3 Speicherverwaltung

Das Betriebsmittel ist jetzt der Hauptspeicher, der verwaltet werden muss.

Bemerkung: **es gibt nichts, was Hauptspeicher ersetzen kann, außer mehr Hauptspeicher.**

Die Strategien zur Hauptspeicherverwaltung unterscheiden sich im Verwendungszweck als:

- Speicherplatz für konkurierende Prozesse
- Speicherplatz des einzelne Prozesses mit Heap- und Stackbereich

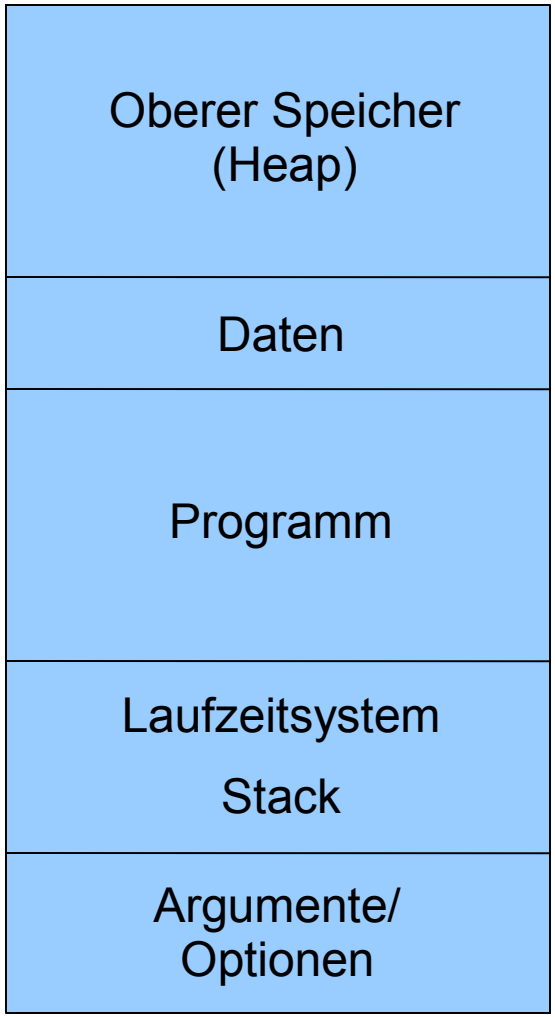


Abbildung 3.1 Speicherbereiche eines Programms

3.1 Direkte Speicherverwaltung

Historisch stand dem Computeranwender der gesamte Computer mit seinem Speicher zur Verfügung.

Das Ein- und Auslagern ganzer Programme (swapping) benötigte Zeit.

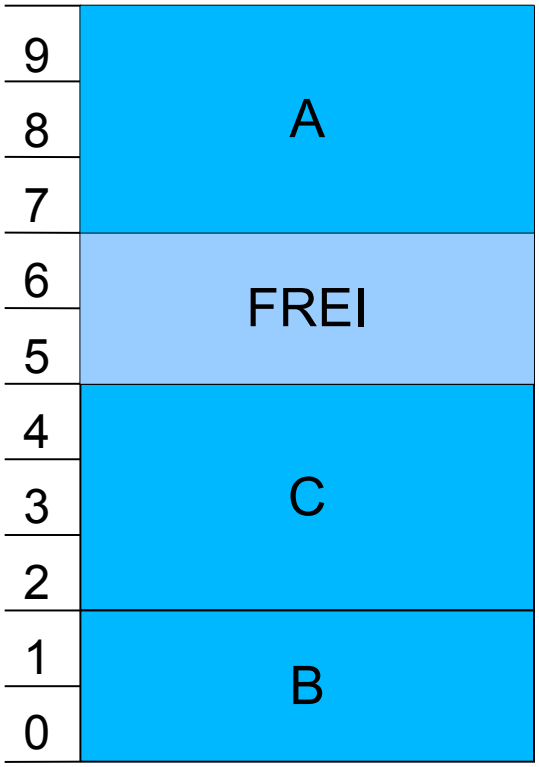
Als mehr Hauptspeicher zur Verfügung stand, konnte man gleichzeitig mehrere Prozesse im Hauptspeicher halten. Die Zuordnung des belegten Hauptspeichers zu den Prozessen musste gelöst werden.

3.1.1 Zuordnung durch feste Tabelle

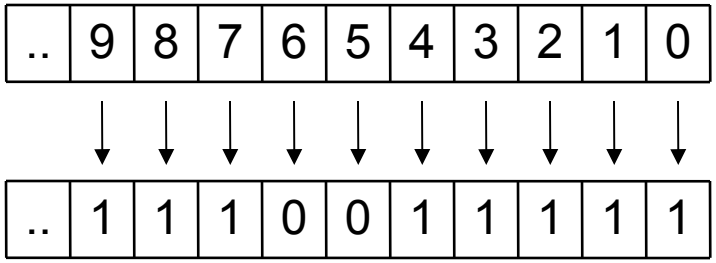
Speicherbelegungstabellen stellen **ein verkleinertes Abbild des Hauptspeichers** dar.

Ein gesetztes Bit in dieser Tabelle markiert ein belegtes Wort oder einen belegten Speicherbereich stets gleicher Größe des Hauptspeichers.

Bei einem Bereich von 4KB Größe reicht eine 100KB große Belegungstabelle aus, um 3,2GB Speicher zu verwalten. 100KB sind 800KBit Markierungsmöglichkeiten á 4KB.



Speicherbelegung



Belegungstabelle

Abbildung 3.2 Eine **direkte Speicherbelegung** und ihre Belegungstabelle

3.1.2 Zuordnung durch Listen

Weniger Speicherplatz als die Einzelmarkierungen benötigt eine Beschreibung, die Startbereich und Länge sowie einen Verweis auf die nächste Belegung enthält.

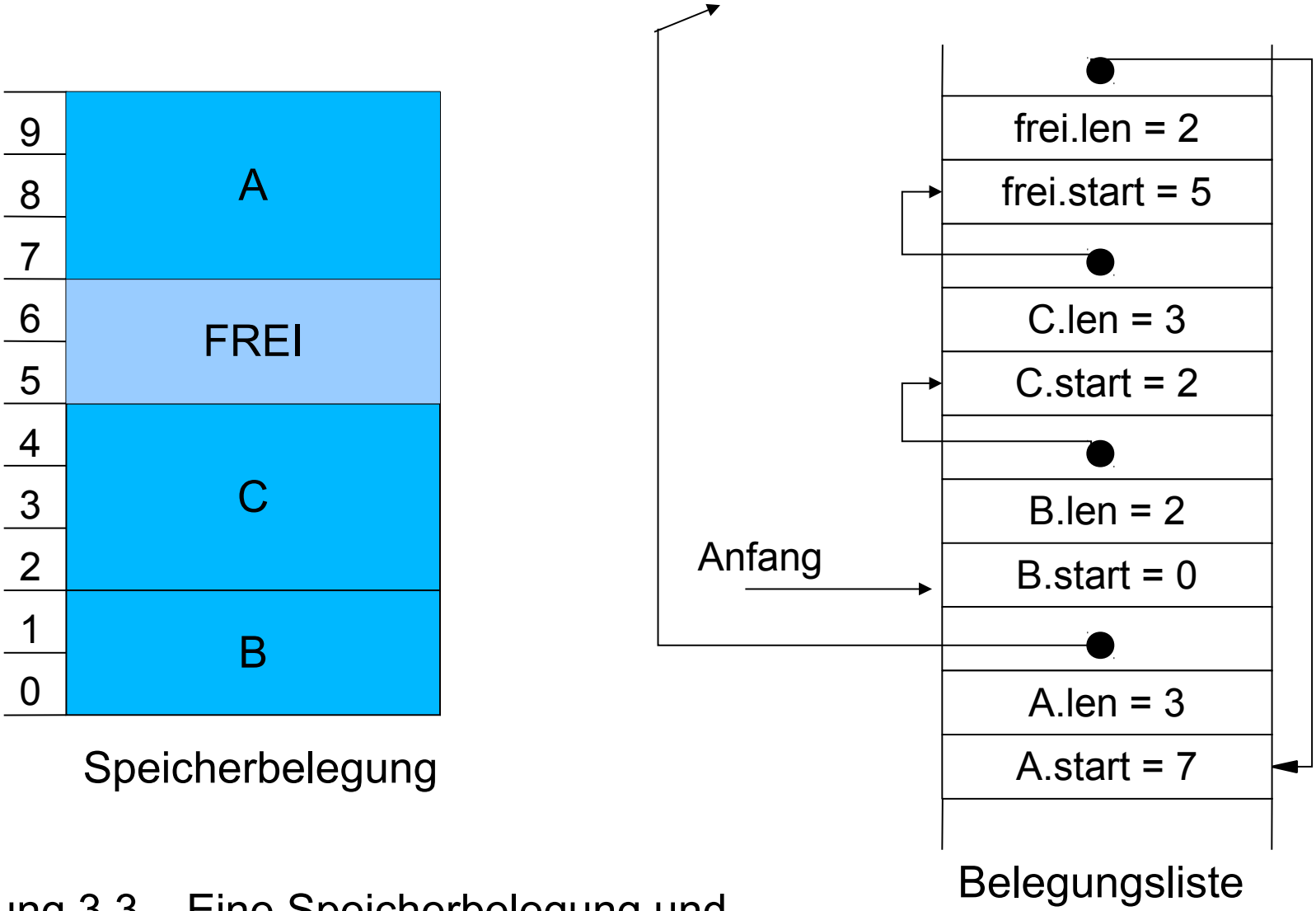


Abbildung 3.3 Eine Speicherbelegung und die verzeigerte Belegungsliste

Die Speicherbereiche für Belegungen und freien Speicher lassen sich in getrennten Listen verwalten. Die Freie-Liste kann nach der Größe der zusammenhängenden Bereiche sortiert werden.

Problem:

Wie können benachbarte, freie, zusammenhängende Bereiche verschmolzen werden, um sie für die nächste große Speicheranforderung nutzen zu können?

3.1.3 Belegungsstrategien

Suche nach freien Bereichen:

- **First Fit** erstes Stück, das ausreichend groß ist
- **Next Fit** wie First Fit, jedoch beginnt die nächste Suche nach dem aktuellen Treffer
- **Best Fit** geeignetes Stück finden, das exakt passt oder nur wenig Verschnitt (s.u.) hat
- **Worst Fit** sucht das größte freie Stück, so dass das Reststück groß bleibt

- **Buddy-Systeme** Es werden Listen mit 2^k, 2^{k+1}, 2^{k+2} usw. großen Stücken verwaltet, aus denen die Belegung durchgeführt wird. Es entsteht Verschnitt aber das Zusammenführen frei werdender Stücke wird vereinfacht, da jedes Stück genau einen Partner hat, mit dem es zu einem größer Stück verschmolzen werden kann.

First Fit ist besser als Next Fit bei der Platzausnutzung.
Worst Fit ist besser als Best Fit, welches dazu neigt, nur sehr kleine unbelegbare Reststücke (Verschnitt) übrig zu lassen.

