: nur Swapping
- ab 3BSD: Paging + Swapping
- dynamische Strategie

Hauptspeicheraufteilung:



Die Verwaltung der Seitenrahmen wird mittels der sogenannten Core Map realisiert.

Die Core-Map enthält für **jeden** Seitenrahmen einen Eintrag à 16 Byte:



Die freien Seitenrahmen werden mittels einer doppelt verketteten Liste, der „Free-List", verwaltet, die in der Core-Map gespeichert ist:

free-Bit=0 $\Leftrightarrow$ Seitenrahmen ist auf der Free–List.

Strategie:

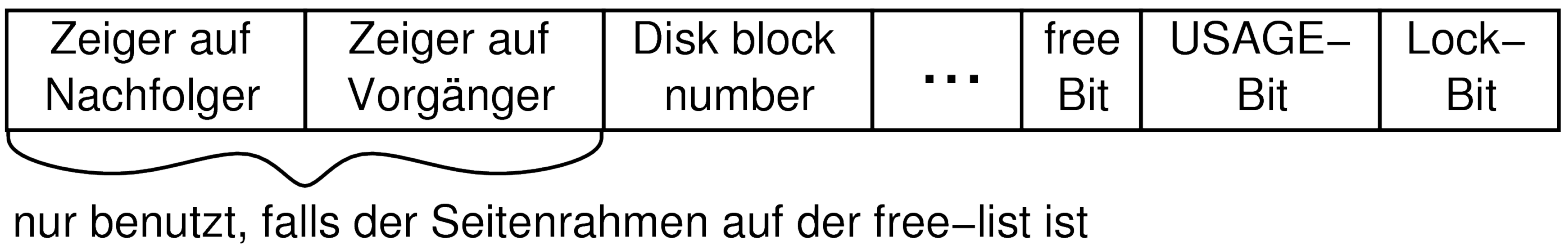
1. Im Falle eines Seitenfehlers wird der erste freie Rahmen auf der Free–List belegt.
2. Periodisches Auffüllen der Free–List: Seiten, die zu keinem Working-Set gehören, werden auf die Free-List gesetzt.

Der Pagedaemon wird alle 250 ms geweckt, um die Free–List aufzufüllen:

- Falls | free list | $<$ `lotsfree`, beginnt der Pagedaemon solange Seiten auszulagern, bis mindestens `lotsfree` Seitenrahmen auf der Free–List sind.
- sonst: sleep()

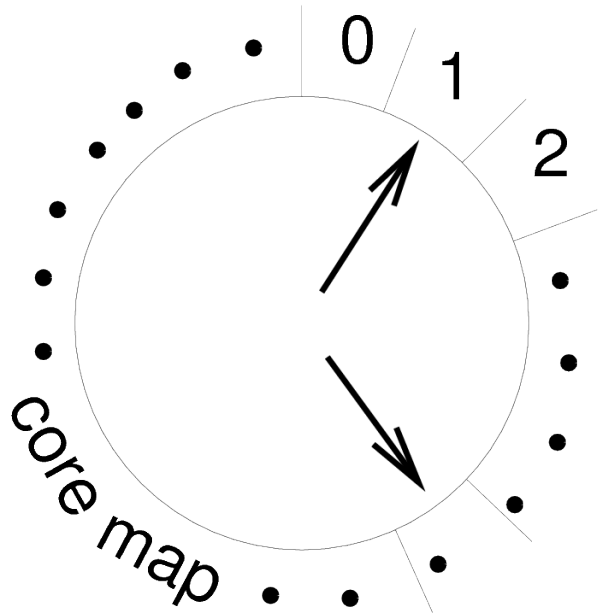
Parameter `lotsfree` $\approx \frac{1}{4}$ Speichergröße

Welche Seiten gehören zu einem Working-Set bzw. zu keinem Working-Set?



Ansatz: USAGE-Bit=0 $\Leftrightarrow$ Seitenrahmen gehört zu keinem Working-Set.

