WWW.PEARSON-STUDIUM.DE
CWS
COMPANION WEBSITE

it informatik

Andrew S. Tanenbaum

Moderne Betriebssysteme

3., aktualisierte Auflage

Prentice Hall

PEARSON Studium

Speicherverwaltung

3

ÜBERBLICK

» Arbeitsspeicher (RAM) ist ein wichtiges Betriebsmittel, das sorgfältig verwaltet werden muss. Ein durchschnittlicher PC hat heute zwar 10.000-mal so viel Speicher wie die IBM 7094, in den frühen Sechzigern der größte Computer der Welt, aber die Programme wachsen schneller als der verfügbare Speicher. Parkinsons Gesetz[1], auf diese Situation übertragen, besagt, dass Programme sich ausdehnen, um den verfügbaren Speicher auszufüllen. In diesem Kapitel untersuchen wir, wie Betriebssysteme Abstraktionen des Speichers erzeugen und verwalten.

Jeder Programmierer hätte am liebsten einen privaten, unendlich großen, unendlich schnellen Speicher, der dazu noch nicht flüchtig ist, das heißt, dessen Inhalt nicht verloren geht, wenn die Stromversorgung unterbrochen wird. Wenn wir schon dabei sind, könnte er eigentlich auch noch billig sein. Dummerweise funktioniert realer Speicher zurzeit so nicht. Vielleicht werden Sie ja eines Tages zum Erfinder des perfekten Speichers.

Bis es so weit ist – womit begnügen wir uns solange? Im Laufe der Jahre wurde das Konzept der **Speicherhierarchie** (*memory hierarchy*) entwickelt: An der Spitze stehen ein paar Megabyte eines sehr schnellen, teuren und flüchtigen Cache-Speichers. Die nächsten Stufen bilden der einige Gigabyte große mittelschnelle und mittelteure Arbeitsspeicher und ein langsamer, billiger und nicht flüchtiger Plattenspeicher, der einige Terabyte groß sein kann. Hinzu kommen noch die entfernbaren Speichermedien wie DVDs und USB-Sticks. Das Betriebssystem hat die Aufgabe, diese Hierarchie in ein nützliches Modell zu verwandeln und diese Abstraktion dann zu verwalten.

Der Teil des Betriebssystems, der (Teile der) Speicherhierarchie verwaltet, heißt **Speicherverwaltung** (*memory manager*). Seine Aufgabe besteht darin, den Speicher effizient zu verwalten: Er verfolgt, welche Speicherbereiche gerade benutzt werden, teilt Prozessen Speicher zu, wenn sie ihn benötigen, und gibt ihn anschließend wieder frei.

In diesem Kapitel untersuchen wir unterschiedliche Strategien zur Speicherverwaltung, darunter einige sehr einfache, aber auch hoch komplizierte. Da das Verwalten der untersten Ebene des Cache-Speichers normalerweise von der Hardware vorgenommen wird, liegt der Fokus dieses Kapitels auf dem Programmiermodell des Arbeitsspeichers und auf dessen effizienter Verwaltung. Die Abstraktionen und die Verwaltung von permanentem Speicher – der Platte – sind das Thema des nächsten Kapitels. Wir beginnen mit einer möglichst einfachen Strategie und arbeiten uns dann langsam zu den ausgefeilteren Methoden vor. «

1 „Jede Arbeit dauert so lange, wie Zeit für sie zur Verfügung steht.“ (Cyril N. Parkinson) (Anm. d. Übers.)

3.1 Systeme ohne Speicherabstraktion

Die einfachste Speicherabstraktion ist, überhaupt keine Abstraktion einzusetzen. Frühe Großrechner (vor 1960), frühe Minicomputer (vor 1970) und frühe PCs (vor 1980) hatten keine Speicherabstraktion. Jedes Programm hatte einfach den physischen Speicher vor sich. Wenn ein Programm einen Befehl wie

```
MOV REGISTER1, 1000
```

ausführte, hat der Computer lediglich den Inhalt der physischen Speicheradresse 1000 in *REGISTER1* abgelegt. Damit war das Modell des Speichers, das dem Programmierer präsentiert wurde, einfach physischer Speicher: eine Menge von Adressen von 0 bis zu einem gewissen Maximum, wobei jeder Adresse eine Zelle zugeordnet war, die eine bestimmte Anzahl von Bits (in der Regel acht) enthielt.

Unter diesen Bedingungen war es nicht möglich, zwei Programme gleichzeitig laufen zu lassen. Wenn das erste Programm einen neuen Wert zum Beispiel an die Speicheradresse 2000 schrieb, wurde damit möglicherweise ein Wert gelöscht, den das zweite Programm hier abgelegt hatte. Nichts würde mehr funktionieren und beide Programme würden fast augenblicklich abstürzen.

Aber selbst mit diesem Speichermodell, bei dem nur der physische Speicher benutzt wird, sind noch mehrere Optionen möglich. ▶Abbildung 3.1 zeigt drei Variationen. Das Betriebssystem liegt entweder am unteren Rand des Speichers im RAM (▶Abbildung 3.1(a)) oder am oberen Rand im ROM (▶Abbildung 3.1(b)) oder die Gerätetreiber liegen oben im ROM und der Rest des Systems liegt unten im RAM (▶Abbildung 3.1(c)). Das erste Modell wurde früher in Großrechnern und Minicomputern verwendet, wird aber heute kaum noch eingesetzt. Das zweite Modell wird in einigen Handheld-Computern und eingebetteten Systemen benutzt. Das dritte Modell wurde von frühen PCs (z. B. unter MS-DOS) verwendet. Der Teil des Systems im ROM wird **BIOS** (**Basic Input Output System**) genannt. Die Modelle (a) und (c) haben den Nachteil, dass ein Fehler im Anwendungsprogramm das Betriebssystem aushebeln kann – möglicherweise mit verheerenden Folgen (wie zum Beispiel der Zerstörung der Platteninhalte).

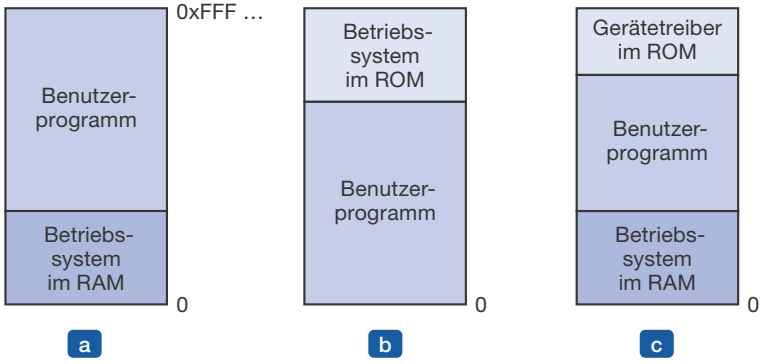

Abbildung 3.1: Drei einfache Möglichkeiten, den Speicher für einen Benutzerprozess und das Betriebssystem zu organisieren. Es gibt aber noch andere Möglichkeiten.

In einem so strukturierten System kann jeweils nur ein Prozess laufen. Sobald der Benutzer ein Kommando eingibt, lädt das Betriebssystem das entsprechende Programm von der Platte in den Speicher und führt es aus. Wenn der Prozess beendet ist, gibt das System ein Prompt-Zeichen aus. Beim nächsten Kommando lädt es das neue Programm in den Speicher und überschreibt dabei das alte.

Eine Möglichkeit, um wenigstens ein wenig Parallelität in einem System ohne Speicherabstraktion zu erhalten, ist die Programmierung von mehreren Threads. Da ohnehin alle Threads in einem Prozess das gleiche Speicherbild sehen sollten, ist dieser Punkt kein Problem. Auch wenn dieser Ansatz grundsätzlich funktioniert, ist er doch nur von begrenztem Nutzen, denn das, was eigentlich erreicht werden sollte – nämlich voneinander *unabhängige* Programme zur selben Zeit laufen zu lassen –, bietet die Abstraktion über Threads gerade nicht. Außerdem ist es fraglich, ob ein System, das so primitiv ist, dass es keine Speicherabstraktion kennt, in der Lage sein wird, Thread-Abstraktionen zur Verfügung zu stellen.

Mehrere Programme ohne Speicherabstraktion ausführen

Dennoch ist es selbst ohne Speicherabstraktion möglich, mehrere Programme gleichzeitig auszuführen. Dazu muss das Betriebssystem den gesamten Inhalt des Speichers in einer Plattendatei ablegen, dann das nächste Programm holen und ausführen. Solange jeweils nur ein Programm im Speicher ist, gibt es auch keine Konflikte. Dieses Konzept (Swapping) besprechen wir weiter hinten noch genauer.

Mit einiger zusätzlicher spezieller Hardware ist es möglich, mehrere Programme auch ohne Swapping parallel laufen zu lassen. Die ersten Modelle des IBM 360 lösten das Problem folgendermaßen: Der Speicher wurde in 2-KB-Blöcke eingeteilt und jedem Block wurde ein 4-Bit-Schutzschlüssel zugewiesen, der wiederum in einem speziellen CPU-Register gespeichert war. Eine Maschine mit einem 1-MB-Speicher brauchte nur 512 dieser 4-Bit-Register für insgesamt 256 Byte Schlüsselspeicher. Das PSW (Programmstatuswort) enthält ebenfalls einen 4-Bit-Schlüssel. Die Hardware der 360er-Serie löste jedes Mal einen Systemaufruf aus, wenn ein laufender Prozess versuchte, auf den Speicher mit einem Schutzcode zuzugreifen, der sich vom PSW-Schlüssel unterschied. Da nur das Betriebssystem die Schutzschlüssel verändern konnte, wurde so verhindert, dass die Benutzerprozesse untereinander und mit dem Betriebssystem selbst in Konflikt gerieten.

Trotzdem hatte diese Lösung einen großen Nachteil, der in ▶Abbildung 3.2 dargestellt ist. Wir haben hier zwei Programme, die beide 16 KB groß sind und jeweils einen unterschiedlichen Speicherschlüssel haben (▶Abbildung 3.2(a) und (b)). Das erste Programm startet mit einem Sprung zur Adresse 24, an der sich ein `MOV`-Befehl befindet. Das zweite Programm springt als Erstes zur Adresse 28, die einen `CMP`-Befehl enthält. In der Abbildung sind nur die Befehle dargestellt, die für unsere Betrachtung relevant sind. Wenn die zwei Programme hintereinander, beginnend bei Adresse 0 in den Speicher geladen werden, haben wir die Situation von ▶Abbildung 3.2(c). Wir nehmen hier an, dass sich das Betriebssystem im oberen Speicherbereich befindet, und somit wird es nicht abgebildet.

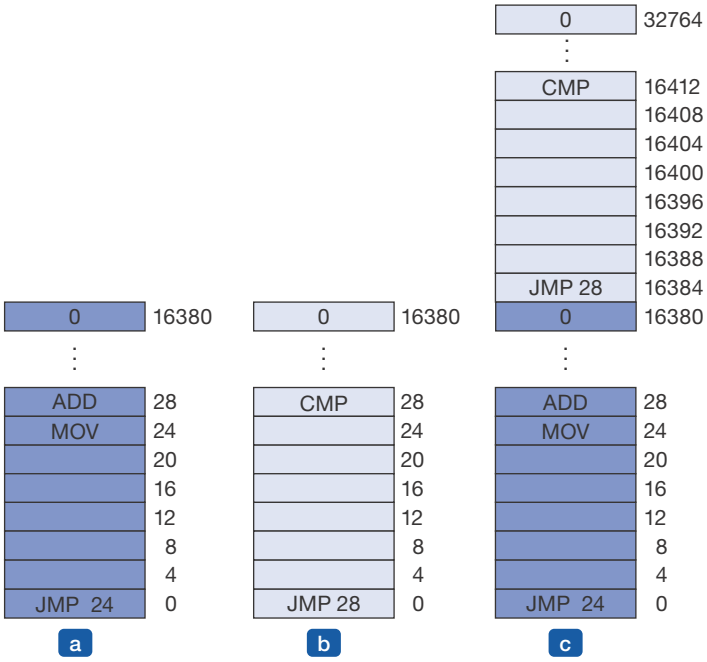

Abbildung 3.2: Darstellung des Relokationsproblems (a) Ein 16-KB-Programm (b) Ein weiteres 16-KB-Programm (c) Die zwei Programme wurden hintereinander in den Speicher geladen.

Nachdem die Programme geladen sind, können sie ausgeführt werden. Da sie unterschiedliche Speicherschlüssel haben, kann erst einmal keines das andere beschädigen. Aber das Problem ist von anderer Art. Das erste Programm führt zunächst den Befehl `JMP 24` aus, der wie erwartet zum `MOV`-Befehl springt. Dieses Programm arbeitet also normal.

Nachdem das erste Programm eine Weile gelaufen ist, entscheidet sich das Betriebssystem möglicherweise, nun das zweite Programm laufen zu lassen, das oberhalb des ersten Programms im Speicher an der Adresse 16.384 geladen wurde. Der erste Befehl, der ausgeführt ist, ist `JMP 28`, der nun zum `ADD`-Befehl des ersten Programms springt anstatt zum `CMP`-Befehl, wohin es eigentlich springen sollte. Das Programm wird vermutlich in weit weniger als 1 Sekunde abstürzen.

Das Kernproblem ist hier, dass beide Programme absolute physische Speicheradressen ansprechen. Dies ist nun genau das, was wir nicht wollten. Wir möchten, dass jedes Programm auf einen privaten Adressbereich zugreift, der lokal für das Programm ist. Wir werden in Kürze zeigen, wie dies erreicht werden kann. Der IBM 360 bot eine Art Überbrückungslösung an, indem er das zweite Programm, sobald es in den Speicher geladen wurde, mittels **statischer Relokation** modifizierte. Diese Technik funktionierte folgendermaßen: Wenn ein Programm beispielsweise an die Adresse 16.384 geladen wurde, dann wurde während des Ladevorganges die Konstante 16.384 zu jeder Programmadresse addiert. Obwohl dieser Mechanismus bei korrekter Anwendung funktioniert, stellt er doch keine allgemeine Lösung dar und verlangsamt das Laden.

Außerdem müssen zusätzliche Informationen für alle ausführbaren Programme mitgeliefert werden, um zu unterscheiden, welches Speicherwort (relozierbare) Adressen enthält und welches nicht. Relozierbare Adressen sind unabhängig von der Position im Arbeitsspeicher immer gültig und müssen etwa beim Verschieben eines Prozesses nicht neu berechnet werden. Beispielsweise müsste für die „28“ in Abbildung 3.2(b) eine neue Adresse berechnet werden, aber ein Befehl wie

```
MOV REGISTER1,28
```

der die Zahl 28 in *REGISTER1* lädt, muss nicht reloziert werden. Das Ladeprogramm benötigt ein Verfahren, um zu herauszufinden, was eine Adresse und was eine Konstante ist.

Schließlich neigt die Geschichte in der Computerwelt dazu, sich zu wiederholen, wie wir schon in Kapitel 1 bemerkt haben. Während die direkte Adressierung des physischen Speichers heute nur noch eine ferne Erinnerung auf Großrechnern, Minicomputern, Desktoprechnern und Notebooks ist, gibt es in eingebetteten Systemen und Smart Cards immer noch keine Speicherabstraktion. Geräte wie Radios, Waschmaschinen und Mikrowellen sind heutzutage alle voller Software (im ROM) und in den meisten Fällen adressiert die Software absoluten Speicher. Dies funktioniert hier, weil alle Programme im Voraus bekannt sind und die Benutzer nicht einfach ihre eigene Software auf ihrem Toaster laufen lassen dürfen.

Einige hoch entwickelte eingebettete Systeme (wie zum Beispiel Mobiltelefone) haben ausgeklügelte Betriebssysteme, einfachere Systeme dagegen nicht. In einigen Fällen ist das Betriebssystem lediglich eine Bibliothek, die mit dem Anwendungsprogramm verbunden ist und Systemaufrufe zur Ausführung von Ein-/Ausgabe und anderer gewöhnlicher Aufgaben bereitstellt. Das populäre Betriebssystem **e-cos** ist ein verbreitetes Beispiel für ein Betriebssystem als Bibliothek.

3.2 Speicherabstraktion: Adressräume

Alles in allem ist das Vorgehen, Prozesse direkt auf den physischen Speicher zugreifen zu lassen, mit mehreren großen Nachteilen verbunden: Erstens können Benutzerprogramme, die jedes Byte des Speichers adressieren dürfen, leicht das Betriebssystem – absichtlich oder zufällig – aushebeln und damit das System zu einem völligen Stillstand bringen (es sei denn, es gibt spezielle Hardware wie das IBM-Modell von Schlüssel und Schloss). Dieses Problem existiert selbst dann, wenn nur ein Benutzerprogramm (Anwendung) läuft. Zweitens ist es mit diesem Modell schwierig, mehrere Programme gleichzeitig auszuführen (die sich abwechseln, falls es nur eine CPU gibt). Auf PCs ist es üblich, mehrere Programme gleichzeitig geöffnet zu haben, zum Beispiel ein Textverarbeitungsprogramm, ein E-Mail-Programm und einen Webbrowser, wobei auf einem Programm der aktuelle Fokus liegt, aber die anderen durch einen Mausklick reaktiviert werden können. Dieser Zustand ist ohne Abstraktion vom physischen Speicher kaum zu erreichen, man musste sich also etwas einfallen lassen.

3.2.1 Das Konzept des Adressraumes

Zwei Probleme müssen gelöst werden, wenn mehrere Anwendungen gleichzeitig im Speicher erlaubt sind, damit sie sich nicht gegenseitig stören: Schutz und Relokation. Auf der IBM 360 wurde eine primitive Lösung zum Schutz benutzt: Markiere Speicherblöcke mit einem Schutzschlüssel und vergleiche den Schlüssel des ausführenden Prozesses mit dem des eben geladenen Speicherwortes. Dieser Ansatz allein löst aber noch nicht das zweite Problem, abgesehen von der langsamen und komplizierten Lösung durch die Relokation von Programmen zur Ladezeit.

Eine bessere Lösung ist die Einführung einer neuen Speicherabstraktion: der **Adressraum** (*address space*). Genau wie das Prozesskonzept eine Art von abstrakter CPU erzeugt, um Programme auszuführen, erzeugt der Adressraum eine Art von abstraktem Speicher, in dem Programme leben können. Ein Adressraum ist die Menge von Adressen, die ein Prozess zur Adressierung des Speichers benutzen kann. Jeder Prozess hat seinen eigenen Adressraum, der unabhängig von den Adressräumen der anderen Prozesse ist (ausgenommen die speziellen Umstände, wenn Prozesse ihre Adressräume teilen wollen).

Das Konzept eines Adressraumes ist sehr allgemein und taucht in vielen Zusammenhängen auf. Betrachten wir beispielsweise Telefonnummern: In den USA, Europa und vielen anderen Ländern ist eine lokale Telefonnummer gewöhnlich eine siebenstellige Zahl. Der Adressraum für Telefonnummern reicht also von 0000000 bis 9999999, wobei einige Nummern nicht benutzt werden, wie z.B. diejenigen, die mit 000 beginnen. Mit der Zunahme von Mobiltelefonen, Modems und Faxgeräten wird dieser (Namens-)Raum zu klein, es müssen also immer mehr Ziffern benutzt werden. Der Adressraum für Ein-/Ausgabeports auf dem Pentium reicht von 0 bis 16.383. IPv4-Adressen sind 32-Bit-Zahlen, ihr Adressraum reicht also von 0 bis $2^{32}-1$ (auch hier gibt es wieder einige reservierte Nummern).

Adressräume müssen nicht numerisch sein. Die Menge der *.com*-Internetdomänen ist ebenfalls ein Adressraum. Dieser Adressraum besteht aus all den Zeichenfolgen der Länge 2 bis 63, die zusammengesetzt sind aus Buchstaben, Zahlen und Bindestrichen, gefolgt von *.com*. Mit diesen Beispielen sollte das Konzept nun klar geworden sein, es ist eigentlich recht einfach.

Etwas schwieriger ist es, wie man jedem Programm seinen eigenen Adressraum zuteilt, so dass die Adresse 28 in dem einen Programm einen anderen physischen Speicherort bezeichnet als Adresse 28 in einem zweiten Programm. Im Folgenden werden wir zunächst einen einfachen Weg beschreiben, der früher üblich war, aber nicht mehr benutzt wird, da man heute viel kompliziertere (und bessere) Modelle auf modernen Chips einsetzen kann.

Basis- und Limitregister

Diese einfache Lösung benutzt eine besonders simple Version der **dynamischen Relokation**. Hierbei wird jeder Adressraum eines Prozesses auf einen unterschiedlichen Teil des physischen Speichers auf einfache Weise abgebildet. Die klassische Lösung, die vom CDC 6600 (dem ersten Supercomputer der Welt) bis zum Intel 8088 (dem Herzen des Ori-

ginal-IBM-PCs) benutzt wurde, stattet jede CPU mit zwei speziellen Hardwareregistern aus, die in der Regel **Basis-** und **Limitregister** (*base/limit register*) genannt werden. Wenn Basis- und Limitregister benutzt werden, dann werden Programme ohne Relokation in möglichst aufeinanderfolgende Bereiche des Speichers geladen, wie in Abbildung 3.2(c) gezeigt. Sobald ein Prozess läuft, wird das Basisregister mit der physischen Adresse geladen, an der das Programm im Speicher anfängt, und das Limitregister wird mit der Länge des Programms geladen. In der Situation von Abbildung 3.2(c) würden also für das erste Programm 0 als Basiswert und 16.384 als Limit in diese Hardwareregister geladen. Die Werte für das zweite Programm wären 16.384 (Basisregister) bzw. 16.384 (Limitregister). Falls ein drittes 16-KB-Programm direkt über dem zweiten geladen und ausgeführt würde, dann wären Basis- und Limitregister 32.768 bzw. 16.384.

Jedes Mal, wenn ein Prozess den Speicher referenziert, um einen Befehl zu holen oder ein Datenwort zu lesen oder zu beschreiben, addiert die CPU-Hardware automatisch den Basiswert zu der Adresse, die von dem Prozess generiert wurde, bevor die Adresse auf den Speicherbus gelegt wird. Gleichzeitig wird geprüft, ob die angebotene Adresse gleich oder größer als der durch das Limitregister bestimmte Wert ist. In diesem Fall wird ein Fehler erzeugt und der Zugriff abgebrochen. Somit führt im Fall des ersten Befehls des zweiten Programms in Abbildung 3.2(c) der Prozess den Befehl

```
JMP 28
```

aus, aber die Hardware behandelt ihn, als ob es

```
JMP 16412
```

wäre, also erfolgt wie erwartet ein Sprung auf den `CMP`-Befehl. Die Belegungen des Basis- und Limitregisters während der Ausführung des zweiten Programms von Abbildung 3.2(c) sind in ▶Abbildung 3.3 zu sehen.

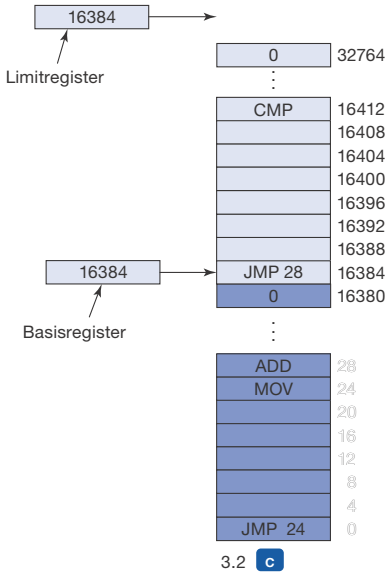

Abbildung 3.3: Basis- und Limitregister können benutzt werden, um jedem Prozess einen separaten Adressraum zu geben.

Die Benutzung von Basis- und Limitregistern bietet eine einfache Möglichkeit, jedem Prozess seinen eigenen privaten Adressraum zu geben, weil zu jeder automatisch erzeugten Speicheradresse die Inhalte des Basisregisters hinzuaddiert werden, bevor sie an den Speicher gesendet wird. In vielen Implementierungen sind Basis- und Limitregister geschützt, so dass nur das Betriebssystem sie verändern kann, wie zum Beispiel beim CDC 6600. Der Intel 8088 dagegen hatte noch nicht einmal ein Limitregister, sondern nur mehrere Basisregister, so dass zum Beispiel Programmtext und Daten unabhängig reloziert werden konnten. Es gab aber keinen Schutz dagegen, Adressen außerhalb des Adressraumes zu referenzieren.

Ein Nachteil der Relokation mit Basis- und Limitregistern ist die Notwendigkeit, bei jedem Speicherzugriff eine Addition und einen Vergleich durchzuführen. Vergleiche dauern nicht lang, aber Additionen sind durch die Übertragung der Carry-Bits relativ langsam, wenn nicht spezielle Additionsschaltkreise benutzt werden.

3.2.2 Swapping

Falls der physische Speicher des Computers groß genug ist, um alle laufenden Prozesse aufzunehmen, dann reichen die bisher besprochenen Modelle mehr oder weniger aus. Aber in der Praxis ist die Gesamtmenge an RAM, die von allen Prozessen benötigt wird, oft viel größer als der Speicher. Auf einem typischen Windows- oder Linux-System werden etwa 40–60 Prozesse gestartet, wenn der Computer hochgefahren wird. Wenn zum Beispiel eine Windows-Anwendung installiert wird, dann werden oft Kommandos ausgegeben, die bewirken, dass bei allen folgenden Systemstarts jedes Mal ein Prozess gestartet wird, der lediglich nach Updates für diese Anwendung sucht. Solch ein Prozess kann leicht 5–10 MB an Speicher belegen. Andere Hintergrundprozesse suchen nach ankommenden E-Mails, Netzwerkverbindungen und vielen weiteren Dingen. Und dies alles findet statt, bevor überhaupt das erste Benutzerprogramm gestartet wurde. Ernsthafte Anwendungsprogramme benötigen heutzutage leicht 50 bis 200 MB und mehr an Speicherplatz. Somit würde es eine riesige Menge an Speicher erfordern, wollte man alle Prozesse die ganze Zeit im Speicher halten. Solange man diesen Platz also nicht hat, müssen andere Methoden gefunden werden.

Im Laufe der Jahre wurden zwei grundlegende Ansätze entwickelt, wie man der Überlastung des Speichers begegnen kann. Die einfachste Strategie ist das sogenannte **Swapping**, bei dem jeder Prozess komplett in den Speicher geladen wird, eine gewisse Zeit laufen darf und anschließend wieder auf die Festplatte ausgelagert wird. Prozesse im Leerlauf werden meistens auf der Platte gespeichert, so dass sie keinen Speicher verbrauchen, wenn sie nicht laufen (wobei einige von ihnen in bestimmten Zeitabständen aufwachen, ihre Aufgaben erledigen und dann wieder schlafen gehen). Bei der anderen Strategie, dem **virtuellen Speicher** (*virtual memory*), können Programme auch dann laufen, wenn sich nur ein Teil von ihnen im Arbeitsspeicher befindet. Zunächst behandeln wir Swapping, in Abschnitt 3.3 besprechen wir dann virtuellen Speicher.

▶Abbildung 3.4 zeigt die Arbeitsweise eines Swapping-Systems. Anfangs liegt nur Prozess *A* im Arbeitsspeicher, später werden die Prozesse *B* und *C* erzeugt oder von

der Platte eingelagert. In ▶Abbildung 3.4(d) wird *A* dann ausgelagert. Als Nächstes wird *D* ein- und *B* ausgelagert. Schließlich kommt *A* noch einmal zurück in den Speicher. Da *A* jetzt an einer anderen Stelle im Speicher liegt, müssen die Adressen, die in *A* enthalten sind, reloziert werden. Dies wird wahrscheinlich hardwaremäßig während der Programmausführung erledigt, kann aber auch durch Software geschehen, wenn der Prozess ein- und ausgelagert wird. Basis- und Limitregister beispielsweise würden hier gut funktionieren.

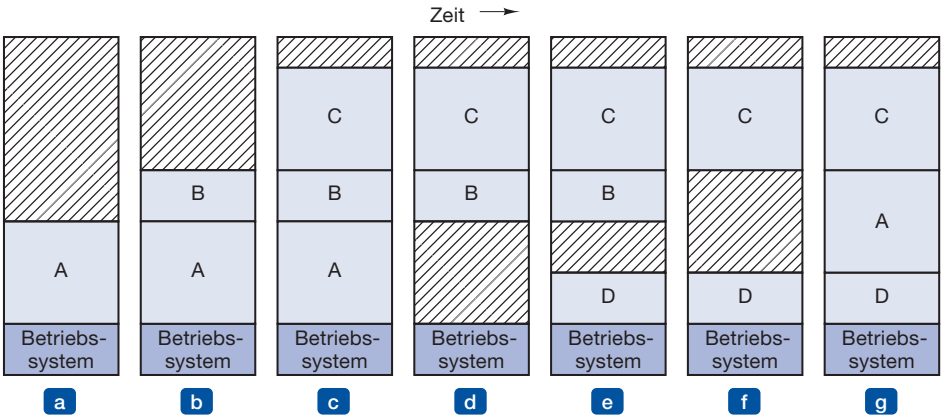

Abbildung 3.4: Mit der Ein- und Auslagerung von Prozessen ändert sich die Speicherbelegung. Die schraffierten Bereiche sind ungenutzt.

Lücken im Speicher, die durch Swapping entstehen, können zu einer großen Lücke zusammengefasst werden, indem man alle Prozesse so weit wie möglich im Speicher nach unten schiebt. Diese Technik ist als **Speicherverdichtung** (*memory compaction*) bekannt. Speicherverdichtung wird normalerweise nicht eingesetzt, weil sie eine Menge Rechenzeit verbraucht. Eine 1-GB-Maschine, die 4 Byte in 20 ns kopieren kann, bräuchte z. B. 5 Sekunden, um ihren gesamten Speicher zu verdichten.

Eine wichtige Frage ist auch, wie viel Speicher für einen Prozess reserviert werden soll, wenn er erzeugt oder eingelagert wird. Wenn jeder Prozess eine feste Größe hat, die sich niemals ändert, ist die Entscheidung einfach: Das Betriebssystem teilt ihm genau so viel zu, wie er braucht, nicht mehr und nicht weniger.

Viele Programmiersprachen erlauben es aber beispielsweise, dynamisch Speicher auf einem Heap zu reservieren. Dadurch können die Datensegmente eines Prozesses größer werden. Wenn der Prozess dann versucht zu wachsen, entsteht ein Problem. Wenn es eine Lücke im Speicher neben dem Prozess gibt, kann diese dem Prozess zugeteilt werden und er kann in die Lücke hineinwachsen. Wenn der Prozess dagegen direkt neben einem anderen Prozess liegt, muss der wachsende Prozess entweder in eine Lücke verschoben werden, die groß genug für ihn ist, oder es müssen Prozesse ausgelagert werden, um eine passende Lücke zu schaffen. Wenn ein Prozess nicht wachsen kann und der Swap-Bereich auf der Platte voll ist, muss der Prozess unterbrochen werden, bis Speicherplatz freigegeben wurde (oder er kann abgebrochen werden).

