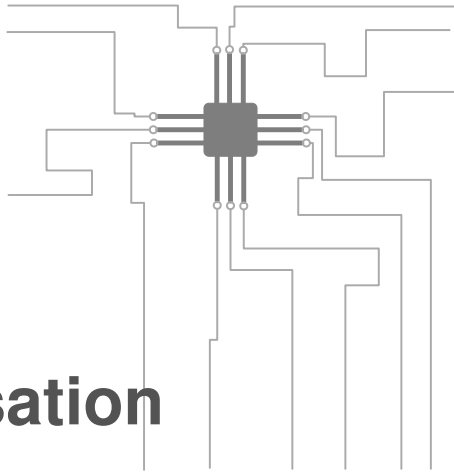

9 Speicherorganisation

In diesem Kapitel behandeln wir...

- verschiedene Speicherkonzepte
- Algorithmen zur effektiven Speichernutzung
- Rechteverwaltung von Speicher

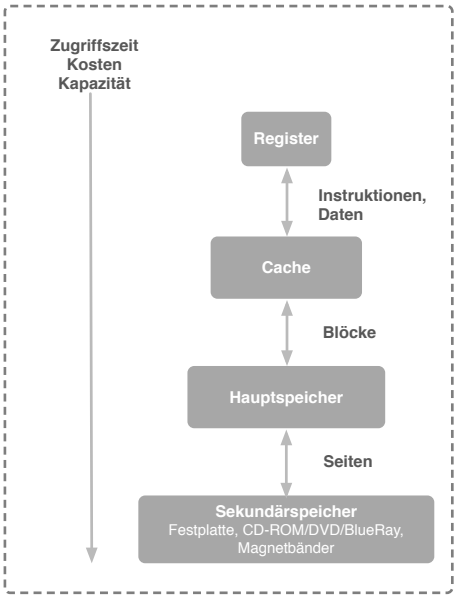

Abbildung 9.1: Speicherpyramide in modernen Rechnern.

9.1 Speicherhierarchie

In heutigen Computern befinden sich Speicher mit verschiedenen Charakteristiken. Generell kann angenommen werden, dass je näher der Speicher an der CPU sitzt, desto schneller ist er in Bezug auf die Zugriffszeit. Gleichzeitig nimmt seine Größe stark ab. Ein Grund ist, dass die Kosten bei einem schnellen Speicher höher sind. Die verschiedenen Speicherarten werden meist für unterschiedliche Zwecke eingesetzt. Wie in Abbildung 9.1 dargestellt, stellen Register den schnellsten und gleichzeitig auch kleinsten Speicher zur Verfügung. Sie sind direkt in die CPU integriert und werden zum Zwischenspeichern von Instruktionen und Daten eingesetzt. Je nach Architektur stehen 15 Register zur Verfügung, die je ein Datenwort fassen. Die geladenen oder berechneten Daten werden oft in Blöcken im Cache abgelegt. Er ist mit seiner Speichergröße ebenfalls klein. Der Cache ist mit dem Hauptspeicher verbunden, welcher seitenweise arbeitet. Hier liegen wir schon deutlich im Gigabytebereich. Der letzte, größte und langsamste Stein in der Pyramide ist der Sekundärspeicher, wie Festplatten, CD-ROM und andere [45, Seite 208-209].

9.2 Speicherverwaltung

Über die Jahre, die mit der Entwicklung der Computer einhergegangen sind, wurden verschiedene Techniken zur Speicherverwaltung entwickelt. Die heute noch eingesetzten Techniken hängen vom Zielsystem ab. So finden sich in eingebetteten Systemen häufig noch ältere Techniken. Diese sind für moderne Mehrzweckbetriebssysteme nicht mehr zeitgemäß. In eingebetteten Systemen, vor allem solchen mit Sicherheitsanforderungen, bestehen andere Anforderungen an die Speicherverwaltung. Oft wird hier der Speicher bereits vorab statisch allokiert, um auch in Situationen mit extremer Speicherauslastung genügend Speicher für alle Dienste zur Verfügung zu haben.

Wenden wir uns im Folgenden von den eingebetteten Systemen ab und betrachten moderne Mehrzweckbetriebssysteme.

9.3 Virtueller Speicher

Was ist die Motivation für das Konzept des virtuellen Speichers?

Der physikalische Speicher ist nie groß genug!