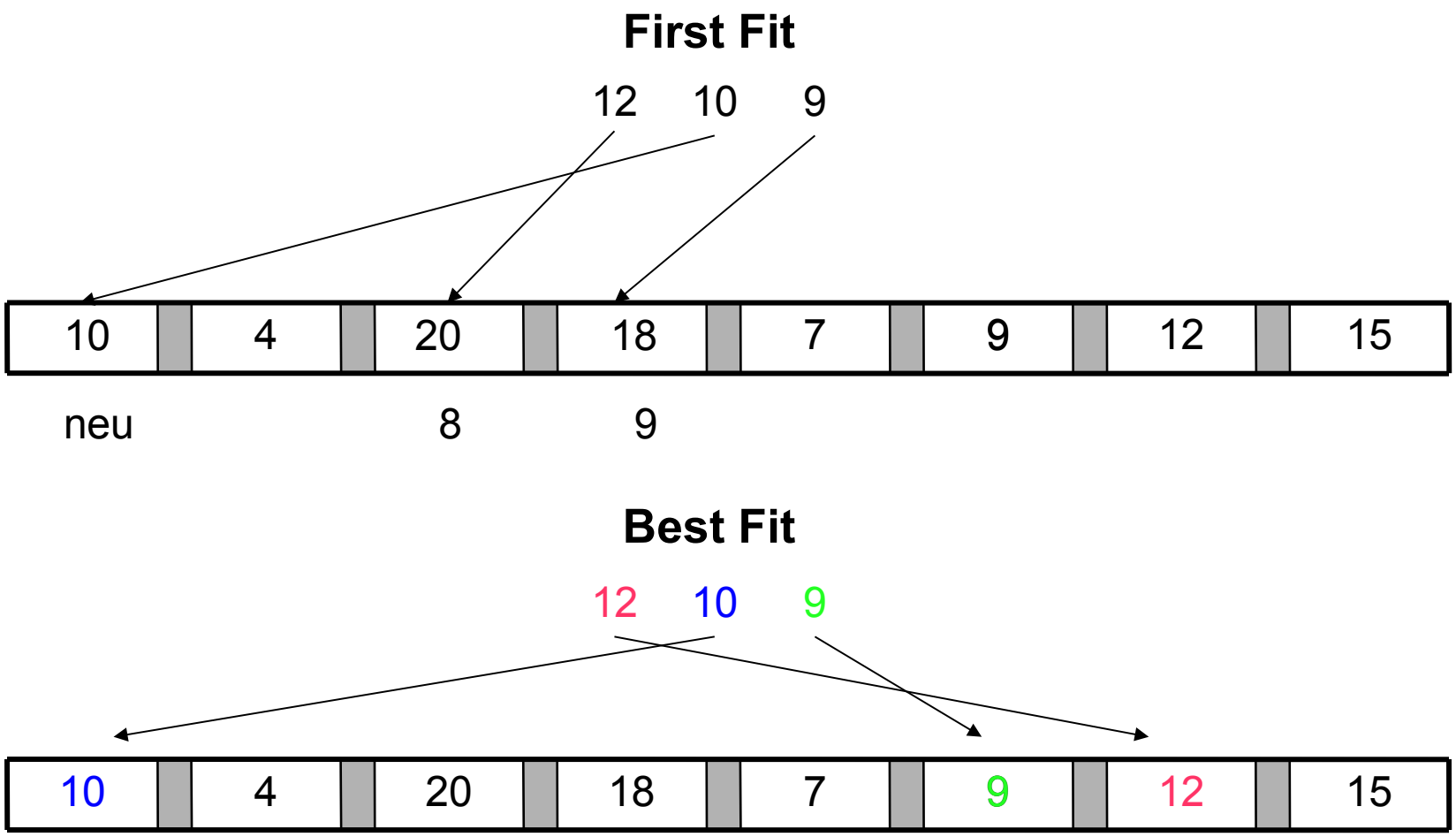


Abbildung 3.4 Die Belegungsstrategien im Vergleich

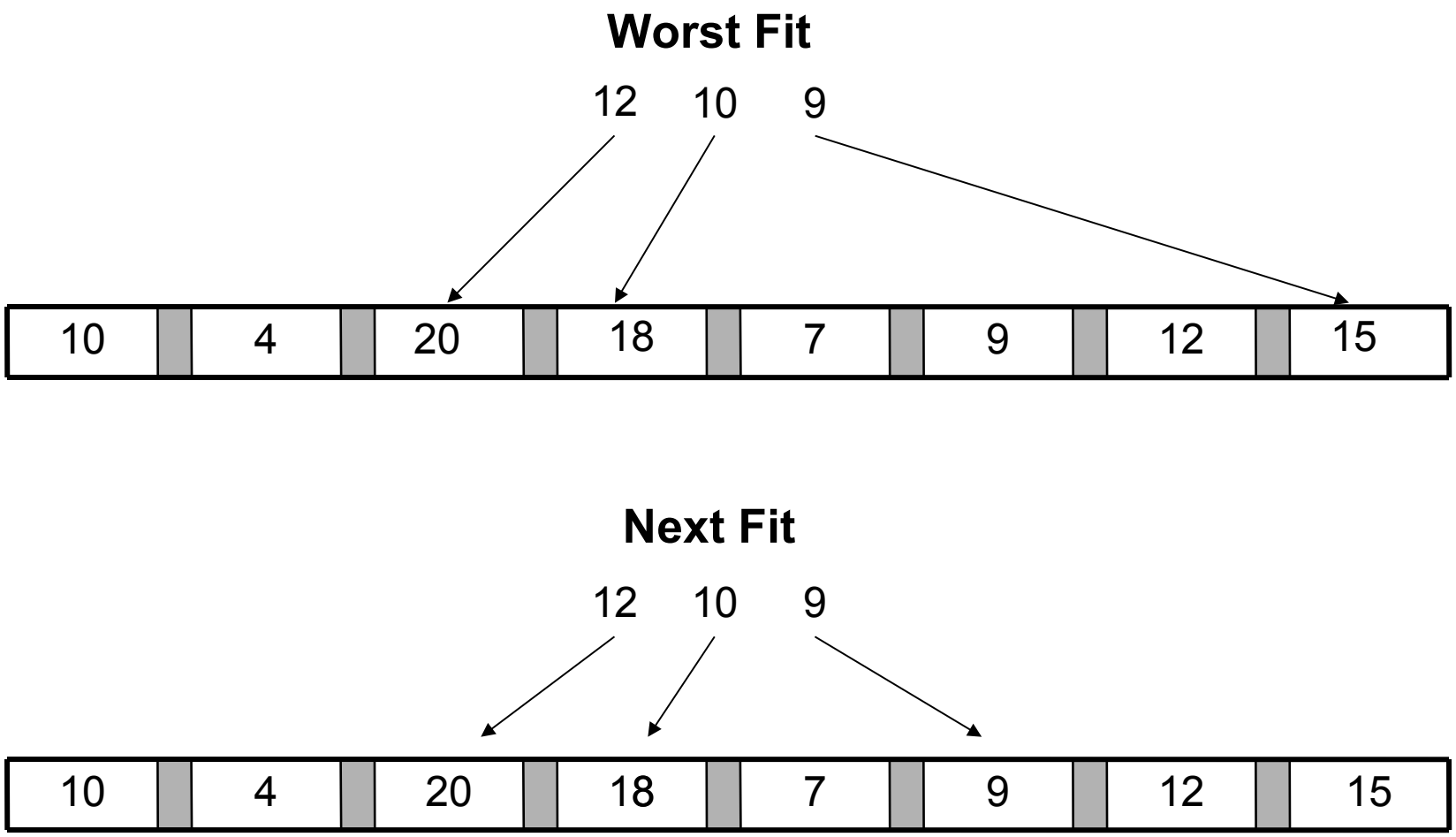


Abbildung 3.4 Die Belegungsstrategien im Vergleich

3.2 Speicherfragmentierung

Zersplitterung des HS in nicht mehr zerlegbare Reststücke.

3.2.1 interner Verschnitt

Tritt bei Prozessausführung auf. Hinterlässt kleine Reststücke auf dem Heap. Abhilfe schafft garbage collection.

3.2.2 externer Verschnitt

Tritt zwischen den geladenen Programmen (Prozessen) auf, die nicht auf Bereichsgrenzen enden. Vergleichbar mit dem Verschnitt beim Schreiben einer Datei auf die Plattenblöcke.

3.3 Logische Adressierung und virtueller Speicher

3.3.1 Speicherprobleme und Lösungsansätze

Neben dem Problem der Fragmentierung, das z.T. sehr aufwendig mit einer garbage collection gelöst wird, treten weitere Probleme auf.

3.3.1.1 Relozierung von Programmcode

Zur Programmlaufzeit werden absolute Speicheradresse benötigt.

Der Binder weist zwar allen Sprungbefehlen und Variablen-adressen eindeutige, absolute Adressen zu, die sich jedoch auf eine voreingestellte Basisadresse (z.B. Null) beziehen. Will man das Programm in einem anderen Speicherbereich laufen lassen, müssen alle Adresse angepasst (reloziert) werden.

Lösungsansätze:

- Programm darf nur relative Adressen z.B. zum PC (program counter) enthalten (nicht für alle Prozessortypen und Befehle möglich).
- Verschiebeinformation wird zusätzlich gespeichert (doppelter Speicherplatzbedarf, ATARI ST).
- Verschiebeinformation ist als Basisadresse in einem speziellen Register der CPU gesetzt und wird bei jedem Zugriff automatisch addiert.