Wenn man davon ausgeht, dass die meisten Prozesse während ihrer Laufzeit wachsen, ist es wahrscheinlich keine schlechte Idee, immer etwas mehr Speicher zu reservieren, wenn ein Prozess eingelagert oder verschoben wird. Dadurch lässt sich der zusätzliche Aufwand vermeiden, wenn ein Prozess nicht mehr in seinen Speicherbereich passt. Wenn ein Prozess ausgelagert wird, sollte natürlich nur der wirklich benutzte Speicher auf die Platte geschrieben werden; den Extraspeicher mit auszulagern, wäre Verschwendung. In ▶Abbildung 3.5(a) sehen wir eine Speicherkonfiguration, in der für zwei Prozesse Platz gelassen wurde, damit sie wachsen können.

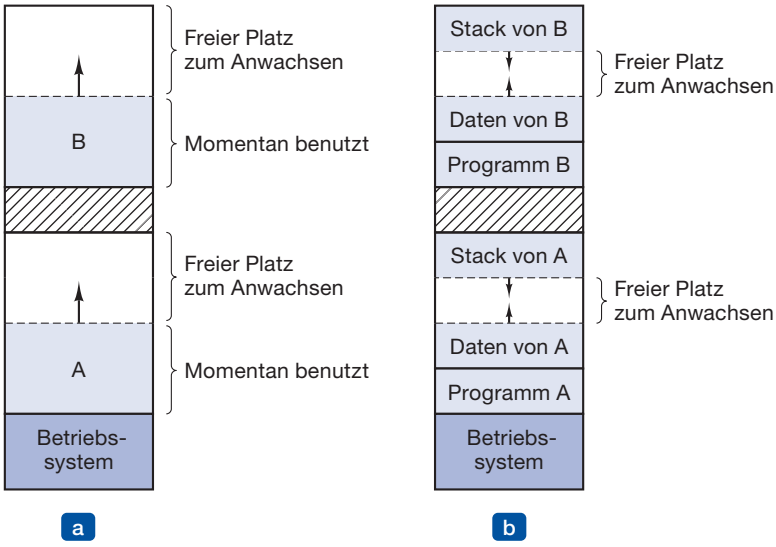

Abbildung 3.5: (a) Speicherreservierung für ein wachsendes Datensegment (b) Speicherreservierung für wachsende Stack- und Datensegmente

Wenn ein Prozess zwei wachsende Segmente haben kann – zum Beispiel ein Datensegment, das als Heap für dynamische Variablen genutzt wird, und ein Stacksegment für lokale Variablen und Rücksprungadressen – liegt es nahe, die Anordnung aus ▶Abbildung 3.5(b) zu verwenden. In dieser Abbildung hat jeder Prozess ein Stacksegment am oberen Ende seines Speicherbereiches, das nach unten wächst, und ein Datensegment genau oberhalb vom Programmcode, das nach oben wächst. Der freie Speicher in der Mitte kann für jedes der beiden Segmente benutzt werden. Wenn er nicht ausreicht, müssen die Prozesse in eine größere Lücke verschoben, ausgelagert (bis eine ausreichend große Lücke erzeugt werden kann) oder abgebrochen werden.

3.2.3 Verwaltung freien Speichers

Wenn der Speicher dynamisch zugeteilt wird, muss das Betriebssystem ihn verwalten. Grob gesagt gibt es dafür zwei Möglichkeiten: Bitmaps und Freibereichslisten (*free list*). In diesem und im nächsten Abschnitt werden wir diese beiden Methoden behandeln.

Speicherverwaltung mit Bitmaps

Bei der Speicherverwaltung mit Bitmaps wird der Speicher in Belegungseinheiten unterteilt, die nur aus ein paar Wörtern bestehen oder einige Kilobyte groß sein können. Jeder Einheit entspricht ein Bit in der Bitmap, wobei eine 0 bedeutet, dass die Einheit frei ist, und eine 1, dass sie belegt ist (oder umgekehrt). ▶Abbildung 3.6 zeigt einen Teil eines Speichers und die zugehörige Bitmap.

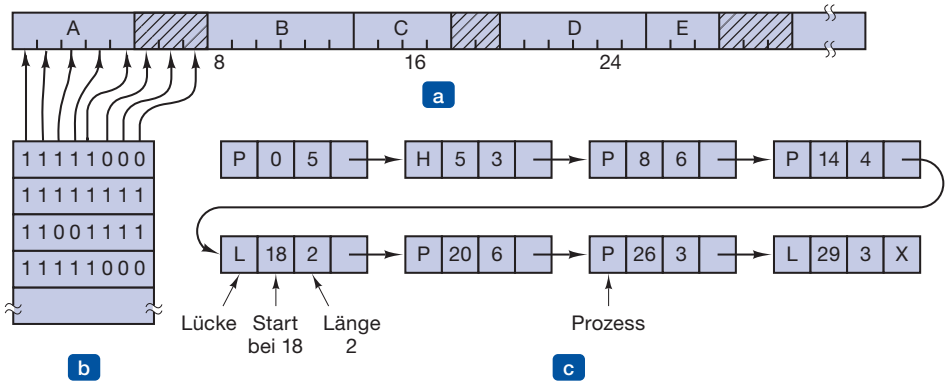

Abbildung 3.6: (a) Ein Teil eines Speichers mit fünf Prozessen und drei Lücken. Die Teilstriche markieren die Grenzen der Belegungseinheiten. Die schraffierten Bereiche sind frei (0 in der Bitmap). (b) Die zugehörige Bitmap (c) Dieselbe Information als Liste

Die Größe der Belegungseinheiten ist eine wichtige Entwurfsfrage. Je kleiner die Einheiten, desto größer wird die Bitmap. Aber selbst wenn die Einheiten nur 4 Byte groß sind, wird für je 32 Bit nur 1 Bit in der Bitmap gebraucht. Ein Speicher der Größe $32n$ Bits verbraucht n Bit für die Bitmap, also 1/33 des Speichers. Für große Einheiten wird die Bitmap kleiner, aber es wird beträchtlicher Speicher an den Rändern von Prozessen verschwendet, wenn ihre Größe kein Vielfaches der Größe einer Einheit ist.

Mit Bitmaps lassen sich sehr einfach Wörter in einem Speicher von fester Größe verwalten, weil die Größe der Bitmap nur von der Größe des Speichers und der Größe der Belegungseinheiten abhängt. Das Hauptproblem ist, dass die Speicherverwaltung jedes Mal die gesamte Bitmap nach einer Folge von k 0-Bits durchsuchen muss, wenn ein Prozess, der k Einheiten benötigt, in den Speicher geladen werden soll. Eine Bitmap nach einer zusammenhängenden Folge von 0-Bits einer gegebenen Länge zu durchsuchen, ist zeitaufwändig (u.a. weil die Folge sich über Wortgrenzen in der Bitmap erstrecken kann). Dies ist ein Argument gegen Bitmaps.

Speicherverwaltung mit verketteten Listen

Eine andere Art, den Überblick über den Speicher zu behalten, ist, eine verkettete Liste von freien und belegten Speichersegmenten zu führen, wobei ein Segment entweder einen Prozess enthält oder eine Lücke zwischen zwei Prozessen darstellt. Der Speicher in ▶Abbildung 3.6(a) wird in ▶Abbildung 3.6(c) als verkettete Liste von Segmenten dargestellt. Jeder Eintrag in der Liste bezeichnet eine Lücke (L) oder einen Prozess (P) und enthält die Startadresse, die Länge und einen Zeiger auf den nächsten Eintrag.

In diesem Beispiel ist die Liste nach Adressen sortiert, was den Vorteil hat, dass die Liste leicht angepasst werden kann, wenn ein Prozess ausgelagert oder beendet wird. Ein terminierender Prozess hat normalerweise zwei Nachbarn (außer er ist ganz außen im Speicher), die entweder Prozesse oder Lücken sein können. Damit ergeben sich die vier möglichen Kombinationen, die in ▶Abbildung 3.7 dargestellt sind. In ▶Abbildung 3.7(a) muss, wenn *X* terminiert, nur ein P durch ein L ersetzt werden. In ▶Abbildung 3.7(b) und ▶Abbildung 3.7(c) müssen jeweils zwei Einträge zu einem zusammengefasst werden und die Liste verkürzt sich um einen Eintrag. In ▶Abbildung 3.7(d) verschmelzen drei Einträge miteinander und die Liste wird um zwei Einträge kürzer.

Da der Prozesstabelleneintrag für den terminierenden Prozess *X* normalerweise einen Zeiger auf den eigenen Listeneintrag enthält, wäre eine doppelt verkettete Liste bequemer als die einfach verkettete aus Abbildung 3.6(c). Die doppelt verkettete Liste vereinfacht das Auffinden des vorherigen Eintrages und die Überprüfung, ob eine Verschmelzung möglich ist.

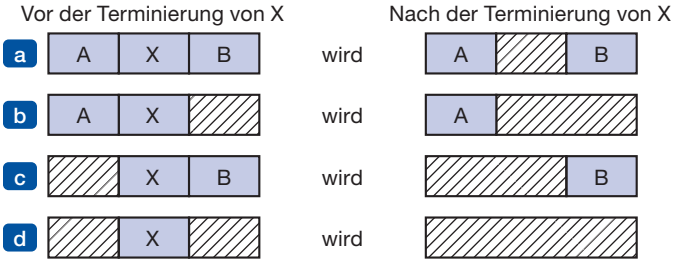

Abbildung 3.7: Die vier möglichen Kombinationen für die Nachbarn des terminierenden Prozesses *X*

Wenn die Liste für die Verwaltung von Prozessen und Lücken nach Adressen geordnet ist, kommen mehrere Algorithmen in Frage, um Speicher zu reservieren, wenn ein Prozess erzeugt oder eingelagert werden soll. Wir gehen davon aus, dass die Speicherverwaltung weiß, wie viel Speicher nötig ist. Der einfachste Algorithmus ist **First Fit**: Die Speicherverwaltung geht die Liste der Reihe nach durch, bis sie auf eine ausreichend große Lücke stößt. Die Lücke wird in zwei Teile geteilt, einen für den Prozess und einen für den unbenutzten Speicher, außer im statistisch höchst unwahrscheinlichen Fall, dass der Prozess exakt passt. First Fit ist ein schneller Algorithmus, weil er so wenig wie möglich sucht.

Eine kleine Variation von First Fit ist **Next Fit**. Next Fit funktioniert genauso wie First Fit, merkt sich aber die Position, an der er zuletzt eine passende Lücke gefunden hat, und beginnt seine nächste Suche an dieser Stelle statt am Anfang des Speichers. In Simulationen (Bays, 1977) hat Next Fit eine etwas schlechtere Leistung als First Fit gezeigt.

Ein anderer bekannter und weit verbreiteter Algorithmus ist **Best Fit**, der die gesamte Liste von Anfang bis Ende durchsucht und die kleinste passende Lücke auswählt. Anstatt eine große Lücke auseinanderzubrechen, die später vielleicht noch gebraucht wird, versucht Best Fit, eine Lücke zu finden, die möglichst gut der tatsächlich benötigten Größe entspricht, um so die Anfragen und die freien Speicherbereiche möglichst gut aufeinander abzustimmen.

Um die Arbeitsweise von First Fit und Best Fit zu verdeutlichen, sehen wir uns noch einmal Abbildung 3.6 an. Wenn ein Block der Länge 2 benötigt wird, belegt First Fit die Lücke bei 5 und Best Fit die Lücke bei 18.

Best Fit ist langsamer als First Fit, weil er bei jedem Aufruf die gesamte Liste durchsuchen muss. Etwas überraschend ist, dass Best Fit auch mehr Speicher verschwendet als First Fit oder Next Fit, weil er dazu neigt, den Speicher mit winzigen, nutzlosen Lücken zu füllen. First Fit erzeugt im Mittel größere Lücken.

Um das Problem zu umgehen, dass durch die Verwendung von Lücken, die fast genau die richtige Größe haben, eine winzige Lücke entsteht, könnte man auch über einen Algorithmus **Worst Fit** nachdenken, der immer die größte verfügbare Lücke wählt, um noch eine nützliche Lücke übrig zu lassen. Simulationen haben gezeigt, dass auch Worst Fit nicht gerade eine gute Idee ist.

Alle vier Algorithmen können durch getrennte Listen für Prozesse und Lücken beschleunigt werden. Auf diese Weise verwenden sie ihre ganze Energie darauf, die Lücken, nicht die Prozesse zu untersuchen. Der Preis für die schnellere Zuteilung ist allerdings eine komplexere und langsamere Freigabe von Speicher, weil jedes freigegebene Segment aus der Prozessliste entfernt und in die Liste der Lücken eingefügt werden muss.

Wenn für Prozesse und Lücken getrennte Listen verwaltet werden, kann man die Lücken-Liste nach Größe sortieren, um Best Fit zu beschleunigen. Wenn die Suche bei der kleinsten Lücke beginnt, ist die erste Lücke, die groß genug ist, garantiert optimal. Damit kann die Suche früher beendet werden als bei einer einzigen Liste für Prozesse und Lücken. Mit einer Liste, die nach Größe sortiert ist, sind First Fit und Best Fit gleich schnell und Next Fit ist überflüssig.

Bei getrennten Listen ist noch eine weitere kleine Verbesserung möglich. Statt einer eigenen Datenstruktur für die Lücken-Liste wie in Abbildung 3.6(c) kann die Information in den Lücken gespeichert werden. Das erste Wort in jeder Lücke könnte deren Größe enthalten, das zweite einen Zeiger auf die nächste Lücke. Die Listeneinträge in Abbildung 3.6(c), die jeweils drei Wörter und ein Bit für P oder L belegen, werden dann nicht mehr gebraucht.

Ein letzter Zuteilungsalgorithmus ist **Quick Fit**, der getrennte Listen für Lücken in einigen gebräuchlicheren Größen unterhält. Er könnte z.B. eine Tabelle mit n Einträgen haben, von denen der erste ein Zeiger auf den Kopf einer Liste von 4-KB-Lücken ist, der zweite Eintrag auf eine 8-KB-Liste zeigt, der dritte Eintrag auf eine 12-KB-Liste und so weiter. Eine 21-KB-Lücke könnte entweder an die 20-KB-Liste gehängt werden oder an eine spezielle Liste für krumme Größen.

Quick Fit findet Lücken einer bestimmten Größe extrem schnell, aber er hat denselben Nachteil wie alle Algorithmen, die nach der Größe der Lücken sortieren: Wenn ein Prozess terminiert oder ausgelagert wird, ist das Finden und Verschmelzen von benachbarten Lücken zu aufwändig. Das Verschmelzen ist aber nötig, weil der Speicher sonst in kurzer Zeit zu einer Menge kleiner Lücken fragmentiert wird, in die kein Prozess mehr passt.

3.3 Virtueller Speicher

Im letzten Abschnitt hatten wir gesehen, wie Basis- und Limitregister genutzt werden können, um die Abstraktion von Adressräumen zu erzeugen. Dieses Konzept löst jedoch ein aktuelles Problem noch nicht: die Handhabung von sogenannter „Bloatware“. Auch wenn Speicherkapazitäten schnell steigen, so wächst die Größe von Computerprogrammen noch schneller. In den 1980er Jahren hatten viele Universitäten ein Timesharing-System, auf dem jede Menge (mehr oder weniger zufriedene) Benutzer simultan arbeiteten und das auf einem 4-MB-VAX lief. Heutzutage empfiehlt Microsoft schon für ein Einbenutzer-Vista-System mindestens 512 MB Arbeitsspeicher, um einfache Anwendungen laufen zu lassen, und 1 GB, wenn man irgendetwas Anspruchsvolles machen will. Der Trend zu Multimedia erhöht die Anforderungen an den Speicher noch weiter.

Diese Entwicklung bringt es mit sich, dass Programme ausgeführt werden müssen, die nicht in den Arbeitsspeicher passen. Und sicherlich braucht man Systeme, die das simultane Ausführen von mehreren Programmen unterstützen, wobei jedes einzelne Programm zwar in den Speicher passt, aber alle gemeinsam den Speicherplatz sprengen. Swapping ist hier nicht sehr erfolgsversprechend, da eine typische SATA-Festplatte eine Spitzenübertragungsgeschwindigkeit von höchstens 100 MB pro Sekunde hat. Das heißt, es dauert mindestens 10 Sekunden, um ein 1-GB-Programm auszulagern, und weitere 10 Sekunden, um ein 1-GB-Programm einzulagern.

Dieses Problem, dass die Programme größer als der Speicher sind, existiert schon seit den Anfängen der EDV, wenn auch nur in überschaubaren Bereichen wie in wissenschaftlichen Anwendungen und im Ingenieurwesen (das Simulieren der Erschaffung des Universums oder auch nur das Simulieren eines neuen Flugzeugs benötigt eine Menge Speicherplatz). In den 1960er Jahren löste man dies, indem man die Programme in kleine Stücke, sogenannte **Overlays**, aufspaltete. Beim Programmstart wurde nur der Overlay-Manager in den Speicher geladen, der wiederum sofort Overlay 0 lud und ausführte.

Im Anschluss daran wurde dem Overlay-Manager mitgeteilt, Overlay 1 zu laden, entweder oberhalb von Overlay 0 im Speicher (falls dafür Platz war) oder durch Überschreiben von Overlay 0 (falls eben kein Platz mehr frei war). Einige Overlay-Systeme waren hochkomplex und konnten viele Overlays gleichzeitig im Speicher halten. Die Overlays wurden auf der Festplatte gespeichert und vom Overlay-Manager bei Bedarf in den Speicher ein- und ausgelagert.

Auch wenn die eigentliche Arbeit des Ein- und Auslagerns der Overlays vom Betriebssystem vorgenommen wurde, so musste der Programmierer das Programm manuell aufteilen. Große Programme in kleine, modulare Teile aufzuspalten, war zeitaufwändig, langweilig und fehleranfällig. Nur wenige Programmierer waren darin wirklich gut. So dauerte es auch nicht lange, bis jemand darüber nachdachte, wie man die ganze Arbeit dem Computer überlassen könnte.

Diese Methode (Fortheringham, 1961) wurde als **virtueller Speicher** (*virtual memory*) bekannt. Die Grundidee dahinter ist, dass der Adressraum eines Programms in Einheiten aufgebrochen wird, die sogenannten **Seiten** (*page*). Jede Seite ist ein aneinander

angrenzender Bereich von Adressen. Die Seiten werden dem physischen Speicherbereich zugeordnet, wobei nicht alle Seiten im physischen Speicher vorhanden sein müssen, damit das Programm läuft. Wenn das Programm auf einen Teil des Adressraumes zugreift, der sich aktuell im physischen Speicher befindet, dann kann die Hardware die notwendige Zuordnung sehr schnell durchführen. Wenn das Programm dagegen auf einen Teil des Adressraumes zugreifen will, der *nicht* im physischen Speicher ist, so wird das Betriebssystem alarmiert, das fehlende Stück zu beschaffen und den fehlgeschlagenen Befehl noch einmal auszuführen.

In gewisser Hinsicht ist virtueller Speicher eine Verallgemeinerung der Basis- und Limitregisteridee. Der 8088 hatte separate Basisregister (aber keine Limitregister) für Text und Daten. Statt gesonderter Relokationen für Text- und Datensegmente kann mit virtuellem Speicher der gesamte Adressraum in ziemlich kleinen Einheiten auf den physischen Speicher abgebildet werden. Wir werden im nächsten Abschnitt zeigen, wie virtueller Speicher implementiert wird.

Virtueller Speicher funktioniert auch sehr gut in Systemen mit Multiprogrammierung, wobei hier einzelne Teile von mehreren Programmen gleichzeitig im Speicher sind. Während ein Programm darauf wartet, dass Teile von ihm eingelesen werden, kann die CPU einem anderen Prozess zugeteilt werden.

3.3.1 Paging

Die meisten Systeme mit virtuellem Speicher verwenden eine Technik namens **Paging**, die wir nun beschreiben wollen. Auf jedem Rechner greifen Programme auf eine Menge von Speicheradressen zu. Führt ein Programm einen Befehl wie

```
MOV REG,1000
```

aus, so wird der Inhalt der Speicheradresse 1.000 in das Register `REG` kopiert (oder umgekehrt, je nach Rechner). Adressen können u.a. mithilfe von Indizierung, Basisregistern und Segmentregistern generiert werden.

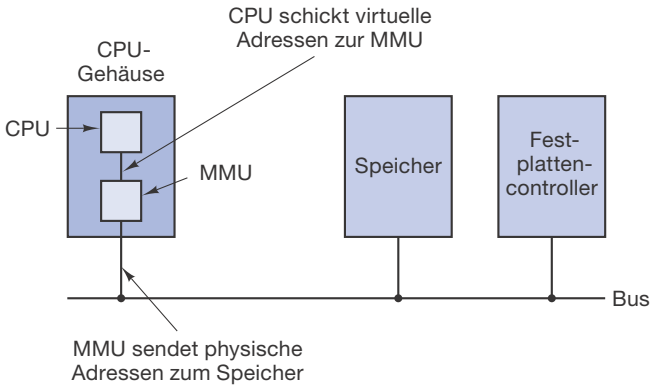

Abbildung 3.8: Position und Funktion der MMU. Die MMU ist hier Teil des CPU-Chips, wie heutzutage üblich. Aus logischer Sicht könnte es aber auch wie in alten Zeiten ein eigenständiger Chip sein.

Diese vom Programm generierten Adressen werden **virtuelle Adressen** genannt und bilden den **virtuellen Adressraum** (*virtual address space*). Bei Rechnern ohne virtuellen Speicher wird die virtuelle Adresse direkt auf den Speicherbus gelegt und das physische Speicherwort mit derselben Adresse wird gelesen oder überschrieben. Bei Systemen mit virtuellem Speicher gehen die virtuellen Adressen nicht direkt auf den Speicherbus, sondern an die **MMU** (**Memory Management Unit**, Speicherverwaltungseinheit), welche die virtuellen Adressen auf die physischen Adressen abbildet, siehe ▶Abbildung 3.8.

Ein sehr einfaches Beispiel, wie diese Zuordnung funktioniert, ist in ▶Abbildung 3.9 zu sehen: Hier haben wir einen Computer, der 16-Bit-Adressen generiert, von 0 bis 16 KB − 1 – die virtuellen Adressen. Der Computer hat jedoch nur 32 KB physischen Speicher. Obwohl also 64-KB-Programme geschrieben werden können, kann man diese nicht in ihrer Gesamtheit in den Speicher laden und ausführen. Eine vollständige Kopie des Programmkerns (bis zu 64 KB) muss aber auf der Festplatte liegen, damit Teile bei Bedarf eingelagert werden können.

Der virtuelle Adressraum ist in Einheiten fester Größe, sogenannte **Seiten** (*page*), unterteilt. Die entsprechenden Einheiten im physischen Speicher werden **Seitenrahmen** (*page frame*) genannt. Seiten und Seitenrahmen sind in der Regel gleich groß, in diesem Beispiel 4 KB. In realen Systemen kommen Seitengrößen zwischen 512 Byte und 64 KB vor. Mit 64 KB virtuellem Adressraum und 32 KB physischen Speicher erhalten wir 16 virtuelle Seiten und 8 Seitenrahmen. Zwischen RAM und Festplatte werden immer ganze Seiten übertragen.

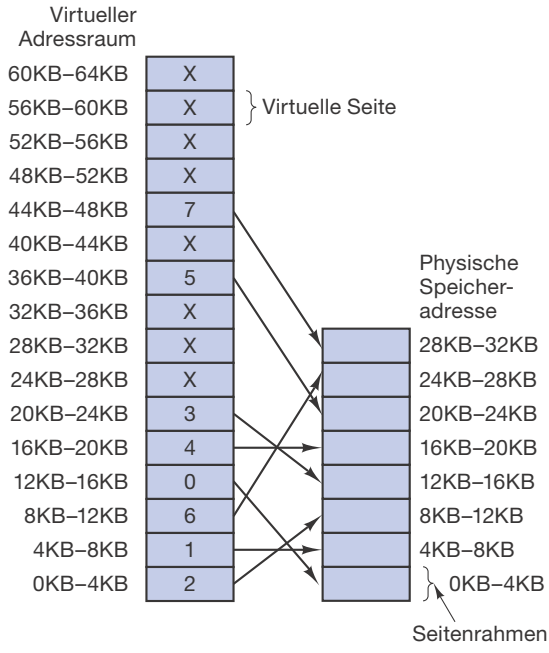

Abbildung 3.9: Die Beziehung zwischen virtuellen und physischen Adressen ist durch eine Seitentabelle vorgegeben. Jede Seite beginnt bei einem Vielfachen von 4.096 und endet 4.096 Adressen höher. 4KB–8KB bedeutet also tatsächlich 4.096–8.191 und 8KB–12KB ist 8.192–12.287.

Abbildung 3.9 ist wie folgt zu lesen: Wenn ein Bereich mit „0KB–4KB“ beschriftet ist, heißt das, die (virtuellen oder physischen) Adressen in diesem Bereich gehen von 0 bis 4.095. Die Beschriftung „4KB–8KB“ verweist dann auf Adressen 4.096 bis 8.191 usw. Jede Seite enthält exakt 4.096 Adressen, angefangen mit einem Vielfachen von 4.096 bis hin zu einem um eins verringerten Wert eines Vielfachen von 4.096.

Versucht das Programm z.B. mit dem Befehl

```
MOV REG,0
```

auf die Adresse 0 zuzugreifen, dann wird die virtuelle Adresse 0 an die MMU geschickt. Die MMU stellt fest, dass diese virtuelle Adresse zur Seite 0 gehört (0 bis 4.095), was dem Seitenrahmen 2 entspricht (8.192 bis 12.287). Die Adresse wird also in 8.192 umgewandelt und als Ausgabe auf den Bus gelegt. Der Speicher weiß nichts von der MMU, er erkennt lediglich eine Anfrage zum Lesen oder Schreiben der Adresse 8.192, die er bearbeitet. Damit hat die MMU alle virtuellen Adressen zwischen 0 und 4.095 auf die physischen Adressen 8.192 bis 12.287 abgebildet.

Analog wird der Befehl

```
MOV REG,8192
```

in

```
MOV REG,24576
```

umgewandelt, denn die virtuelle Adresse 8.192 (in virtueller Seite 2) wird auf 24.576 (im physischen Seitenrahmen 6) abgebildet. Ein drittes Beispiel: Die virtuelle Adresse 20.500 ist 20 Byte vom Anfang der virtuellen Seite 5 (virtuelle Adressen 20.480 bis 24.575) entfernt und wird auf die physische Adresse 12.288 + 20 = 12.308 abgebildet.

Die Fähigkeit allein, die 16 virtuellen Seiten durch die MMU auf jeden beliebigen der 8 Seitenrahmen abzubilden, löst noch nicht das Problem, dass der virtuelle Adressraum größer als der physische Speicher ist. Da wir nur 8 physische Seitenrahmen haben, werden auch nur 8 der virtuellen Seiten in Abbildung 3.9 auf den physischen Speicher abgebildet. Den anderen, in der Abbildung durch ein Kreuz gekennzeichnet, werden keine physischen Adressen zugewiesen. In realen Systemen zeigt ein **Present-/Absent-Bit** (anwesend/abwesend) an, welche Seiten physisch im Speicher vorhanden sind und welche nicht.

Was passiert nun, wenn ein Programm eine Adresse anspricht, die nicht im physischen Speicher liegt, zum Beispiel mit dem Befehl

```
MOV REG,32780KB
```

der auf Byte 12 innerhalb von Seite 8 (angefangen bei 32.768) zugreift? Die MMU bemerkt, dass die Seite ausgelagert ist (angezeigt durch ein Kreuz in der Abbildung), und löst eine Systemaufruf aus. Dieser Aufruf wird **Seitenfehler** (*page fault*) genannt. Das Betriebssystem wählt einen wenig benutzten Seitenrahmen aus und schreibt dessen Inhalt zurück auf die Festplatte (falls er dort nicht schon ist). Dann lädt es die angeforderte Seite in den frei gewordenen Seitenrahmen, ändert die Zuordnungstabelle und führt den unterbrochenen Befehl noch einmal aus.

Wenn das Betriebssystem sich zum Beispiel entscheidet, den Seitenrahmen 1 zu räumen, dann würde es die virtuelle Seite 8 an die physische Adresse 4.096 laden und zwei Änderungen an der Zuordnungstabelle vornehmen: Zuerst müsste der Eintrag von virtueller Seite 1 als ausgelagert markiert werden, um beim nächsten Zugriff auf eine virtuelle Adresse zwischen 4.096 und 8.191 einen Seitenfehler auszulösen. Anschließend wird das Kreuz in dem Eintrag der virtuellen Seite 8 durch eine 1 ersetzt, so dass – wenn der unterbrochene Befehl erneut ausgeführt wird – die virtuelle Adresse 32.780 auf die physische Adresse 4.108 (4.096 + 12) abgebildet wird.

Lassen Sie uns nun in eine MMU hineinschauen und sehen, wie sie arbeitet und warum wir eine Zweierpotenz als Seitengröße gewählt haben. In ▶Abbildung 3.10 sehen wir ein Beispiel, wie eine virtuelle Adresse, 8.196 (binär: 0010000000000100), mit der MMU-Zuordnung aus Abbildung 3.9 verarbeitet wird: Die ankommende virtuelle 16-Bit-Adresse wird in eine 4-Bit-Seitennummer und einen 12-Bit-Offset zerlegt. Mit 4 Bit für die Seitennummer lassen sich 16 Seiten ansprechen und mit einem 12-Bit-Offset können wir alle 4.096 Byte innerhalb einer Seite adressieren.

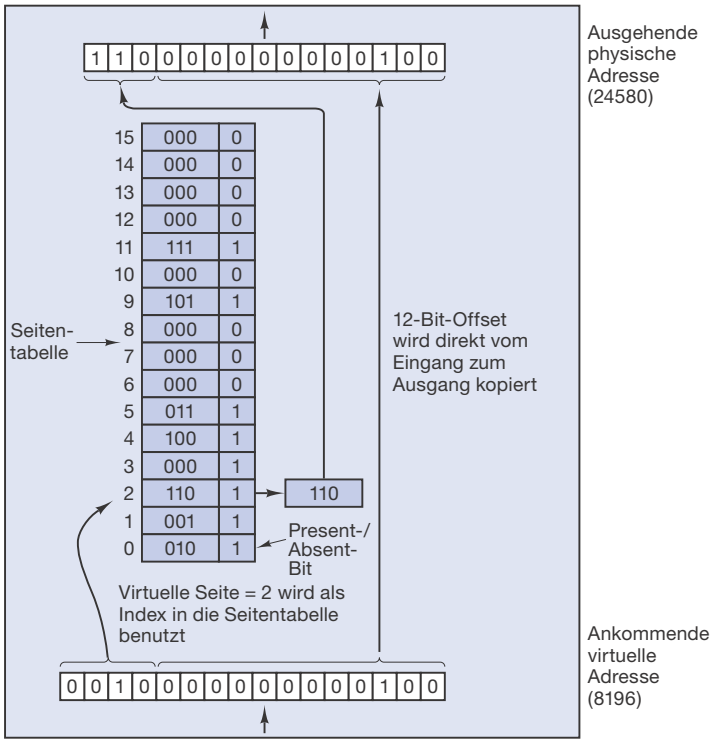

Abbildung 3.10: Interne Operation der MMU mit 16 4-KB-Seiten

Die Seitennummer wird als Index für die **Seitentabelle** (*page table*) benutzt. Diese wiederum enthält die Nummer des Seitenrahmens, welcher der virtuellen Seite entspricht. Wenn das Present-/Absent-Bit 0 ist, wird ein Seitenfehler ausgelöst. Ist das Bit 1, so wird die Nummer des Seitenrahmens aus der Tabelle in die oberen 3 Bit des Ausgaberegisters kopiert. Der 12-Bit-Offset wird unverändert von der ankommenden virtuellen Adresse

übernommen. Zusammen ergibt sich eine 15 Bit große physische Adresse, die als physische Speicheradresse mit dem Ausgaberegister auf den Speicherbus gelegt wird.