Dafür gibt es verschiedene Gründe. Entweder der Speicher ist zu teuer, oder der Speicherbedarf von Anwendungen steigt zu schnell. Hinzu kommt eine weitere Komponente, der Programmierer. Unabhängig davon, wie groß der Speicher gerade sein kann, schafft es der Programmierer immer, an dessen Grenzen und darüber hinaus zu gehen. Weiter ist es so, dass nicht jeder Rechner mit der maximalen Anzahl an physikalischem Speicher ausgestattet ist.

Über die Zeit kam es zu verschiedenen Lösungsstrategien. Der erste Ansatz war der Versuch, mit alternativen Algorithmen den Verbrauch eines Programms zu senken. Liefern zwei Algorithmen dasselbe Resultat, wird derjenige mit dem geringeren Speicherverbrauch gewählt, auch wenn er langsamer in der Durchführung ist.

Als nächste Technik wurden Overlays eingeführt. Hier mussten die Programmierer ihre Programme unterteilen und mittels speziellen Anweisungen zur Bedarfszeit entladen und laden. Bei diesem Ansatz mussten die Programmierer stark in ihr Programm eingreifen und viel über die Zielarchitektur wissen. In der heutigen Zeit, wo universelle portable Programme ein Ziel sind, undenkbar.

Eine Verbesserung brachte ein zweiter Ansatz. Hier wird die Verwaltung der Overlays vom Betriebssystem übernommen. Das nimmt dem Programmierer einiges an Arbeit ab. Was weiter bei ihm hängen bleibt, ist die Aufteilung seines Programmes.

Ein richtiger Durchbruch wurde erst 1961 von John Fotheringham erzielt. *Dynamic storage allocation in*

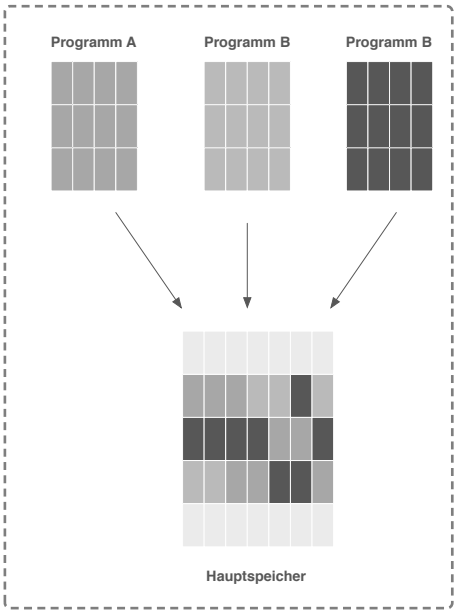

Abbildung 9.2: Virtueller Speicher.

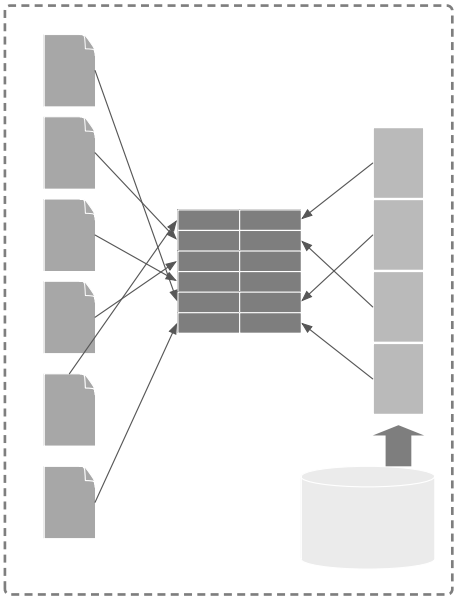

Abbildung 9.3: Beim Seitenverfahren (paging), ist der physikalische Speicher in Einheiten, den Seitenrahmen (page frames), unterteilt. Der virtuelle Adressraum wird in Seiten (pages) unterteilt.

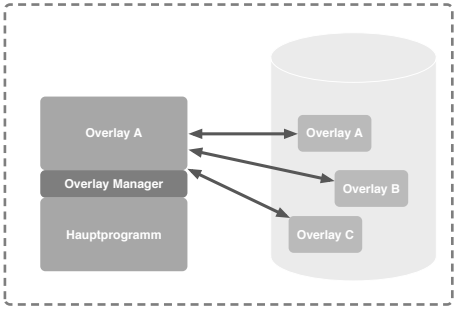

Abbildung 9.4: Speichererweiterung durch Überlagerung.

the Atlas computer, including an automatic use of a backing store [54, Seite 435, 436]. Er führte das Konzept des virtuellen Speichers ein. Für Entwickler wird ein großer zusammenhängender Speicherbereich simuliert. Dieser kann größer sein, als der tatsächlich zur Verfügung stehende physikalische Speicher. Außerdem ist er plattformunabhängig. Der Programmierer muss auch keine speziellen Adressen mehr kennen.