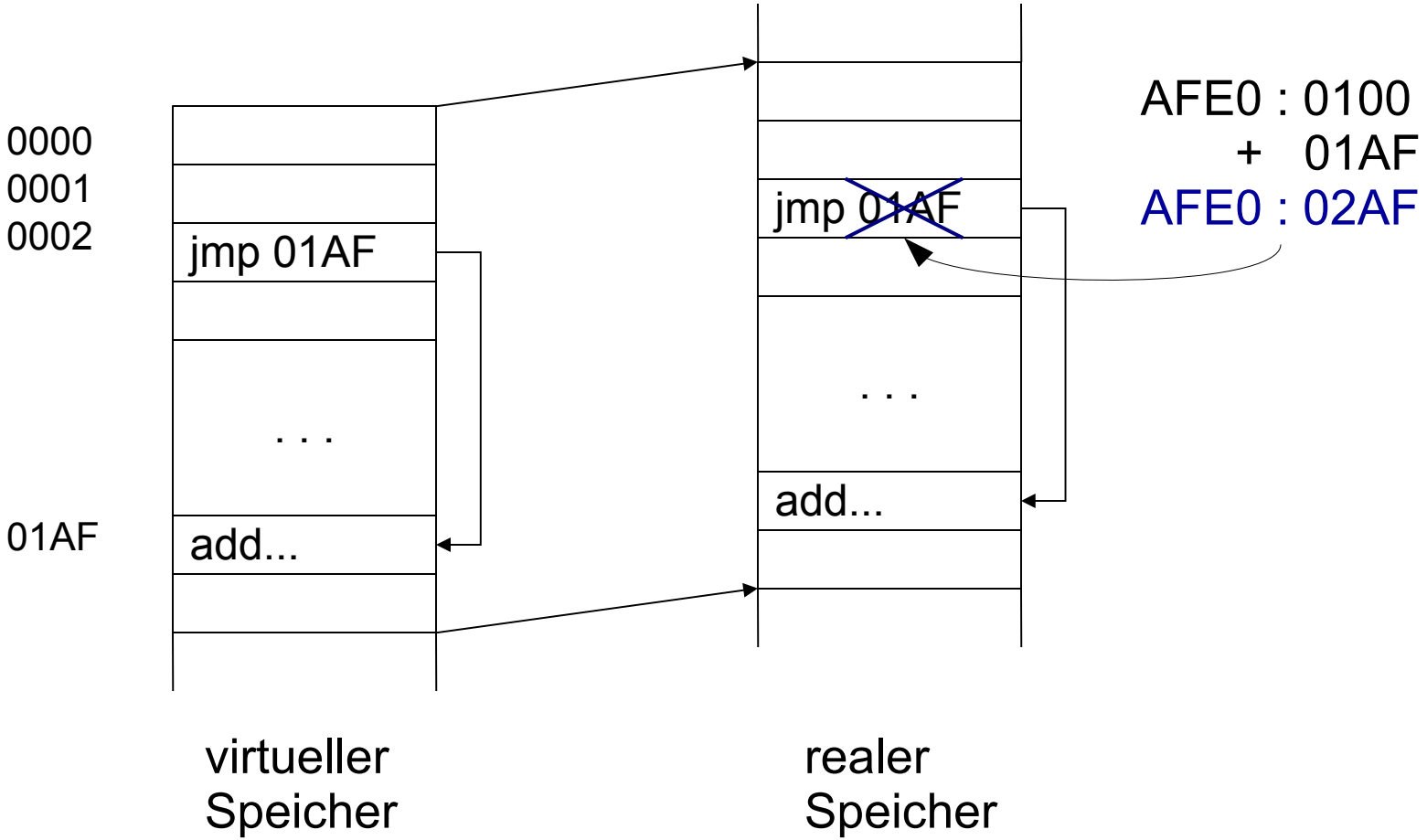


Abbildung 3.5 Relozieren von Programmcode

3.3.1.2 Speicherzusatzbelegung

Prozess benötigt während seiner Ausführung weiteren Speicher-platz.

Lösungsansätze:

- Prozess stilllegen und auslagern, neue Speichergröße berechnen und beim nächsten Einlagern berücksichtigen.
- Externen Verschnitt (zwischen den Prozessen) nutzen.

3.3.1.3 Speicherschutz

Ein Teil des Hauptspeichers ist dauerhaft vom Betriebssystemkern, den Systempuffern usw. belegt. Auf diesen Speicherbereich darf kein Benutzerprogramm zugreifen.

Lösungsansatz: Betriebssystemkern wird an die Unter- oder Obergrenze des Hauptspeichers gelegt.

Jetzt kann ständig kontrolliert werden, ob ein Prozess in diesen Speicherplatz zugreifen möchte. Diese Überprüfung geschieht hardwareseitig durch Vergleich mit den Limits (fences).

3.3.2 Virtueller Speicher

Dem Programmierer sollte der Zugriff auf den Hauptspeicher so einfach wie möglich gemacht werden. So wird die Programmierung und die Portierung erleichtert. Das Wunschbild des Programmierers für jedes seiner Programme besteht in einem Speicher, der mit der Adresse Null beginnt und sich kontinuierlich bis ins Unendliche erstreckt.
Leider sieht die Realität anders aus:

- Betriebssystemkern muss geladen sein, der meist den unteren Speicherbereich belegt.
- Andere Prozesse sind ebenfalls in den HS geladen, sodass es meistens keinen kontinuierlichen Bereich gibt, der passend groß ist.

- Speicherplatz ist immer teuer, sodass nicht unendlich viel zur Verfügung steht.

Konzept des virtuellen Speichers:

- Fragmente müssen als kontinuierliche Bereiche (ab Adresse Null) dargestellt werden.

- Falls ein Programm mehr Speicher verlangt, werden inaktive Speicherbereiche ausgelagert (swapping, paging) und neu benutzt.

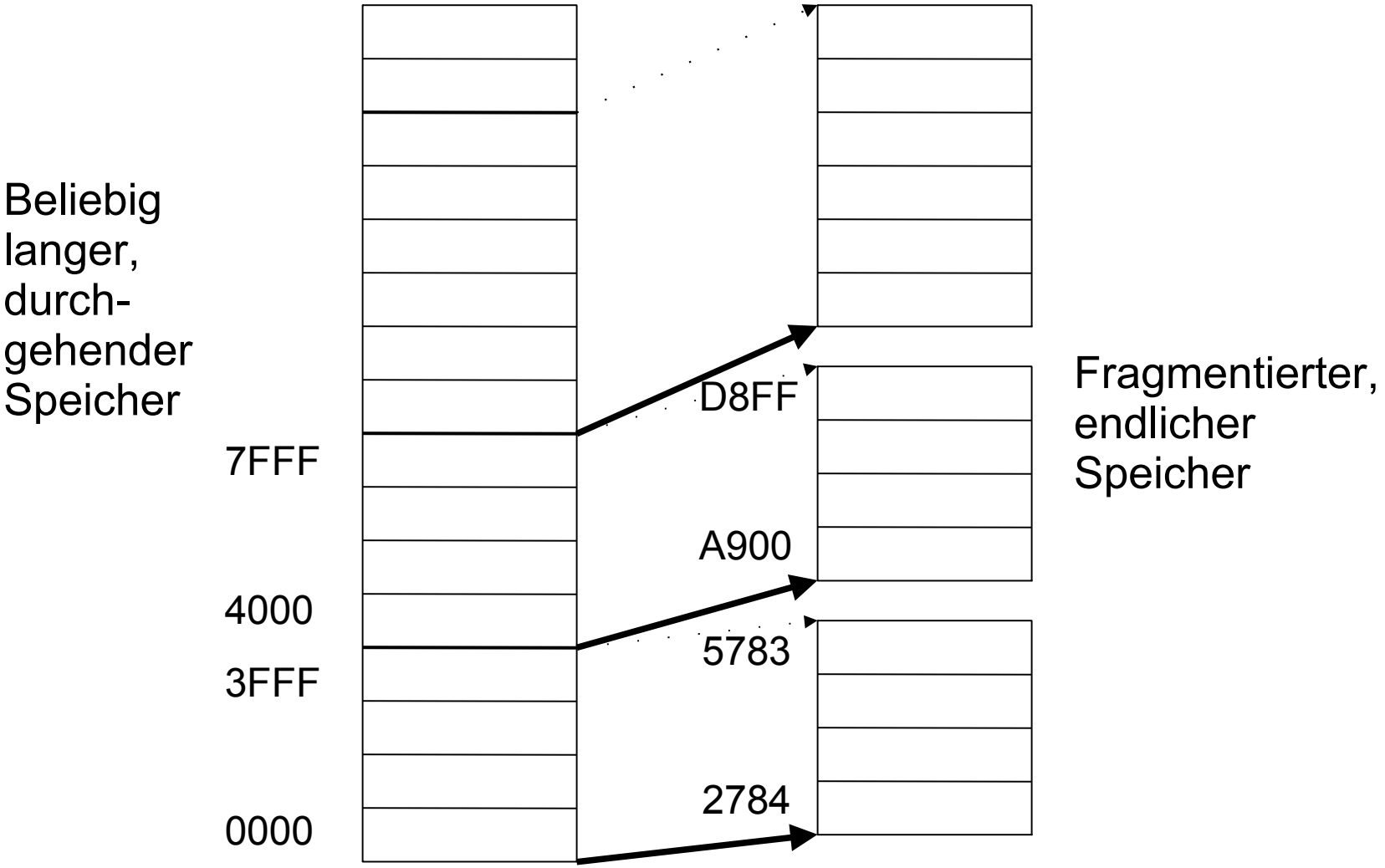


Abbildung 3.6 Die Abbildung physikalischer Speicherbereiche auf virtuelle Bereiche

3.4 Seitenverwaltung (paging)

Einfaches Verfahren zur Implementierung von virtuellem Speicher:

Speicheranforderung des Programms wird in gleich große Bereiche, sogenannte Seiten (pages) mit einer festen Seitengröße von z.B. 1KB, 4KB oder 8KB eingeteilt.

Der Hauptspeicher wird ebenfalls in Seiten dieser Länge eingeteilt. Die Seiten des HS werden zur Unterscheidung zu den Seiten des Programms auch Kacheln genannt. In einer Seitentabelle (Seiten-Kachel-Tabelle, page table) wird die Zuordnung der Seiten zu den Kacheln verwaltet.

3.4.1 Adresskonversion

Die virtuelle Adress setzt sich aus zwei Teilen zusammen.

Ein Teil mit den geringstwertigen Bits (least significant bits, LSBs), der **offset** genannt wird, bestimmt die relative Lage zu einer

Basisadresse, die aus den verbleibenden höherwertigen Bits (high significant bits, HSBs) errechnet wird.