3.3.2 Seitentabellen

Für eine einfache Implementierung kann die Abbildung von virtuellen auf physische Adressen wie folgt zusammengefasst werden: Die virtuelle Adresse wird in eine virtuelle Seitennummer (höherwertige Bits) und einen Offset (niederwertige Bits) geteilt. Mit einer 16-Bit-Adresse und 4 KB großen Seiten würden z. B. die oberen vier Bits eine der 16 virtuellen Seiten auswählen und die restlichen zwölf Bits wären der Offset (0 bis 4.095) innerhalb der Seite. Es ist aber auch möglich, drei oder fünf oder eine andere Zahl von Bits für die Seitennummer zu verwenden. Verschiedene Aufteilungen ergeben verschiedene Seitengrößen.

Die virtuelle Seitennummer wird als Index benutzt, um in der Seitentabelle den Eintrag für diese Seite zu finden. Der Seitentabelleneintrag enthält die Nummer des entsprechenden Seitenrahmens, wenn vorhanden. Die Seitenrahmennummer wird an das höherwertige Ende des Offsets angehängt und ersetzt die virtuelle Seitennummer. Zusammen ergeben Seitenrahmennummer und Offset die physische Adresse, die an den Speicher geschickt wird.

Der Zweck der Seitentabelle ist es, virtuelle Seiten auf Seitenrahmen abzubilden. Mathematisch betrachtet, ist die Tabelle eine Funktion, mit der virtuellen Seitennummer als Argument und der Seitenrahmennummer als Ergebnis. Mit dem Ergebnis dieser Funktion kann der Teil einer virtuellen Adresse, der die Seitennummer enthält, durch einen Adressteil für den Seitenrahmen ersetzt werden, wodurch dann eine physische Adresse gebildet wird.

Der Aufbau eines Seitentabelleneintrages

Wir wollen nun die Details eines einzelnen Seitentabelleneintrages betrachten. Der genaue Aufbau ist stark maschinenabhängig, aber die darin enthaltene Information ist von Maschine zu Maschine etwa gleich. ▶Abbildung 3.11 zeigt ein Beispiel für einen Seitentabelleneintrag. Die Größe unterscheidet sich von Computer zu Computer, aber 32 Bit sind typisch. Das wichtigste Feld ist natürlich die *Seitenrahmennummer*. Die Ausgabe dieses Werts ist schließlich der Grund für die ganze Tabelle. Daneben haben wir das *Present-/Absent*-Bit, das anzeigt, ob die Seite momentan im Speicher liegt. Ein Zugriff auf einen Tabelleneintrag, bei dem dieses Bit auf 0 gesetzt ist, erzeugt einen Seitenfehler.

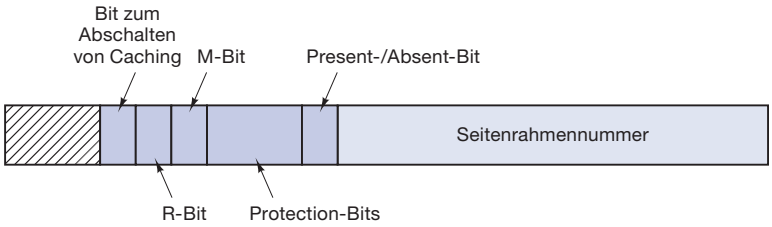

Abbildung 3.11: Ein typischer Seitentabelleneintrag

Die *Protection*-Bits oder Schutzbits regeln den Zugriff auf die Seite. In der einfachsten Form enthält dieses Feld nur ein Bit, mit 0 für Lese- und Schreibzugriff und 1 für Lesezugriff. Eine ausgefeiltere Methode benutzt drei Bits, jeweils eines für das Leserecht, eines für das Schreibrecht und eines für das Recht, die Seite auszuführen.

Das *Modified*-Bit (*M*-Bit) und das *Referenced*-Bit (*R*-Bit) protokollieren die Zugriffe auf die Seite. Wenn ein Programm in eine Seite schreibt, setzt die Hardware automatisch das *M*-Bit. Das Betriebssystem benutzt dieses Bit, wenn es eine Seite auslagert. Wenn die Seite verändert wurde, muss der Seitenrahmen zurück auf die Platte geschrieben werden, wenn nicht, kann er einfach überschrieben werden, weil die Kopie der Seite auf der Platte noch aktuell ist. Dieses Bit wird manchmal auch **Dirty-Bit** genannt, weil es angibt, ob eine Seite „dreckig“ oder „sauber“ ist, d. h., ob auf sie geschrieben wurde oder nicht.

Das *R*-Bit wird bei jedem Zugriff auf eine Seite gesetzt, egal ob Lese- oder Schreibzugriff. Es hilft dem Betriebssystem bei der Entscheidung, welche Seite es bei einem Seitenfehler auslagern soll. Seiten, die nicht benutzt werden, sind geeignetere Kandidaten als solche, auf die ständig zugegriffen wird. Dieses Bit spielt eine wichtige Rolle in mehreren Seitenersetzungsalgorithmen, die wir später in diesem Kapitel noch behandeln werden.

Das letzte Bit erlaubt es schließlich, das Caching für eine Seite zu regeln. Diese Eigenschaft ist für Seiten wichtig, die auf Geräteregister statt auf Speicher abgebildet werden. Nehmen wir an, das Betriebssystem hat einen Ein-/Ausgabebefehl an ein Gerät gesendet und wartet nun in einer Schleife auf die Antwort. Dann ist es wichtig, dass die Hardware in jedem Schleifendurchlauf die Register des Gerätes ausliest und nicht immer wieder die gleiche Kopie im Cache benutzt. Mit diesem Bit kann das Caching abgeschaltet werden. Rechner, die einen getrennten E/A-Adressraum haben und keine Memory-Mapped-Ein-/Ausgabe benutzen, brauchen dieses Bit nicht.

Die Plattenadressen, an denen ausgelagerte Seiten liegen, werden aus einem einfachen Grund nicht in der Seitentabelle gespeichert: Die Tabelle enthält nur die Informationen, die die Hardware braucht, um eine virtuelle Adresse in eine physische umzurechnen. Die Informationen, die das Betriebssystem zur Behandlung von Seitenfehlern benötigt, werden in eigenen Tabellen im Betriebssystem gespeichert. Die Hardware braucht diese Informationen nicht.

Bevor wir in die Implementierungsdetails eintauchen, sollte noch einmal betont werden, dass die wesentliche Funktion des virtuellen Speichers die Erzeugung einer neuen Abstraktion ist – dem Adressraum. Dieser stellt eine Abstraktion des physischen Speichers dar, genauso wie ein Prozess die Abstraktion des physischen Prozessors (CPU) ist. Virtueller Speicher kann implementiert werden, indem der virtuelle Adressraum in Seiten aufgebrochen wird, die dann entweder auf einen Seitenrahmen des physischen Speichers abgebildet werden oder (zeitweise) ohne Zuordnung bleiben. In diesem Kapitel geht es also im Grunde um eine Abstraktion, die vom Betriebssystem erzeugt wurde, und darum, wie diese Abstraktion verwaltet wird.

3.3.3 Beschleunigung des Paging

Wir haben gerade die Grundlagen des virtuellen Speichers und des Paging kennengelernt. Jetzt ist es Zeit, mehr in die Details über mögliche Implementierungen einzusteigen. In jedem Paging-System sehen wir uns mit zwei großen Problemen konfrontiert:

1. Die Umrechnung von der virtuellen Adresse in die physische Adresse muss sehr schnell erfolgen.
2. Wenn der virtuelle Adressraum groß ist, dann wird die Seitentabelle groß.

Der erste Punkt folgt aus der Notwendigkeit, für jeden Speicherzugriff eine virtuelle Adresse in eine physische umzurechnen. Alle Befehle müssen letztendlich vom Speicher kommen und viele Befehle greifen auch auf Operanden im Speicher zu. Also sind für jeden Befehl ein, zwei oder oft noch mehr Zugriffe auf die Seitentabelle nötig. Wenn die Ausführung eines Befehls zum Beispiel 1 ns dauert, dann darf der Zugriff auf die Tabelle nicht länger als 0,2 ns dauern. Ansonsten würden die Tabellenzugriffe zum Engpass.

Der zweite Punkt ergibt sich aus der Tatsache, dass die virtuellen Adressen moderner Computer mindestens 32 Bit lang sind, wobei 64 Bit immer häufiger werden. Bei einer Seitengröße von 4 KB hat ein 32-Bit-Adressraum eine Million Seiten und ein 64-Bit-Adressraum hat mehr, als man sich vorstellen will. Wenn eine Million Seiten im virtuellen Adressraum sind, dann muss die Seitentabelle eine Million Einträge haben. Nicht zu vergessen, dass jeder Prozess seinen eigenen Adressraum hat, also auch eine eigene Seitentabelle braucht.

Die Notwendigkeit von großen Seitentabellen mit schnellem Zugriff ist eine wichtige Randbedingung beim Entwurf von Computern. Das einfachste Design, zumindest vom Konzept her, ist eine einzige Seitentabelle, die aus einer Reihe von schnellen Hardwareregistern besteht. Die Tabelle enthält einen Eintrag für jede virtuelle Seite, wie in ▶Abbildung 3.10 gezeigt. Wenn ein Prozess gestartet wird, lädt das Betriebssystem seine Seitentabelle aus dem Speicher in die Register. Während der Prozess läuft, werden keine Speicherzugriffe für die Seitentabelle gebraucht. Der Vorteil dieser Methode ist, dass sie einfach ist und dass keine Speicherzugriffe für die Umrechnung nötig sind. Ein Nachteil ist, dass sie für große Seitentabellen untragbar teuer ist, ein weiterer, dass das Laden der ganzen Seitentabelle bei jedem Kontextwechsel die Performanz beeinträchtigt.

Das andere Extrem ist, die Seitentabelle komplett im Arbeitsspeicher zu halten. Die Hardware braucht dann nur ein einziges Register, das auf den Anfang der Seitentabelle zeigt. Die Zuordnung vom virtuellen zum physischen Speicher kann dann bei einem Kontextwechsel einfach geändert werden, indem man dieses Register überschreibt. Der offensichtliche Nachteil sind die Speicherreferenzen auf die Seitentabelle, die für die Ausführung jedes einzelnen Maschinenbefehls nötig sind, was diese Methode sehr langsam macht.

TLB – der Translation Lookaside Buffer

Nun wollen wir einen Blick auf oft implementierte Modelle zur Beschleunigung des Paging und zur Behandlung großer virtueller Adressräume werfen. Wir beginnen hier mit der Beschleunigung. Der Ausgangspunkt der meisten Optimierungstechniken ist die Tatsache, dass sich die Seitentabelle im Speicher befindet. Darunter könnte die Leistung möglicherweise enorm leiden. Nehmen wir als Beispiel einen 1-Byte-Befehl, der ein Register in ein anderes kopiert. Ohne Paging ist für diesen Befehl nur ein Speicherzugriff nötig, um den Befehl aus dem Speicher zu holen. Mit Paging wird noch mindestens ein weiterer Speicherzugriff auf die Seitentabelle benötigt. Die Ausführungsgeschwindigkeit ist dadurch beschränkt, wie schnell die CPU Daten und Befehle aus dem Speicher holen kann. Wenn also zwei Seitentabellenzugriffe pro Speicherzugriff nötig sind, bricht die Leistung um die Hälfte ein. Unter diesen Umständen würde niemand Paging benutzen.

Den Computerentwicklern ist dieses Problem seit Jahren bekannt. Ihre Lösung basiert auf der Beobachtung, dass Programme dazu neigen, sehr viele Zugriffe auf sehr wenige Seiten auszuführen (und nicht umgekehrt). Ein kleiner Anteil der Seiten wird also ständig gelesen, der Rest fast nie.

Computer werden also mit einem kleinen Hardwaregerät ausgestattet, das virtuelle Adressen ohne Umweg über die Seitentabelle auf physische Adressen abbildet. Dieses Gerät heißt **TLB** (**Translation Lookaside Buffer**) oder auch **Assoziativspeicher** und wird in ▶Abbildung 3.12 dargestellt. Es ist normalerweise ein Teil der MMU und besteht aus einer kleinen Zahl von Einträgen. In diesem Beispiel sind es acht, in Wirklichkeit selten mehr als 64. Jeder Eintrag enthält Informationen über eine Seite, darunter die virtuelle Seitennummer, ein Bit, das gesetzt wird, wenn die Seite modifiziert wird, der Schutzcode (Erlaubnis zum Lesen/Schreiben/Ausführen) und der physische Seitenrahmen, in dem die Seite liegt. Außer der virtuellen Seitennummer, die in der Seitentabelle nicht benötigt wird, sind diese Felder eins zu eins aus der Seitentabelle übernommen. Ein weiteres Bit gibt an, ob der Eintrag gültig ist (d. h. momentan benutzt wird).

Gültig	Virtuelle Seite	Verändert	Schutz	Seitenrahmen
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Abbildung 3.12: Ein TLB zur Beschleunigung des Paging

Ein Beispiel für ein Programm, das den TLB aus ▶Abbildung 3.12 erzeugen könnte, ist ein Prozess in einer Schleife, die in den virtuellen Seiten 19, 20 und 21 liegt, so dass diese Seiten Schutzcodes haben, die das Lesen und Ausführen erlauben. Die Daten, die das Programm gerade benutzt (z.B. ein Feld, das es gerade bearbeitet), finden sich auf den Seiten 129 und 130. Seite 140 enthält die Feldindizes, die bei der Bearbeitung gebraucht werden. Die Seiten 860 und 861 enthalten den Stack.

Sehen wir uns jetzt an, wie der TLB funktioniert. Wenn eine virtuelle Adresse zur Übersetzung an die MMU geschickt wird, überprüft die Hardware zunächst, ob der entsprechende Eintrag im TLB liegt, indem sie die virtuelle Seitennummer mit allen Einträgen gleichzeitig (d.h. parallel) vergleicht. Wenn ein passender Eintrag gefunden wird und der Zugriff nicht gegen den Schutzcode verstößt, wird die Seitenrahmennummer aus dem TLB verwendet, ohne auf die Seitentabelle zuzugreifen. Wenn der TLB den richtigen Eintrag enthält, der Befehl aber versucht, in eine schreibgeschützte Seite zu schreiben, löst das eine Schutzverletzung aus.

Interessanter ist, was passiert, wenn die virtuelle Seitennummer nicht im TLB steht. Die MMU stellt fest, dass der Eintrag fehlt, holt den Eintrag ganz normal aus der Seitentabelle und schreibt ihn in den TLB, wobei einer der älteren Einträge überschrieben wird. Beim nächsten Zugriff auf die Seite steht der Eintrag dann im TLB und muss nicht mehr aus der Seitentabelle geholt werden. Wenn ein Eintrag aus dem TLB verbannt wird, wird das *M*-Bit in den Seitentabelleneintrag der Seite zurückkopiert, die anderen Felder außer dem *R*-Bit sind unverändert. Wenn der TLB aus der Seitentabelle geladen wird, werden alle Felder aus dem Speicher kopiert.

Verwaltung des TLB durch Software

Bis jetzt haben wir angenommen, dass jede Maschine mit virtuellem Speicher und Paging Seitentabellen hat, die von der Hardware verwaltet werden, und zusätzlich einen TLB. In diesem Design erledigt die MMU auch die Verwaltung des TLB und der TLB-Fehler. Systemaufrufe kommen nur bei Seitenfehlern vor.

Früher war diese Annahme richtig. Doch viele moderne RISC-Prozessoren, darunter SPARC, MIPS, Alpha und HP PA, erledigen fast ihre gesamte Seitenverwaltung durch Software. In diesen Maschinen werden die TLB-Einträge explizit durch das Betriebssystem geladen. Wenn ein gesuchter Seiteneintrag nicht im TLB steht, erzeugt die MMU – statt den Eintrag aus der Seitentabelle zu holen – einen TLB-Fehler und schiebt das Problem dem Betriebssystem zu. Das System muss dann die Seite finden, einen Eintrag aus dem TLB entfernen, den neuen Eintrag einfügen und den Befehl, der den Fehler ausgelöst hat, neu starten. Und natürlich muss es das alles mit nur einer Handvoll Befehle bewerkstelligen, weil TLB-Fehler viel häufiger sind als Seitenfehler.

Erstaunlicherweise ist die Softwareverwaltung des TLB einigermaßen effizient, solange der TLB groß genug ist, um die Fehlerrate zu reduzieren (z.B. 64 Einträge). Der Hauptvorteil ist hier, dass die MMU sehr viel einfacher wird, was auf dem CPU-Chip Platz für mehr Cache und andere leistungssteigernde Vorrichtungen schafft. Die Verwaltung von TLBs durch Software wird von Uhlig et al. (1994) diskutiert.

Verschiedene Strategien wurden entwickelt, um die Leistung von Maschinen zu verbessern, die den TLB durch Software verwalten. Ein Ansatz versucht, sowohl den Aufwand für jeden TLB-Fehler als auch deren Häufigkeit zu reduzieren (Bala et al., 1994). Um die Anzahl der TLB-Fehler zu reduzieren, kann das Betriebssystem mitunter seine Intuition benutzen, um herauszufinden, welche Seiten wohl als Nächstes gebraucht werden, und dann die entsprechenden Einträge im Voraus in den TLB laden. Wenn z.B. ein Client-Prozess eine Nachricht an einen Server-Prozess auf derselben Maschine sendet, ist es wahrscheinlich, dass der Server-Prozess bald gebraucht wird. Darum kann das Betriebssystem, während es den `send`-Systemaufruf ausführt, die Code-, Daten- und Stackseiten des Servers finden und in den TLB eintragen, noch bevor diese einen TLB-Fehler auslösen können.

Die übliche Art, einen TLB-Fehler durch Hardware oder Software zu behandeln, ist, die Indizes aus der virtuellen Adresse zu benutzen, um den entsprechenden Seiteneintrag zu finden. Wenn diese Suche durch Software ausgeführt wird, besteht das Problem, dass die Seiten, die die Seitentabelle enthalten, vielleicht nicht im TLB liegen, was zusätzliche TLB-Fehler auslöst. Diese Fehler können durch einen großen Software-Cache (z.B. 4 KB) für TLB-Einträge reduziert werden, der an einer festen Adresse liegt und dessen Seiteneinträge immer im TLB bleiben. Das Betriebssystem kann dann viele TLB-Fehler vermeiden, indem es zuerst den Cache überprüft.

Wenn die Verwaltung des TLB durch die Software durchgeführt wird, ist es wichtig, zwischen zwei Arten von Fehlern zu unterscheiden. Ein sogenannter **weicher Seitenfehltalarm** (*soft miss*) tritt auf, wenn sich die angesprochene Seite nicht im TLB, sondern im Speicher befindet. In diesem Fall muss lediglich der TLB aktualisiert werden, eine Plattenein-/-ausgabe ist nicht notwendig. Typischerweise benötigt die Behebung eines weichen Fehlers 10–20 Befehle und kann in ein paar Nanosekunden beendet werden. Im Gegensatz dazu tritt ein **harter Seitenfehltalarm** (*hard miss*) auf, wenn die Seite selbst nicht im Speicher ist (und damit natürlich auch nicht im TLB). Ein Plattenzugriff wird erforderlich, um die Seiten einzulagern, was einige Millisekunden in Anspruch nimmt. Ein harter Fehler ist leicht Millionen Mal langsamer als ein weicher Fehler.

3.3.4 Seitentabellen für große Speicherbereiche

TLBs können eingesetzt werden, um die Übersetzung der virtuellen in die physischen Adressen zu beschleunigen, wenn das traditionelle Schema mit Seitentabellen im Speicher benutzt wird. Aber das ist nicht das einzige Problem, das wir angehen müssen. Es müssen auch Möglichkeiten gefunden werden, mit sehr großen virtuellen Adressräumen umzugehen. Im Folgenden werden wir zwei dieser Möglichkeiten besprechen.

Mehrstufige Seitentabellen

Als ersten Ansatz betrachten wir den Einsatz von **mehrstufigen Seitentabellen** (*multilevel page table*). ▶Abbildung 3.13 zeigt ein einfaches Beispiel. In ▶Abbildung 3.13(a) wird ein 32-Bit-Adressraum in ein 10-Bit-*PT1*-Feld, ein 10-Bit-*PT2*-Feld und einen 12-Bit-*Offset* unterteilt. Dadurch ergeben sich 2^{20} 4-KB-Seiten.

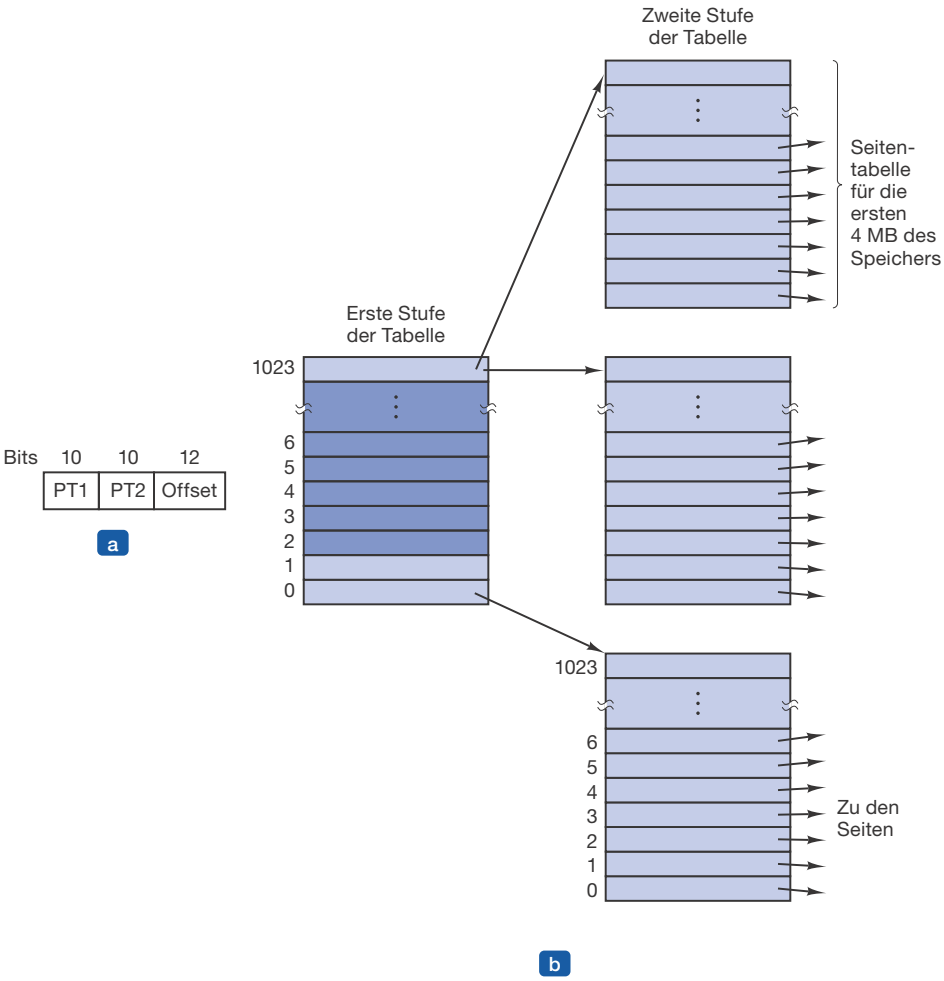

Abbildung 3.13: (a) Eine 32-Bit-Adresse mit zwei 10-Bit-Feldern für Seitennummern (b) Zweistufige Seitentabellen

Der springende Punkt bei den mehrstufigen Seitentabellen ist, dass nicht mehr alle Seitentabellen gleichzeitig im Speicher gehalten werden müssen. Besonders die nicht benötigten Tabellen sollten nicht nutzlos im Speicher herumliegen. Nehmen wir zum Beispiel an, ein Prozess belegt 12 MB Speicher, die unteren 4 MB für den Programmcode, die mittleren 4 für Daten und die oberen 4 für den Stack. Zwischen der Obergrenze der Daten und der Untergrenze des Stacks ist eine riesige Lücke, die nicht benutzt wird.

▶Abbildung 3.13(b) zeigt, wie die zweistufige Seitentabelle für dieses Beispiel funktioniert. Links sehen wir die Seitentabelle der ersten Stufe mit 1.024 Einträgen, die dem 10-Bit-Feld *PT1* entspricht. Wenn die MMU eine virtuelle Adresse umrechnet, liest sie zunächst das *PT1*-Feld aus und benutzt es als Index für diese erste Seitentabelle. Jeder der 1.024 Einträge entspricht 4 MB, da der gesamte 4 Gigabyte große Adressraum (d.h. 32-Bit-Adressraum) in 4.096-Byte-Stücke aufgespalten ist.

Der Eintrag aus der Seitentabelle der ersten Stufe enthält die Adresse oder die Seitenrahmennummer einer Seitentabelle der zweiten Stufe. Eintrag 0 der ersten Seitentabelle zeigt auf die Seitentabelle für den Programmcode, Eintrag 1 zeigt auf die Tabelle für die Daten und Eintrag 1.024 zeigt auf die für den Stack. Die anderen (dunkel gezeichneten) Einträge sind ungenutzt. Als Nächstes wird das *PT2*-Feld als Index für die ausgewählte Seitentabelle der zweiten Stufe benutzt, um die Seitenrahmennummer der gesuchten Seite selbst zu finden.

Nehmen wir als Beispiel die virtuelle 32-Bit-Adresse 0x00403004 (entspricht dezimal 4.206.596), also 12.292 Byte innerhalb der Programmdaten. Diese virtuelle Adresse entspricht *PT1* = 1, *PT2* = 3 und *Offset* = 4. Die MMU benutzt zunächst *PT1* als Index für die Seitentabelle der ersten Stufe und erhält Eintrag 1, der den Adressen 4 MB − 1 bis 8 MB − 1 entspricht. Anschließend benutzt sie *PT2* als Index für die Seitentabelle der zweiten Stufe und findet den Eintrag 3, der den Adressen 12.288 bis 16.383 innerhalb des 4-MB-Bereiches der Tabelle entspricht, das heißt den absoluten Adressen 4.206.592 bis 4.210.687. Dieser Eintrag enthält die Seitenrahmennummer der Seite, die die virtuelle Adresse 0x00403004 enthält. Wenn diese Seite nicht im Speicher ist, ist das *Present-/Absent*-Bit 0 und die MMU löst einen Seitenfehler aus. Ansonsten kombiniert die MMU die Seitenrahmennummer mit dem Offset (4) zu einer physischen Adresse, die dann auf den Speicherbus gelegt wird.

Das Bemerkenswerte an ▶Abbildung 3.13: Obwohl der Adressraum über eine Million Seiten enthält, werden nur vier Seitentabellen wirklich gebraucht: die Seitentabelle der ersten Stufe und die Tabellen der zweiten Stufe für 0 bis 4 MB (für den Programmtext), für 4 bis 8 MB (für die Daten) und die obersten 4 MB (für den Stack). In 1.021 Einträgen der Tabelle der ersten Stufe sind die *Present-/Absent*-Bits auf 0 gesetzt, was bei einem Zugriff einen Seitenfehler auslösen würde. In diesem Fall merkt das Betriebssystem, dass der Prozess versucht, unerlaubt auf fremden Speicher zuzugreifen, und ergreift geeignete Maßnahmen. Zum Beispiel könnte es dem Prozess ein Signal senden oder ihn abbrechen. In diesem Beispiel waren *PT1* und *PT2* gleich groß und wir haben runde Zahlen für die Größe gewählt. In der Praxis sind natürlich auch andere Werte möglich.

Die zweistufige Seitentabelle aus ▶Abbildung 3.13 könnte auch auf drei, vier oder noch mehr Stufen erweitert werden. Zusätzliche Ebenen erhöhen die Flexibilität, aber ob eine Tabelle mit mehr als drei Stufen den zusätzlichen Aufwand wert ist, ist zweifelhaft.

Invertierte Seitentabellen

Für einen virtuellen 32-Bit-Adressraum funktioniert die mehrstufige Seitentabelle einigermaßen gut, doch mit der Verbreitung von 64-Bit-Computern ändert sich die Situation dramatisch. Mit einem Adressraum von 2^{64} Byte und 4 KB großen Seiten hätte die Seitentabelle 2^{52} Einträge. Bei 8 Byte pro Eintrag wäre die Tabelle dann 30 Millionen Gigabyte groß (30 PB). 30 Millionen Gigabyte allein für die Seitentabelle zu benutzen, ist keine gute Idee – heute nicht und wahrscheinlich auch noch nicht im nächsten Jahr. Für virtuelle 64-Bit-Adressräume mit Paging ist also eine andere Lösung nötig.

Eine mögliche Lösung ist die **invertierte Seitentabelle** (*inverted page table*). Bei diesem Ansatz wird in der Seitentabelle ein Eintrag für jeden physischen Seitenrahmen gespeichert anstatt eines Eintrages für jede Seite im virtuellen Adressraum. Eine invertierte Seitentabelle für einen Computer mit einem virtuellen 64-Bit-Adressraum, 4 KB großen Seiten und 4 GB an RAM hätte zum Beispiel nur 262.144 Einträge. Jeder Eintrag speichert zu einem Seitenrahmen das zugehörigen Paar (Prozess, Seitennummer).

Invertierte Seitentabellen sparen zwar enorme Mengen an Speicherplatz, zumindest dann, wenn der virtuelle Adressraum wesentlich größer als der physische Speicher ist, sie haben aber einen gravierenden Nachteil: Es wird wesentlich aufwändiger, eine virtuelle Adresse auf eine physische abzubilden. Wenn Prozess *n* auf die virtuelle Seite *p* zugreifen will, kann die Hardware den physischen Seitenrahmen nicht einfach finden, indem sie die virtuelle Seitennummer als Index für die Seitentabelle benutzt, sondern sie muss die gesamte Seitentabelle nach dem Eintrag (*n*, *p*) durchsuchen. Außerdem ist diese Suche für jeden einzelnen Speicherzugriff nötig, nicht nur bei Seitenfehlern. Bei jedem Speicherzugriff eine Tabelle mit 264 KB Einträgen zu durchsuchen, ist kein guter Weg, Geschwindigkeitsrekorde zu brechen.

Die Lösung für dieses Dilemma bietet ein TLB. Wenn alle häufig benutzten Seiten in den TLB passen, ist die Adressumrechnung genauso schnell wie mit herkömmlichen Seitentabellen. Bei einem TLB-Fehler muss die invertierte Seitentabelle allerdings von der Software durchsucht werden. Eine praktikable Methode für diese Suche ist die Nutzung einer Hashtabelle mit den virtuellen Adressen als Hashwerten. Alle virtuellen Seiten im Speicher, die denselben Hashwert haben, sind wie in ▶Abbildung 3.14 miteinander verkettet. Wenn die Hashtabelle so viele Einträge wie das System physische Seiten hat, dann hat jede Kette im Durchschnitt nur einen Eintrag, was die Suche sehr beschleunigt. Wenn die Seitenrahmennummer gefunden ist, wird das neue Paar aus virtueller Seite und physischem Seitenrahmen in den TLB eingetragen.