Virtueller Speicher ist eine Abstraktionsschicht für einen Prozess. Unabhängig vom tatsächlich installierten Hauptspeicher, kann ein Prozess hier die volle Bit-Breite adressieren. Er trennt den Adressraum von der physikalischen Speicherstelle.

9.4 Überlagerungen

Eine der ersten Techniken, um den Speicherhunger von Programmierern zu stillen, war die Overlay-Methode. Passte ein Programm nicht vollständig in den Hauptspeicher, war es an dem Programmierer, dieses in Häppchen zu unterteilen. Diese Häppchen wurden dann nach Bedarf zur Laufzeit geladen [55, Seite 486]. Es galt Abschnitte zu identifizieren, welche sich wechselseitig ausschließen. Beim Start des Programms wurde dann standardgemäß das Overlay 0 geladen [56, Seite 241]. Von hieraus musste der Programmierer dafür sorgen, das neue Häppchen nachgeladen wurden, wenn der Bedarf entstand. Gleichzeitig musste der aktuelle Happen aus dem Speicher entfernt werden. Eine Herausforderung war es, die Häppchen zur Laufzeit so zu organisieren, das stets alle erforderlichen in den Hauptspeicher passten und dass bei der Ausführung keines der Häppchen unwillentlich überschritten wurde [55, Seite 486].

Vom mangelnden Spaßfaktor abgesehen, war es eine Mammutaufgabe, die bei der Komplexität der heutigen Programme schlicht nicht mehr zu bewältigen wäre. Die Unterteilung war auch sehr stark hardwaregebunden. Sie erfolgte mit einer klaren Zielplattform, die weitestgehend unverändert bliebt. Dies ist ein weiterer Faktor, der heute so nicht mehr gegeben ist.

9.5 Swapping

Swapping war die erste einfache Technologie, die im Betriebssystem verwendet wurde, um das Problem des Speicherüberlaufs zu lösen. Es werden die Segmente eines Prozesses in Sekundärspeicher ausgelagert. Bei Linux existiert zu diesem Zweck die swap-Partition, Windows verwendet die Auslagerungsdatei. Ziel ist es, Wartezeiten zu überbrücken, was häufig der Fall ist, wenn auf Eingabe- / Aufgabeoperationen gewartet wird. In dieser Zeit kann das RAM des Prozesses an andere Prozesse vergeben werden. Es sollen möglichst viele ausführungsbereite Prozesse im Hauptspeicher sein. Ein Prozess kann nur ausgelagert werden, wenn es sicher ist, dass er sich im von idling, benennt das Fehlen von Bewegung-Zustand befindet. Am Ende der Wartezeit werden die Segmente wieder eingelagert. Da der Vorgang zeitaufwendig ist, wird er praktisch nicht mehr verwendet. Moderne Betriebssysteme greifen erst auf diese Technik zurück, wenn der Speicherverbrauch zu hoch wird und Techniken wie Paging alleine nicht mehr helfen. Die meisten Betriebssysteme kombinieren somit Swapping und Paging.

Die geltenden Kriterien dabei sind Prioritäten und Fairness. Die Prozesse sollen möglichst rotierend ausgelagert werden.

Tritt unter Linux die Situation ein, dass sowohl Haupt- als auch Swapping-Speicher voll ist, schlägt der Out-of-memory (OOM)-Killer zu. Dieser beendet einen beliebigen Prozess, um wieder Speicher zu gewinnen.

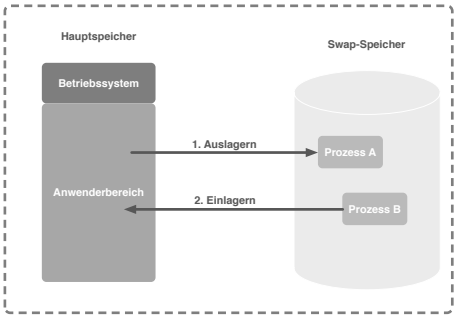

Abbildung 9.5: Auslagern des Speichers mittels Swapping.