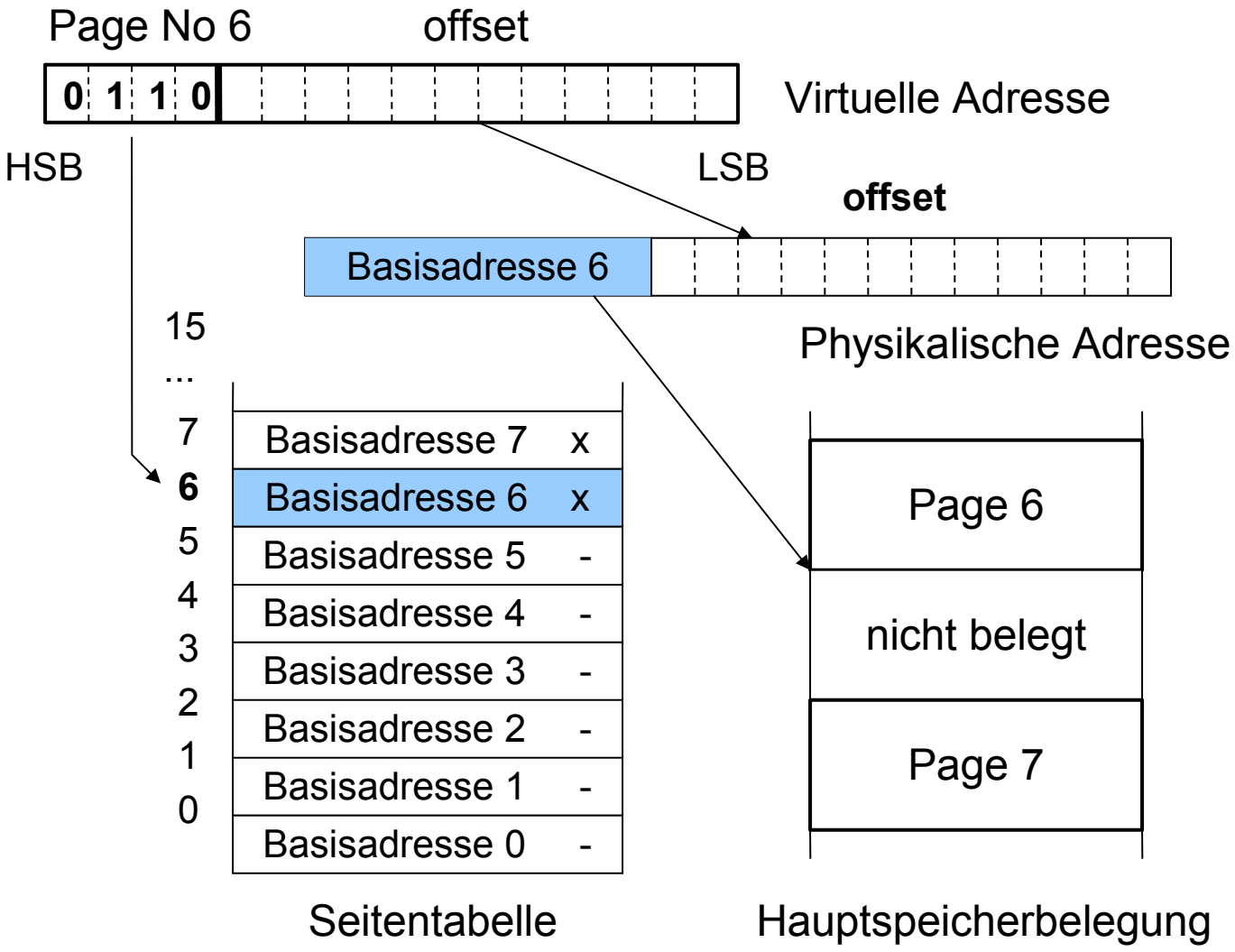


Abbildung 3.7 Die Umsetzung einer virtuellen Adresse in eine physikalische Adresse

Eine **Memory Management Unit** (MMU) sorgt für eine möglichst schnelle Ausführung der Operation, die bei jedem Speicherzugriff benötigt wird.

Die MMU erzeugt auch das Signal "**Seitenfehler**" (**page fault**) um anzuzeigen, dass eine gewünschte virtuelle Adresse noch nicht im Hauptspeicher abgebildet ist. Sind keine Hauptspeicherseiten mehr frei, so muss ein **Verdrängungsalgorithmus** eine Seite auswählen, die vorübergehend aus dem Hauptspeicher entfernt wird, um die neue virtuelle Seite einzulagern.

3.4.2 Adresskonversionsverfahren

Wir gehen von einer Wortbreite von 16 Bit aus. Die Seitengröße soll durch 12Bit auf 4KB Speicherwörter festgelegt werden. Damit enthält die Seitentabelle 16 Einträge (16 – 12 = 4, $2^4 = 16$). Eine solche Tabelle ist leicht zu handhaben und wurde bei den PDP-11-Rechnern der Firma Digital verwendet.

Erhöht man die Wortbreite auf 32Bit, ergibt sich bei einer gleichen Seitengröße von 4KB Speicherwörtern eine Seitentabelle mit $2^{20} \sim 10^6$, also etwa 1 Millionen Einträgen.

Bei einer Wortbreite von 64Bit wächst die Seitentabelle auf 2^{52}, also ungefähr $4*10^{15}$ Seiteneinträge an.

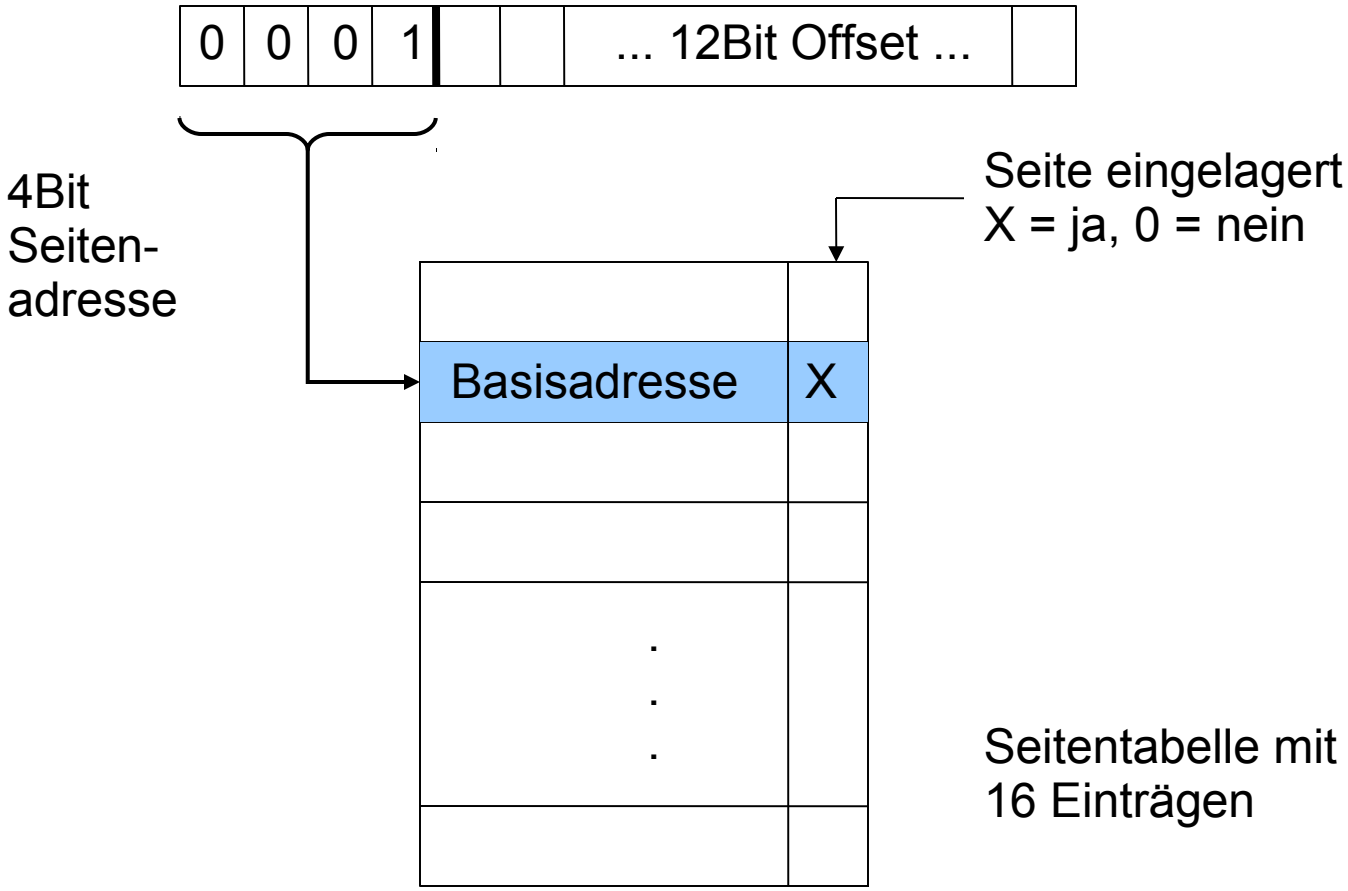


Abbildung 3.8 Eine einfache Seitentabelle

Hinweis:

Die Größe der Seitentabelle errechnet man aus der Anzahl der Einträge und der Bitanzahl für die Basisadresse + 1.

Für jeden Prozess muss diese Seitentabelle im Hauptspeicher gehalten werden, damit die Umrechnung schnell durchgeführt werden kann.

3.4.2.1 Adressbegrenzung

Bei einer Begrenzung des Adressraumes auf z.B. 30 Bit, was einem virtuellen Speicher von 1GB Speicherworten pro Prozess entspricht,

enthält die Seitentabelle 2^{18}, also 256 K Einträge, so dass eine ungefähr 0,5 MB große Seitentabelle unabhängig von der Prozessgröße entsteht.

Für einen 1 GB speicherwortgroßen Prozess ist dies angemessen, nicht jedoch für einen kleinen Prozess von 50 KB Worten.

3.4.2.2 Multi-Level-Tabellen

Die Information, ob ein Speicherbereich überhaupt genutzt und die Information, wie bei einem genutzten und existierenden Bereich die Adresse transformiert wird, wird getrennt gehalten.

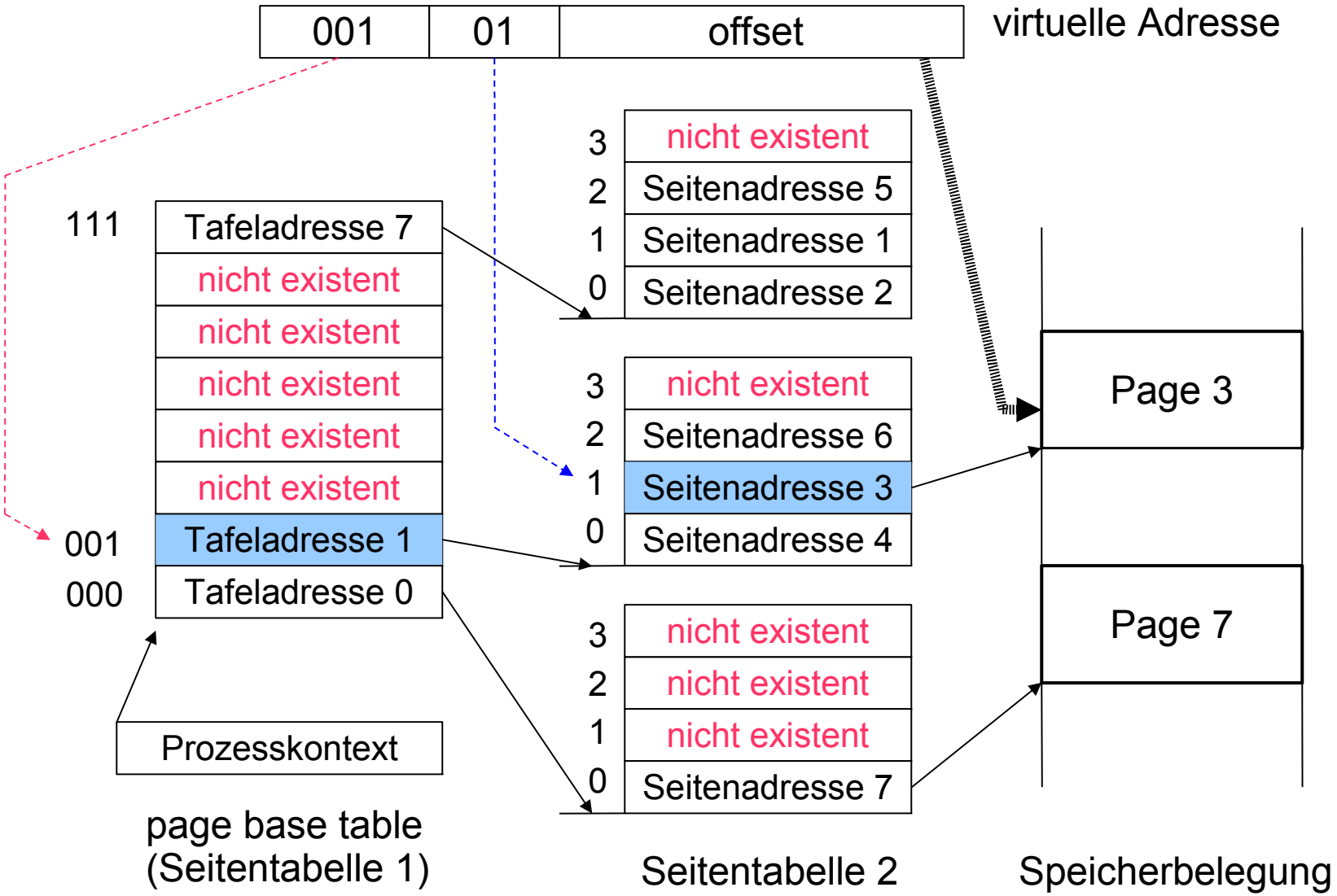


Abbildung 3.9 Eine zweistufige Adresskonversion

3.4.2.3 Invertierte Seitentabellen

Hier werden zu den tatsächlich vorhandenen physikalischen Adressen die virtuellen Adressen eines Prozesses abgespeichert.