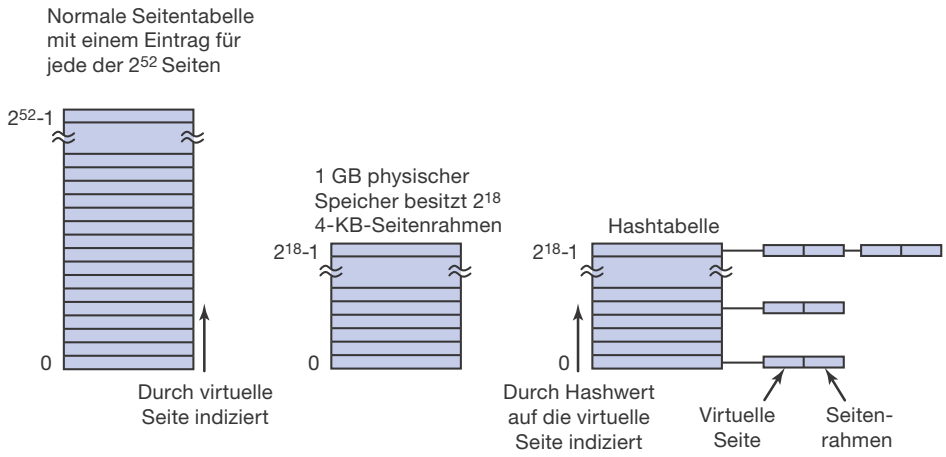

Abbildung 3.14: Vergleich zwischen herkömmlicher und invertierter Seitentabelle

Invertierte Seitentabellen sind auf den 64-Bit-Rechnern sehr verbreitet, weil selbst bei sehr groß gewählten Seitengrößen die Anzahl der Seitentabelleneinträge enorm ist. Zum Beispiel sind bei 4-MB-Seiten und virtuellen 64-Bit-Adressen 2^{42} Seiteneinträge nötig. Andere Ansätze für das Problem großer virtueller Adressräume finden sich in (Talluri et al., 1995).

3.4 Seitenersetzungsalgorithmen

Jedes Mal, wenn ein Seitenfehler auftritt, muss das Betriebssystem eine Seite auswählen, die verdrängt wird (d.h. aus dem Speicher entfernt wird), um für die neue Seite Platz zu machen. Wenn die Seite, die ausgelagert werden soll, seit ihrer Einlagerung modifiziert wurde, muss sie auf die Festplatte zurückgeschrieben werden, damit die Kopie auf der Platte aktuell bleibt. Wenn sie aber z.B. Programmcode enthält und nicht verändert wurde, ist die Kopie auf der Platte noch aktuell und muss nicht neu geschrieben werden. Die alte Seite kann dann einfach mit der neuen überschrieben werden.

Man könnte bei jedem Seitenfehler einfach eine zufällige Seite auslagern, aber die Systemleistung ist wesentlich besser, wenn man Seiten auslagert, die nur selten benutzt werden. Wenn eine viel benutzte Seite aus dem Speicher entfernt wird, muss sie wahrscheinlich sehr bald wieder eingelagert werden, was zusätzlichen Aufwand bedeutet. Das Gebiet der Seitenersetzungsalgorithmen wurde intensiv erforscht, sowohl theoretisch als auch experimentell. In diesem Abschnitt beschreiben wir ein paar der wichtigsten Algorithmen.

Das Problem der „Seitenersetzung“ tritt auch in anderen Gebieten der Computertechnik auf. Beispielsweise haben die meisten Computer einen oder mehrere Caches für 32 oder 64 Byte große Speicherblöcke, die in letzter Zeit benutzt wurden. Wenn so ein Cache voll ist, muss einer der Blöcke ausgewählt und entfernt werden. Dieses Problem ist genau dasselbe wie bei der Seitenersetzung, nur auf einer wesentlich kürzeren Zeitskala (einige Nanosekunden statt Millisekunden wie bei der Seitenersetzung). Der Grund für die kürzere Zeitskala ist die höhere Geschwindigkeit, weil fehlende Blöcke aus dem Arbeitsspeicher geholt werden, nicht von der Festplatte, d.h., Suchzeit und Rotationsverzögerung entfallen.

Ein weiteres Beispiel ist ein Webserver. Der Server kann eine bestimmte Anzahl von viel benutzten Seiten in seinem Cache im Speicher halten. Wenn der Cache voll ist und eine neue Seite benötigt wird, muss der Server entscheiden, welche Seite er aus dem Speicher entfernen soll. Der einzige größere Unterschied zu Seiten im virtuellen Speicher ist, dass Webseiten niemals im Cache modifiziert werden und deshalb auch nicht „auf die Platte“ zurückgeschrieben werden müssen. In einem System mit virtuellem Speicher sind die Seiten im Arbeitsspeicher entweder unverändert (*clean*) oder verändert (*dirty*).

In allen Seitenersetzungsalgorithmen, die wir im Folgenden betrachten werden, taucht immer wieder ein bestimmter Aspekt auf: Wenn eine Seite aus dem Speicher verdrängt werden soll, muss es dann eine Seite sein, die zu dem Prozess gehört, der den Seitenfehler verursacht hat, oder kann es auch eine Seite sein, die zu einem anderen Prozess gehört? Im ersten Fall müssten alle Prozesse auf eine festgelegte Anzahl von Seiten begrenzt werden, im anderen Fall nicht. Beides ist möglich und wir werden in Abschnitt 3.5.1 noch darauf zurückkommen.

3.4.1 Der optimale Algorithmus zur Seitenersetzung

Der optimale Seitenersetzungsalgorithmus ist einfach zu beschreiben, aber leider unmöglich zu implementieren. Er funktioniert so: In dem Moment, in dem ein Seitenfehler auftritt, sind eine bestimmte Menge von Seiten im Arbeitsspeicher. Der nächste Befehl wird auf eine dieser Seiten zugreifen (die Seite, die den Befehl enthält). Auf andere Seiten wird vielleicht erst in 10, 100 oder 1.000 Befehlen zugegriffen. Jede Seite wird mit der Anzahl der Befehle markiert, die bis zum nächsten Zugriff auf diese Seite ausgeführt werden.

Der optimale Algorithmus entfernt einfach die Seite mit der höchsten Zahl von Befehlen bis zum nächsten Zugriff. Wenn eine Seite nach acht Millionen Befehlen wieder gebraucht wird und eine andere schon nach sechs Millionen, zögert man den nächsten Seitenfehler so lange wie möglich hinaus, indem man die erste der beiden Seiten auslagert. Auch Computer schieben unangenehme Ereignisse gern vor sich her.

Das einzige Problem mit diesem Algorithmus ist, dass er nicht realisierbar ist. Im Moment des Seitenfehlers kann das Betriebssystem nicht wissen, wann auf welche Seite das nächste Mal zugegriffen wird. (Der Scheduling-Algorithmus Shortest Job First war ein ähnliches Beispiel – wie soll das Betriebssystem wissen, welcher Auftrag der kürzeste ist?) Trotzdem ist es möglich, ein Programm auf einem Simulator laufen zu lassen, sich jeden Seitenzugriff zu merken und diese Informationen dann zu benutzen, um für den *zweiten* Ablauf einen optimalen Seitenersetzungsalgorithmus zu implementieren.

So kann man zumindest die Leistung von realisierbaren Algorithmen mit der bestmöglichen Leistung vergleichen. Wenn ein Betriebssystem z.B. eine nur 1% schlechtere Leistung erreicht als der optimale Algorithmus, kann ein verbesserter Algorithmus die Leistung höchstens um 1% steigern.

Es sollte klargestellt werden, dass die gespeicherte Folge von Seitenzugriffen nur für ein Programm und für einen einzigen Satz Eingabedaten gilt. Der Seitenersetzungsalgorithmus, der sich aus dieser Information ergibt, funktioniert also nur für genau dieses Programm mit genau dieser Eingabe. Diese Methode eignet sich nur zum Testen von Algorithmen, für reale Systeme ist sie nutzlos. Ab jetzt behandeln wir Algorithmen, die in realen Systemen eingesetzt werden können.

3.4.2 Der Not-Recently-Used-Algorithmus (NRU)

Computer mit virtuellem Speicher haben meistens zwei Statusbits, die es dem Betriebssystem erlauben, nützliche Statistiken über die Benutzung von Seiten aufzustellen. Das *R*-Bit wird immer dann gesetzt, wenn auf die Seite zugegriffen wird (Lese- und Schreibzugriffe). Das *M*-Bit wird gesetzt, wenn auf die Seite geschrieben wird (d.h., wenn die Seite modifiziert wird). Diese Bits sind, wie in ▶Abbildung 3.11, ein Teil des Seitentabelleneintrages. Es ist wichtig, dass sie von der Hardware gesetzt werden, weil sie bei jedem einzelnen Speicherzugriff aktualisiert werden müssen. Wenn ein Bit einmal auf 1 gesetzt wird, bleibt es 1, bis das Betriebssystem es zurücksetzt.

Falls die Hardware diese Bits nicht zur Verfügung stellt, können sie folgendermaßen simuliert werden: Wenn ein Prozess gestartet wird, werden zunächst alle seine Seitentabelleneinträge als ausgelagert markiert. Sobald auf eine Seite zugegriffen wird, gibt es einen Seitenfehler. Das Betriebssystem setzt dann das *R*-Bit (in einer internen Tabelle), ändert den Tabelleneintrag, so dass er auf den richtigen Seitenrahmen im READ-ONLY-Modus zeigt, und führt den Befehl noch einmal aus. Wenn der Prozess dann auf die Seite schreibt, gibt es noch einmal einen Seitenfehler und das Betriebssystem kann das *M*-Bit setzen und den Zugriffsmodus auf READ/WRITE ändern.

Der folgende einfache Seitenersetzungsalgorithmus basiert auf den *R*- und *M*-Bits. Wenn ein Prozess gestartet wird, setzt das Betriebssystem die *R*- und *M*-Bits für alle seine Seiten auf 0. In bestimmten Zeitabständen (z.B. bei jedem Timerinterrupt) werden alle *R*-Bits gelöscht. Damit ist nur bei den Seiten, die in letzter Zeit gebraucht wurden, das *R*-Bit gesetzt.

Bei einem Seitenfehler teilt das Betriebssystem alle Seiten nach dem Zustand der *R*- und *M*-Bits in vier Kategorien auf:

- Klasse 0: nicht referenziert, nicht modifiziert
- Klasse 1: nicht referenziert, modifiziert
- Klasse 2: referenziert, nicht modifiziert
- Klasse 3: referenziert und modifiziert

Auf den ersten Blick scheint es so, als wären Seiten der Klasse 1 unmöglich, aber sie können entstehen, wenn bei einer Seite in Klasse 3 das *R*-Bit durch einen Timerinterrupt gelöscht wird. Timerinterrupts löschen das *M*-Bit nicht, weil es für die Entscheidung gebraucht wird, ob eine Seite auf die Platte zurückgeschrieben werden muss oder nicht. Wenn *R* gelöscht wird und *M* nicht, entstehen Seiten der Klasse 1.

Der **NRU-Algorithmus** (**Not Recently Used**) entfernt eine zufällige Seite aus der niedrigsten nicht leeren Klasse. Die Ordnung der Klassen impliziert, dass es besser ist, eine modifizierte Seite auszulagern, die im letzten Timerintervall (typischerweise ungefähr 20 ms) nicht referenziert wurde, anstatt eine saubere Seite, die ständig benutzt wird. Die Vorteile des NRU-Algorithmus sind, dass er leicht verständlich und einigermaßen effizient zu implementieren ist. Die Leistung ist sicher nicht optimal, aber in vielen Fällen ausreichend.

3.4.3 Der First-In-First-Out-Algorithmus (FIFO)

Ein weiterer Seitenersetzungsalgorithmus mit geringem Aufwand ist der **FIFO-Algorithmus** (**First In First Out**). Um zu verstehen, wie er funktioniert, stellen wir uns einen Supermarkt vor, der genügend Regale hat, um genau k verschiedene Produkte aufzustellen. Eines Tages kommt ein neues Fertiggericht auf den Markt – Tiefkühl-Bio-Instantjoghurt für die Mikrowelle. Das Produkt ist ein voller Erfolg, also muss unser räumlich begrenzter Supermarkt eines seiner alten Produkte loswerden, um Platz zu schaffen.

Er könnte feststellen, welches Produkt schon am längsten verkauft wird (z. B. etwas, was schon seit 120 Jahren verkauft wird), und es mit der Begründung abschaffen, dass es keinen mehr interessiert. Der Supermarkt würde dann eine verkettete Liste von allen Produkten, die gerade verkauft werden, verwalten, geordnet nach dem Datum ihrer Einführung. Das neue Produkt wird an das Ende der Liste angehängt und das Produkt ganz vorne wird gelöscht.

Dieselbe Methode ist auch als Seitenersetzungsalgorithmus anwendbar. Das Betriebssystem verwaltet eine Liste von allen Seiten im Speicher. Am Ende der Liste steht der jüngste Eingang und am Kopf der älteste. Bei einem Seitenfehler wird die Seite am Kopf der Liste entfernt und die neue Seite wird an das Ende angehängt. Der FIFO-Algorithmus würde aus einem Supermarkt Dinge wie Schnurrbartwachs entfernen, er könnte aber auch Mehl, Salz oder Butter abschaffen. Auf Computer angewendet bereitet FIFO ähnliche Probleme und wird deshalb nur selten unverändert eingesetzt.

3.4.4 Der Second-Chance-Algorithmus

Eine einfache Variante von FIFO, die das Problem vermeidet, dass Seiten ausgelagert werden, obwohl sie häufig benutzt werden, ist der **Second-Chance-Algorithmus**. Er überprüft zunächst das R-Bit der ältesten Seite. Wenn es nicht gesetzt ist, ist die Seite nicht nur alt, sondern auch unbenutzt, und wird sofort ersetzt. Ansonsten wird das R-Bit gelöscht, die Seite wird an das Ende der Liste verschoben und die Ladezeit wird so gesetzt, als wäre die Seite eben erst geladen worden. Dann wird die Suche fortgesetzt.

▶Abbildung 3.15 zeigt, wie der Algorithmus funktioniert. ▶Abbildung 3.15(a) zeigt eine verkettete Liste der Seiten A bis H, geordnet nach der Zeit, zu der sie geladen wurden.

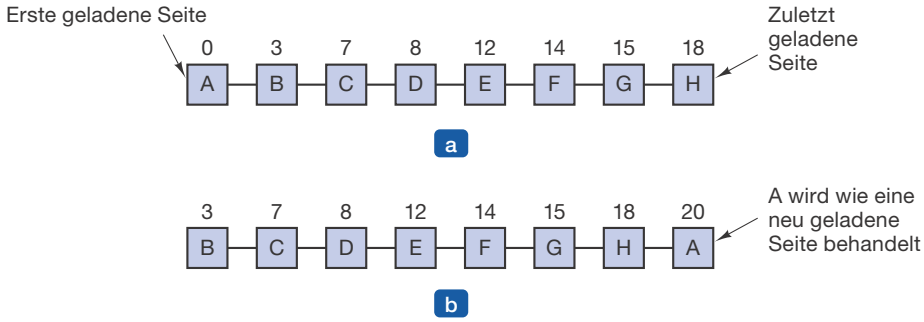

Abbildung 3.15: Arbeitsweise von Second Chance (a) Seiten in FIFO-Ordnung sortiert (b) Die Seitenliste nach einem Seitenfehler zum Zeitpunkt 20, falls bei A das R-Bit gesetzt war. Die Zahlen sind Ladezeiten.

Nehmen wir an, zum Zeitpunkt 20 gibt es einen Seitenfehler. Die älteste Seite ist *A*, die beim Prozessstart zum Zeitpunkt 0 geladen wurde. Wenn das *R*-Bit von *A* nicht gesetzt ist, wird *A* aus dem Speicher genommen: entweder auf die Platte zurückgeschrieben (wenn das *M*-Bit gesetzt ist) oder einfach überschrieben (falls $M = 0$). Andernfalls (wenn das *R*-Bit gesetzt ist) wird *A* an das Ende der Liste verschoben und die „Ladezeit" wird auf die aktuelle Zeit (20) gesetzt. Außerdem wird das *R*-Bit gelöscht. Die Suche nach einer passenden Seite geht dann bei *B* weiter.

Second Chance sucht eigentlich nur nach einer möglichst alten Seite, auf die in den letzten Timerintervallen nicht zugegriffen wurde. Wenn alle Seiten referenziert wurden, degeneriert Second Chance zu einem reinen FIFO-Algorithmus. Nehmen wir z. B. an, dass die *R*-Bits aller Seiten in ▶Abbildung 3.15(a) gesetzt sind. Das Betriebssystem verschiebt dann eine Seite nach der anderen an das Ende der Liste und löscht dabei die *R*-Bits. Schließlich rückt *A* wieder an den Anfang der Liste vor, diesmal aber mit gelöschtem *R*-Bit, und wird ausgelagert. Der Algorithmus terminiert also auf jeden Fall.

3.4.5 Der Clock-Algorithmus

Second Chance ist ein vernünftiger Algorithmus, aber er ist unnötig ineffizient, weil er ständig Seiten in der Liste verschiebt. Besser wäre es, alle Seiten in einer ringförmigen Liste von der Form einer Uhr zu halten, siehe ▶Abbildung 3.16. Der Uhrzeiger zeigt auf die älteste Seite.

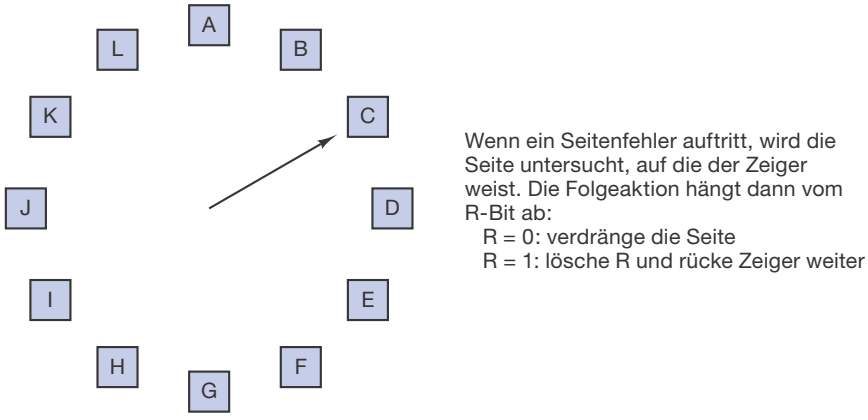

Abbildung 3.16: Der Clock-Algorithmus

Bei einem Seitenfehler wird zunächst die Seite geprüft, auf die der Uhrzeiger zeigt. Wenn das *R*-Bit 0 ist, wird die Seite ausgelagert, die neue Seite wird an derselben Stelle in die Uhr eingefügt und der Zeiger rückt um eine Seite vor. Wenn das *R*-Bit 1 ist, wird es gelöscht, der Zeiger wird vorgerückt und der Vorgang wird so lange wiederholt, bis eine Seite mit $R = 0$ gefunden ist. Wenig überraschend heißt dieser Algorithmus **Clock**.

3.4.6 Der Least-Recently-Used-Algorithmus (LRU)

Eine gute Annäherung an den optimalen Algorithmus basiert auf der folgenden Beobachtung: Eine Seite, die von den letzten paar Befehlen häufig benutzt wurde, wird wahrscheinlich auch für die nächsten Befehle gebraucht. Umgekehrt werden Seiten, die seit einer Ewigkeit unbenutzt sind, wahrscheinlich noch lange Zeit unbenutzt bleiben. Damit drängt sich ein realisierbarer Algorithmus auf: Entferne bei einem Seitenfehler die Seite, die am längsten unbenutzt ist. Diese Strategie heißt **LRU** (**Least Recently Used**).

LRU ist zwar theoretisch realisierbar, aber es ist nicht billig. Für eine vollständige Implementierung von LRU ist eine verkettete Liste von allen Seiten im Speicher nötig, wobei die zuletzt benutzte Seite am Anfang steht und die am längsten nicht benutzte Seite am Ende. Das Problem ist, dass die Liste bei jedem Speicherzugriff aktualisiert werden muss. Eine Seite in der Liste zu finden, sie zu löschen und anschließend an den Anfang der Liste zu verschieben, ist eine sehr aufwändige Operation, selbst wenn sie in der Hardware implementiert ist (falls solche Hardware überhaupt gebaut werden kann).

Es gibt aber andere Möglichkeiten, LRU mit Spezialhardware zu implementieren. Sehen wir uns zunächst die einfachste an. Für diese Methode muss die Hardware mit einem 64-Bit-Zähler C ausgestattet werden, der nach jedem Maschinenbefehl automatisch erhöht wird. Außerdem muss jeder Eintrag in der Seitentabelle ein Feld haben, das groß genug für den Zähler ist. Bei jedem Speicherzugriff wird der aktuelle Zählerstand im Tabelleneintrag der Seite gespeichert, auf die zugegriffen wurde. Bei einem Seitenfehler durchsucht das Betriebssystem die Seitentabelle nach der Seite mit dem niedrigsten Zählerstand. Diese Seite wurde am längsten von allen nicht benutzt.

Sehen wir uns nun eine andere Hardwareversion von LRU an. Bei diesem Ansatz enthält die LRU-Hardware für eine Maschine mit n Seitenrahmen eine Matrix aus $n \times n$ Bits, die am Anfang alle 0 sind. Wenn auf einen Seitenrahmen k zugegriffen wird, setzt die Hardware zunächst alle Bits der Zeile k auf 1 und dann alle Bits der Spalte k auf 0. Zu jedem Zeitpunkt ist die Zeile mit dem niedrigsten Binärwert die am längsten nicht benutzte, die Zeile mit dem nächsthöheren Wert ist die am zweitlängsten nicht benutzte und so weiter. ▶Abbildung 3.17 zeigt ein Beispiel für diesen Algorithmus mit vier Seitenrahmen und der Zugriffsfolge

0 1 2 3 2 1 0 3 2 3

Nach dem ersten Zugriff auf Seite 0 haben wir den Zustand in ▶Abbildung 3.17(a). Der nächste Zugriff auf Seite 1 ergibt ▶Abbildung 3.17(b) und so weiter.

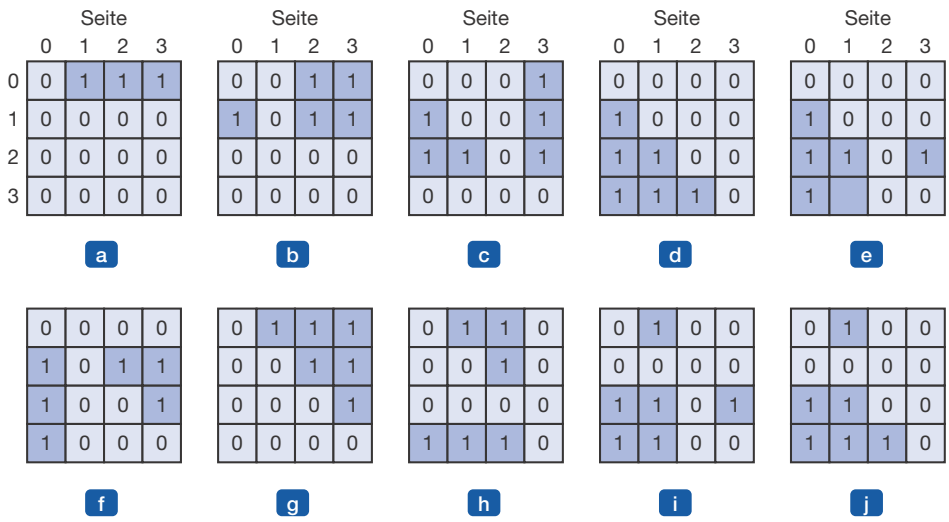

Abbildung 3.17: LRU mit einer Matrix. Auf die Seiten wird in der Reihenfolge 0 1 2 3 2 1 0 3 2 3 zugegriffen.

3.4.7 Simulation von LRU durch Software

Die beiden vorgestellten LRU-Algorithmen sind zwar (im Prinzip) realisierbar, in der Praxis haben aber, wenn überhaupt, nur wenige Maschinen die erforderliche Hardware. Stattdessen wird eine Lösung gebraucht, die als Software implementiert werden kann. Eine Möglichkeit ist der sogenannte **NFU-Algorithmus** (**Not Frequently Used**). Diese Lösung benötigt für jede Seite einen Softwarezähler, der zu Beginn auf 0 gesetzt ist. Bei jedem Timerinterrupt durchläuft das Betriebssystem alle Seiten im Speicher und addiert zu jedem Zähler das *R*-Bit der zugehörigen Seite, also 0 oder 1. Die Zähler zählen grob mit, wie oft auf jede Seite zugegriffen wird. Bei einem Seitenfehler wird die Seite mit dem niedrigsten Zählerstand ersetzt.

Das Hauptproblem von NFU ist, dass er niemals etwas vergisst. Bei einem Multipass-Compiler könnten z.B. die Seiten, die im ersten Durchlauf viel benutzt wurden, auch in späteren Durchläufen noch einen hohen Zählerstand haben. Wenn der erste Durchlauf länger dauert als alle anderen, könnten die Seiten, die den Code für die späteren Durchläufe enthalten, sogar bis zum Schluss einen niedrigeren Zählerstand haben. Das Betriebssystem wird dann nützliche Seiten auslagern und Seiten, die nie mehr gebraucht werden, im Speicher lassen.

Zum Glück ist nur eine kleine Veränderung an NFU nötig, um ihn zu einer recht guten Annäherung an LRU zu machen. Die Veränderung besteht aus zwei Teilen. Erstens werden die Zähler immer ein Bit nach rechts geschoben, bevor das *R*-Bit addiert wird. Zweitens wird das *R*-Bit zum ganz linken (höchstwertigen) Bit des Zählers addiert, nicht zum rechten.

Wie der verbesserte Algorithmus, bekannt als **Aging**, funktioniert, zeigt ▶Abbildung 3.18. Nehmen wir an, dass nach dem ersten Zeitintervall die *R*-Bits der Seiten 0 bis 5 die Werte 1, 0, 1, 0, 1, 1 haben (d.h., Seite 0 ist 1, Seite 1 ist 0, Seite 2 ist 1 usw.). Mit anderen Worten: Im ersten Intervall wurden die Seiten 0, 2, 4 und 5 referenziert und die *R*-Bits dieser Seiten wurden auf 1 gesetzt. Die *R*-Bits der restlichen Seiten sind 0, weil auf sie nicht zugegriffen wurde. Nachdem die sechs Zähler der Seiten um ein Bit nach rechts verschoben und die *R*-Bits von links eingefügt wurden, ergeben sich die Werte in ▶Abbildung 3.18(a). Die übrigen vier Spalten zeigen die Zustände nach den nächsten vier Intervallen.

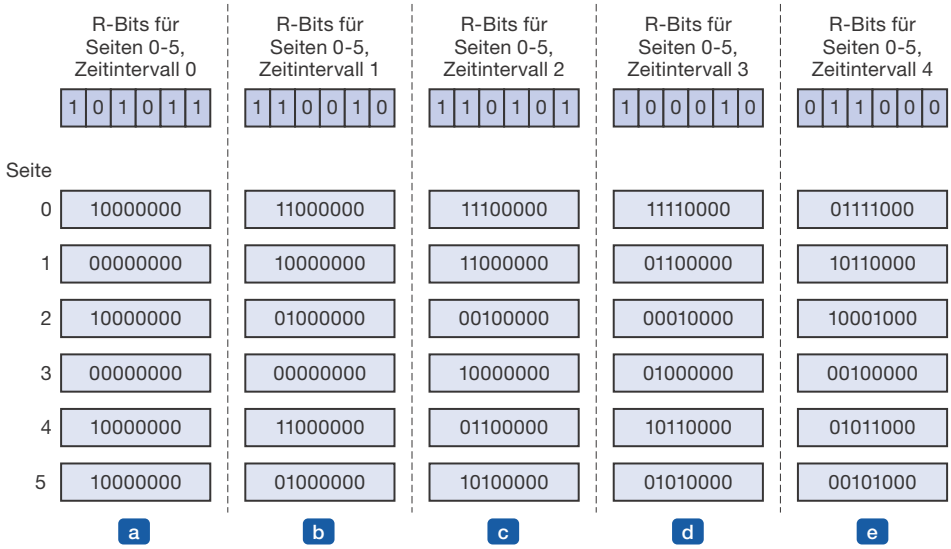

Abbildung 3.18: Der Aging-Algorithmus ist eine Software-Simulation von LRU. Dargestellt werden die Zähler von sechs Seiten für fünf Intervalle. (a) bis (e) zeigen die Zustände nach den Intervallen 1–5.

Bei einem Seitenfehler wird die Seite mit dem niedrigsten Zählerstand entfernt. Eine Seite, auf die in den letzten vier Intervallen nicht zugegriffen wurde, hat logischerweise vier führende Nullen in ihrem Zähler und damit einen niedrigeren Zählerstand als eine Seite, auf die erst seit drei Intervallen nicht zugegriffen wurde.

Zwischen Aging und LRU gibt es zwei Unterschiede. Sehen wir uns die Seiten 3 und 5 in ▶Abbildung 3.18(e) an. Auf beide Seiten wurde zuletzt vor drei Intervallen zugegriffen. Wenn eine Seite ausgelagert wird, sollte es nach LRU eine dieser beiden sein. Das Problem ist, wir wissen nicht, auf welche der beiden im zweiten Intervall zuletzt zugegriffen wurde. Dadurch, dass pro Intervall nur ein Bit gespeichert wird, können wir nicht mehr zwischen Zugriffen am Anfang und am Ende eines Intervalls unterscheiden. Das Beste, was wir tun können, ist, Seite 3 auszulagern, weil auf Seite 5 schon zwei Intervalle zuvor zugegriffen wurde und auf Seite 3 nicht.

Der zweite Unterschied ist, dass die Zähler beim Aging-Algorithmus eine endliche Anzahl von Bits haben (in diesem Beispiel 8), was das Aufzeichnen vergangener Zugriffe begrenzt. Wenn zwei Seiten beide einen Zählerstand von 0 haben, können

wir nur willkürlich eine der beiden auswählen. In Wirklichkeit wurde auf die Seite, die wir auslagern, vielleicht vor neun Intervallen zugegriffen und auf die andere vor 1.000 Intervallen. Es gibt keine Möglichkeit, das herauszufinden. In der Praxis reichen 8 Bit aber normalerweise aus, wenn ein Intervall 20 ms dauert. Eine Seite, die 160 ms lang nicht gebraucht wird, ist wahrscheinlich nicht besonders wichtig.

3.4.8 Der Working-Set-Algorithmus

In der reinsten Form des Paging startet ein Prozess mit keiner einzigen Seite im Arbeitsspeicher. Sobald die CPU versucht, den ersten Befehl zu laden, gibt es einen Seitenfehler und das Betriebssystem lagert die Seite ein, die den ersten Befehl enthält. Kurz darauf folgen weitere Seitenfehler für den Stack und für globale Variablen. Nach einer Weile hat der Prozess dann die meisten Seiten zusammen, die er braucht, und läuft mit relativ wenigen Seitenfehlern weiter. Diese Strategie heißt **Demand Paging** oder **Einlagern bei Bedarf**, weil die Seiten nur auf Anforderung, nicht im Voraus geladen werden.