9.6 Effektivität von virtuellem Speicher

Durch das Einführen von virtuellem Speicher haben wir das Problem adressiert, dass der Hauptspeicher effektiver genutzt wird. Er kann auch größer erscheinen als er ist, was die Verwendung aller Adressen ermöglicht. Unabhängig davon, wieviel physikalischer Speicher verbaut ist. Auf der Minusseite haben wir uns eine Indirektion für den Zugriff auf den Speicherinhalt eingehandelt. Wie sich diese auswirkt, wollen wir uns in dem folgenden Abschnitt ansehen.

Effective Access Time (EAT) bezeichnet die Zeit, welche tatsächlich für den Zugriff auf den Inhalt einer virtuellen Seite benötigt wird. Beim virtuellen Speicher haben wir einen Zeitverlust, da hier mindestens zwei Speicherzugriffe erfolgen. Zunächst wird die Page Table nach dem physikalischen Rahmen gefragt, anschließend wird dieser gelesen. Beides sind RAM Zugriffe. Nehmen wir ein System, das für den Speicherzugriff 200 ns benötigt und eine Page Fault Rate von 1 % hat. Muss die Seite aus dem Sekundärspeicher geladen werden, dauert dies 10 ms [57, Seite 258]. Die EAT t_{eff} bestimmt sich durch die Wahrscheinlichkeit des Eintretens eines Page faults $0 < p <= 1.0$ und der Zeit für einen Speicherzugriff t_{ma}, multipliziert mit dessen Wahrscheinlichkeit:

$$t_{eff} = (1-p) \times t_{ma} + Overhead \times p$$

Setzen wir in diese Formel die Werte von oben ein, erhalten wir:

$$\begin{aligned} t_{eff} &= 0,99 \times (2 \times 200 \text{ ns}) + 0,01 \times (10 \text{ ms}) \\ &= 1000,396 \text{ ns} \end{aligned}$$

Die mittlere Zugriffszeit beträgt also 1.000,396 ns. Wie ist es mit der theoretisch bestmöglichen Zugriffszeit? Wir können die Formel und die Eckdaten des Beispiels weiter verwenden. Lediglich die Wahrscheinlichkeit eines Page faults beträgt im bestmöglichen Fall 0 %:

$$\begin{aligned} t_{eff} &= 1,0 \times (2 \times 200 \text{ ns}) \\ &= 400 \text{ ns} \end{aligned}$$

Mit gerade einmal 400 ns liegt die bestmögliche Zeit deutlich unter der durchschnittlichen. Die Flexibilität unseres Speichers haben wir uns recht teuer erkauft.

9.7 Seitentabelle

Die Seitentabelle kann beliebig viele Bits haben. Eine Beschränkung gibt es nur für die Adresse, die angefragt wird. Aus dieser muss sich der Seitenrahmen und der Offset in der physikalischen Seite abbilden lassen. Schutzbits werden nicht in dieser Adresse kommuniziert, sie können die Tabelle also erweitern. Womit beispielsweise bei einem 32 Bit System, Seitentabellen mit mehr als 32 Bit vorstellbar sind.

Betrachten wir ein System, welches über 32 kByte physikalischen Speicher verfügt. Der virtuelle Speicher soll 64 kByte groß sein. Die Seitengröße soll 4 kByte und die Adresslänge 16 Bit betragen.

Wie viele virtuelle Seiten werden benötigt? Die Anzahl der benötigten Seiten, c_{vS}, ergibt sich, indem wir den gewünschten virtuellen Speicher durch die Größe einer Seite teilen:

$$\begin{aligned} c_{vS} &= \frac{\text{virtueller Speicher}}{\text{Seitengröße}} \\ &= \frac{64\ \text{kByte}}{4\ \text{kByte}} \\ &= 16 \end{aligned}$$

Wir kennen nun die Anzahl virtueller Seiten. Diese müssen von physikalischen Seiten abgebildet werden. Es stellt sich die Frage, wie viele physikalische Seiten, c_{pS}, das System hat.

$$\begin{aligned} c_{pS} &= \frac{\text{physikalischer Speicher}}{\text{Seitengröße}} \\ &= \frac{32\ \text{kByte}}{4\ \text{kByte}} \\ &= 8 \end{aligned}$$

Jetzt stellt sich noch die Frage nach der Anzahl Bit die für eine Seite benötigt werden. Wir wollen pro Seite 4 kByte Daten speichern.