Aus vielen großen Seitentabellen wird eine relativ kleine Tabelle, in der in jeder Zuordnungszeile noch zusätzlich die Prozessnummer notiert werden muss, um eine eindeutige Zuordnung zu erhalten.

ProzessId = 1

virtuell	reell
0	5
1	-
2	2
3	-
4	7
5	-

ProzessId = 2

virtuell	reell
0	0
1	-
2	-
3	-
4	1
5	3

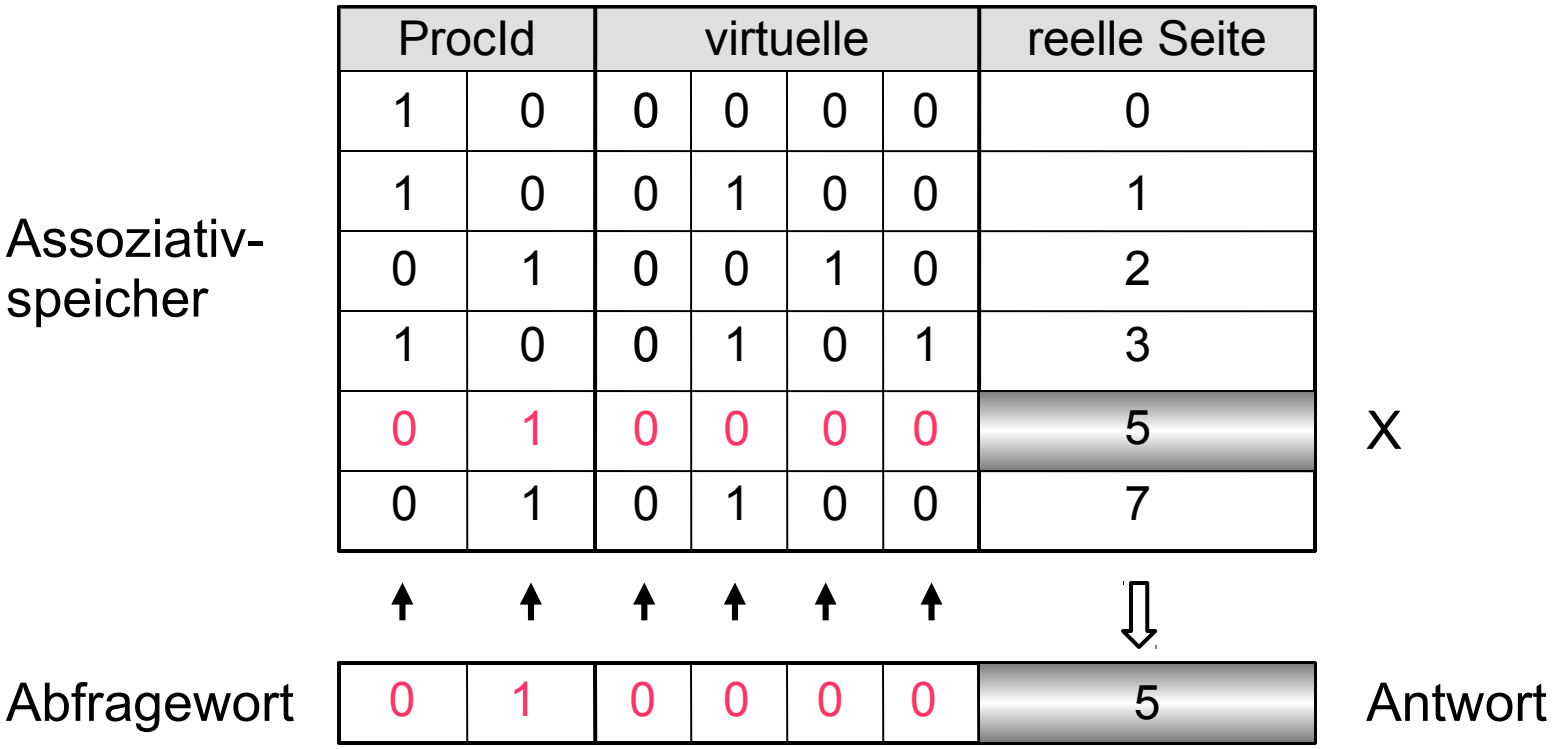
reell	virtuell	ProzId
0	0	2
1	4	2
2	2	1
3	5	2
4	-	-
5	0	1
6	-	-
7	4	1

Abbildung 3.10 **Zusammenfassen von Seitentabellen zu einer inversen Seitentabelle**

Ein großer Nachteil von invertierten Seitentabellen ist das ständige u.U. vollständige Durchsuchen der Tabelle bei jedem Zugriff.

3.4.3 Assoziativer Tabellencache

Ständiges Durchsuchen der Tabellen verbraucht viel Zeit, sodass es sinnvoll ist, Tabellenteile in einem schnellen Zwischenspeicher (cache) zu halten.



ProcId		virtuelle				reelle Seite
1	0	0	0	0	0	0
1	0	0	1	0	0	1
0	1	0	0	1	0	2
1	0	0	1	0	1	3
0	1	0	0	0	0	5
0	1	0	1	0	0	7
0	1	0	0	0	0	5

Abbildung 3.11 Funktion eines assoziativen Tabellencache

3.4.4 Gemeinsam genutzter Speicher (shared memory)

Gemeinsam genutzten Speicher kann es für denselben Programmcode geben, den mehrere Prozesse ausführen oder für Datenbereiche, auf die von verschiedenen Prozessen zugegriffen wird.

Um Speicherbereich eines Prozesses als globalen gemeinsamen Speicher (shared memory) zu deklarieren, müssen Betriebsystemaufrufe genutzt werden.

3.4.5 Virtueller Adressraum in UNIX

In UNIX HP-UX ist der virtuelle Adressraum 4 GB Worte groß, wird also durch eine 32-Bit-Adresse angesprochen. Der gesamte virtuelle Adressraum von 4 GB ist in vier einzelne Unterabschnitte (Quadranten) aufgeteilt.

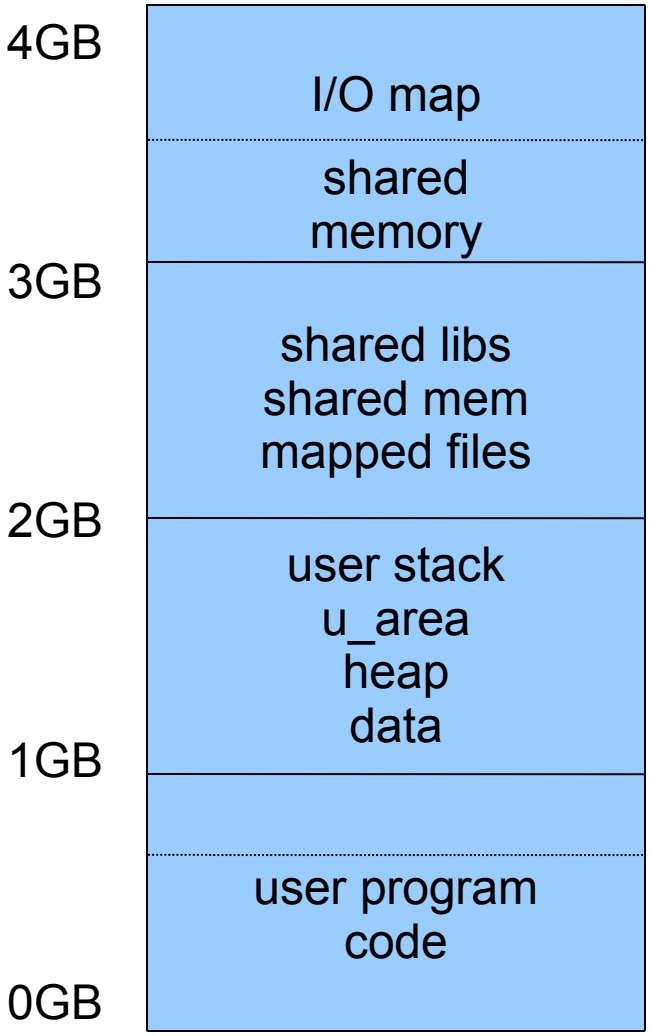


Abbildung 3.12 Der Adressraum für HP-UX/800

3.4.6 Seitenersetzungsstratgien

Eine fehlende Seite muss vom Massenspeicher in den Hauptspeicher kopiert werden, damit der Zugriff erfolgen kann.

Solange noch genügend Hauptspeicherseiten zur Verfügung stehen, kann schnell eingelagert werden.

Sind alle Hauptspeicherseiten belegt, findet eine Ersetzung statt.
Es muss eine Seite gefunden werden, die ausgelagert werden kann, d.h. falls sie verändert wurde, auf den Massenspeicher zurückkopiert wird.
Eine Strategie dazu kann als Schedulingstrategie für einen Seitenaustauschprozessor angesehen werden.

3.4.6.1 Referenzfolgen

Man kann den Zugriff auf die Seiten eines Prozesses protokollieren und erhält dann eine Nummernfolge der benötigten, referenzierten Seiten - die Referenzfolge (reference string).

Wüsste man diese Folge im Voraus, könnte man bei der Ersetzungsstrategie darauf reagieren. Kurzzeitstatistiken helfen, sich einer zu erwartenden Referenzfolge anzunähern. Für jede Seite werden Statusbits R und M gehalten. R = referenced, Seite benutzt. M = modified, Seite wurde verändert und muss zurück-geschrieben werden. Das R-Bit kann in regelmäßigen Abständen zurückgesetzt werden, so dass R = 1 aussagt, dass die Seite erst kürzlich benutzt wurde.

3.4.6.2 Optimale Strategie

Belady konnte zeigen, dass genau dann die wenigsten Ersetzungen vorgenommen werden, wenn man die Seite wählt, die am spätesten in der Zukunft benutzt werden wird.