Seitenersetzungsstrategie: “Two - handed clock algorithm”



Der 1. Zeiger löscht das USAGE-Bit.

Der 2. Zeiger prüft, ob das USAGE-Bit gelöscht ist.

- Falls das USAGE-Bit gelöscht $\Longrightarrow$ Modifizierte Seiten werden auf den Hintergrundspeicher zurückgeschrieben und der Seitenrahmen an die Free–List gehängt.
- Pro Aufruf werden die Zeiger solange vorgerückt, bis genug Seitenrahmen auf der Free–List stehen.
  maximal: weniger als 1 Umdrehung

Lokale oder Globale Strategie?

Bemerkung: Ein One–handed Clock Algorithm wurde schon in MULTICS benuzt.

4.5 Beispiel: Windows NT und Nachfolger

Solomon/Russinovich: "On multiprocessor systems, Windows 2000 implements a variation of a local FIFO replacement policy. On uniprocessor systems, it implements something closer to a least recently used policy (LRU) (known as the clock algorithm, as implemented in most versions of UNIX).

Parameter für x86-Systeme: Seitengröße 4 KB

Seitenersetzungsstrategie:

- auf x86-Einprozessorsystemen: ähnlich Clock Algorithm: Falls `Access-Bit` im PTE (Page Table Entry) gelöscht ist, wird die Seite freigegeben, d.h. aus dem Working-Set entfernt. Andernfalls wird das `Access-Bit` gelöscht und die nächste Seite geprüft.
- auf Mehrprozessorsystemen und Alpha-Systemen: lokale FIFO–Seitenersetzungsstrategie

Seitenersetzungsstrategie

Definition 29:

Diejenigen Seiten des Prozesses, die im Hauptspeicher eingelagert sind, heißen Working Set des Prozesses („NT-Working-Set").

Die Anzahl an Seiten, die ein Prozess mindestens im Hauptspeicher belegen darf, heißt Minimum-Working-Set des Prozesses, die maximale Anzahl heißt Maximum-Working-Set.

Hauptspeichergröße	
Small	$\leq$ 19 MB
Medium	20-32 MB
Large	$\geq$ 32 MB, falls Windows NT Workstation $\geq$64 MB, falls Windows NT Server

Haupt-speichergröße	Default Minimum-Working-Set Size (in Pages)	Default Maximum-Working-Set Size (in Pages)
Small	20	45
Medium	30	145
Large	50	345

Windows benutzt ein hybrides Verfahren, daß eine lokale und eine globale Strategie kombiniert:

1. Unter „normalen Umständen“ erhält jeder Prozeß den Minimum-Working-Set und muß damit auskommen.
2. Für einen Prozeß mit hoher Seitenfehlerrate wird der aktuelle Working-Set vergrößert, falls genügend freier Speicher vorhanden ist (Aufgabe des `Memory Managers`).

Variablen für das Erweitern des NT-Working-Sets

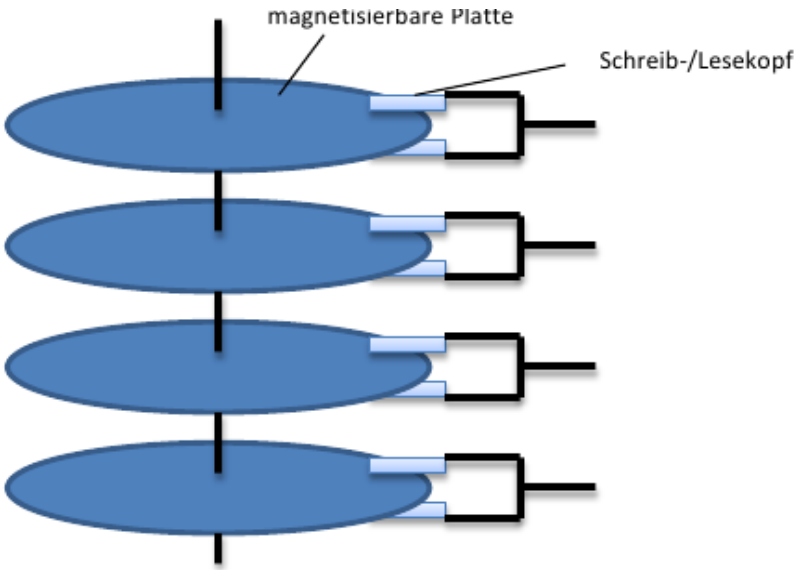
Variable	Wert	Bedeutung
`MmWsExpandThreshold`	90	Anzahl Seiten, die verfügbar sein müssen, damit der Working-Set über den Maximalwert erweitert werden darf.
`MmWorkingSetSizeIncrement`	6	Anzahl an Seiten, um die der Working-Set vergrößert wird, falls genügend Speicher vorhanden ist und der Working-Set unter dem Maximalwert liegt.
`MmWorkingSetSizeExpansion`	20	Anzahl an Seiten, um die der Working-Set vergrößert wird, falls genügend Speicher vorhanden ist und der Working-Set seinen Maximalwert erreicht hat.