Natürlich ist es leicht, ein Testprogramm zu schreiben, das systematisch auf alle Seiten in einem großen Adressraum zugreift, um so viele Seitenfehler zu erzeugen, dass der Speicher nicht ausreicht, um alle Seiten einzulagern. Zum Glück funktionieren die meisten Prozesse nicht so, sondern neigen zu einem Verhalten, das als **Lokalitätseigenschaft** (*locality of reference*) bezeichnet wird, d.h., sie beschränken ihre Zugriffe in jeder Phase ihrer Ausführung auf einen relativ kleinen Teil ihrer Seiten. Jeder Durchgang eines Multipass-Compilers greift z.B. nur auf einen Bruchteil seiner Seiten zu, der sich außerdem mit jedem Durchgang ändert.

Die Menge von Seiten, die ein Prozess zu einem bestimmten Zeitpunkt benutzt, wird **Arbeitsbereich** (*working set*) genannt (Denning, 1968a; Denning, 1980). Wenn der gesamte Arbeitsbereich im Speicher ist, läuft der Prozess ohne viele Seitenfehler bis zur nächsten Phase seiner Ausführung (z.B. bis zum nächsten Durchgang eines Compilers). Wenn der verfügbare Speicher nicht für alle Seiten im Arbeitsbereich ausreicht, erzeugt der Prozess viele Seitenfehler und läuft sehr langsam ab, weil es nur ein paar Nanosekunden dauert, einen Befehl auszuführen, aber typischerweise 10 ms, eine Seite von der Platte zu lesen. Bei nur ein oder zwei Befehlen alle 10 ms dauert es ewig, bis der Prozess fertig ist. Denning (1968a) hat für dieses Verhalten den Begriff **Thrashing** (**Seitenflattern**) eingeführt.

In einem System mit Multiprogrammierung werden Prozesse häufig auf die Platte ausgelagert (d.h., alle ihre Seiten werden aus dem Speicher entfernt), um andere Prozesse zum Zug kommen zu lassen. Dabei stellt sich die Frage, was man tun soll, wenn ein Prozess wieder eingelagert wird. Im Prinzip reicht es aus, gar nichts zu tun: Der Prozess wird einfach so lange Seitenfehler erzeugen, bis sein Arbeitsbereich geladen ist. Das Problem ist nur, dass diese Strategie sehr langsam ist und eine Menge CPU-Zeit verschwendet, weil dann jedes Mal, wenn ein Prozess geladen wird, 20, 100 oder sogar 1.000 Seitenfehler erzeugt werden und jeder Seitenfehler einige Millisekunden CPU-Zeit verbraucht.

Viele Betriebssysteme merken sich deshalb den Arbeitsbereich eines Prozesses, wenn sie ihn auslagern, und sorgen dafür, dass er wieder geladen wird, bevor sie den Pro-

zess weiter ausführen. Dieser Ansatz wird **Working-Set-Modell** genannt (Denning, 1970) und soll die Seitenfehlerrate stark reduzieren. Man sollte dabei nicht vergessen, dass der Arbeitsbereich sich mit der Zeit ändert. Strategien, die Seiten laden, noch *bevor* sie gebraucht werden, werden auch **Prepaging** genannt.

Es ist schon lange bekannt, dass die Speicherzugriffe eines Programms nicht gleichmäßig über den Adressraum verteilt sind, sondern sich auf einige wenige Seiten konzentrieren. Jeder Speicherzugriff holt entweder einen Maschinenbefehl, liest Daten aus dem Speicher oder schreibt Daten in den Speicher. Zu jedem Zeitpunkt t gibt es eine Menge von Seiten, die in den letzten k Speicherzugriffen benutzt wurden. Diese Menge $w(k, t)$ ist der Arbeitsbereich. Da die letzten $k = 1$ Zugriffe mindestens alle Seiten der letzten $k > 1$ Zugriffe referenziert haben (und möglicherweise noch mehr), ist $w(k, t)$ eine (nicht streng) monoton steigende Funktion von k. Der Grenzwert von $w(k, t)$ für große k ist endlich, weil ein Programm nicht auf mehr Seiten zugreifen kann als in seinen Adressraum passen und nur wenige Programme jede einzelne Seite benutzen. ▶Abbildung 3.19 zeigt die Größe des Arbeitsbereiches in Abhängigkeit von k.

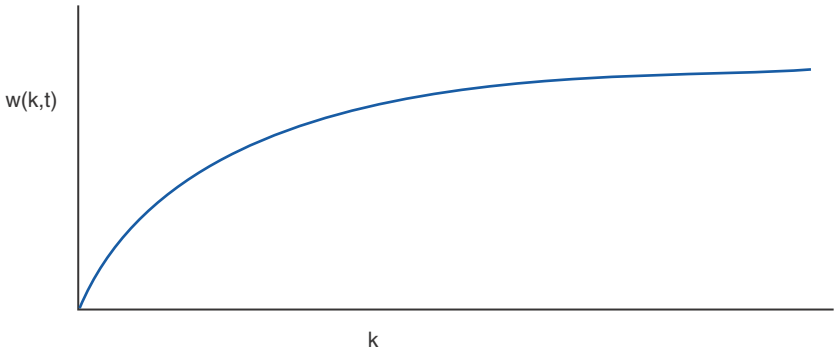

Abbildung 3.19: Der Arbeitsbereich ist die Menge der Seiten, die von den letzten k Speicherzugriffen benutzt werden. Die Funktion $w(k, t)$ ist die Größe des Arbeitsbereiches zum Zeitpunkt t.

Die Tatsache, dass die meisten Programme zufällig auf eine kleine Anzahl von Seiten zugreifen, dass diese Menge sich aber mit der Zeit langsam ändert, erklärt den anfänglichen steilen Anstieg und das Abflachen der Kurve für große k. Beispielsweise könnte ein Programm eine Schleife ausführen, deren Code auf zwei Seiten liegt, und dabei Daten auf vier verschiedenen Seiten benutzen. Es würde dann ständig (z. B. jeweils nach 1.000 Befehlen) auf diese sechs Seiten zugreifen, aber der letzte Zugriff auf eine andere Seite könnte eine Million Befehle zurückliegen, in der Initialisierungsphase. Dieses asymptotische Verhalten führt dazu, dass der Inhalt des Arbeitsbereiches nicht sehr stark von der Wahl von k abhängt. Anders ausgedrückt: Es gibt einen großen Bereich von Werten für k, für die der Arbeitsbereich unverändert bleibt. Da der Arbeitsbereich eines Prozesses sich nur langsam ändert, kann man bei der Auslagerung des Prozesses relativ sicher vorhersagen, welche Seiten er später brauchen wird. Prepaging bedeutet, diese Seiten zu laden, bevor der ausgelagerte Prozess fortgesetzt wird.

Für die Implementierung des Working-Set-Modells muss das Betriebssystem zu jedem Zeitpunkt wissen, welche Seiten im Arbeitsbereich eines Prozesses sind. Aus dieser Information ergibt sich direkt ein Seitenersetzungsalgorithmus: Wenn ein Seitenfehler auftritt, finde eine Seite, die nicht zum Arbeitsbereich gehört, und lagere sie aus. Um so einen Algorithmus umzusetzen, brauchen wir ein präzise definiertes Kriterium dafür, welche Seiten zu einem bestimmten Zeitpunkt zum Arbeitsbereich gehören. Definitionsgemäß ist der Arbeitsbereich die Menge der Seiten, die in den letzten *k* Speicherzugriffen benutzt wurden (einige Autoren verwenden gleichwertig die letzten *k* Seitenreferenzen). Für die Implementierung eines Working-Set-Algorithmus muss der Wert von *k* im Voraus festgelegt werden. Damit ist dann nach jedem Zugriff die Menge der Seiten, die von den letzten *k* Speicherzugriffen benutzt wurden, eindeutig festgelegt.

Eine korrekte Definition des Arbeitsbereiches bedeutet natürlich noch lange nicht, dass er während der Programmausführung effizient berechnet werden kann. Es wäre z.B. ein Schieberegister der Länge *k* vorstellbar, in das bei jedem Speicherzugriff von rechts die entsprechende Seitennummer geschoben wird. Die Menge der *k* Seitennummern in dem Register wären dann der Arbeitsbereich. Bei einem Seitenfehler könnte der Inhalt des Registers theoretisch ausgelesen und sortiert werden. Man müsste nur die doppelt vorkommenden Seiten entfernen und das Ergebnis wäre der Arbeitsbereich. Leider wäre es viel zu aufwändig, das Schieberegister auf dem neuesten Stand zu halten und es bei einem Seitenfehler zu verarbeiten. Deshalb wird diese Technik nicht angewendet.

Es gibt allerdings verschiedene Annäherungen, die in realen Systemen einsetzbar sind. Eine häufig benutzte Annäherung besteht darin, sich nicht mehr die letzten *k* Speicherzugriffe zu merken, sondern die Ausführungszeit des Prozesses zu benutzen. Beispielsweise könnten wir den Arbeitsbereich nicht mehr als die Seiten definieren, die von den letzten zehn Millionen Speicherzugriffen benutzt wurden, sondern als die Seiten, auf die in den letzten 100 ms zugegriffen wurde. In der Praxis ist diese Definition völlig gleichwertig und viel einfacher zu implementieren. Wichtig ist, dass für jeden Prozess nur seine eigene Ausführungszeit zählt. Wenn ein Prozess also zum Zeitpunkt T gestartet wird und 40 ms CPU-Zeit bis zum Zeitpunkt $T + 100$ ms bekommt, beträgt seine Zeit für den Working-Set-Algorithmus 40 ms. Die CPU-Zeit, die ein Prozess seit seinem Start benutzt hat, wird oft als **virtuelle Zeit** (*current virtual time*) bezeichnet. Bei dieser Annäherung ist der Arbeitsbereich eines Prozesses die Menge der Seiten, auf die er in den letzten τ Sekunden virtueller Zeit zugegriffen hat.

Sehen wir uns jetzt einen Seitenersetzungsalgorithmus an, der auf dem Arbeitsbereich eines Prozesses basiert. Die Grundidee ist, bei einem Seitenfehler eine Seite auszulagern, die nicht zum Arbeitsbereich gehört. ▶Abbildung 3.20 zeigt einen Ausschnitt aus einer Seitentabelle. Weil nur Seiten, die im Speicher liegen, für die Auslagerung in Frage kommen, übergeht der Algorithmus die ausgelagerten Seiten. Jeder Eintrag enthält (mindestens) zwei Informationen: die (ungefähre) Zeit des letzten Zugriffes auf die Seite und das *R*-Bit. Der leere Bereich steht für die anderen Felder, die dieser Algorithmus nicht braucht, wie z.B. die Seitenrahmennummer, das *M*-Bit oder die *Protection*-Bits.

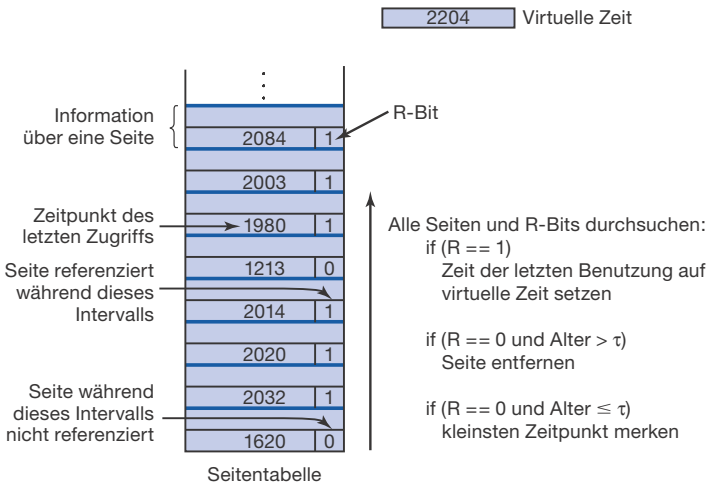

Abbildung 3.20: Der Working-Set-Algorithmus

Der Algorithmus funktioniert folgendermaßen: Zunächst setzen wir voraus, dass die Hardware die *R*- und *M*-Bits setzt, wie oben bereits angesprochen. Außerdem nehmen wir an, dass ein periodischer Timerinterrupt dafür sorgt, dass die *R*-Bits softwaremäßig gelöscht werden. Bei jedem Seitenfehler wird die Seitentabelle nach einer Seite durchsucht, die ausgelagert werden kann.

Bei jedem Eintrag wird zunächst das *R*-Bit untersucht. Wenn es gesetzt ist, wird die virtuelle Zeit in das Feld für den *Zeitpunkt des letzten Zugriffs* eingetragen. Das bedeutet, dass die Seite zum Zeitpunkt des Seitenfehlers benutzt wurde. Da die Seite im aktuellen Timerintervall verwendet wurde, gehört sie offensichtlich zum Arbeitsbereich und kommt nicht für die Auslagerung in Frage (wir nehmen an, dass τ mehrere Timerintervalle dauert).

Wenn $R = 0$ ist, wurde auf die Seite seit dem letzten Timerinterrupt nicht zugegriffen und die Seite ist ein möglicher Kandidat für die Auslagerung. Um zu entscheiden, ob die Seite zum Arbeitsbereich gehört, wird ihr Alter (die virtuelle Zeit minus *Zeitpunkt des letzten Zugriffs*) berechnet und mit τ verglichen. Wenn das Alter größer als τ ist, gehört die Seite nicht mehr zum Arbeitsbereich und sie wird durch die neue Seite ersetzt. Der Rest der Tabelle wird noch durchlaufen, um die Zugriffszeiten auf den neuesten Stand zu bringen.

Wenn *R* nicht gesetzt, aber das Alter gleich oder geringer als τ ist, gehört die Seite noch zum Arbeitsbereich und sie wird vorläufig verschont, aber die Seite mit dem höchsten Alter (d.h. dem kleinsten Wert von *Zeitpunkt des letzten Zugriffs*) wird vermerkt. Wenn die ganze Tabelle durchsucht und kein Kandidat gefunden wurde, bedeutet das, dass alle Seiten im Arbeitsbereich liegen. Wenn eine oder mehrere Seiten mit $R = 0$ gefunden wurden, wird dann die mit dem höchsten Alter ausgelagert. Schlimmstenfalls wurde im letzten Intervall auf alle Seiten zugegriffen (d.h., alle *R*-Bits sind gesetzt), so dass eine willkürlich gewählte Seite ausgelagert werden muss, möglichst eine, deren Inhalt nicht verändert wurde.

3.4.9 Der WSClock-Algorithmus

Der einfache Working-Set-Algorithmus ist umständlich, weil er bei jedem Seitenfehler die gesamte Seitentabelle durchläuft, bis ein passender Kandidat gefunden ist. Ein verbesserter Algorithmus, der auf dem Clock-Algorithmus aufbaut, aber auch Informationen über den Arbeitsbereich nutzt, heißt **WSClock** (Carr und Hennessey, 1981). Wegen seiner guten Leistung und einfachen Implementierung ist er in realen Systemen weit verbreitet.

Genau wie der Clock-Algorithmus benutzt WSClock eine ringförmige Liste von Seitenrahmen (▶Abbildung 3.21). Anfänglich ist die Liste leer. Wenn die erste Seite geladen wird, wird sie in die Liste eingefügt. Mit der Zeit kommen immer mehr Seiten hinzu und die Liste wird zu einem Ring. Jeder Listeneintrag enthält ein *R*-Bit, ein *M*-Bit (nicht in der Abbildung) und ein Feld für den *Zeitpunkt des letzten Zugriffs* aus dem einfachen Working-Set-Algorithmus.

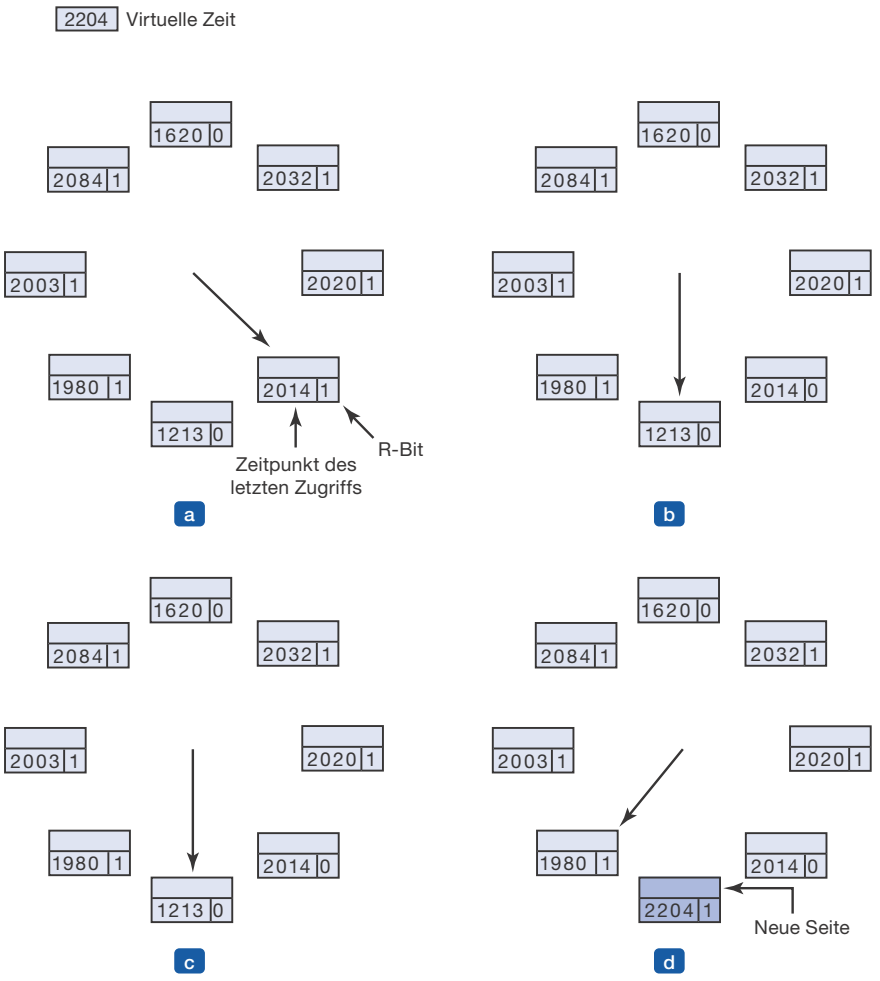

Abbildung 3.21: Die Funktionsweise des WSClock-Algorithmus (a) und (b) zeigen, was bei $R = 1$ passiert. (c) und (d) geben ein Beispiel für $R = 0$.

Wie beim Clock-Algorithmus wird bei einem Seitenfehler zunächst die Seite untersucht, auf die der Uhrzeiger zeigt. Wenn das *R*-Bit gesetzt ist, wurde die Seite im letzten Intervall benutzt. Sie ist damit kein idealer Kandidat zur Auslagerung. Das *R*-Bit wird auf 0 gesetzt, der Zeiger rückt vor zur nächsten Seite und der Algorithmus wird für diese Seite wiederholt. ▶Abbildung 3.21(b) zeigt den Zustand nach diesen Ereignissen.

Überlegen wir uns jetzt, was passiert, wenn das *R*-Bit 0 ist, wie in ▶Abbildung 3.21(c). Wenn das Alter der Seite größer ist als τ und die Seite nicht verändert wurde, gehört sie nicht zum Arbeitsbereich und es existiert eine gültige Kopie auf der Platte. Die Seite wird dann einfach wie in ▶Abbildung 3.21(d) gelöscht und durch die neue Seite ersetzt. Wenn der Inhalt der Seite verändert wurde, kann sie nicht einfach gelöscht werden, weil die Kopie auf der Platte nicht aktuell ist. Um einen Prozesswechsel zu vermeiden, wird vorgemerkt, den Seiteninhalt auf die Platte zu schreiben, aber der Zeiger wird vorgerückt und der Algorithmus macht mit der nächsten Seite weiter. Die Seite wird nicht sofort ausgelagert, weil es weiter unten in der Liste vielleicht eine alte, aber saubere Seite gibt, die einfach gelöscht werden kann.

Im Prinzip könnten nach einem kompletten Listendurchlauf alle Seiten dafür vorgemerkt sein, auf die Platte geschrieben zu werden. Um die Belastung der Festplatte zu beschränken, könnte man ein Limit setzen, so dass maximal *n* Seiten auf die Platte geschrieben werden können. Sobald dieses Limit erreicht ist, werden keine neuen Seiten vorgemerkt.

Wenn der Zeiger die ganze Liste durchläuft und wieder am Anfang ankommt, muss man zwei Fälle unterscheiden:

1. Es wurde mindestens eine Seite dafür vorgemerkt, auf die Platte geschrieben zu werden.
2. Es wurde keine Seite vorgemerkt.

Im ersten Fall läuft der Zeiger einfach weiter und sucht nach einer sauberen Seite. Irgendwann wurde eine der Seiten auf die Platte geschrieben und das *M*-Bit wird gelöscht (d.h., die Seite ist sauber). Die erste saubere Seite, auf die der Algorithmus trifft, wird ersetzt. Diese Seite muss nicht unbedingt die zuerst vorgemerkte sein, weil der Festplattentreiber die Schreibbefehle neu ordnen kann, um die Leistung zu optimieren.

Im zweiten Fall gehören alle Seiten zum Arbeitsbereich, weil ansonsten mindestens eine Seite vorgemerkt wäre. Ohne zusätzliche Informationen besteht die einfachste Strategie darin, irgendeine Seite auszulagern und den Seitenrahmen für die neue Seite zu benutzen. Während des Listendurchlaufs könnte sich der Algorithmus eine saubere Seite merken, die er dann im Notfall auslagert. Existiert keine saubere Seite, wird einfach die aktuelle Seite geopfert und zurück auf die Platte geschrieben.

3.4.10 Zusammenfassung der Seitenersetzungsstrategien

In diesem Abschnitt fassen wir die verschiedenen Seitenersetzungsalgorithmen, die wir gesehen haben, noch einmal zusammen. ▶Abbildung 3.22 enthält eine Liste der behandelten Algorithmen.

Algorithmus	Kommentar
Optimal	Nicht realisierbar, aber nützlich als Maßstab
NRU (Not Recently Used)	Sehr grobe Annäherung an LRU
FIFO (First In First Out)	Entfernt evtl. auch wichtige Seiten
Second Chance	Enorme Verbesserung gegenüber FIFO
Clock	Realistisch
LRU (Least Recently Used)	Exzellent, aber schwierig zu implementieren
NFU (Not Frequently Used)	Ziemlich grobe Annäherung an LRU
Aging	Effizienter Algorithmus, gute Annäherung an LRU
Working Set	Etwas aufwändig zu implementieren
WSClock	Guter und effizienter Algorithmus

Abbildung 3.22: Die behandelten Seitenersetzungsalgorithmen

Der optimale Algorithmus verdrängt die Seite, deren Aufruf am weitesten in der Zukunft liegt. Leider ist es unmöglich, herauszufinden, welche Seite dies ist, also ist dieser Algorithmus in der Praxis nicht einsetzbar. Er kann jedoch als Referenz benutzt werden, um die Leistung anderer Algorithmen zu messen.

Der NRU-Algorithmus teilt die Seiten anhand ihrer *R*- und *M*-Bits in vier Klassen ein und wählt zufällig eine Seite aus der niedrigsten nicht leeren Klasse. NRU ist sehr einfach zu implementieren, aber auch sehr primitiv. Es gibt bessere Algorithmen.

Der FIFO-Algorithmus merkt sich die Reihenfolge, in der die Seiten in den Speicher geladen wurden, indem er sie in eine verkettete Liste einträgt. Die älteste Seite auszuwählen, ist trivial, aber die Seite könnte noch gebraucht werden, obwohl sie schon am längsten im Speicher ist. Deshalb ist FIFO eine schlechte Wahl.

Second Chance ist eine modifizierte Version von FIFO. Er überprüft, ob eine Seite benutzt wird, bevor er sie aus dem Speicher entfernt. Diese Veränderung verbessert die Leistung enorm. Der Clock-Algorithmus ist eine andere Implementierung von Second Chance. Er hat dieselbe Leistung, es dauert aber nicht so lange, den Algorithmus auszuführen.

LRU ist ein hervorragender Algorithmus, der aber nicht ohne spezielle Hardware auskommt. NFU ist der Versuch einer groben Annäherung an LRU und kein sehr guter Algorithmus. Aging stellt dagegen eine wesentlich bessere Annäherung an LRU dar und kann effizient implementiert werden. Aging ist eine gute Wahl.

Die letzten beiden Algorithmen benutzen den Arbeitsbereich eines Prozesses. Der Working-Set-Algorithmus bietet anständige Leistung, ist aber recht aufwändig zu implementieren. WSClock ist eine Variante, die gute Leistung und effiziente Implementierung vereint.

Alles in allem sind die beiden besten Algorithmen Aging und WSClock. Sie basieren jeweils auf LRU und dem Arbeitsbereich eines Prozesses. Beide zeigen gute Leistung bei der Seitenersetzung und lassen sich effizient implementieren. Es gibt noch einige andere Algorithmen, aber diese beiden sind in der Praxis wohl die wichtigsten.

3.5 Entwurfskriterien für Paging-Systeme

In den letzten Abschnitten haben wir erklärt, wie Paging funktioniert und einige grundlegende Seitenersetzungsalgorithmen vorgestellt. Das alles reicht aber noch nicht aus, um ein gut funktionierendes System zu entwerfen, genauso wie man noch lange kein guter Schachspieler ist, wenn man weiß, wie der Turm, der Läufer und die anderen Figuren ziehen dürfen. In den folgenden Abschnitten behandeln wir Themen, über die man sich als Betriebssystementwickler Gedanken machen muss, wenn man ein leistungsstarkes Paging-System erstellen will.

3.5.1 Lokale versus globale Zuteilungsstrategien

In den letzten Abschnitten haben wir mehrere Algorithmen vorgestellt, die bei einem Seitenfehler eine Seite zur Auslagerung auswählen. Ein wichtiger Punkt bei dieser Entscheidung (den wir bisher sorgfältig unter den Teppich gekehrt haben) ist, wie der Speicher zwischen den konkurrierenden lauffähigen Prozessen aufgeteilt werden soll.

Sehen wir uns ▶Abbildung 3.23(a) an. In dieser Abbildung sind drei Prozesse lauffähig, *A*, *B* und *C*. Wenn *A* einen Seitenfehler erzeugt, sollte der Seitenersetzungsalgorithmus bei der Suche nach der am wenigsten benutzten Seite nur die sechs Seiten in Betracht ziehen, die momentan *A* zugeteilt sind, oder sollte er alle Seiten im Speicher durchsuchen? Wenn er nur die Seiten von *A* betrachtet, ist die älteste Seite *A5* und es entsteht die Situation in ▶Abbildung 3.23(b).

Wenn der Algorithmus dagegen die älteste Seite im Speicher sucht, egal welchem Prozess sie gehört, wählt er die Seite *B3* und es entsteht die Situation in ▶Abbildung 3.23(c). Die erste der beiden Strategien ist eine sogenannte **lokale** Paging-Strategie, die zweite ist eine **globale** Strategie. Eine lokale Strategie führt dazu, dass jedem Prozess ein fester Speicherbereich zugeteilt wird. Globale Strategien verteilen die Seitenrahmen dynamisch unter den lauffähigen Prozessen und die Anzahl der Seitenrahmen eines Prozesses ist veränderlich.

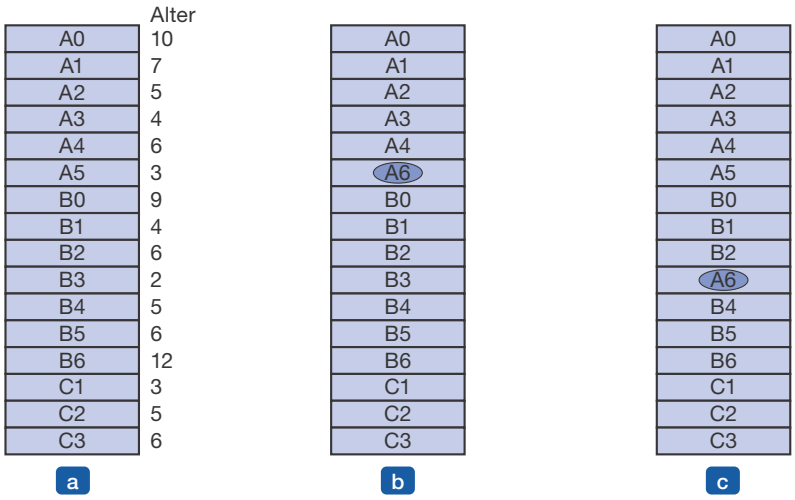

Abbildung 3.23: Lokale gegenüber globaler Seitenersetzung (a) Ausgangszustand (b) Lokale Seitenersetzung (c) Globale Seitenersetzung

Im Allgemeinen funktionieren globale Strategien besser, besonders wenn die Größe des Arbeitsbereiches eines Prozesses sich mit der Zeit ändert. Wenn eine lokale Strategie benutzt wird und der Arbeitsbereich eines Prozesses wächst, erzeugt der Prozess ständig Seitenfehler (Thrashing), obwohl genügend Seitenrahmen frei wären. Wenn der Arbeitsbereich schrumpft, verschwendet eine lokale Strategie Speicher. Bei einer globalen Strategie muss das Betriebssystem ständig entscheiden, wie viele Seitenrahmen es einem Prozess zuteilen soll. Eine Möglichkeit ist, die Größe des Arbeitsbereiches mithilfe der Aging-Bits zu überwachen, aber dadurch wird Thrashing nicht unbedingt verhindert. Der Arbeitsbereich kann sich innerhalb von wenigen Mikrosekunden ändern und die Aging-Bits sind nur ein grobes Maß für die Speichernutzung über mehrere Timerintervalle hinweg.

Ein anderer Ansatz ist die Verwendung eines eigenen Algorithmus für die Zuteilung von Seitenrahmen zu Prozessen. Im Beispiel könnte man periodisch die Anzahl der laufenden Prozesse bestimmen und dann jedem Prozess gleich viel Speicher zuteilen. Bei zehn laufenden Prozessen und 12.416 verfügbaren Seitenrahmen (d.h. Seitenrahmen, die nicht vom System belegt sind), würde jeder Prozess dann 1.241 Seitenrahmen bekommen. Die übrigen sechs werden gemeinsam genutzt, wenn Seitenfehler auftreten.

Diese Methode erscheint zwar fair, es hat aber wenig Sinn, einem 10-KB-Prozess und einem 300-KB-Prozess gleich viel Speicher zu geben. Stattdessen könnten Seiten im Verhältnis zur Gesamtgröße eines Prozesses zugeteilt werden. Der 300-KB-Prozess würde dann 30-mal soviel Speicher bekommen wie der 10-KB-Prozess. Wahrscheinlich ist es eine gute Idee, jedem Prozess ein gewisses Minimum an Seiten zuzuteilen, so dass auch sehr kleine Prozesse laufen können. Auf manchen Maschinen kann z.B. ein einziger Befehl auf bis zu sechs Seiten zugreifen, weil sowohl der Befehl als auch der Quell- und Zieloperand über Seitengrenzen hinweg verlaufen können. Wenn dem Prozess nur maximal fünf Seiten zugeteilt werden, kann so ein Befehl nicht ausgeführt werden.