Belady, L. "A Study of Replacement Algorithmus for a virtual Storage Computer.„
IBM Systems Journal, Nov. 2, 1966

Referenzfolge 0, 1, 2, 3, 4, 0, 1, 5, 6, 0, 1, 2, 3, 5, 6, ...

RAM	0	1	2	3	4	0	1	5	6	0	1
RAM		0	1	1	1	4	0	1	1	6	0
RAM			0	0	0	1	4	0	0	1	6
DISK				2	3	3	3	4	5	5	5
DISK					2	2	2	3	4	4	4
DISK								2	3	3	3
DISK									2	2	2

Abbildung 3.13 Optimale Seitenersetzungsstrategie

Die eingekreisten Zahlen der Tabelle sind das Ergebnis einer Seitenersetzung.

Mit der vertikalen Position einer Seitenzahl wird der relative Zeitpunkt eines Zugriffs auf eine Seite dargestellt.
Somit wandern einige Seiten anscheinend durchs RAM, was tatsächlich jedoch nicht der Fall ist.

Die vertikale Position der Seitenzahlen im DISK Bereich haben keine Bedeutung.

3.4.6.3 FIFO - Strategie

Es wird die Seite ausgelagert, die zeitlich die älteste ist.

In Abbildung 3.14 wird immer die Seite gewählt, die an der Grenze RAM/DISK liegt.

Referenzfolge 0, 1, 2, 3, 4, 0, 1, 5, 6, 0, 1, 2, 3, 5, 6, ...

RAM	0	1	2	(3)	(4)	(0)	(1)	(5)	(6)	(0)	(1)
RAM		0	1	2	3	4	0	1	5	6	0
RAM			0	1	2	3	4	0	1	5	6
DISK				0	1	2	3	4	0	1	5
DISK					0	1	2	3	4	4	4
DISK								2	3	3	3
DISK									2	2	2

Abbildung 3.14 FIFO Seitenersetzungsstrategie

Der Algorithmus berücksichtigt nicht, dass einige Seiten öfter referenziert werden.

Wird eine Seite im RAM erneut referenziert, betrachtet man sie wie eine neu referenzierte Seite. Sie erhält eine "Zweite Chance" (second chance) zum längeren Verbleib im RAM.

Man nennt diese Strategie Fifo – second chance.

3.4.6.4 NRU/RNU - Stategie

Bei der NRU- (Not Recently Used) bzw. RNU-Strategie (Recently Not Used) werden die Statusbits R und M zur Seitenersetzung ausgewertet. Alle Seiten im RAM erhalten folgende Klassennummer:

1: R=0, M=0
2: R=0, M=1
3: R=1, M=0
4: R=1, M=1.

Seiten mit der kleinsten Nummer werden zuerst ersetzt.

Referenzfolge 0, 1, 2, 3, 4, 0, 1, 5, 6, 0, 1, 2, 3, 5, 6, ...

RAM	0	1	2	(3)	(4)	(0)	(1)	(5)	(6)	0	1
RAM		0	1	2	3	4	0	1	1		
RAM			0	1	2	3	4	0	0		
DISK				0	1	2	3	4	5		
DISK					0	1	2	3			
DISK								2			
DISK											

Abbildung 3.15 NRU Seitenersetzungsstrategie

Ab der Anforderung von Seite 6 ergeben sich Änderungen zur FIFO-Strategie, da man nicht Seite 0, sondern Seite 5 verdrängt hat.

Bemerkung:

Aus der Referenzreihenfolge allein kann man dies jedoch nicht mehr ableiten. Es müssen die R- und M-Bits jeweils mitangegeben werden.

3.4.6.5 LRU - Stategie

Genauer als die FIFO-Strategie und das qualitative Messen einer Seitenreferenz der NRU-Strategie ist das quantitative Messen der Zeit, die eine Seite unbenutzt war. So kann bei mehreren Seiten mit R=0 die älteste ersetzt werden, was zur LRU-Strategie (Least Recently Used) führt.

Die Zeitmessung kann über den normalen Zeitzähler für die Uhrzeit erfolgen. Bei jeder Referenz einer Seite wird die aktuelle Uhrzeit abgespeichert. Die älteste Seite hat damit den kleinsten Zeitstempel.

Steht keine Hardware zur Verfügung, kann man das Altern durch ein Schieberegister pro Seite simulieren. In regelmäßigen Abständen wird das R-Bit in das Schieberegister jeder Seite geschoben.

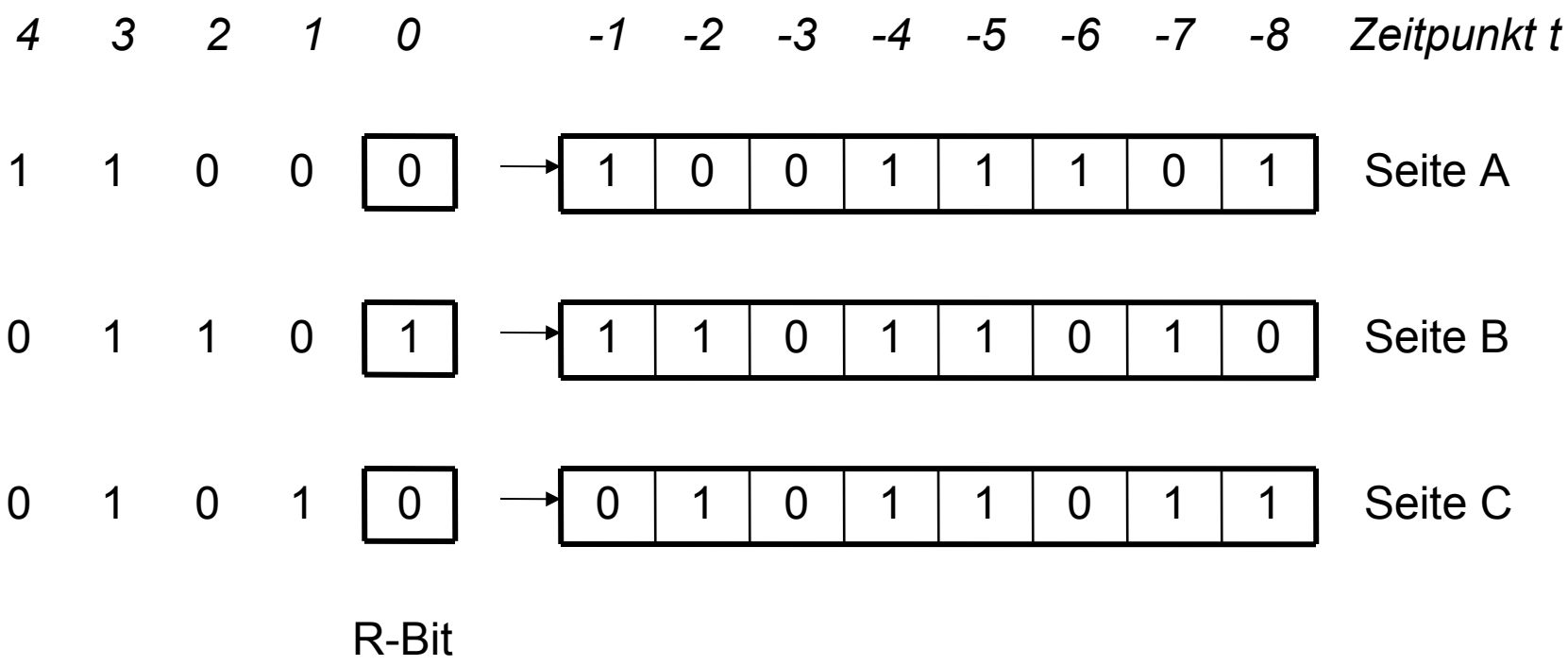


Abbildung 3.16 Realisierung von LRU mit Schieberegistern

Der Inhalt der Schieberegister wird als Dualzahl interpretiert. Die Seite mit der kleinsten Zahl wird ersetzt.

3.4.6.6 NFU/LFU- Stategie

Bei der NFU/LFU-Strategie (Not Frequently Used bzw. Least Frequently Used) wird die bisher am seltensten benutzte Seite ersetzt.

In Abb. 3.16 müsste man die Anzahl der 1en in den Schieberegistern zählen. Die Seite mit der kleinsten Summe wird ersetzt.

3.4.7 Analyse der Seitenersetzung

3.4.7.1 Optimale Seitenlänge

Sei k die Hauptspeichergröße und s die Größe einer Seite. Sei weiter die Länge der Daten zufallsmäßig gleich verteilt, d.h. alle Werte aus dem Intervall]0,s] treten als Verschnitt auf, dann ist der **mittlere Verschnitt s/2 Speicherworte** pro Prozess. Bei einer **einstufigen Seitentabelle** werden **[k/s] Einträge** (~Speicherwort) benötigt.

Pro Prozess entsteht ein **mittlerer Verlust** von **V = (k/s + s/2)**.

Mittlerer Verlust

V = (k/s + s/2).

Wird dieser durch einen Verlustfaktor f_k ausgedrückt, ergibt sich **V = k f_k**.

Je größer die Seite, desto kleiner die Seitentabelle und desto größer der mittlere Verschnitt und umgekehrt.

Ein lokales Minimum erhält man für **s_{opt} = sqrt (2k)**.

3.4.7.2 Optimale Seitenzahl

Ausgangspunkt ist die Frage, wieviele Prozesse im Hauptspeicher zugelassen werden sollen. Sie ist äquivalent mit der Frage, wieviele Seiten pro Prozess im Hauptspeicher reserviert sein müssen.
Die Annahme, je mehr Seiten pro Prozess im Hauptspeicher zur Verfügung stehen, desto weniger Seitenfehler treten auf, ist jedoch bei einigen Seitenersetzungsstrategien ungültig und wird als sogenannte "Belady-Anomalie" bezeichnet.

Am Beispiel der FIFO-Strategie für 4 und 5 RAM-Seiten sei diese Anomalie gezeigt.

Referenzfolge 0, 1, 2, 3, 4, 0, 1, 5, 6, 0, 1, 2, 3, 5, 6, ...

RAM	3	(4)	(0)	(1)	(5)	(6)	6	6	(2)	(3)	3	3
RAM	2	3	4	0	1	5	5	5	6	2	2	2
RAM	1	2	3	4	0	1	1	**1**	5	6	6	**6**
RAM	0	1	2	3	4	0	**0**	0	1	5	**5**	5
DISK		0	1	2	3	4	4	4	0	1	1	1
DISK					2	3	3	3	4	0	0	0
DISK						2	2	2	3	4	4	4

Abbildung 3.17a FIFO Strategie mit 4 RAM-Seiten

Referenzfolge 0, 1, 2, 3, 4, 0, 1, 5, 6, 0, 1, 2, 3, 5, 6, ...