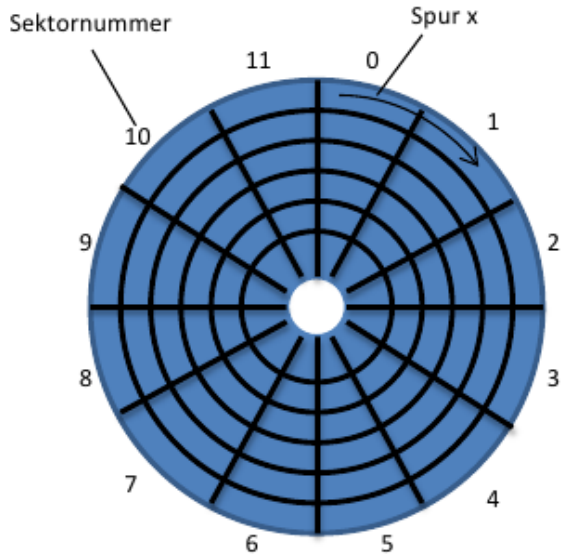
Windows benutzt ein hybrides Verfahren, daß eine lokale und eine globale Strategie kombiniert:

1. Unter „normalen Umständen“ erhält jeder Prozeß den Minimum-Working-Set und muß damit auskommen.
2. Für einen Prozeß mit hoher Seitenfehlerrate wird der aktuelle Working-Set vergrößert, falls genügend freier Speicher vorhanden ist (Aufgabe des `Memory Managers`).
3. Automatic Working Set Trimming

Mit Automatic Working Set Trimming wird ein Verfahren bezeichnet, das versucht stets freie Seiten im System zu haben. Hierbei wird für jeden Prozeß geprüft, ob die Größe seines Working-Sets den Minimum-Working-Set-Wert überschreitet. Falls ja, werden dem Prozeß Seitenrahmen entzogen und somit sein Working-Set verkleinert. Die Zeit zwischen zwei Trimmzyklen beträgt 6s.

Physikalischer Aufbau Festplatte





Quelle: Mario Schölzel

Das mechanische Positionieren der Schreib-/Leseköpfe ist deutlich zeitaufwendiger als der eigentliche Lese- bzw. Schreibvorgang.

Demand-Paging with „Clustering”: Im Falle eines Seitenfehlers wird nicht nur die angeforderte Seite eingelagert, sondern auch eine gewisse Anzahl von Seiten aus der „Umgebung”.

Hauptspeichergröße	Clustergröße für Datenseiten		Clustergröße für Programmseiten	
	NT	2000	NT	2000
$\leq$ 19 MB	0	2	1	3
$>$ 19 MB	1-3	4	2-7	8

Strategie: Clock-Algorithm mit Optimierungen

1. Optimierung: Es soll möglichst zusammenhängender Speicher belegt werden, um effizientes Kopieren zwischen Haupt- und Hintergrundspeicher (Platte) zu ermöglichen.