Bei einer globalen Strategie ist es vielleicht möglich, jeden Prozess mit einer bestimmten Anzahl von Seiten proportional zu seiner Größe zu starten und die Seitenzuteilung dynamisch anzupassen, während der Prozess läuft. Eine Möglichkeit zur Überwachung der Speicherzuteilung ist der **PFF-Algorithmus** (**Seitenfehlerrate,** *Page Fault Frequency*), der entscheidet, ob dem Prozess mehr oder weniger Speicher zugeteilt werden muss, aber nicht, welche Seite ausgelagert werden soll. Der Algorithmus dient nur dazu, die Größe des zugeteilten Speichers zu kontrollieren.

Wie bereits erwähnt, ist es für eine große Klasse von Seitenersetzungsalgorithmen, einschließlich LRU, bekannt, dass die Seitenfehlerrate abnimmt, wenn mehr Speicher zur Verfügung steht. Dies ist der Grundgedanke hinter PFF. ▶Abbildung 3.24 illustriert diese Eigenschaft.

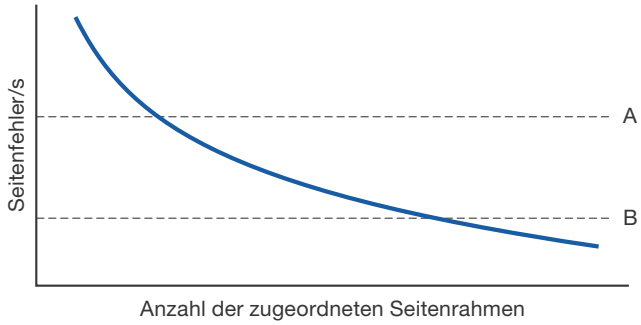

Abbildung 3.24: Die Seitenfehlerrate in Abhängigkeit von der Anzahl der zugewiesenen Seitenrahmen

Die Seitenfehlerrate ist einfach zu messen: Man muss nur die Anzahl der Fehler pro Sekunde zählen und eventuell noch einen Mittelwert über den Messungen der letzten Sekunden bilden. Eine einfache Möglichkeit dafür ist, die Anzahl der Seitenfehler während der unmittelbar vorhergehenden Sekunde zu dem aktuellen Mittelwert zu addieren und das Ergebnis durch zwei zu teilen. Die gestrichelte Linie *A* markiert eine zu hohe Seitenfehlerrate, also muss dem Prozess mehr Speicher zugeteilt werden, um die Fehlerrate zu reduzieren. Die Linie *B* markiert eine Fehlerrate, die so niedrig ist, dass man annehmen kann, dass der Prozess zu viel Speicher hat. In diesem Fall können dem Prozess Seitenrahmen entzogen werden. PFF versucht also, die Seitenfehlerrate aller Prozesse innerhalb akzeptabler Grenzen zu halten.

Es ist wichtig zu beachten, dass manche Seitenersetzungsalgorithmen sowohl mit einer lokalen als auch mit einer globalen Paging-Strategie arbeiten können. Beispielsweise kann FIFO die älteste Seite im gesamten Speicher (global) oder die älteste Seite des aktuellen Prozesses (lokal) ersetzen. Analog kann LRU oder eine Annäherung die kürzlich am wenigsten benutzte Seite im gesamten Speicher (global) oder die am wenigsten genutzte Seite des aktuellen Prozesses (lokal) auslagern. In manchen Fällen ist die Entscheidung zwischen lokaler und globaler Strategie also unabhängig vom Seitenersetzungsalgorithmus.

Für andere Algorithmen ist dagegen nur eine lokale Strategie sinnvoll. Insbesondere beziehen sich die Working-Set- und WSClock-Algorithmen auf den Arbeitsbereich

eines speziellen Prozesses und müssen in diesem Zusammenhang angewendet werden. Es gibt keinen Arbeitsbereich für die gesamte Maschine und die Vereinigung aller Arbeitsbereiche zu benutzen, würde die Lokalität der Referenzen zerstören und nicht gut funktionieren.

3.5.2 Lastkontrolle

Selbst mit dem besten Seitenersetzungsalgorithmus und der optimalen globalen Speicherzuteilung kommt es vor, dass das System zu viele Seitenfehler erzeugt. Wenn die Arbeitsbereiche aller Prozesse größer sind als der verfügbare Speicher, ist das sogar zu erwarten. Ein Symptom für diese Situation ist, dass der PFF-Algorithmus anzeigt, dass mehrere Prozesse zu wenig Speicher haben, aber keiner zu viel hat. In diesem Fall gibt es keine Möglichkeit, einem Prozess mehr Speicher zu geben, ohne einem anderen Prozess zu schaden. Die einzige Lösung ist, einige Prozesse zeitweise ganz aus dem Speicher zu entfernen.

Ein gutes Mittel, um die Anzahl der Prozesse zu reduzieren, die sich um den Speicher streiten, besteht darin, einige von ihnen auf die Festplatte auszulagern und alle ihre Seiten freizugeben. Zum Beispiel könnte man einen Prozess auslagern und seinen Speicher zwischen den Prozessen aufteilen, die zu viele Seitenfehler erzeugen. Wenn die Fehlerrate genügend sinkt, kann das System so eine Zeit lang weiterlaufen. Wenn sie nicht sinkt, wird noch ein Prozess ausgelagert und so weiter, bis sich die Situation verbessert. Swapping wird also auch in einem Paging-System gebraucht, nur dient es jetzt dazu, die möglichen Speicheranforderungen zu reduzieren und nicht mehr dazu, Seiten freizumachen.

Ganze Prozesse zur Reduzierung der Speicherlast auszulagern, erinnert an das zweistufige Scheduling, bei dem einige Prozesse auf die Festplatte ausgelagert werden und ein zweiter Scheduler die CPU-Zeit unter den übrigen Prozessen aufteilt. Offensichtlich können diese zwei Ideen kombiniert werden, so dass gerade genug Prozesse ausgelagert werden, um die Seitenfehlerrate auf ein akzeptables Maß zu senken. In regelmäßigen Abständen können die Prozesse dann wieder eingelagert und andere ausgelagert werden.

Man sollte dabei jedoch nicht den Grad der Multiprogrammierung vergessen. Die CPU könnte längere Zeit im Leerlauf sein, wenn zu wenige Prozesse im Speicher sind. Aus diesem Gedankengang ergibt sich der Vorschlag, bei der Entscheidung, welchen Prozess man auslagert, nicht nur dessen Größe und Fehlerrate in Betracht zu ziehen, sondern auch andere Kriterien wie z.B. ob er CPU-intensiv oder E/A-intensiv ist sowie die Eigenschaften der übrigen Prozesse.

3.5.3 Seitengröße

Die Seitengröße ist oft ein Parameter, den das Betriebssystem wählen kann. Auch wenn die Hardware beispielsweise für 512-Byte-Seiten entworfen wurde, kann das Betriebssystem leicht die Seitenrahmen 0 und 1, 2 und 3, 4 und 5 usw. als 1-KB-Seiten behandeln, indem es einer Seite immer zwei aufeinanderfolgende Seitenrahmen zuordnet.

Bei der Wahl der besten Seitengröße muss man zwischen mehreren gegenläufigen Faktoren abwägen. Ein allgemeines Optimum gibt es deshalb nicht. Es gibt zwei Faktoren, die für eine kleine Seitengröße sprechen. Erstens wird ein willkürlich gewähltes Text-, Daten- oder Stacksegment keine ganze Zahl von Seiten füllen. Im Durchschnitt bleibt immer die Hälfte der letzten Seite leer und der übrige Speicherplatz wird verschwendet. Dieser Verschnitt heißt **interne Fragmentierung** (*internal fragmentation*). Für n Segmente im Speicher und eine Seitengröße von p Byte, werden $np/2$ Byte für interne Fragmentierung verschwendet. Dies ist ein Argument für kleine Seiten.

Das zweite Argument für kleine Seiten wird klar, wenn man sich ein Programm vorstellt, das aus acht aufeinanderfolgenden Phasen besteht, die jeweils 4 KB groß sind. Bei einer Seitengröße von 32 KB belegt das Programm die ganze Zeit 32 KB. Mit 16-KB-Seiten belegt es nur 16 KB. Bei einer Seitengröße von 4 KB oder weniger belegt es zu jedem Zeitpunkt nur 4 KB. Allgemein führen große Seiten dazu, dass mehr Speicher verschwendet wird.

Auf der anderen Seite bedeuten kleine Seiten, dass die Programme viele Seiten belegen, was zu größeren Seitentabellen führt. Ein 32-KB-Programm braucht nur vier 8-KB-Seiten, aber 64 Seiten mit der Größe von 512 Byte. Zwischen Festplatte und Speicher werden normalerweise ganze Seiten übertragen, wobei die meiste Zeit für Suchzeit und Rotationsverzögerung verbraucht wird, so dass es fast genauso lang dauert, eine kleine Seite zu übertragen wie eine große. Es könnte z. B. 64×10 ms dauern, 64.512-Byte-Seiten zu laden, aber nur 4×12 ms für vier 8-KB-Seiten.

Auf manchen Rechnern muss die Seitentabelle bei jedem Prozesswechsel in einen Satz Hardwareregister geladen werden. Je kleiner dann die Seiten werden, desto länger dauert es, die Seitenregister zu laden. Außerdem belegt die Seitentabelle für kleinere Seiten mehr Platz.

Diesen letzten Punkt kann man mathematisch analysieren. Die durchschnittliche Prozessgröße sei s Byte und die Seitengröße sei p Byte. Außerdem sei jeder Seitentabelleneintrag e Byte groß. Jeder Prozess belegt dann ungefähr s/p Seiten und damit se/p Byte in der Seitentabelle. Durch die interne Fragmentierung gehen außerdem noch einmal $p/2$ Byte verloren. Der gesamte Speicherverbrauch durch die Seitentabelle und interne Fragmentierung beträgt also

$$\text{Verbrauch} = se/p + p/2$$

Der erste Term (die Seitentabelle) ist groß, wenn die Seitengröße klein ist. Der zweite Term (interne Fragmentierung) ist groß, wenn auch die Seiten groß sind. Das Optimum muss also irgendwo in der Mitte liegen. Wenn wir die erste Ableitung nach p gleich null setzen, erhalten wir

$$-se/p^2 + 1/2 = 0$$

Aus dieser Gleichung können wir eine Formel ableiten, die die optimale Seitengröße angibt, wobei nur die Größe der Seitentabelle und die interne Fragmentierung berücksichtigt werden. Das Ergebnis ist:

$$p = \sqrt{2se}$$

Für $s = 1$ MB und $e = 8$ Byte pro Tabelleneintrag ist die optimale Seitengröße 4 KB. In kommerziellen Computern wurden Seitengrößen zwischen 512 Byte und 64 KB verwendet. Früher war 1 KB typisch, heute sind es eher 4 KB oder 8 KB. Die Seitengröße neigt dazu, mit dem Speicher zu wachsen, aber nicht linear. Wenn sich der Speicher vervierfacht, wird die Seitengröße meistens nicht einmal verdoppelt.

3.5.4 Trennung von Befehls- und Datenräumen

Die meisten Computer haben einen einzigen Adressraum, der sowohl die Programme als auch die Daten enthält (▶Abbildung 3.25(a)). Wenn dieser Adressraum groß genug ist, funktioniert alles hervorragend. Leider ist er aber oft zu klein, was die Programmierer zu extremen Verrenkungen zwingt, um alles hineinzubekommen.

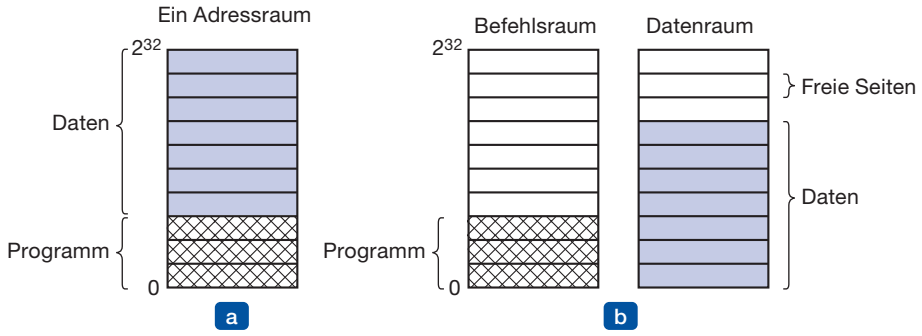

Abbildung 3.25: (a) Ein Adressraum (b) Getrennte Adressräume für Code und Daten

Eine Lösung, die zuerst auf der PDP-11, einem 16-Bit-Rechner, eingesetzt wurde, sind getrennte Adressräume für Befehle (Programmtext) und Daten. Die beiden Adressräume werden **I-Space** oder **Befehlsraum** und **D-Space** oder **Datenraum** genannt (siehe ▶Abbildung 3.25(b)). Die Adressen in jedem Raum reichen von 0 bis zu einem bestimmten Maximum, normalerweise $2^{16} - 1$ oder $2^{32} - 1$. Der Binder muss wissen, ob getrennte Adressräume verwendet werden, weil die Daten dann an die virtuelle Adresse 0 anstatt an die Adresse hinter dem Programmtext reloziert werden.

Bei einem Computer mit diesem Design können die Paging-Mechanismen der beiden Adressräume voneinander unabhängig sein. Jeder hat seine eigene Seitentabelle und seine eigene Abbildung von virtuellen auf physische Adressen. Wenn die Hardware einen Befehl holt, weiß sie, dass sie den Befehlsraum und dessen Seitentabelle benutzen muss. Ebenso laufen Zugriffe auf Daten über die Seitentabelle für den Datenraum. Außer dieser Unterscheidung entstehen durch getrennte Adressräume keine Komplikationen und der verfügbare Adressraum wird verdoppelt.

3.5.5 Gemeinsame Seiten

Ein weiterer Punkt beim Entwurf eines Paging-Systems sind **gemeinsame Seiten** (*shared page*). In großen Mehrbenutzersystemen kommt es häufig vor, dass mehrere Benutzer zur selben Zeit das gleiche Programm benutzen. Offensichtlich ist es effizienter, die Seiten gemeinsam zu benutzen, als gleichzeitig zwei Kopien derselben Seite im Speicher zu halten. Ein Problem dabei ist, dass nicht alle Seiten gemeinsam benutzt werden können. Seiten, die nur gelesen werden, z.B. Programmtext, können gemeinsam benutzt werden, aber Seiten, die Daten enthalten, nicht.

Wenn das System getrennte Adressräume für Programmtext und Daten unterstützt, besteht eine unkomplizierte Methode darin, für zwei oder mehr Prozesse dieselbe Seitentabelle für den Befehlsraum und getrennte Seitentabellen für den Datenraum zu verwenden. In einem System, das gemeinsame Seiten auf diese Art unterstützt, sind die Seitentabellen normalerweise von der Prozesstabelle unabhängig. Jeder Prozess hat dann zwei Zeiger in seinem Prozesstabelleneintrag: einen auf die Seitentabelle für den Befehlsraum und einen auf die Tabelle für den Datenraum, wie in ▶Abbildung 3.26 dargestellt. Wenn der Scheduler einen Prozess zur Ausführung bringt, findet er über diese Zeiger die richtigen Seitentabellen und sorgt dafür, dass die MMU sie benutzt. Es ist auch ohne getrennte Adressräume möglich, dass Prozesse Programmtext (oder Bibliotheken) gemeinsam benutzen, aber der Mechanismus ist dann wesentlich komplizierter.

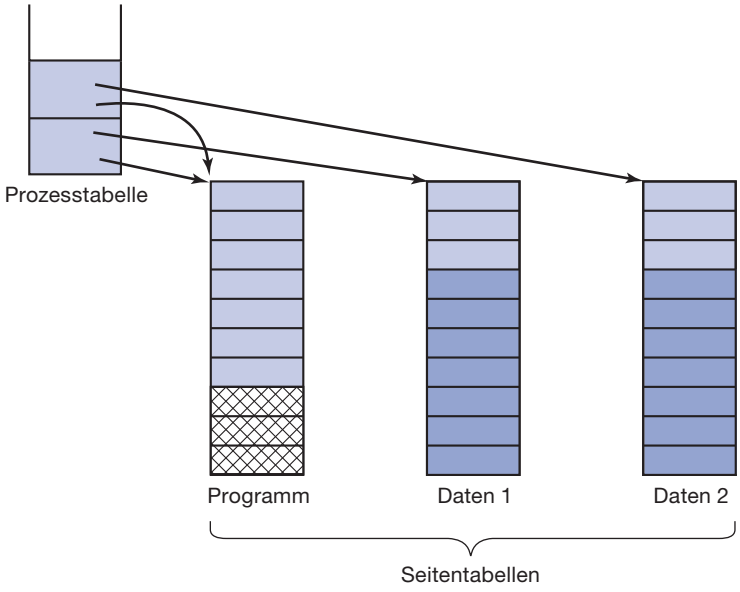

Abbildung 3.26: Zwei Prozesse benutzen ein Programm gemeinsam, indem sie auf dieselbe Seitentabelle zugreifen.

Wenn zwei oder mehr Prozesse gemeinsam Programmcode benutzen wollen, entsteht ein Problem mit den gemeinsamen Seiten. Nehmen wir an, dass die Prozesse *A* und *B* beide denselben Editor laufen lassen und seine Seiten gemeinsam benutzen. Wenn der Scheduler *A* aus dem Speicher entfernt, dabei alle seine Seiten auslagert und die lee-

ren Seitenrahmen für andere Prozesse verwendet, wird *B* eine Menge Seitenfehler erzeugen, um die Seiten in den Speicher zurückzubringen.

Auch wenn *A* terminiert, ist es wichtig, dass der Scheduler mitbekommt, dass die Seiten noch gebraucht werden, damit sie nicht aus Versehen von der Festplatte entfernt werden. Die ganze Seitentabelle zu durchsuchen, um herauszufinden, ob eine Seite noch von einem anderen Prozess benutzt wird, ist normalerweise zu aufwändig; deshalb werden spezielle Datenstrukturen gebraucht, um die gemeinsamen Seiten zu verwalten, besonders wenn gemeinsamer Speicher in Einheiten von einer Seite (oder zusammenhängenden Blöcken von Seiten) vergeben wird, anstatt die gesamte Seitentabelle gemeinsam zu verwenden.

Die gemeinsame Benutzung von Daten ist nicht unmöglich, aber sie ist komplizierter als bei Programmcode. Nach einem `fork`-Systemaufruf unter UNIX müssen der Vater- und der Kindprozess sowohl gemeinsamen Code als auch gemeinsame Daten haben. In einem Paging-System bekommt oft jeder dieser Prozesse eine eigene Seitentabelle, die beide auf dieselben Seitenrahmen zeigen. So werden durch den `fork`-Aufruf keine Seiten kopiert. In beiden Tabellen sind aber alle Datenseiten als READ ONLY markiert.

Solange beide Prozesse nur lesen, ist alles in Ordnung. Sobald aber einer der Prozesse ein Wort in den Speicher schreibt, wird durch die Verletzung der Zugriffsrechte ein Systemaufruf ausgelöst. Die entsprechende Seite wird dann kopiert, so dass jetzt jeder Prozess seine eigene Kopie hat. Beide Kopien werden auf READ/WRITE gesetzt, so dass spätere Schreibzugriffe keine Schutzfehler mehr auslösen. Diese Strategie läuft darauf hinaus, dass Seiten, die niemals modifiziert werden (darunter alle Seiten, die Code enthalten), auch nicht kopiert werden müssen. Nur die Datenseiten, auf die geschrieben wird, werden kopiert. Diese Methode heißt **Copy-on-Write** und erhöht die Leistung, weil weniger Seiten kopiert werden müssen.

3.5.6 Gemeinsame Bibliotheken

Die gemeinsame Nutzung kann auch in anderen Einheiten als einzelnen Seiten erfolgen. Wenn ein Programm zweimal aufgerufen wird, dann werden die meisten Betriebssysteme automatisch alle Textseiten zur gemeinsamen Nutzung vorsehen, damit nur eine einzige Kopie im Speicher ist. Da Textseiten immer nur gelesen werden können, gibt es hier keine Probleme. Je nach Betriebssystem kann jeder Prozess entweder seine eigene private Kopie der Datenseiten haben oder sie werden bei gemeinsamer Nutzung als nur zum Lesen markiert. Wenn ein Prozess eine Datenseite verändert, dann wird eine private Kopie für den Prozess angelegt, d. h., das Copy-on-Write-Verfahren kommt zum Einsatz.

In modernen Systemen gibt es viele große Bibliotheken, die von zahlreichen Prozessen benutzt werden, zum Beispiel die Bibliothek, die für den Dialog zum Durchsuchen nach zu öffnenden Dateien zuständig ist, und viele Grafikbibliotheken. Wenn all diese Bibliotheken statisch an jedes ausführbare Programm auf der Festplatte gebunden würden, dann wären sie noch aufgeblähter, als sie es ohnehin schon sind.

Stattdessen ist es eine üblich Technik, **gemeinsame Bibliotheken** (*shared library*) zu benutzen (die unter Windows **DLL** oder **Dynamic Link Library** heißen). Um die Idee einer gemeinsamen Bibliothek noch klarer zu machen, betrachten wir zuerst das traditionelle Binden. Wenn ein Programm gebunden wird, dann werden ein oder mehrere Objektdateien und eventuell einige Bibliotheken in dem Befehl zum Binden aufgezählt, wie zum Beispiel bei dem folgenden UNIX-Kommando:

```
ld *.o -lc -lm
```

Der Befehl bewirkt, dass alle *.o*-Dateien (Objektdateien) in dem aktuellen Verzeichnis gebunden werden und dann zwei Bibliotheken durchsucht werden, */usr/lib/libc.a* und */usr/lib/libm.a*. Jede Funktion, die zwar in den Objektdateien aufgerufen wird, sich dort aber nicht befindet (z.B. *printf*), wird **undefinierte externe Funktion** (*undefined external*) genannt und in den Bibliotheken gesucht. Wenn sie gefunden wird, wird sie in den ausführbaren Code eingefügt. Ebenso wird jede Funktion, die innerhalb der soeben eingebundenen Funktion aufgerufen wird und noch nicht vorhanden ist, zu einer undefinierten externen Funktion. Zum Beispiel braucht *printf* die Funktion *write*, also muss der Binder eventuell nach *write* suchen und es einbinden, sobald er es gefunden hat. Wenn der Bindevorgang beendet ist, dann wird eine ausführbare Binärdatei auf die Festplatte zurückgeschrieben, die alle benötigten Funktionen enthält. Funktionen, die in den Bibliotheken vorkommen, aber nicht aufgerufen werden, werden nicht eingebunden. Wenn das Programm schließlich in den Speicher geladen und ausgeführt wird, sind alle benötigten Funktionen da.

Nehmen wir an, ein normales Programm benutzt 20–50 MB an Grafik- und Benutzungsschnittstellenfunktionen. Wenn man Hunderte von solchen Programmen mit allen benutzten Bibliotheken statisch binden wollte, würde dies eine ungeheure Menge an Platz sowohl auf der Platte als auch im RAM verschwenden. Das System hat keine Möglichkeit herauszufinden, ob teilweise auch eine gemeinsame Nutzung infrage käme. An diesem Punkt kommen die gemeinsamen Bibliotheken ins Spiel. Wenn ein Programm mit gemeinsamen Bibliotheken verbunden ist (die sich leicht von den statisch gebundenen Bibliotheken unterscheiden), dann schließt der Binder statt der aktuellen Funktionsaufrufe eine kleine Stub-Routine ein, die an die aufgerufene Funktion zur Laufzeit gebunden wird. Abhängig vom System und den Konfigurationsdetails werden gemeinsame Bibliotheken entweder zur gleichen Zeit wie das Programm oder beim ersten Aufruf einer ihrer Funktionen geladen. Wenn ein anderes Programm die gemeinsame Bibliothek bereits geladen hat, dann besteht selbstverständlich keine Veranlassung, diese noch einmal zu laden – das ist der springende Punkt. Beim Laden oder bei der Nutzung einer gemeinsamen Bibliothek wird nicht die gesamte Bibliothek auf einmal in den Speicher eingelesen, sondern sie wird bei Bedarf Seite für Seite eingelagert. Damit liegen keine Funktionen im RAM, die aktuell nicht aufgerufen werden.

Gemeinsame Bibliotheken helfen also, ausführbare Dateien klein zu halten und Platz im Speicher zu sparen. Darüber hinaus haben sie noch einen weiteren Vorteil: Wenn eine Funktion innerhalb einer gemeinsamen Bibliothek verändert wird, um beispielsweise einen Fehler zu beheben, ist es nicht nötig, alle Programme, die diese Funktion auf-

rufen, neu zu übersetzen. Die alten binären Dateien können weiterhin benutzt werden. Dieses Merkmal ist besonders für kommerzielle Software wichtig, bei der der Quellcode dem Kunden nicht ausgehändigt wird. Falls zum Beispiel Microsoft eine Sicherheitslücke in einer Standard-DLL findet und ausbessert, dann lädt *Windows Update* die neue DLL und ersetzt damit die alte DLL. Alle Programme, die die DLL benutzen, werden bei ihrer nächsten Ausführung automatisch die neue Version benutzen.

Gemeinsame Bibliotheken bringen allerdings ein kleines Problem mit sich, das einer Lösung bedarf. Das Problem ist in ▶Abbildung 3.27 dargestellt. Wir sehen hier zwei Prozesse, die eine Bibliothek der Größe 20 KB gemeinsam benutzen (d.h., jedes Feld stellt 4 KB dar). Die Bibliothek liegt in beiden Prozessen jeweils an einer anderen Stelle, da die Programme selbst vermutlich nicht die gleiche Größe haben. In Prozess 1 beginnt die Bibliothek an der Adresse 36 KB; in Prozess 2 beginnt sie bei 12 KB. Angenommen, die erste Aktion der ersten Bibliotheksfunktion ist ein Sprung zu Adresse 16 in der Bibliothek. Wenn die Bibliothek nicht gemeinsam genutzt wird, könnte die Zieladresse zum Zeitpunkt des Ladens reloziert werden, so dass (in Prozess 1) zur virtuellen Adresse 36 KB + 16 gesprungen werden könnte. Beachten Sie, dass die physische Adresse im RAM, wo die Bibliothek liegt, nicht interessiert, da die Zuordnung der Seiten von der virtuellen zur physischen Adresse von der MMU-Hardware vorgenommen wird.

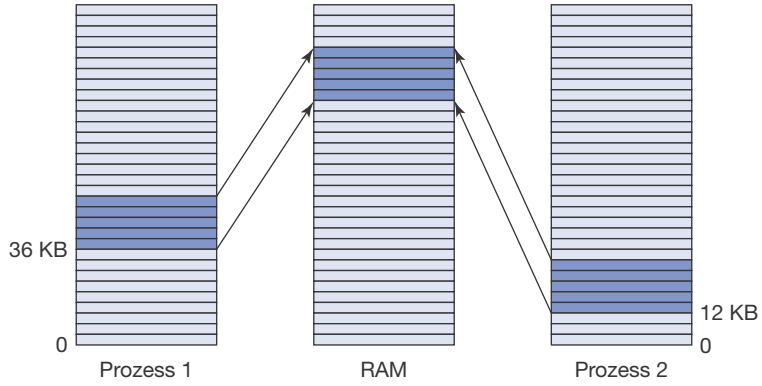

Abbildung 3.27: Eine gemeinsame Bibliothek, die von zwei Prozessen benutzt wird

Da die Bibliothek jedoch gemeinsam genutzt wird, ist diese Art der Relokation nicht möglich. Denn wenn die erste Bibliotheksfunktion von Prozess 2 aufgerufen wird (dessen Startadresse 12 KB ist), dann muss der Sprungbefehl zu 12 KB + 16 und nicht zu 36 KB + 16 führen. Genau hier liegt das kleine Problem. Eine mögliche Lösung wäre, das Copy-on-Write-Verfahren einzusetzen und für jeden Prozess, der die Bibliothek mitbenutzt, neue Seiten zu erzeugen, die dann zum Zeitpunkt ihrer Erzeugung reloziert werden. Leider durchkreuzt dieses Modell den Zweck der gemeinsamen Bibliotheken.

Eine bessere Lösung besteht darin, die gemeinsamen Bibliotheken mit einem speziellen Compiler-Flag zu übersetzen, durch das der Compiler angewiesen wird, keine Befehle mit absoluten Adressen zu erzeugen. Stattdessen werden nur Befehle mit rela-

tiven Adressen benutzt. Zum Beispiel gibt es fast immer einen Befehl wie „Springe *n* Byte vor“ (oder zurück) – im Gegensatz zu einem Befehl, der eine spezielle Sprungadresse vorgibt. Diese Befehle funktionieren korrekt, unabhängig davon, wo die gemeinsame Bibliothek im virtuellen Adressraum platziert ist. Durch das Vermeiden von absoluten Adressen kann das Problem also gelöst werden. Ein Code, der nur relative Offsets benutzt, heißt **positionsunabhängig** (*position independent* oder auch PIC).

3.5.7 Memory-Mapped-Dateien

Gemeinsame Bibliotheken sind eigentlich ein Spezialfall einer allgemeineren Einrichtung, den **Memory-Mapped-Dateien**. Die Idee hierbei ist, dass ein Prozess einen Systemaufruf ausgeben kann, um eine Datei in einen Teilbereich seines virtuellen Adressraumes einzublenden. In den meisten Implementierungen werden die Seiten nicht zum Zeitpunkt der Einblendung in den Speicher geholt, sondern auf Anforderung eine nach der anderen eingelagert. Die jeweilige Plattendatei wird dabei als Hintergrundspeicher benutzt. Wenn der Prozess beendet ist oder explizit die Datei wieder ausblendet, werden alle veränderten Seiten zurück in die Datei geschrieben.

Memory-Mapped-Dateien bieten ein alternatives Modell zur Ein-/Ausgabe. Anstatt die Lese- und Schreibvorgänge auszuführen, kann auf die Datei wie auf ein großes Zeichenfeld im Speicher zugegriffen werden. Programmierer finden dieses Modell für manche Situationen geeigneter.

Falls zwei oder mehr Prozesse zur gleichen Zeit dieselbe Datei einblenden, können sie über gemeinsam genutzten Speicher miteinander kommunizieren. Schreibvorgänge von einem Prozess sind unmittelbar sichtbar, wenn der andere von dem Teil seines virtuellen Adressraumes liest, in den die Datei eingeblendet ist. Dieser Mechanismus stellt somit einen Kanal mit hoher Bandbreite zwischen Prozessen zur Verfügung und wird oft als solcher benutzt (sogar bis zum Einblenden einer Scratch-Datei). Wenn also der Mechanismus des Einblendens von Dateien in den Speicher verfügbar ist, kann dieser von den gemeinsamen Bibliotheken genutzt werden.