RAM	3	4	4	4	5	6	0	1	2	3	5	6
RAM	2	3	3	3	4	5	6	0	1	2	3	5
RAM	1	2	2	2	3	4	5	6	0	1	2	3
RAM	0	1	1	**1**	2	3	4	5	6	0	1	2
RAM		0	**0**	0	1	2	3	4	5	6	0	1
DISK					0	1	2	3	4	5	6	0
DISK						0	1	2	3	4	4	4

Abbildung 3.17b FIFO Strategie mit 5 RAM-Seiten

Für die FIFO - Strategie mit “second chance” tritt diese Anomalie nicht auf.

Sie gehört zu den sogenannten Stack-Strategien, da man die im RAM gehaltenen Seiten über einen Stack verwalten kann. Wird die maximale Stacktiefe überschritten, wird nicht die oberste Seite wie gewöhnlich für eine Stackverwaltung ersetzt, sondern der Stack entlässt seine unterste Seite und sackt somit wieder auf seine normale Größe zusammen.

Referenzfolge 0, 1, 2, 3, 4, 0, 1, 5, 6, 0, 1, 2, 3, 5, 6, ...

RAM	3	(4)	(0)	(1)	(5)	(6)	**0**	**1**	(2)	(3)	(5)	(6)
RAM	2	3	4	0	1	5	6	0	1	2	3	5
RAM	1	2	3	4	0	1	5	6	0	1	2	3
RAM	0	1	2	3	4	0	1	5	6	0	1	2
DISK		0	1	2	3	4	4	4	5	6	0	1
DISK					2	3	3	3	4	5	6	0
DISK						2	2	2	3	4	4	4

Abbildung 3.18a FIFO Strategie Second Chance mit 4 RAM-Seiten

Referenzfolge 0, 1, 2, 3, 4, 0, 1, 5, 6, 0, 1, 2, 3, 5, 6, ...

RAM	3	4	**0**	**1**	(5)	(6)	**0**	**1**	(2)	(3)	(5)	(6)
RAM	2	3	4	0	1	5	6	0	1	2	3	5
RAM	1	2	3	4	0	1	5	6	0	1	2	3
RAM	0	1	2	3	4	0	1	5	6	0	1	2
RAM		0	1	2	3	4	4	4	5	6	0	1
DISK					2	3	3	3	4	5	6	0
DISK						2	2	2	3	4	4	4

Abbildung 3.18b FIFO Strategie Second Chance mit 5 RAM-Seiten

3.4.7.3 Working Set

Denning definierte den working set als minimale Anzahl Seiten, die im RAM vorhanden sein müssen, damit ein Programm ohne Seitenfehler ablaufen kann. Betrachtet man die Maschinenbefehle

MOVE A, B
MOVE C, D

so ist der working set = 5, da die Codeseite und jede Variablenreferenz auf einer anderen Seite liegen können. Der mittlere erwartete working set $\langle W(t, \Delta t) \rangle_t$ charakterisiert einen gesamten Prozessverlauf. Er enthält die zu einem Zeitpunkt t in einem Zeitfenster Δt zuvor referenzierten Seiten.

Bisher haben unsere Strategien das bedarfsgerechte Einlagern (und Ersetzen) von Seiten durchgeführt. Es wird als "demand paging" bezeichnet.

Im Gegensatz dazu wird das voraussehende Einlagern von (wahrscheinlich) benötigten Seiten als "prepaging" bezeichnet. Der working set enthält Kandidaten für das prepaging.

3.4.7.4 Trashing

Ist die mittlere Seitenverweilzeit t_s (die Phase, in der ein Prozess im Zustand running ist) größer als die mittlere Wartezeit t_w (Prozess im Zustand blocked), so kann stets ein Prozess gefunden werden, der rechenbereit ist.

Die gesamte Bearbeitungsdauer eines einzelnen Prozesses ist dann proportional seiner Bearbeitungsdauer und der Anzahl Prozesse im System.

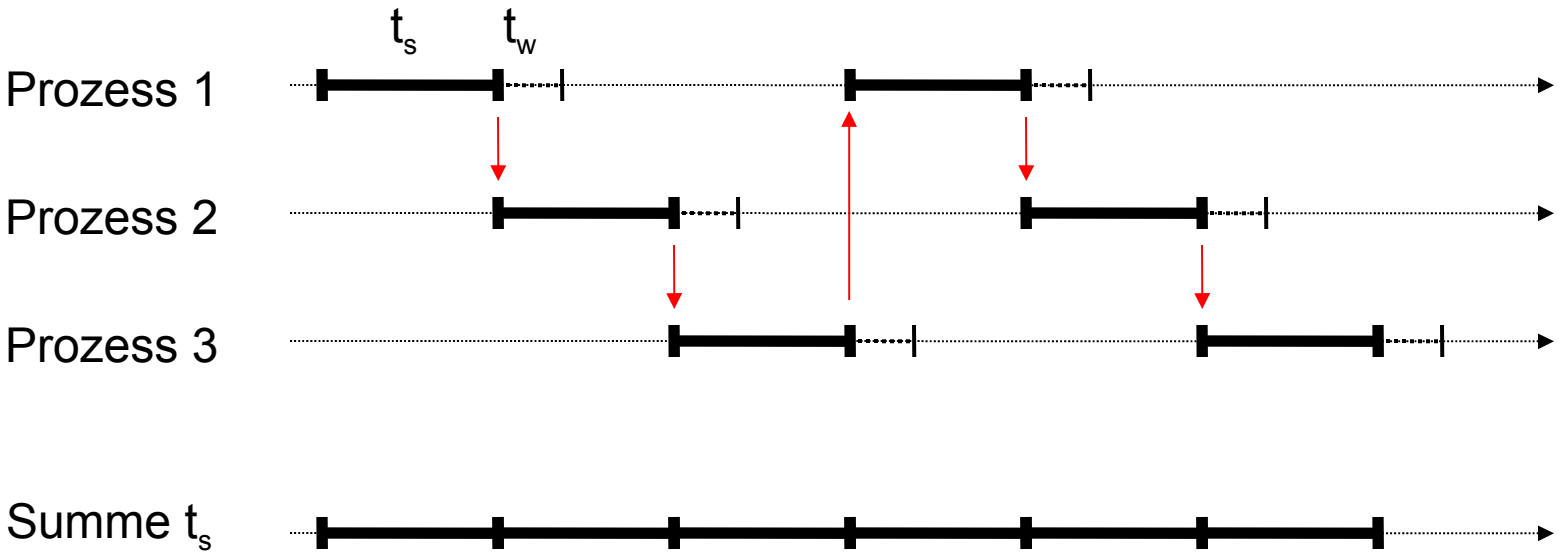


Abbildung 3.19 Überlappung von Rechen- und Seitenaustausch-phasen bei $t_s > t_w$

Erhöht man die Anzahl der Prozesse im System drastisch, wird t_s kleiner.

Wird t_w größer t_s, ist nicht mehr der Prozessor das knappste Betriebsmittel, sondern der DMA-Kontroller, der für die Ausführung des Seitenaustausch verantwortlich ist.

Die Gesamtbearbeitungsdauer der einzelnen Prozesse steigt stark an, was "trashing" genannt wird.

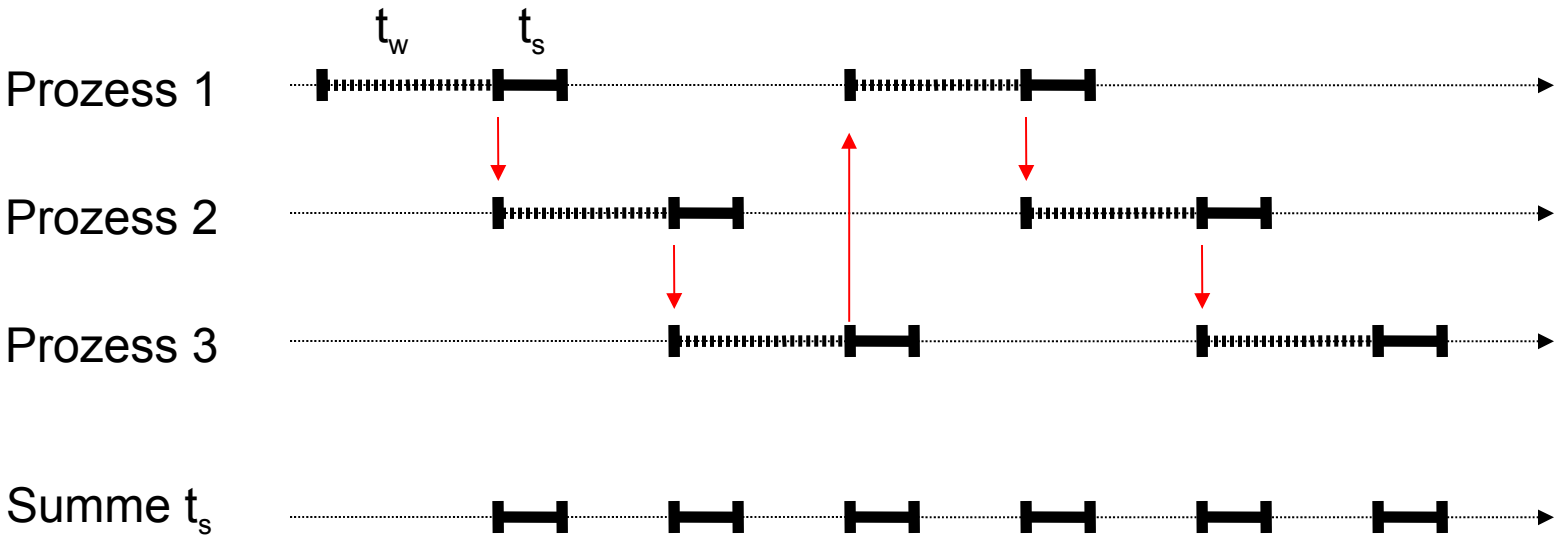


Abbildung 3.20 Überlappung von Rechen- und Seitenaustauschphasen bei $t_s < t_w$

3.4.8 Seitenersetzungsstrategie in UNIX

In HP-UX werden die Mechanismen swapping und paging unterstützt. Beim swapping werden ganze Prozesse ein- und ausgelagert. Swap-Space kann unterschiedlich zur Verfügung stehen.

Abhängig von der Zugriffsgeschwindigkeit und Datensicherheit klassifiziert man:

- swap space auf swap disk: eigenständige physikalische Einheit
- swap space auf logischem Laufwerk: logische Einheit, swap partition
- swap space als Teil des Dateisystem: swap directory

Paging ist das Ein- und Auslagern von einzelnen Seiten. Es arbeiten die zwei Hintergrundprozesse (pageout demon, swapper demon) wie folgt zusammen:

(a) Der **pageout demon setzt** in regelmäßigen Abständen das **R-Bit zurück** und lagert alle Seiten mit gelöschtem R-Bit aus, wenn mehr als 75% der Seiten benutzt sind.

(b) Steht nur sehr **wenig Hauptspeicherplatz** zur Verfügung,

deaktiviert der **swapper demon** solange **Prozesse**, bis das handhabbare Minimum wieder erreicht wird.

Es kann ein sogenanntes “Seitenflattern" auftreten, d.h. ein gerade ausgelagerter Prozess wird sofort wieder aktiviert.