$$4\ \text{kByte} = 2^2 \times 2^{10} = 2^{12}$$

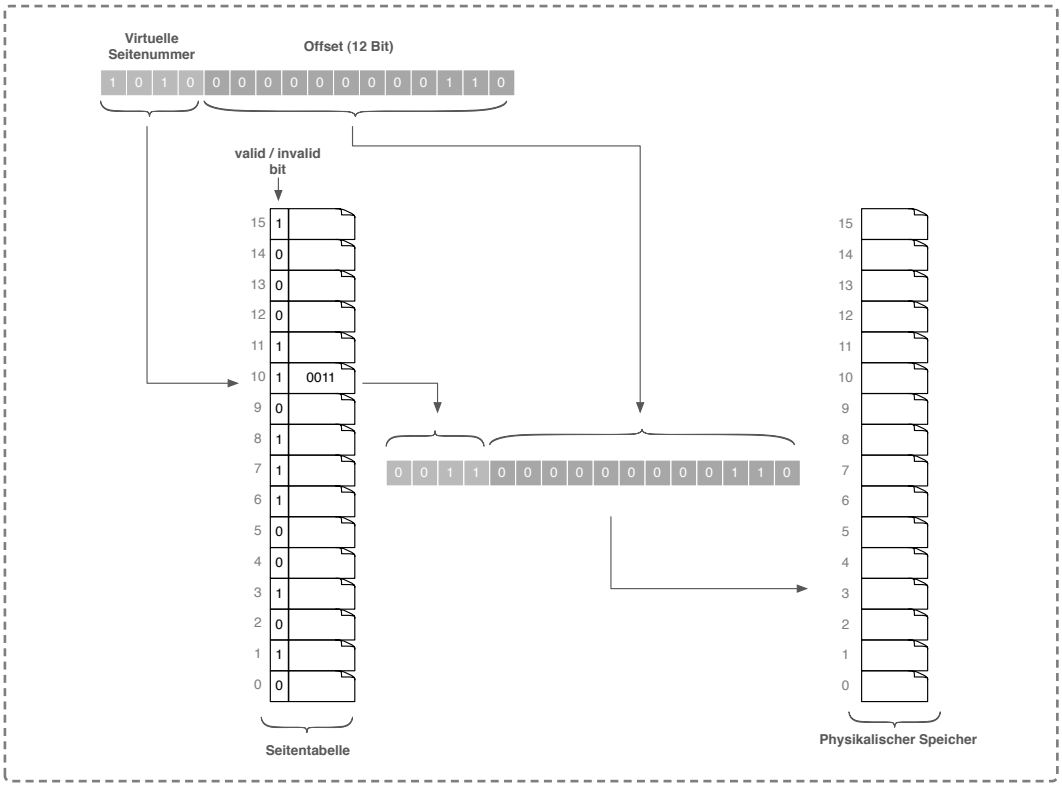

Abbildung 9.6: Umsetzung einer virtuellen Adresse in eine physikalische. Die virtuelle Seitennummer wird abgespalten und dient als Index für die Seitentabelle. Die Einträge an der Indexstelle, kombiniert mit dem Offset der virtuellen Adresse, ergibt die physikalische Adresse.

Es braucht also 12 Bit um 4 kByte darzustellen.

Das von uns betrachtete System verfügt über eine Adresslänge von 16 Bit. Von diesen benötigen wir 12 zur Adressierung innerhalb einer Seite. Die verbleibenden 4 Bit stehen zum Adressieren der gewünschten Seite zur Verfügung. Hier lassen sich exakt 16 Seiten realisieren.

9.8 Seitenersetzung

Befindet sich eine Seite nicht im Hauptspeicher, muss sie in diesen nachgeladen werden. Wie wir in Abschnitt 9.6 gesehen haben, kostet dies empfindlich viel

Zeit. Es gilt also, das Nachladen auf ein Minimum zu reduzieren. Für die Ersetzung von Seiten im Hauptspeicher gibt es verschiedene Algorithmen.

Untersuchen wir zunächst, was der optimale Algorithmus wäre. Der Algorithmus müsste die Zeitdauer bis zum nächsten Zugriff auf die entsprechende Seite kennen und berücksichtigen. Ferner würde er nur eine minimale Anzahl von Ersetzungen vornehmen. Er wäre auch gegen die Belady Anomalie immun. Als Fazit: Er würde immer die Seite mit dem größten Vorwärtsabstand ersetzen.