3.5.8 Bereinigungsstrategien

Paging funktioniert am besten, wenn jederzeit genügend Seitenrahmen frei sind, die bei Seitenfehlern angefordert werden können. Wenn alle Seitenrahmen voll sind und ihr Inhalt im Speicher verändert wurde, muss die alte Seite auf die Platte zurückgeschrieben werden, bevor eine neue Seite eingelagert werden kann. Um jederzeit die Versorgung mit freien Seitenrahmen sicherzustellen, haben viele Paging-Systeme einen Hintergrundprozess, einen sogenannten **Paging-Daemon**, der die meiste Zeit schläft, aber in regelmäßigen Abständen aufwacht und den Zustand des Speichers überprüft. Wenn nicht genug Seitenrahmen frei sind, wählt er mit dem Seitenersetzungsalgorithmus Seiten aus, die ausgelagert werden. Seiten, die im Speicher verändert wurden, werden dabei auf die Platte zurückgeschrieben.

Auf jeden Fall bleibt der Inhalt der Seite vorerst im Speicher. So kann die Seite, falls sie gebraucht wird, bevor sie überschrieben ist, aus der Menge der freien Seitenrahmen zurückgeholt werden, ohne den Inhalt neu von der Platte zu lesen. Bei einem Seitenfehler immer freie Seitenrahmen verfügbar zu halten anstatt den Speicher jedes Mal nach einem passenden Seitenrahmen zu durchsuchen, erhöht die Leistung. Zumindest sorgt der Paging-Daemon dafür, dass alle freien Seitenrahmen sauber sind, so dass sie nicht panisch auf die Platte geschrieben werden müssen, wenn sie gebraucht werden.

Eine Möglichkeit, diese Bereinigungsstrategie zu implementieren, benutzt eine Uhr mit zwei Zeigern. Der erste Zeiger wird vom Paging-Daemon kontrolliert. Wenn er auf eine saubere Seite zeigt, rückt der Zeiger auf die nächste Seite vor. Wenn er auf eine modifizierte Seite zeigt, wird sie auf die Platte geschrieben und der Zeiger rückt auch eine Seite vor. Der zweite Zeiger wird wie beim normalen Clock-Algorithmus für die Seitenersetzung verwendet, allerdings mit dem Unterschied, dass die Wahrscheinlichkeit, auf eine saubere Seite zu treffen, durch den Paging-Daemon erhöht wurde.

3.5.9 Schnittstelle des virtuellen Speichersystems

Bis jetzt haben wir angenommen, dass der virtuelle Speicher für Prozesse und Programmierer transparent ist, d.h., sie sehen nur einen großen virtuellen Adressraum auf einem Computer mit einem klein(er)en physischen Speicher. Für viele Systeme stimmt das, aber in einigen fortgeschrittenen Systemen hat der Programmierer etwas Kontrolle über die Speicherabbildung und kann das Verhalten seiner Programme auf unkonventionelle Weise verbessern. In diesem Abschnitt sehen wir uns einige dieser Möglichkeiten an.

Einer der Gründe dafür, dem Programmierer die Kontrolle über die Speicherabbildung zu geben, ist, es zu ermöglichen, dass zwei oder mehr Prozesse denselben Speicher verwenden. Wenn ein Programmierer einem Speicherbereich einen Namen geben kann, könnte ein Prozess den Namen an einen anderen Prozess weitergeben, so dass dieser den Speicherbereich in seinen Adressraum einblenden kann. Über den gemeinsamen Speicher könnten dann zwei (oder mehr) Prozesse mit hoher Bandbreite Daten austauschen – der eine Prozess schreibt Daten in den gemeinsamen Speicher und der andere Prozess liest sie.

Gemeinsame Seiten können auch für ein Nachrichtenaustauschsystem mit hoher Leistung benutzt werden. Bei normalem Nachrichtenaustausch werden die Daten von einem Adressraum in den anderen kopiert, was hohe Kosten verursacht. Prozesse, die die Kontrolle über ihre Speicherabbildungen haben, können Nachrichten austauschen, indem der Sender die Seiten mit der Nachricht aus seinem Adressraum ausblendet und der Empfänger sie einblendet. Dabei werden statt der Daten nur die Namen der Seiten kopiert.

Eine weitere fortgeschrittene Speicherverwaltungstechnik ist der **verteilte gemeinsame Speicher** (*distributed shared memory*) (Feeley et al., 1995; Li, 1986; Li und Hudak, 1989; Zekauskas et al., 1994). Die Grundidee besteht darin, dass mehrere Prozesse eine Menge von Seiten über ein Netzwerk gemeinsam benutzen dürfen, möglicherwei-

se in der Form eines einzigen, gemeinsamen linearen Adressraumes. Wenn ein Prozess auf eine Seite zugreift, die gerade nicht eingeblendet ist, wird ein Seitenfehler erzeugt. Die Behandlungsroutine für den Seitenfehler, der im Kern- oder im Benutzermodus laufen kann, findet die Maschine, die die Seite im Speicher hält, und fordert sie über das Netzwerk auf, die Seite aus ihrem Speicher auszublenden und über das Netz zu senden. Sobald die Seite ankommt, wird sie in den Speicher eingeblendet und der Befehl, der den Fehler ausgelöst hat, wird neu gestartet. In Kapitel 8 beschäftigen wir uns näher mit verteiltem gemeinsamen Speicher.

3.6 Implementierungsaspekte

Bei der Implementierung eines Systems mit virtuellem Speicher muss man zunächst eine Auswahl unter den verschiedenen theoretischen Algorithmen treffen, z.B. Second Chance oder Aging, lokale oder globale Paging-Strategie, Demand Paging oder Prepaging. Man muss sich aber auch der praktischen Probleme bewusst sein, die sich bei der Implementierung stellen. In diesem Abschnitt beschreiben wir ein paar der typischen Probleme und einige Lösungen.

3.6.1 Aufgaben des Betriebssystems beim Paging

Linux-Lab

Im Leben eines Prozesses gibt es vier Punkte, an denen das Betriebssystem Arbeit leisten muss, die etwas mit Paging zu tun hat: bei der Erzeugung des Prozesses, bei der Ausführung des Prozesses, bei einem Seitenfehler und bei der Terminierung des Prozesses. Wir nehmen uns nun jeden dieser Punkte vor und sehen uns an, was das Betriebssystem jeweils zu tun hat.

Wenn ein neuer Prozess erzeugt wird, muss das Betriebssystem zunächst die (anfängliche) Größe des Programmcodes und der Daten feststellen und dafür eine Seitentabelle erzeugen. Die Tabelle muss Speicher zugeteilt bekommen und initialisiert werden. Die Tabelle muss nur im Speicher liegen, wenn der Prozess läuft. Wenn er ausgelagert ist, ist dies nicht erforderlich. Außerdem muss auf der Festplatte Platz für ausgelagerte Seiten reserviert werden und der Swap-Bereich muss mit dem Programmtext und den Daten initialisiert werden, damit Seitenfehler behandelt werden können. Einige Systeme laden den Programmtext direkt aus der Programmdatei in den Speicher, um Plattenplatz und Zeit bei der Initialisierung zu sparen. Schließlich müssen noch die Informationen über die Seitentabelle und den Swap-Bereich in die Prozesstabelle eingetragen werden.

Wenn der Scheduler einen Prozess zur Ausführung bringt, muss die MMU auf den neuen Prozess eingestellt werden und der TLB geleert werden, um die Spuren des vorherigen Prozesses zu beseitigen. Die aktuelle Seitentabelle muss auf die Seitentabelle des neuen Prozesses gesetzt werden, was normalerweise geschieht, indem ein Zeiger auf die Tabelle in ein (oder mehrere) Hardwareregister geladen wird. Vielleicht werden auch einige oder alle Seiten des Prozesses eingelagert, um die anfängliche Seitenfehlerrate zu senken (z.B. ist es sicher, dass die Seite, auf die der Befehlszähler zeigt, gebraucht wird).

Bei einem Seitenfehler muss das Betriebssystem Hardwareregister auslesen, um festzustellen, welche virtuelle Adresse den Fehler erzeugt hat. Aus dieser Information muss es herleiten, welche Seite benötigt wird, und diese auf der Festplatte finden. Dann muss es für die neue Seite einen verfügbaren Seitenrahmen suchen, wenn nötig eine alte Seite auslagern und die neue Seite in den Seitenrahmen einlesen. Als Letztes muss es den Befehlszähler auf den Befehl zurücksetzen, der den Seitenfehler ausgelöst hat, damit er noch einmal ausgeführt wird.

Wenn ein Prozess terminiert, muss das Betriebssystem seine Seitentabelle, seine Seitenrahmen und den Plattenplatz für ausgelagerte Seiten freigeben. Seiten, die gemeinsam mit anderen Prozessen benutzt werden, können erst gelöscht werden, wenn der letzte Prozess, der sie benutzt, terminiert.

3.6.2 Behandlung von Seitenfehlern

Jetzt sind wir endlich in der Lage, im Detail zu beschreiben, was bei einem Seitenfehler genau passiert. Die Folge von Ereignissen ist:

1. Die Hardware schreibt den Befehlszähler auf den Stack und löst einen Sprung in den Kern aus. Auf den meisten Maschinen werden einige Informationen über den Zustand des aktuellen Befehls in speziellen CPU-Registern gespeichert.
2. Eine Assembler-Routine wird gestartet, die Mehrzweckregister und andere flüchtige Informationen speichert, damit das Betriebssystem sie nicht zerstört. Diese Routine ruft das Betriebssystem als Prozedur auf.
3. Das Betriebssystem stellt fest, dass ein Seitenfehler erzeugt wurde, und versucht herauszufinden, welche virtuelle Seite gebraucht wird. Meistens enthält eines der Hardwareregister diese Information. Wenn nicht, muss das Betriebssystem über den Befehlszähler den Befehl laden und interpretieren, um softwaremäßig herauszufinden, was den Seitenfehler ausgelöst hat.
4. Sobald die virtuelle Adresse bekannt ist, die den Fehler ausgelöst hat, überprüft das System, ob die Adresse gültig ist und ob der Zugriff erlaubt ist. Wenn nicht, schickt es dem Prozess ein Signal oder bricht ihn ab. Wenn die Adresse gültig ist und keine Schutzverletzung vorliegt, sucht das System nach einem freien Seitenrahmen. Wenn es keine freien Seitenrahmen gibt, wird der Seitenersetzungsalgorithmus gestartet, um ein Opfer auszuwählen.
5. Wenn der gewählte Seitenrahmen modifiziert wurde, gibt das System Anweisung, die Seite auf die Platte zu schreiben, und es findet ein Kontextwechsel statt, der den Prozess unterbricht und einen anderen Prozess laufen lässt, bis die Seite zurückgeschrieben wurde. Der Seitenrahmen wird inzwischen als belegt markiert, um zu verhindern, dass er für andere Zwecke benutzt wird.
6. Sobald der Seitenrahmen sauber ist (entweder sofort oder nachdem er auf die Platte geschrieben wurde), findet das System die Festplattenadresse der benötigten Seite heraus und gibt eine Anweisung an die Festplatte, die Seite zu laden.

Während die Seite geladen wird, ist der Prozess immer noch unterbrochen und ein anderer lauffähiger Prozess wird ausgeführt, wenn vorhanden.

7. Sobald ein Festplatteninterrupt anzeigt, dass die Seite angekommen ist, wird die Seitentabelle angepasst, so dass die virtuelle Seite, die den Fehler ausgelöst hat, auf den Seitenrahmen abgebildet wird, in den sie geladen wurde. Der Zustand des Seitenrahmens wird als normal markiert.
8. Der Befehl, der den Fehler ausgelöst hat, wird in seinen Anfangszustand versetzt und der Befehlszähler wird auf diesen Befehl gesetzt.
9. Der Prozess wird wieder zur Ausführung ausgewählt und das Betriebssystem springt zurück in die (Assembler-)Routine, die es aufgerufen hat.
10. Diese Routine lädt die Register und andere Zustandsinformationen und wechselt wieder in den Benutzermodus, als sei gar nichts passiert.

3.6.3 Sicherung von unterbrochenen Befehlen

Wenn ein Programm auf eine Seite zugreift, die nicht im Speicher liegt, wird der Befehl, der den Fehler auslöst, auf halbem Weg gestoppt und es kommt zu einem Sprung ins Betriebssystem. Nachdem das Betriebssystem die fehlende Seite geladen hat, muss es den Befehl neu starten. Das ist leichter gesagt als getan.

Um das Problem in seiner schlimmsten Form kennenzulernen, sehen wir uns eine CPU an, die Befehle mit zwei Adressen hat, wie z.B. der Motorola 680x0, der in eingebetteten Systemen weit verbreitet ist. Der Befehl

```
MOVE.L #6(A1),2(A0)
```

ist beispielsweise sechs Byte lang (siehe ▶Abbildung 3.28). Um den Befehl neu zu starten, muss das Betriebssystem herausfinden, wo das erste Byte des Befehls ist. Der Wert des Befehlszählers zum Zeitpunkt des Seitenfehlers hängt davon ab, welcher Operand den Fehler ausgelöst hat und wie der Befehl im Mikrocode der CPU implementiert ist.

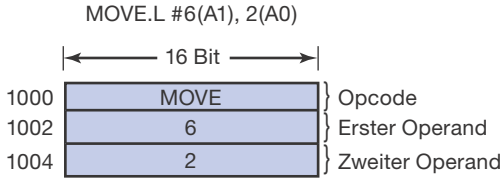

Abbildung 3.28: Ein Befehl, der einen Seitenfehler auslöst

▶Abbildung 3.28 zeigt einen Befehl an Adresse 1.000, der drei Speicherzugriffe ausführt: das Befehlswort selbst und zwei Offsets für die Operanden. Je nachdem, welche dieser Referenzen den Seitenfehler ausgelöst hat, könnte der Befehlszähler zum Zeitpunkt des Seitenfehlers auf 1.000, 1.002 oder 1.004 zeigen. Für das Betriebssystem ist es häufig unmöglich, eindeutig zu entscheiden, wo der Befehl anfängt. Wenn der

Befehlszähler zum Zeitpunkt des Fehlers 1.002 ist, hat das System keine Möglichkeit herauszufinden, ob das Wort bei 1.002 ein Opcode ist oder eine Adresse, die zu einem Befehl bei 1.000 gehört (z.B. die Adresse eines Operanden).

So schlimm dieses Problem auch ist, es kann noch schlimmer kommen. Einige Adressierungsmodi des 680x0 benutzen Auto-Inkrementierung, d.h., mit der Ausführung eines Befehls werden automatisch ein oder mehrere Register erhöht. Auch Befehle im Auto-Inkrementierungsmodus können Seitenfehler auslösen. Abhängig von den Details im Mikrocode kann die Inkrementierung vor dem Speicherzugriff ausgeführt werden. In diesem Fall muss das Betriebssystem die Register softwaremäßig zurücksetzen, bevor es den Befehl neu startet. Oder die Auto-Inkrementierung kann nach dem Speicherzugriff stattgefunden haben, was dann bedeutet, dass sie zum Zeitpunkt des Seitenfehlers noch nicht ausgeführt wurde und deshalb nicht rückgängig gemacht werden darf. Es gibt auch einen Auto-Dekrementierungsmodus, der ähnliche Probleme verursacht. Die genauen Details, ob und wann Auto-Inkrementierung und -Dekrementierung ausgeführt werden, können sich von Befehl zu Befehl und von CPU-Modell zu CPU-Modell ändern.

Zum Glück bieten die CPU-Entwickler bei manchen Prozessoren eine Lösung an, normalerweise in Form eines versteckten internen Registers, in das der Befehlszähler jedes Mal kopiert wird, kurz bevor ein Befehl ausgeführt wird. Diese Maschinen haben vielleicht auch ein zweites Register, das Informationen darüber speichert, welche Register automatisch um wie viel inkrementiert oder dekrementiert wurden. Mit diesen Informationen kann das Betriebssystem alle Effekte des unterbrochenen Befehls ohne Mehrdeutigkeiten rückgängig machen und ihn noch einmal starten. Ohne diese Informationen muss das Betriebssystem alle möglichen Tricks anwenden, um herauszufinden, was passiert ist und wie man es reparieren kann. Es scheint fast so, als hätten die Hardwareentwickler das Problem nicht lösen können, kurz mit den Schultern gezuckt und es den Betriebssystementwicklern zugeschoben. Nette Leute.

3.6.4 Sperren von Seiten im Speicher

Wir haben uns zwar in diesem Kapitel noch nicht viel mit Ein-/Ausgabe beschäftigt, aber die Tatsache, dass ein Computer virtuellen Speicher hat, bedeutet nicht, dass keine Ein- und Ausgaben stattfinden. Virtueller Speicher und Ein-/Ausgabe interagieren auf subtile Art und Weise. Nehmen wir an, ein Prozess hat gerade einen Systemaufruf gestartet, um Daten aus einer Datei oder von einem Gerät in einen Puffer in seinem Adressraum zu lesen. Während der Prozess auf das Ende des Ein-/Ausgabebefehls wartet, wird er unterbrochen und ein anderer Prozess bekommt die CPU. Dieser andere Prozess erzeugt einen Seitenfehler.

Bei einer globalen Paging-Strategie besteht eine – wenn auch sehr kleine – Chance, dass die Seite, die den Puffer enthält, zur Auslagerung ausgewählt wird. Wenn ein Ein-/Ausgabegerät gerade einen DMA-Transfer in diesen Puffer ausführt, wird ein Teil der Daten in die richtige Seite geschrieben und der Rest in die soeben geladene Seite. Eine Lösung für dieses Problem ist, die Seiten zu sperren, die mit Ein-/Ausgabebefehlen zu tun

haben, so dass sie nicht ausgelagert werden können. Das Sperren von Seiten wird häufig als **Pinning** bezeichnet. Alternativ könnten Ein-/Ausgabedaten zunächst in einen Kern-Puffer geschrieben und dann später in den richtigen Adressraum kopiert werden.

3.6.5 Hintergrundspeicher

Unsere Diskussion von Seitenersetzungsstrategien hat sich hauptsächlich damit befasst, welche Seite aus dem Speicher entfernt werden soll. Was wir noch nicht behandelt haben, ist, wohin sie auf der Festplatte geschrieben wird, wenn sie ausgelagert wird. In diesem Abschnitt beschreiben wir deshalb einige Punkte, die mit der Verwaltung der Festplatte zu tun haben.

Die einfachste Art, Platz für ausgelagerte Seiten auf der Platte zu schaffen, ist die Einrichtung einer speziellen Swap-Partition oder – noch besser – einer separaten Platte des Dateisystems (um die Ein-/Ausgabelast auszugleichen). Die meisten UNIX-Systeme arbeiten nach diesem Prinzip. Diese Partition hat kein normales Dateisystem, das all den Aufwand des Konvertierens von Offsets in Dateien zu Blockadressen eliminiert. Stattdessen werden durchweg Blocknummern relativ zum Anfang der Partition benutzt.

Beim Systemstart ist diese Swap-Partition leer und wird im Speicher durch einen einzigen Eintrag repräsentiert, der seinen Anfang und seine Größe enthält. Im einfachsten Modell reserviert das System beim Start des ersten Prozesses einen Block der Swap-Partition, der genauso groß ist wie der Prozess. Jeder weitere Prozess bekommt beim Start einen Block von der Größe seines Speicherabbilds zugeteilt. Wenn ein Prozess terminiert, wird sein Plattenplatz wieder freigegeben. Der Swap-Partition wird als Liste von freien Blöcken verwaltet. In Kapitel 10 werden wir besseren Strategien begegnen.

Mit jedem Prozess wird die Festplattenadresse seines Swap-Bereiches verbunden, d.h., an welcher Stelle der Swap-Partition das Speicherabbild liegt. Diese Information wird in der Prozesstabelle festgehalten. Wenn eine Seite ausgelagert werden soll, ist die Festplattenadresse leicht zu berechnen: Man muss nur den Offset der Seite im virtuellen Adressraum zur Anfangsadresse des Swap-Bereiches addieren. Bevor ein Prozess starten kann, muss der Swap-Bereich aber initialisiert werden. Eine Möglichkeit ist, das gesamte Kernbild des Prozesses in den Swap-Bereich zu kopieren, so dass alle Seiten bei Bedarf *eingelagert* werden können. Die andere Möglichkeit ist, den ganzen Prozess in den Speicher zu laden und bei Bedarf Seiten *auszulagern*.

Leider gibt es bei dieser simplen Methode ein Problem: Prozesse können wachsen, während sie laufen. Der Programmtext bleibt zwar normalerweise gleich, aber die Daten wachsen manchmal und der Stack wächst fast immer. Es wäre also besser, verschiedene Swap-Bereiche für Text, Daten und Stack zu reservieren und es zu ermöglichen, dass jeder dieser Bereiche aus mehr als einem Block besteht.

Das andere Extrem ist, zunächst gar nichts zu reservieren und für jede Seite einzeln Platz auf der Festplatte zu reservieren, wenn sie ausgelagert wird, und wieder freizugeben, wenn sie wieder eingelagert wird. So belegen Prozesse im Speicher überhaupt keinen Platz auf der Festplatte. Der Nachteil ist, dass für jede einzelne ausgelagerte

Seite eine Festplattenadresse im Speicher gehalten werden muss. Mit anderen Worten, jeder Prozess braucht eine Tabelle, die für jede Seite speichert, wo sie gerade ist. ▶Abbildung 3.29 illustriert die beiden Alternativen.

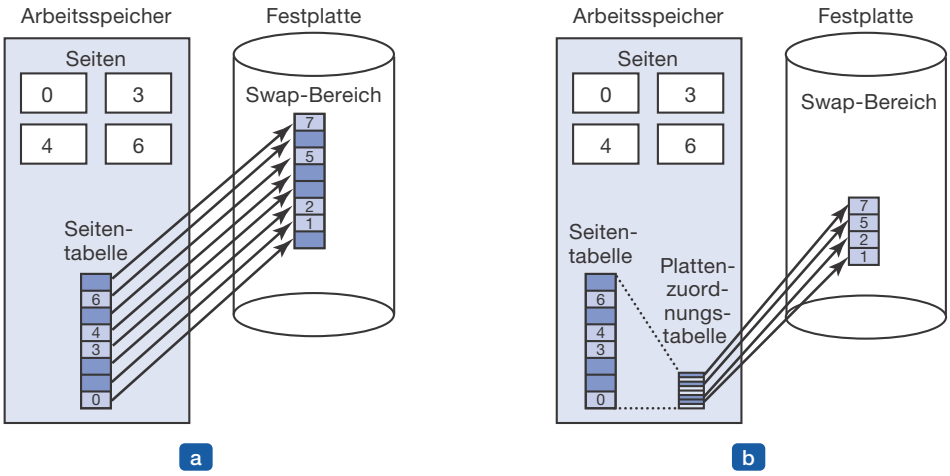

Abbildung 3.29: (a) Paging mit statischem Swap-Bereich (b) Dynamischer Hintergrundspeicher

▶Abbildung 3.29(a) zeigt eine Seitentabelle mit acht Seiten. Die Seiten 0, 3, 4 und 6 liegen im Arbeitsspeicher, die Seiten 1, 2, 5 und 7 auf der Platte. Der Swap-Bereich ist genauso groß wie der virtuelle Adressraum (acht Seiten). Jede Seite hat eine feste Adresse auf der Platte, an die sie geschrieben wird, falls sie ausgelagert wird. Für die Berechnung dieser Adresse muss man nur die Anfangsadresse des Swap-Bereiches kennen, weil die Seiten geordnet nach ihren virtuellen Seitennummern gespeichert werden. Eine Seite im Speicher hat immer eine Schattenkopie auf der Platte, die aber nicht unbedingt aktuell ist. Die dunkel dargestellten Seiten im Speicher zeigen an, dass sich diese Seiten nicht im Speicher befinden. Die dunkel gezeichneten Seiten auf der Platte sind (im Prinzip) von den Kopien im Speicher verdrängt worden. Falls allerdings eine Speicherseite zurück auf die Platte geschrieben wird, die nicht verändert wurde, dann wird die (dunkelblaue) Plattenkopie benutzt.

In ▶Abbildung 3.29(b) haben die Seiten keine festen Adressen auf der Festplatte. Wenn eine Seite ausgelagert wird, sucht das Betriebssystem eine leere Seite auf der Platte und trägt die Adresse in die Tabelle ein, die virtuelle Seiten auf Festplattenadressen abbildet. Eine Seite im Speicher hat keine Festplattenkopie und ihre Einträge in der Tabelle für Plattenadressen sind ungültig oder als unbenutzt markiert.

Nicht immer ist es möglich, Swap-Partitionen eine feste Größe zuzuordnen. Es kann zum Beispiel vorkommen, dass keine Plattenpartitionen verfügbar sind, in diesem Fall können ein oder mehrere große, im Voraus reservierte Dateien innerhalb des normalen Dateisystems benutzt werden. Windows verwendet diese Methode. Eine Optimierung kann hier eingesetzt werden, um die Menge des benötigten Plattenplatzes zu reduzieren. Da der Programmtext jedes Prozesses in einer (ausführbaren) Datei im Dateisystem liegt, kann diese Datei als Swap-Bereich benutzt werden. Mehr noch: Da der

Programmtext in der Regel nur gelesen werden darf, können die Programmseiten einfach gelöscht und bei Bedarf wieder eingelesen, wenn der Speicherplatz knapp wird und Seiten ausgelagert werden müssen. Gemeinsame Bibliotheken können auch nach diesem Prinzip arbeiten.

3.6.6 Trennung von Strategie und Mechanismus

Ein wichtiges Hilfsmittel, um die Komplexität eines beliebigen Systems in den Griff zu bekommen, ist die Trennung von Strategie und Mechanismus. Dieses Prinzip kann man bei der Speicherverwaltung anwenden, indem man den größten Teil der Speicherverwaltung in einen Benutzerprozess verlagert. Eine solche Trennung wurde zum ersten Mal bei dem Betriebssystem Mach (Young et al., 1987) vollzogen, die folgende Diskussion ist daran angelehnt.

▶Abbildung 3.30 zeigt ein einfaches Beispiel für die Trennung von Strategie und Mechanismus. Die Speicherverwaltung besteht hier aus drei Teilen:

1. einer maschinennahen MMU-Behandlungsroutine;
2. einer Seitenfehler-Behandlungsroutine im Kern;
3. einem externen Pager im Benutzermodus.

Die Details der MMU sind in der MMU-Behandlungsroutine gekapselt, die aus maschinenabhängigem Code besteht und die für jede Plattform, auf die das Betriebssystem portiert wird, neu geschrieben werden muss. Die Seitenfehlerbehandlung ist maschinenunabhängig und enthält den Hauptteil der Paging-Mechanismen. Die Strategie wird hauptsächlich vom externen Pager bestimmt, der als Benutzerprozess läuft.

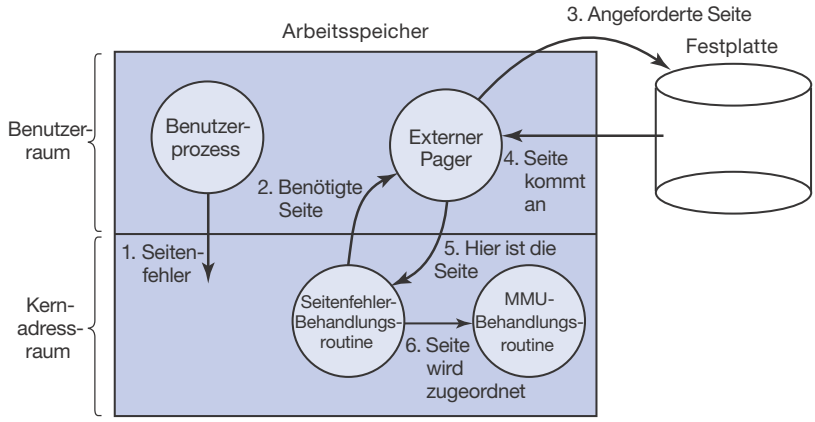

Abbildung 3.30: Seitenfehlerbehandlung mit externem Pager

Wenn ein neuer Prozess gestartet wird, wird der externe Pager benachrichtigt, um die Seitentabelle zu initialisieren und wenn nötig Hintergrundspeicher auf der Festplatte zu reservieren. Wenn der Prozess während seiner Laufzeit neue Speicherobjekte in seinen Adressraum einblendet, wird der Pager erneut benachrichtigt.

Sobald der Prozess läuft, ist es möglich, dass er Seitenfehler auslöst. Die Seitenfehler-Behandlungsroutine findet heraus, welche virtuelle Seite benötigt wird, und schickt eine Nachricht an den externen Pager, um ihn über das Problem zu informieren. Der externe Pager liest die Seite von der Platte und kopiert sie in seinen eigenen Adressraum. Dann teilt er der Behandlungsroutine mit, wo die Seite liegt. Die Behandlungsroutine blendet die Seite aus dem Adressraum des externen Pagers aus und ruft die MMU-Behandlungsroutine auf, die sie an der richtigen Stelle im Adressraum des Benutzerprozesses einblendet. Dann kann der Prozess wieder gestartet werden.

Diese Implementierung lässt offen, wo der Seitenersetzungsalgorithmus untergebracht ist. Am saubersten wäre es, ihn als Teil des externen Pagers zu implementieren, aber dieser Ansatz bereitet einige Probleme. Das größte Problem ist, dass der externe Pager nicht auf die *R*- und *M*-Bits aller Seiten zugreifen kann, die in vielen Seitenersetzungsalgorithmen eine wichtige Rolle spielen. Der Seitenersetzungsalgorithmus muss also entweder im Kern laufen oder die *R*- und *M*-Bits müssen irgendwie an den externen Pager weitergegeben werden. Im letzteren Fall teilt die Seitenfehler-Behandlungsroutine dem externen Pager mit, welche Seite der Algorithmus auslagern will, und überträgt die Daten, indem die Fehlerbehandlungsroutine entweder die Seite in den Adressraum des Pagers einblendet oder indem sie die Daten in einer Nachricht schickt. Auf jeden Fall schreibt der externe Pager die Daten auf die Festplatte.

Der Hauptvorteil dieser Implementierung ergibt sich aus der größeren Modularität und Flexibilität. Der Hauptnachteil ist der Leistungsverlust einerseits durch die Kontextwechsel zwischen Kern und Benutzermodus und andererseits durch den Nachrichtenaustausch zwischen den Teilen des Systems. Im Moment wird dieses Thema noch kontrovers diskutiert. Die Computer werden aber immer schneller und die Software wird immer komplexer, so dass die meisten Betriebssystementwickler sich auf lange Sicht wahrscheinlich dafür entscheiden werden, etwas Performanz zu opfern, um die Software verlässlicher zu machen.

3.7 Segmentierung

Der bisher behandelte virtuelle Speicher ist eindimensional, weil die virtuellen Adressen linear von 0 bis zu einem bestimmten Maximum wachsen, eine Adresse nach der anderen. Für viele Probleme wäre es aber viel besser, zwei oder mehr separate Adressräume zu haben. Ein Compiler hat z.B. mehrere Tabellen, die während der Übersetzung aufgebaut werden, darunter

1. der Quellcode, der für den Ausdruck des Listings gespeichert wird (auf Stapelverarbeitungssystemen);
2. die Symboltabelle, die die Namen und Attribute von Variablen enthält;
3. eine Tabelle für die benutzten Ganzzahl- und Gleitkomma-Konstanten;
4. der Strukturbaum, der während der syntaktischen Analyse des Programms aufgebaut wird;
5. der Stack für Prozeduraufrufe innerhalb des Compilers.