Abhilfe schafft das Festlegen von minimalen und maximalen Schwellenwerten zur Prozessdeaktivierung.

Wenn a+b nicht zum Erfolg führen, muss

mehr Hauptspeicher und/ oder **schnellerer swap space** bereitgestellt werden.

3.5 Segmentierung

Bisher wurde der benötigte Speicher eines Programms als homogenes, kontinuierliches Feld betrachtet. Tatsächlich benötigt ein Programm in unterschiedlichen Bereichen mehr oder weniger Hauptspeicher bei der Programmausführung. Als Bereiche lassen sich leicht identifizieren: Programm, Heap und Stack. Die Verwaltung dieser Bereiche erleichtert sich, wenn jeder Bereich bei Adresse Null beginnen kann. Durch eine sogenannte segmentierte Speicheradressierung kann man die gewünschte Verwaltung erreichen. Die virtuelle Adresse setzt sich aus einer Segmentadresse und einem Offset zusammen. Die Segmentadresse bestimmt die Lage des durch den Offset referenzierten Speicherbereichs im Hauptspeicher.

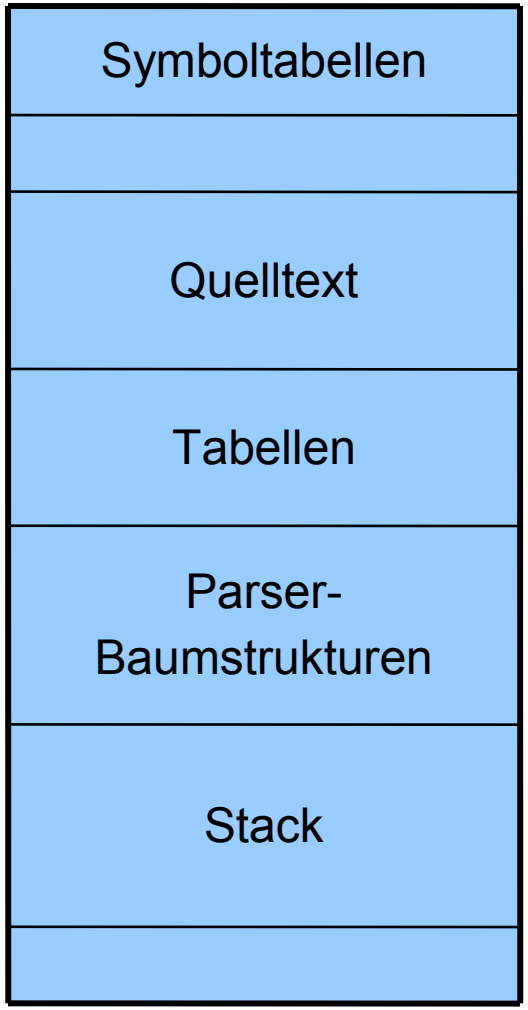


Abbildung 3.21 Speichereinteilung eines Compilers

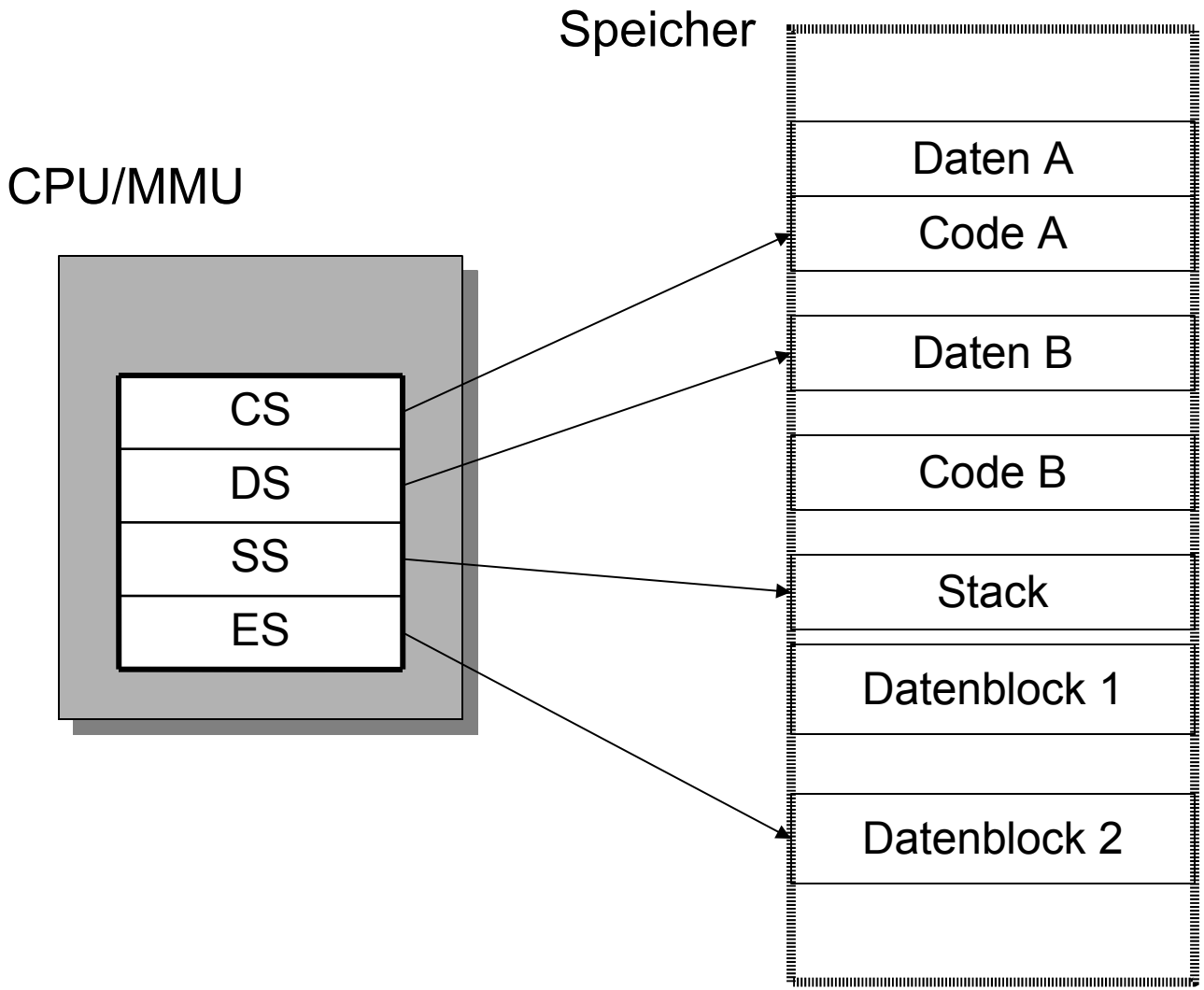


Abbildung 3.22 Speicherverwaltung beim Intel 80286

Um Adresslöcher zu vermeiden, kombiniert man Segmentierung mit Paging. Es müssen getrennte Segment- und Seitentabellen gehalten werden.

Segmente besitzen individuelle Zugriffsrechte. Man kann damit Speicherbereiche schützen bzw. gemeinsam genutzten Speicher mehrerer Prozesse implementieren.

3.6 Cache

Ein Cache ist ein schneller Zwischenspeicher, in dem Kopien der Daten des Speichers liegen, auf den eigentlich zugegriffen werden soll.

Man unterscheidet:

1. pipeline-burst-cache
2. Hit/Miss - Mechanismus zur Cache-Aktualisierung

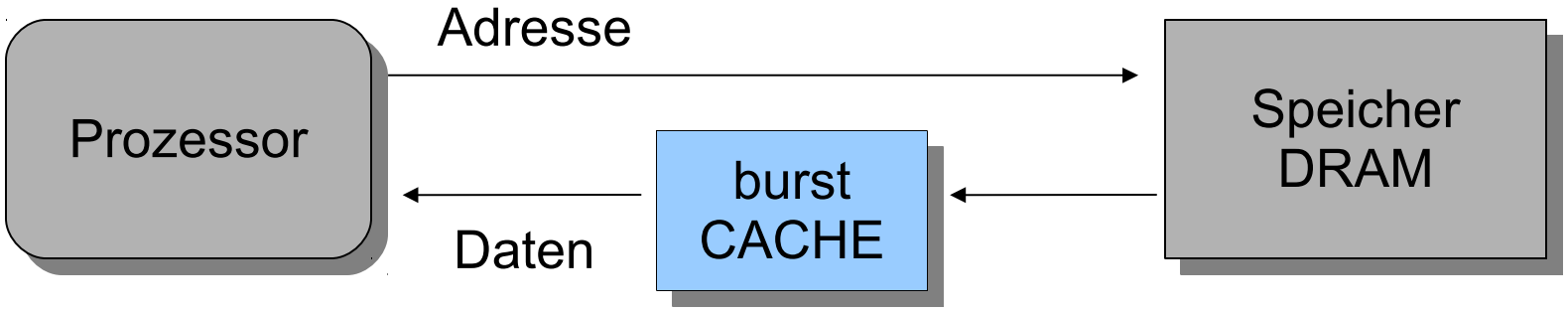


Abbildung 3.23 Benutzung eines pipeline-burst-Cache

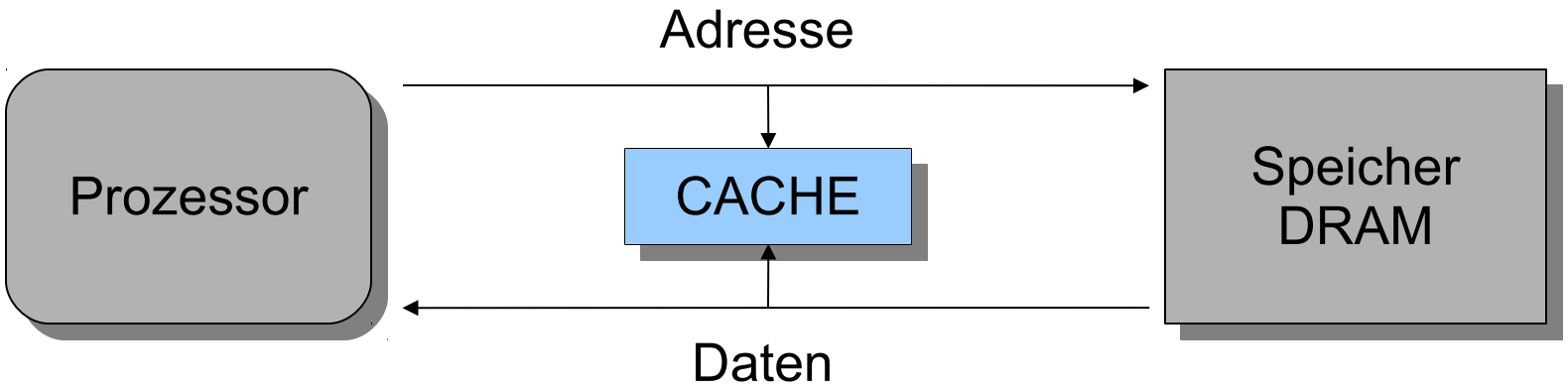


Abbildung 3.24 Adress- und Datenanschluss des Cache