Buddy-Liste `free_area`: Für $0 \leq i \leq 9$ enthält die `free_area` Listen mit 2^i zusammenhängenden freien Seiten

2. Optimierung: (stammt aus SunOS)
Ziel: den **internen** Verschnitt für **oft benutzte Datenstrukturen** minimieren

Beispiele: Einträge in der Prozesstabelle (Prozessleitblock), Eintrag in der Deskriptortabelle, ...

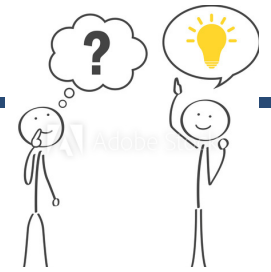
Slab Allocator: hält für häufig benutzte Datentypen passenden Speicherplatz vor

Ablauf:

1. Der Slab Allocator holt sich freie Seitenrahmen von der Buddy-Liste.
2. Der Slab Allocator schneidet einen Seitenrahmen in „Scheibchen“ passender Größe und verwaltet diese in Listen.
3. Speicherplatz, der frei gegeben wird, wird weiterhin vom Slab Allocator verwaltet und wiederverwendet.

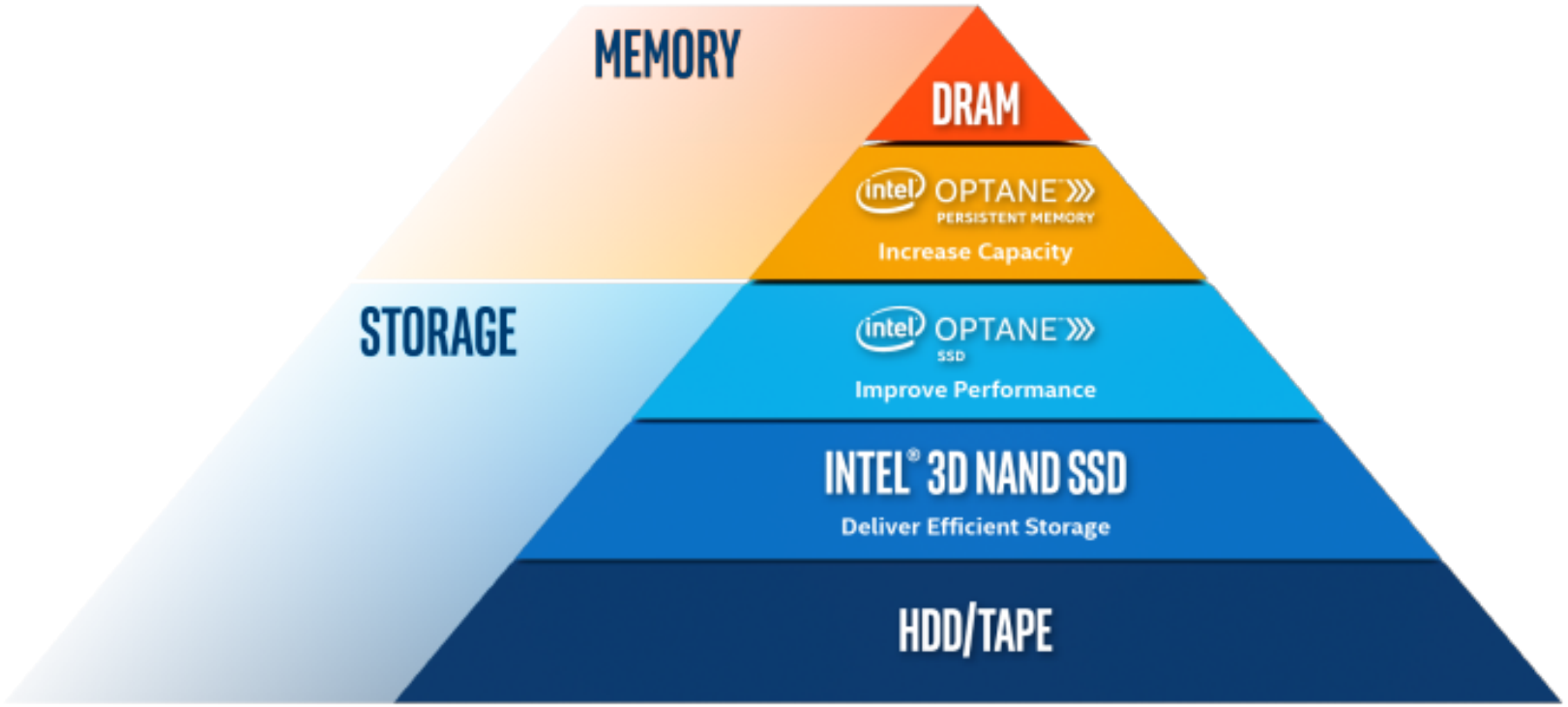
(siehe auch /proc/slabinfo)