Dieser Algorithmus müsste also entweder in die Zukunft schauen können oder es müsste sich um ein deterministisches Programm handeln. Da es unmöglich ist, das Verhalten eines Prozesses bezüglich seiner Speicherzugriffe zu prognostizieren, handelt es sich um einen theoretischen Algorithmus. Belady war der erste Forscher, der diesen optimalen Algorithmus entdeckt hat [35, Seite 461].

Ungeachtet der Tatsache, dass dieser Algorithmus nicht implementierbar ist, wird er gerne als Referenz verwendet, um die Qualität realer Algorithmen zu vergleichen.

Wie in Tabelle 9.1 zu sehen, benötigt der optimale Algorithmus 7 Ersetzungen für die gegebene Referenzfolge.

Der Einsatz der Technik von virtuellem Speicher funktioniert, weil zwei Prinzipien oft greifen:

Temporale Lokalität : Wenn ein Zugriff auf ein Datum erfolgt, wird mit großer Wahrscheinlichkeit in Kürze erneut darauf zugegriffen.

Räumliche Lokalität : Wenn auf ein Datum zugegriffen wird, erfolgt mit großer Wahrscheinlichkeit auch auf Daten oder Adresse in der Nähe ein Zugriff.

Die räumliche Lokalität wird durch die Anordnung und Organisation des Speichers erreicht. Dieser ist in Blöcke, Seiten und Rahmen aufgeteilt, welche mit Daten aufeinanderfolgender Adressen befüllt sind.

Um die temporale Lokalität auszunutzen, gibt es verschiedene Verfahren.

Referenzfolge	Kachel 1	Kachel 2	Kachel 3
1	**1**		
2	1	**2**	
3	1	2	**3**
4	1	2	**4**
1	1	2	4
2	1	2	4
5	1	2	**5**
1	1	2	5
2	1	2	5
3	**3**	2	5
4	**4**	2	5
5	4	2	5

Tabelle 9.1: Der optimale Ersetzungsalgorithmus.

Referenzfolge	Kachel 1	Kachel 2	Kachel 3	Kachel 4
1	**1**			
2	1	**2**		
3	1	2	**3**	
4	1	2	3	**4**
1	1	2	3	4
2	1	2	3	4
5	**5**	2	3	4
1	5	**1**	3	4
2	5	1	**2**	4
3	5	1	2	**3**
4	**4**	1	2	3
5	4	**5**	2	3

Tabelle 9.3: FIFO Variante mit vier Kacheln.

First in first out (FIFO) bei der Seitenersetzung mittels eines einfachen FIFO Ansatzes erwarten wir, dass je größer das FIFO dimensioniert ist, desto weniger Einlagerungen benötigt werden. Ein Beispiel ist in Tabelle **??** gezeigt.

Führen wir die FIFO Strategie mit der gleichen Anzahl Kacheln sowie der gleichen Sequenz wie beim optimalen Algorithmus aus, benötigt es 9 Einlagerungen. 2 mehr als mit dem theoretisch besten Algorithmus.

Führen wir die Folge nochmals aus, diesmal mit 4 Kacheln, ergibt sich das in Tabelle 9.3 gezeigte Bild.

Trotz mehr Kacheln verschlechtert sich der Algorithmus und benötigt nun 10 Einlagerungen.

Dieses Verhalten wurde bereits 1969 von Belady et al. [58, Seite 349 - 353] erkannt. Es wird von der Belady Anomalie gesprochen, wenn Systeme mit größerem Hauptspeicher mehr Seitenfehler produzieren als Systeme mit geringem Hauptspeicher. Der Grund liegt im Vorgehen bei der Ersetzung des FIFO Algorithmus. Es wird stets der älteste Block als erstes überschrieben, unabhängig davon, wie häufig oder wann zuletzt er benutzt wurde.

Last Recently Used (LRU) basiert auf der Annahme, dass Seiten, welche von den letzten Befehlen benutzt wurden, wahrscheinlich auch für die folgenden gebraucht werden. Hier werden Seiten, die schon lange nicht mehr benutzt wurden, ausgelagert. Dies wird durch eine verkettete Liste aller möglichen Seiten realisiert. Den Anfang bildet die aktuellste, zuletzt benutzte Seite. Die am längsten unbenutzte Seite bildet das Ende der Liste. Es handelt sich hier um kein einfaches FIFO, da die Liste mit Zeitstempeln angereichert ist. Diese werden meist durch Hardware generiert.

Last Frequently Used (LFU) ähnelt der LRU Ersetzung. Hier wird die Seite entfernt, welche über