Was Sie jetzt wissen sollten ...



- Grundproblem: externe und interne Fragmentierung
- virtuelle Adressierung und Paging vermeiden externe Fragmentierung und ermöglichen es, Programme ablaufen zu lassen, die größer als der verfügbare Hauptspeicher sind.
- In der Theorie geeignet: LRU
- Approximativ umgesetzt im Working-Set-Verfahren
- Implementierung des Working-Set-Verfahrens mittels USAGE-Bit (UNIX, Two-handed Clock Algorithm) bzw. `Access-Bit` (Windows NT)
- Buddy-Verfahren: ursprünglich im Kontext von Swapping vorgeschlagen, um externe Fragmentierung zu reduzieren, wird eingesetzt
  - Linux: `free_area` Listen
  - Doug Lea's `dlmalloc()` für die Verwaltung des virtuellen Adressraums (siehe Übung)

Aktuelle Entwicklungen



Quelle: Intel

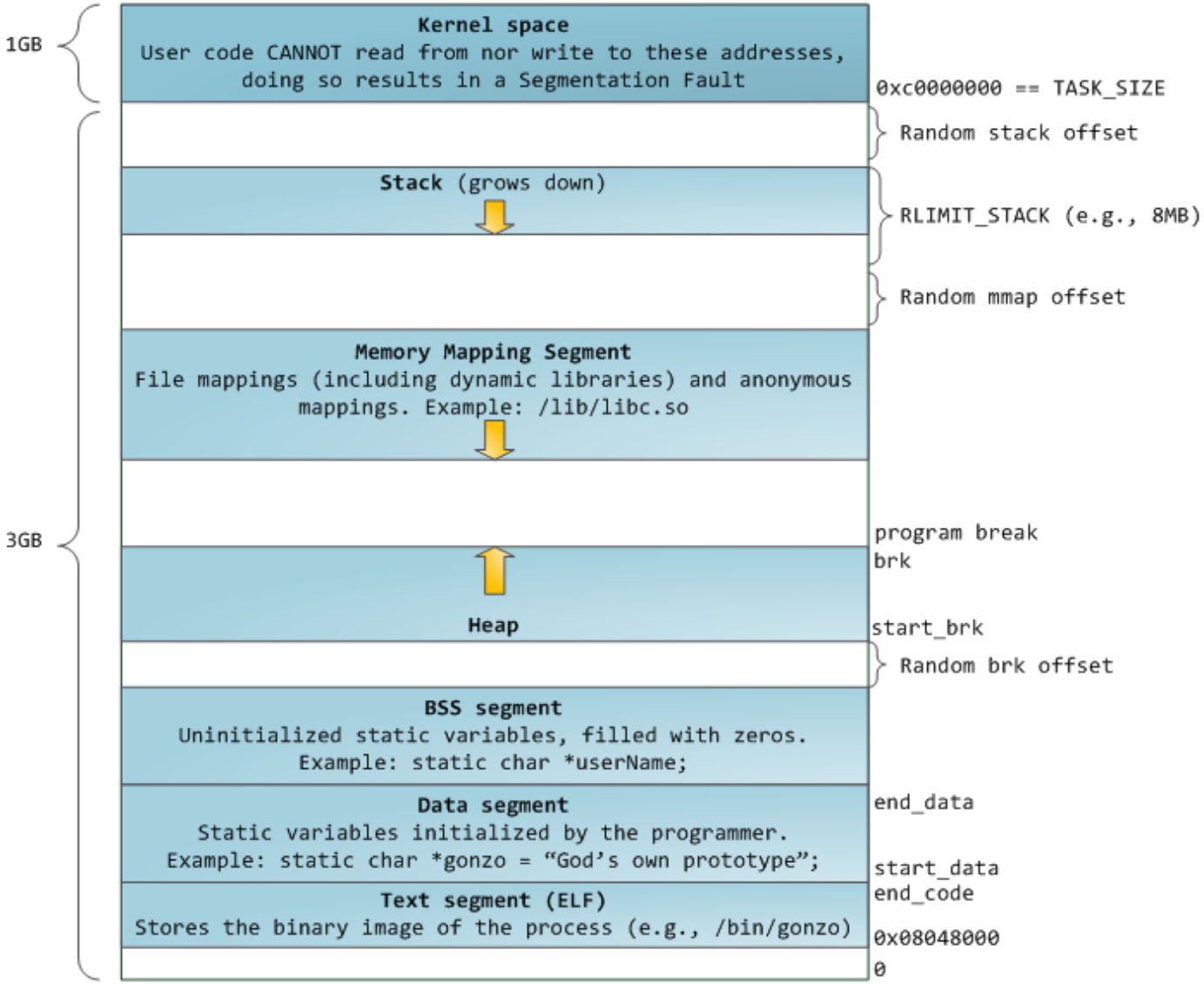
siehe auch Intel Optane Persistent Memory aims to fill the gap DRAM can't, ComputerWeekly, 10.11.2020

Memory Allocator aus der Standard C Bibliothek:

```
MALLOC(3)                          Linux Programmer's Manual

NAME
  malloc, free, calloc, realloc - allocate and free dynamic
  memory

SYNOPSIS
  #include <stdlib.h>

  void *malloc(size_t size);
  void free(void *ptr);
  void *calloc(size_t nmemb, size_t size);
  void *realloc(void *ptr, size_t size);
```

Malloc-Implementierung:

1. Platz im virtuellen Adressraum besorgen
2. ggf. neuen Platz im Hauptspeicher besorgen $\Longrightarrow$ Aufgabe des Betriebssystems (Seite von der Free-List, Seitentabelle aktualisieren, ...)



Quelle: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

BSS: Block Started by Symbol

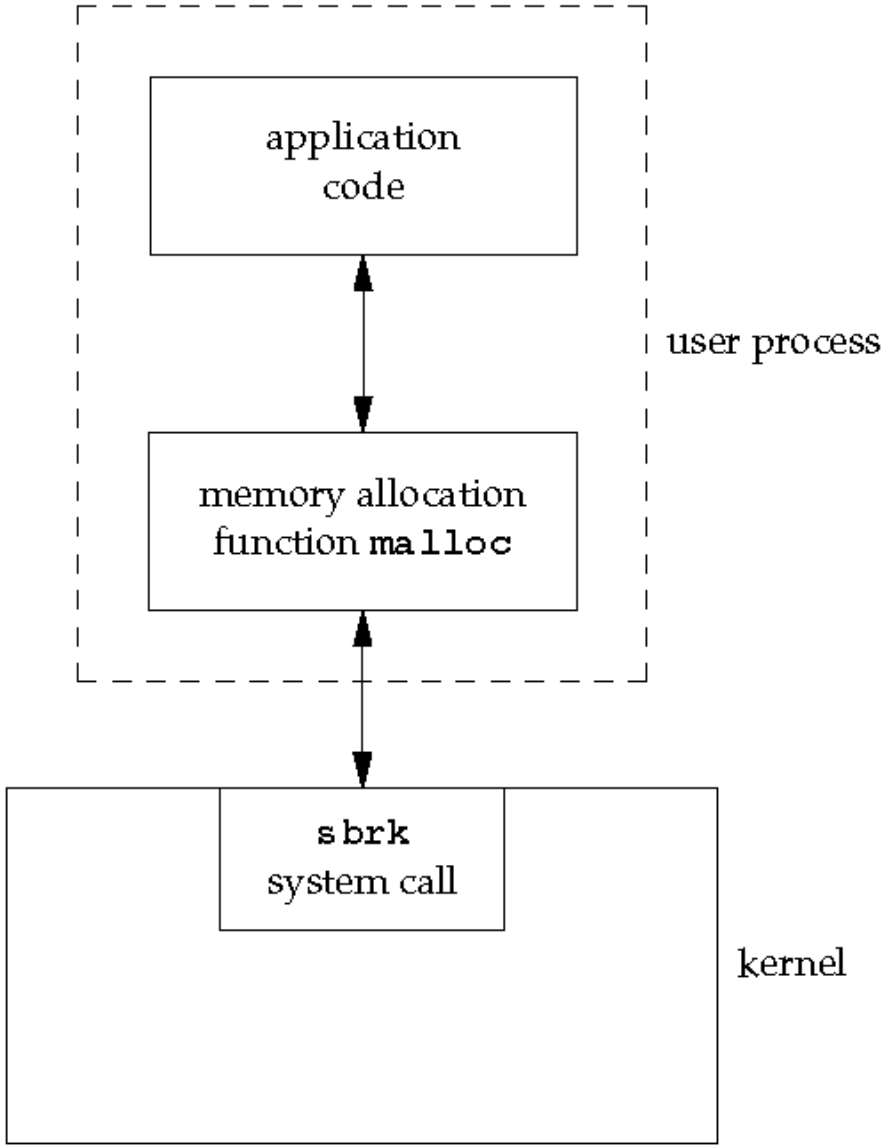


Figure 1.11 Separation of `malloc` function and `sbrk` system call

Quelle: Stevens, Rago: *Advanced Programming in the UNIX Environment*

```
BRK(2)                          Linux Programmer's Manual

NAME
  brk, sbrk - change data segment size

SYNOPSIS
  #include <unistd.h>
  void *sbrk(intptr_t increment);

DESCRIPTION

brk() and sbrk() change the location of the program break
...

Increasing the program break has the effect of allocating
memory to the process; decreasing the break deallocates
memory.
```

sbrk() increments the program's data space by increment bytes. Calling sbrk() with an increment of 0 can be used to find the current location of the program break.

RETURN VALUE

On success, sbrk() returns the previous program break. (If the break was increased, then this value is a pointer to the start of the newly allocated memory). On error, (void *) -1 is returned, and errno is set to ENOMEM.

malloc() Implementierung?

```c
/* An horrible dummy malloc */

#include <sys/types.h>
#include <unistd.h>

void          *malloc(size_t size)
{
  void *p;
  p = sbrk(0);
  /* If sbrk fails, we return NULL */
  if (sbrk(size) == (void*)-1)
    return NULL;
  return p;
}
```

Quelle: Marwan Burelle: *A Malloc Tutorial*

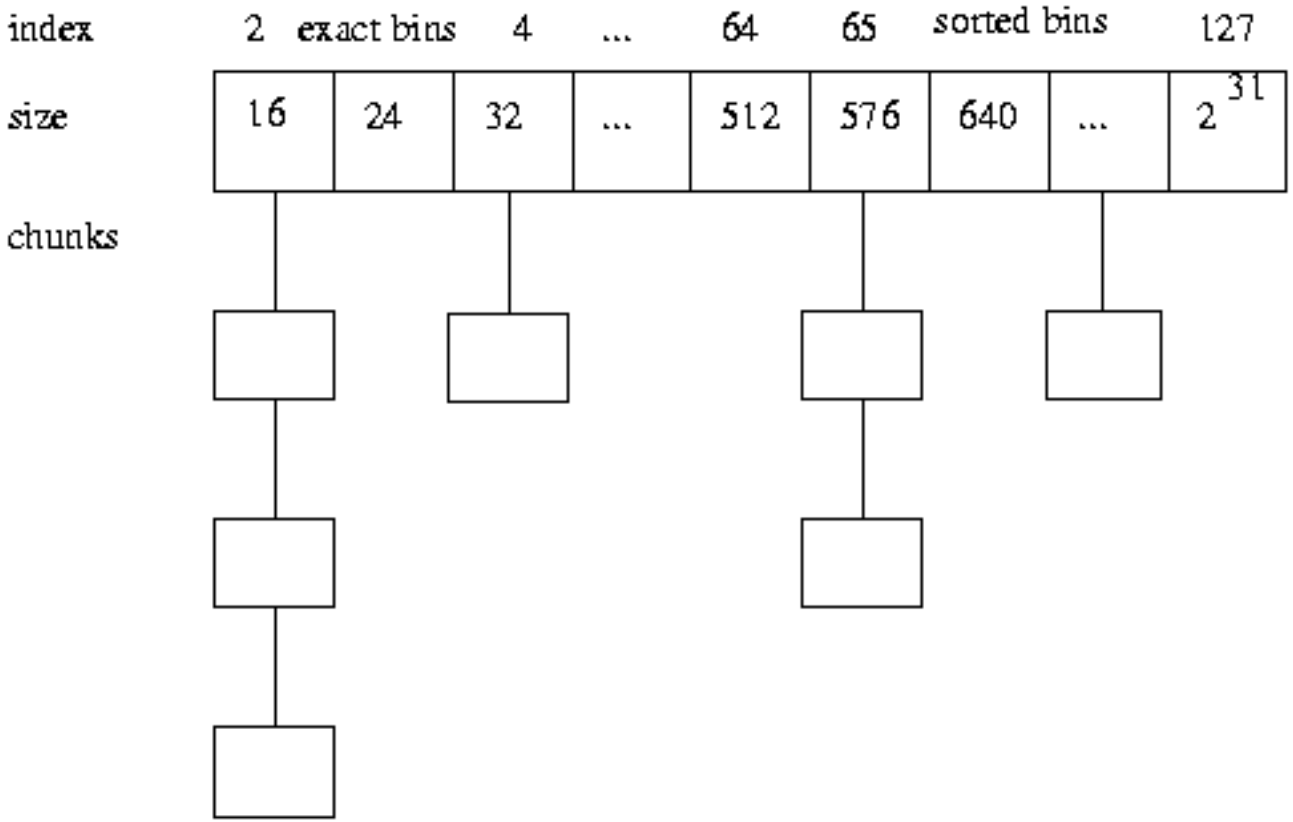
Problem: Nach `free()` entstehen Lücken im virtuellen Adressraum.
$\Longrightarrow$ Durch die dynamische Speicherverwaltung kommt es zu Fragmentierung auch im virtuellen Adressraum.

Ziele:

- Lücken wiederverwenden,
- schnell freie Adressraumlücke finden,
- Fragmentierung reduzieren,
- Programm/Prozeß klein halten,
- ...

Beispiel: Doug Lea's `dlmalloc()`: Benutzt eine Variante des Buddy-Verfahrens

- Die verwalteten Bereiche im virtuellen Adressraum heißen *Chunks*.
- Chunks werden nach der Größe sortiert auf Listen verwaltet, sogenannten *Bins*.
- Searches for available chunks are processed in smallest-first, best-fit order: best-first with coalescing.



64 Bins mit Chunks fester Größe: 16, 24, 32, 40, ..., 512
64 Bins für Chunks der Größe g: $g_{i-1} < g \leq g_i$

Quelle: `http://g.oswego.edu/dl/html/malloc.html`

Literatur:

- Marwan Burelle: *A Malloc Tutorial*, Laboratoire Système et Sécurité de l'EPITA, 2009.
- Doug Lea: `dlmalloc - A Memory Allocator`, 1995 (etwas veraltet, aber lesenswert)
- Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles: Dynamic Storage Allocation: A survey and Critical Review, University of Texas, 1995.