Die ersten vier Tabellen wachsen während der Übersetzung ständig. Die letzte Tabelle wächst und schrumpft unvorhersehbar während der Übersetzung. In einem eindimensionalen Speicher müsste jeder dieser Tabellen wie in ▶Abbildung 3.31 ein zusammenhängender Speicherblock zugeteilt werden.

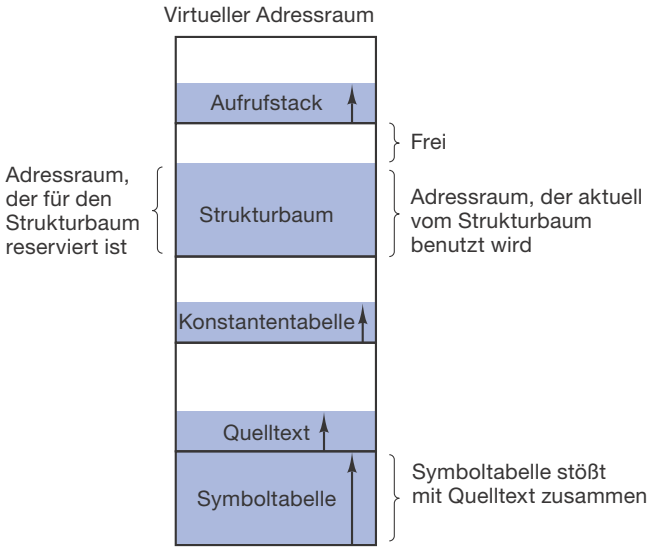

Abbildung 3.31: In einem eindimensionalen Adressraum können wachsende Tabellen kollidieren.

Überlegen wir uns, was passiert, wenn ein Programm viel mehr Variablen als gewöhnlich hat, aber von allem anderen eine normale Menge. Dann würde der Symboltabelle bald der Platz ausgehen, aber alle anderen Tabellen könnten noch Platz übrig haben. Der Compiler könnte die Übersetzung natürlich mit der Meldung abbrechen, dass das Programm zu viele Variablen hat, aber wenn in den anderen Tabellen noch Platz ist, wäre das eher unelegant.

Eine andere Möglichkeit wäre, Robin Hood zu spielen: den Tabellen mit zu viel Platz Speicher wegnehmen und ihn denen geben, die zu wenig Platz haben. Diese Umverteilung ist möglich, aber es gilt dasselbe wie bei der Verwaltung von Overlays – im besten Fall ist sie unbequem und im schlimmsten Fall bedeutet sie eine Menge nervtötender, unbefriedigender Arbeit.

Am besten wäre es, den Programmierer nicht mit der Verwaltung von wachsenden und schrumpfenden Tabellen zu belasten, so wie er durch den virtuellen Speicher von der Verwaltung der Overlays befreit wurde.

Eine einfache und äußerst allgemeine Lösung besteht darin, die Maschine mit vielen voneinander völlig unabhängigen Adressräumen auszustatten, den sogenannten **Segmenten**. Jedes Segment besteht aus einer linearen Folge von Adressen, von 0 bis zu einem bestimmten Maximum. Die Größe eines Segmentes liegt zwischen 0 und dem

erlaubten Maximum. Verschiedene Segmente können verschieden groß sein und sind es normalerweise auch. Außerdem kann sich die Größe eines Segmentes während der Ausführung ändern. Die Länge eines Stacksegmentes würde z.B. durch Push-Operationen erhöht und durch Pop-Operationen verringert.

Weil jedes Segment einen eigenen Adressraum darstellt, können Segmente unabhängig voneinander ihre Größe ändern. Wenn ein Stack in einem bestimmten Segment mehr Platz braucht, kann er ihn haben, weil nichts in seinem Adressraum ist, mit dem er kollidieren könnte. Natürlich kann einem Segment der Platz ausgehen, aber das kommt nur selten vor, weil Segmente normalerweise sehr groß sind. Eine Adresse in diesem segmentierten oder zweidimensionalen Speicher besteht aus zwei Teilen: einer Segmentnummer und einer Adresse innerhalb des Segmentes. ▶Abbildung 3.32 zeigt, wie der segmentierte Speicher für den erwähnten Compiler aufgebaut sein könnte. In diesem Beispiel benutzt der Compiler für die Programmdaten fünf Segmente.

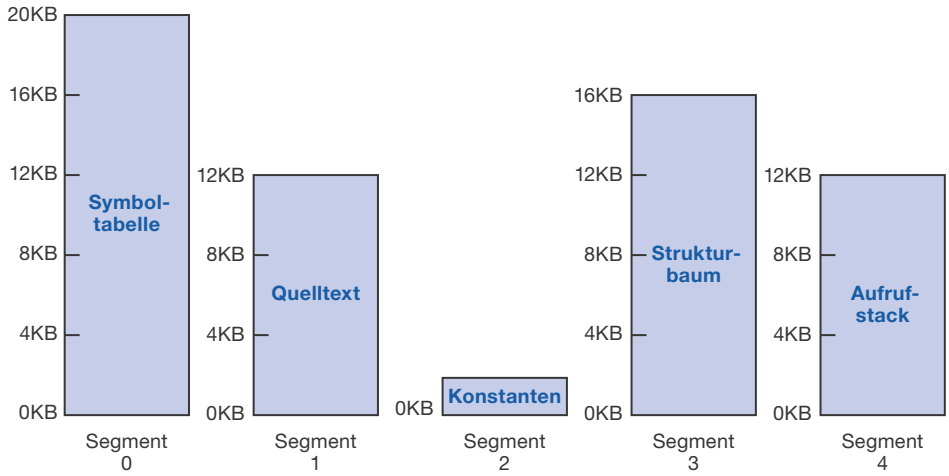

Abbildung 3.32: Segmentierter Speicher ermöglicht Tabellen, die unabhängig voneinander ihre Größe ändern

Ein Segment bildet eine logische Einheit. Der Programmierer ist sich dessen bewusst und benutzt das Segment auch einheitlich. Zum Beispiel könnte ein Segment eine Prozedur, ein Feld, einen Stack oder eine Menge von Variablen enthalten, aber normalerweise keine Mischung verschiedener Typen.

Neben der einfacheren Verwaltung von Datenstrukturen, die ihre Größe ändern, hat segmentierter Speicher noch andere Vorteile. Wenn jede Prozedur eines Programms in einem separaten Segment an Adresse 0 liegt, vereinfacht sich das Binden von getrennt übersetzten Programmteilen enorm. Nachdem alle Prozeduren, aus denen das Programm besteht, übersetzt und gebunden sind, kann man als Einstiegspunkt in die Prozedur im Segment n die zweiteilige Adresse (n, 0) verwenden, die auf das erste Wort im Segment n zeigt.

Wenn die Prozedur im Segment *n* später geändert und neu übersetzt wird, müssen die anderen Prozeduren nicht angepasst werden, weil sich keine Anfangsadressen geändert haben, auch wenn die neue Prozedur länger ist als die alte. In einem eindimensionalen Speicher liegen die Prozeduren direkt hintereinander, so dass kein freier Adressraum dazwischen existiert. Wenn nun eine Prozedur ihre Größe ändert, können sich dadurch die Anfangsadressen von anderen (nicht verbundenen) Prozeduren ändern. Dadurch muss dann jede Prozedur, die eine der verschobenen Prozeduren aufruft, geändert werden, um sie an die neue Anfangsadresse anzupassen. Dieser Vorgang kann sehr teuer werden, wenn das Programm Hunderte von Prozeduren enthält.

Segmentierung vereinfacht auch die gemeinsame Nutzung von Programmcode oder Daten durch mehrere Prozesse. Ein typisches Beispiel sind gemeinsame Bibliotheken. Moderne Workstations mit ausgefeilten grafischen Oberflächen haben oft extrem große Grafikbibliotheken, die in fast jedes Programm eingebunden werden. In einem segmentierten System kann diese Bibliothek in ein Segment geladen und von allen Prozessen gemeinsam genutzt werden. Dadurch muss sie nicht mehr separat in den Adressraum jedes Prozesses geladen werden. In reinen Paging-Systemen sind gemeinsame Bibliotheken zwar auch möglich, aber wesentlich komplizierter zu verwirklichen. Meistens läuft es darauf hinaus, dass diese Systeme Segmentierung simulieren.

Jedes Segment bildet eine logische Einheit, wie beispielsweise eine Prozedur, ein Feld oder einen Stack, und der Programmierer ist sich dessen bewusst. Deshalb können verschiedene Segmente auch unterschiedlich geschützt werden. Ein Segment, das eine Prozedur enthält, kann z. B. als nur ausführbar markiert sein, um Lese- und Schreibzugriffe zu verhindern. Ein Feld von Gleitkommazahlen könnte als lesbar und beschreibbar, aber als nicht ausführbar markiert sein, so dass Versuche, in das Feld hineinzuspringen, abgefangen werden. Diese Art von Speicherschutz hilft, Programmierfehler zu entdecken.

Es ist wichtig zu verstehen, warum Speicherschutz in segmentiertem Speicher, aber nicht in eindimensionalem Speicher mit Paging sinnvoll ist. In segmentiertem Speicher weiß der Programmierer, welche Art von Daten sich in jedem Segment befindet. Ein Segment kann z. B. eine Prozedur oder einen Stack enthalten – entweder das eine oder das andere, aber niemals beide gleichzeitig. Der Schutz für ein Segment kann also speziell an die Art der Objekte in diesem Segment angepasst werden. ▶Abbildung 3.33 stellt Paging und Segmentierung gegenüber.

Überlegung	Paging	Segmentierung
Muss der Programmierer wissen, dass diese Technik benutzt wird?	Nein	Ja
Wie viele lineare Adressräume gibt es?	1	Viele
Kann der gesamte Adressraum die Größe des physischen Speichers übersteigen?	Ja	Ja
Können Prozeduren und Daten unterschieden und getrennt voneinander geschützt werden?	Nein	Ja
Können Tabellen mit schwankender Größe verwaltet werden?	Nein	Ja
Wird das gemeinsame Benutzen von Prozeduren durch Anwender unterstützt?	Nein	Ja
Warum wurde diese Technik eingeführt?	Um einen großen linearen Adressraum benutzen zu können, ohne weiteren physischen Speicher zu kaufen	Um Programme und Daten in unabhängige logische Adressräume aufzuspalten und um gemeinsame Nutzung und Schutz zu unterstützen

Abbildung 3.33: Vergleich von Paging und Segmentierung

3.7.1 Implementierung von Segmentierung

Zwischen der Implementierung von Segmentierung und der von Paging gibt es einen wichtigen Unterschied: Seiten haben eine feste Größe, Segmente nicht. ▶Abbildung 3.34(a) zeigt einen physischen Speicher, der fünf Segmente enthält. Wenn Segment 1 entfernt wird und das kleinere Segment 7 an seine Stelle tritt, entsteht die Situation in ▶Abbildung 3.34(b). Zwischen Segment 7 und Segment 2 ist ein unbenutzter Bereich entstanden – eine Lücke. Als Nächstes wird, wie in ▶Abbildung 3.34(c) zu sehen, Segment 4 durch Segment 5 ersetzt und anschließend Segment 3 durch Segment 6, siehe ▶Abbildung 3.34(d).Wenn das System eine Zeit lang gelaufen ist, ist der Speicher in eine Menge von Blöcken unterteilt, von denen einige Segmente und andere Lücken sind. Dieses Phänomen heißt **Checkerboarding** oder **externe Fragmentierung** und verschwendet Speicher. Es kann durch Verdichtung verhindert werden, wie in ▶Abbildung 3.34(e) gezeigt.

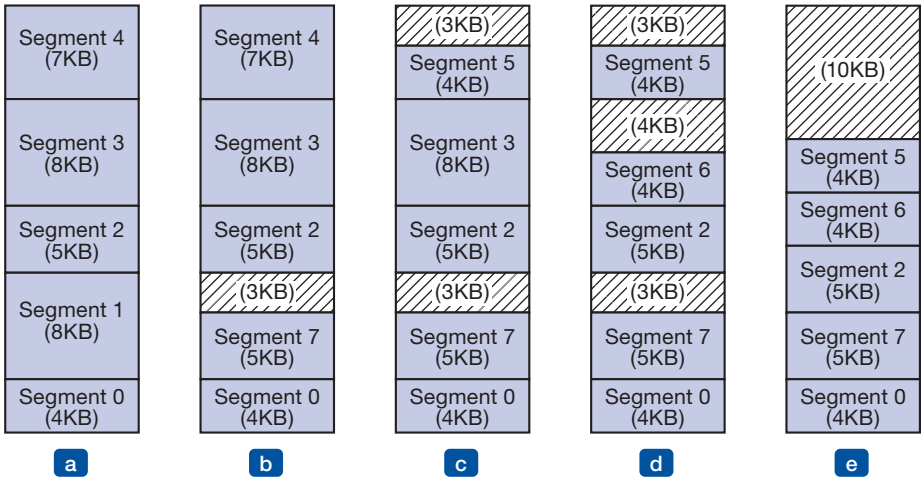

Abbildung 3.34: (a)–(d) Entstehung von externer Fragmentierung (e) Behebung durch Speicherverdichtung

3.7.2 Segmentierung mit Paging: MULTICS

Wenn die Segmente groß sind, ist es möglicherweise unbequem oder sogar unmöglich, sie komplett im Arbeitsspeicher zu halten. Dies führt zu der Idee, sie in Seiten aufzuteilen und die Seiten nach Bedarf ein- und auszulagern. Mehrere wichtige Systeme haben Segmentierung mit Paging unterstützt. In diesem Abschnitt beschreiben wir das allererste: MULTICS. Im nächsten Abschnitt sehen wir uns dann ein neueres System an, den Intel Pentium.

MULTICS lief auf Honeywell-6000-Maschinen und deren Nachfolgern. Unter MULTICS konnte jedes Programm bis zu 2^{18} Segmente benutzen (über 250.000), die jeweils bis zu 65.536 36-Bit-Wörter lang waren. Um dies zu implementieren, behandelten die MULTICS-Entwickler jedes Segment als virtuellen Speicher und teilten es in Seiten auf. Dadurch verbanden sie die Vorteile von Paging (einheitliche Seitengröße und die Möglichkeit, nur den Teil eines Segmentes im Speicher zu halten, der gebraucht wird) mit denen der Segmentierung (einfache Programmierung, Modularität, Speicherschutz, gemeinsame Nutzung von Speicher).

Jedes MULTICS-Programm hat eine Segmenttabelle, die für jedes Segment einen Deskriptor enthält. Da die Tabelle bis zu einer viertel Million Einträge haben kann, ist sie selbst ein Segment und kann seitenweise ausgelagert werden. Ein Segmentdeskriptor zeigt an, ob das Segment im Speicher ist oder nicht. Wenn nur ein Teil des Segmentes im Speicher liegt, gilt das Segment als im Speicher vorhanden und die Seitentabelle liegt auch im Speicher. In diesem Fall enthält der Deskriptor einen 18-Bit-Zeiger auf die Seitentabelle (siehe ▶Abbildung 3.35(a)). Da die physischen Adressen 24 Bit haben und die Seiten immer an 64-Byte-Grenzen ausgerichtet sind (wodurch die unteren 6 Bit einer Seitenadresse immer 000000 sind), reichen 18 Bit im Deskriptor aus, um die Adresse einer Seitentabelle zu speichern. Der Deskriptor enthält außerdem noch die Segmentgröße, die Schutzbits und einige andere Informationen. ▶Abbildung 3.35(b) zeigt einen MULTICS-

Segmentdeskriptor. Die Adresse eines Segmentes im Sekundärspeicher steht nicht im Deskriptor, sondern in einer Tabelle der Segmentfehler-Behandlungsroutine.

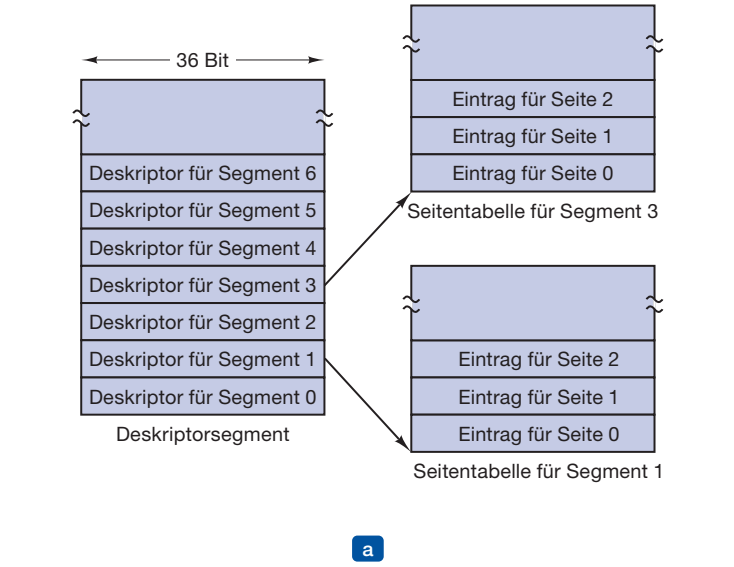

Abbildung 3.35: Der virtuelle Speicher unter MULTICS (a) Die Deskriptoren im Deskriptorsegment zeigen auf die Seitentabellen. (b) Ein Segmentdeskriptor. Die Zahlen bezeichnen die Länge des Feldes.

Jedes Segment ist ein normaler virtueller Adressraum, der aus Seiten besteht, die genau wie beim Paging ohne Segmentierung ein- und ausgelagert werden. Die normale Seitengröße ist 1.024 Wörter, aber ein paar kleine Segmente, die von MULTICS selbst benutzt werden, werden in Einheiten von 64 Byte ein- und ausgelagert, um physischen Speicher zu sparen.

Eine Adresse besteht in MULTICS aus zwei Teilen: dem Segment und der Adresse innerhalb des Segmentes. Die Adresse innerhalb des Segmentes unterteilt sich weiter in eine Seitennummer und ein Wort innerhalb der Seite. ▶Abbildung 3.36 zeigt eine virtuelle Adresse in MULTICS. Bei jedem Speicherzugriff wird der folgende Algorithmus ausgeführt:

1. Über die Segmentnummer wird der Segmentdeskriptor geladen.
2. Es wird überprüft, ob die Seitentabelle des Segmentes im Speicher liegt. Wenn ja, wird sie über den Deskriptor lokalisiert. Wenn nicht, wird ein Segmentfehler ausgelöst. Bei einer Schutzverletzung wird ebenfalls ein Fehler (Sprung ins System) ausgelöst.
3. Der Seitentabelleneintrag für die gesuchte Seite wird untersucht. Ist die Seite selbst nicht im Speicher, wird ein Seitenfehler ausgelöst. Ansonsten wird die Speicheradresse des Seitenanfangs aus dem Seitentabelleneintrag ausgelesen.
4. Der Offset innerhalb der Seite wird zum Seitenanfang addiert. Das Ergebnis ist die gesuchte physische Adresse.
5. Der Speicherzugriff wird schließlich ausgeführt.

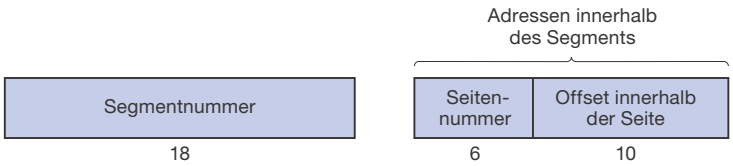

Abbildung 3.36: Eine virtuelle 34-Bit-Adresse unter MULTICS

▶Abbildung 3.37 illustriert diesen Vorgang noch einmal. Der Einfachheit halber wurde ignoriert, dass das Deskriptorsegment selbst aus Seiten besteht, die ausgelagert sein könnten. In Wirklichkeit steht die Anfangsadresse der Seitentabelle des Deskriptorsegmentes in einem speziellen Register (dem Deskriptor-Basisregister). Diese Tabelle enthält die physischen Adressen des Deskriptorsegmentes. Sobald der Deskriptor für das gesuchte Segment gefunden ist, geht die Adressberechnung wie in ▶Abbildung 3.37 weiter.

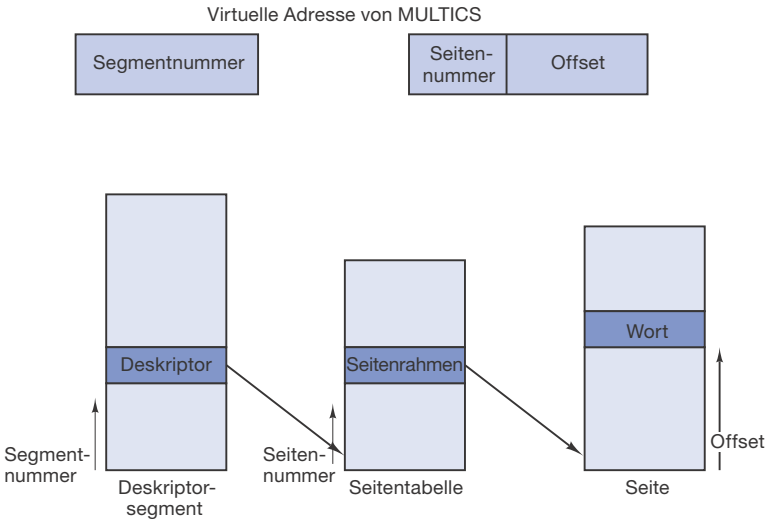

Abbildung 3.37: Umwandlung einer zweiteiligen MULTICS-Adresse in eine physische Speicheradresse

Wie Sie zweifellos inzwischen erkannt haben, würde dieser Algorithmus die Programme nicht gerade beschleunigen, wenn er für jeden Speicherzugriff ausgeführt werden müsste. In Wirklichkeit enthält die MULTICS-Hardware einen schnellen, 16-Bit-Wörter großen TLB, dessen Einträge parallel nach einem Schlüssel durchsucht werden können, siehe ▶Abbildung 3.38. Wenn ein Programm auf eine Adresse zugreifen will, prüft die Hardware für die Adressberechnung zunächst, ob die virtuelle Adresse im TLB steht. Wenn ja, liest sie die Seitenrahmennummer direkt aus dem TLB und bildet die Adresse, ohne auf das Deskriptorsegment oder die Seitentabelle zuzugreifen.

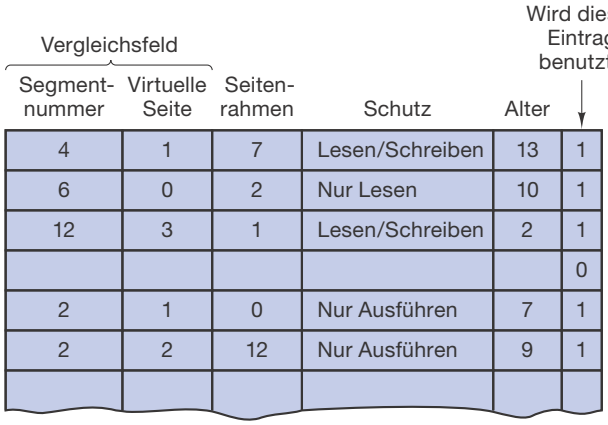

Abbildung 3.38: Eine vereinfachte Version des MULTICS-TLB. In Wirklichkeit wird der TLB durch die zwei verschiedenen Seitengrößen komplizierter.

Der TLB enthält die Adressen der 16 letzten Seiten. Programme, deren Arbeitsbereich kleiner als der TLB ist, erreichen einen Gleichgewichtszustand mit den Adressen ihres gesamten Arbeitsbereiches im TLB und laufen deshalb effizient. Wenn eine Seite nicht im TLB ist, wird die Adresse des Seitenrahmens über die Deskriptor- und Seitentabelle ermittelt und in den TLB an der Stelle der am längsten unbenutzten Seite eingefügt. Das *Alter*-Feld dient zur Ermittlung der am längsten nicht benutzten Seite. Ein TLB wird eingesetzt, damit die Segment- und Seitennummern von allen Einträgen parallel mit der gesuchten Adresse verglichen werden können.

3.7.3 Segmentierung mit Paging: der Intel Pentium

Der virtuelle Speicher des Intel Pentium ähnelt dem von MULTICS in vielen Bereichen, darunter die Verwendung von Segmentierung und Paging. Im Gegensatz zu MULTICS, das 256 KB unabhängige Segmente mit jeweils 64 KB 36-Bit-Wörter verwaltet, hat der Pentium 16-KB-Segmente, von denen jedes bis zu einer Milliarde 32-Bit-Wörter enthält. Der Pentium hat zwar weniger Segmente, die Größe ist aber wesentlich wichtiger, weil ein Programm häufig sehr große Segmente benötigt, aber nur selten mehr als 1.000.

Das Herz des virtuellen Speichers des Pentium wird von zwei Tabellen gebildet, die **LDT** (**lokale Deskriptortabelle**, *Local Descriptor Table*) und die **GDT** (**globale Deskriptortabelle**, *Global Descriptor Table*). Jedes Programm hat seine eigene LDT, aber es gibt nur eine GDT, die gemeinsam von allen laufenden Programmen benutzt wird. Die LDT beschreibt die lokalen Segmente eines Programms, darunter Code-, Daten- und Stacksegment. Die GDT beschreibt dagegen die Systemsegmente, darunter das Betriebssystem selbst.

Um auf ein Segment zuzugreifen, lädt ein Pentium-Programm zunächst einen Selektor für dieses Segment in eines der sechs Segmentregister. Während ein Programm ausgeführt wird, enthält das *CS*-Register den Selektor für das Codesegment und *DS* enthält den Selektor für das Datensegment. Die anderen Segmentregister sind weniger wichtig. Jeder Selektor ist eine 16-Bit-Zahl (siehe ▶Abbildung 3.39).

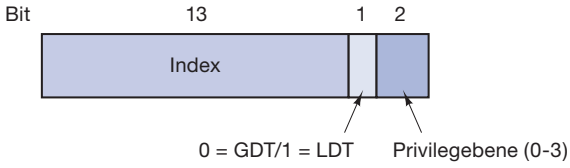

Abbildung 3.39: Pentium-Selektor

Eines der Selektor-Bits bestimmt, ob das Segment lokal oder global ist, d.h., ob der Deskriptor in der LDT oder GDT steht. Weitere 13 Bits enthalten die Nummer des Eintrages in der LDT oder GDT, so dass diese Tabellen jeweils maximal 8 KB Segmentdeskriptoren enthalten können. Die restlichen zwei Bits sind ein Schutzcode und werden später behandelt. Der Deskriptor 0 ist verboten und kann in ein Register geladen werden, um anzuzeigen, dass das Segment nicht verfügbar ist. Wenn er benutzt wird, löst das einen Fehler aus.

Sobald ein Selektor in ein Segmentregister geladen wird, wird der entsprechende Deskriptor aus der LDT oder GDT geholt und zum schnelleren Zugriff in einem Mikroprogrammregister gespeichert. Ein Deskriptor besteht aus 8 Byte und enthält die Basisadresse des Segmentes, die Größe und andere Informationen (siehe ▶Abbildung 3.40).

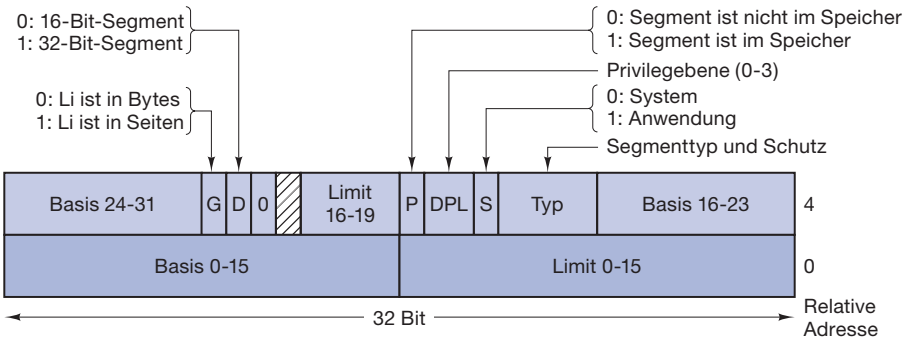

Abbildung 3.40: Pentium-Deskriptor für ein Codesegment. Datensegmente sehen etwas anders aus.

Das Format eines Selektors ist geschickt gewählt, so dass der Deskriptor einfach zu finden ist. Zunächst wird, abhängig vom Selektor-Bit 2, die LDT oder die GDT ausgewählt. Dann wird der Selektor in ein internes Zwischenregister kopiert und die niedrigsten drei Bit werden auf 0 gesetzt. Schließlich wird die Adresse der LDT bzw. GDT zum Selektor addiert, der dann direkt auf den Deskriptor zeigt. Der Selektor 72 bezieht sich z.B. auf den neunten Eintrag in der GDT, der an der Adresse GDT + 72 liegt.

Gehen wir jetzt die Schritte durch, die ein (Selektor, Offset)-Paar in eine physische Adresse umwandeln. Sobald das Mikroprogramm weiß, welches Segmentregister benutzt wird, kann es den zugehörigen Deskriptor in seinen internen Registern finden. Wenn das Segment nicht existiert (Selektor 0) oder ausgelagert ist, löst es einen Sprung ins Betriebssystem aus.

Die Hardware benutzt dann das *Limit*-Feld, um zu überprüfen, ob der Offset über das Segment hinauszeigt. In diesem Fall springt es ebenfalls ins Betriebssystem. Eigentlich sollte der Deskriptor ein 32-Bit-Feld für die Größe des Segmentes enthalten, das Feld hat aber nur 20 Bit. Deshalb wird ein anderes Schema benutzt. Wenn das *G*-Bit (*Granularity*) 0 ist, steht im *Limit*-Feld die exakte Größe, maximal 1 MB. Ansonsten gibt es die Größe anstatt in Byte in Seiten an. Der Pentium hat eine feste Seitengröße von 4 KB, so dass 20 Bit für bis zu 2^{32} Byte ausreichen.

Wenn das Segment im Speicher ist und der Offset sich im erlaubten Bereich befindet, addiert der Pentium das 32 Bit große *Basis*-Feld zum Offset. Das Ergebnis ist eine sogenannte **lineare Adresse** wie in ▶Abbildung 3.41. Aus Kompatibilitätsgründen zum 286er, wo die *Basis* nur 24 Bit hat, ist das *Basis*-Feld im Deskriptor nicht zusammenhängend, sondern in drei Blöcken über den Deskriptor verteilt. Der Zweck des *Basis*-Feldes ist, Segmente an einem beliebigen Punkt im virtuellen 32-Bit-Adressraum beginnen zu lassen.

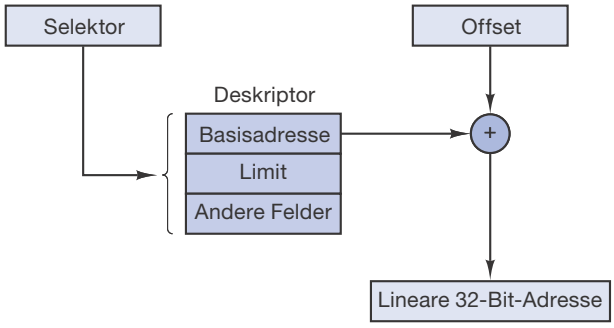

Abbildung 3.41: Berechnung einer linearen Adresse aus einem (Selektor, Offset)-Paar

Wenn das Paging abgeschaltet ist (durch ein Bit in einem globalen Kontrollregister), wird die lineare Adresse als physische Adresse behandelt und direkt auf den Speicherbus gelegt. In diesem Fall haben wir also reine Segmentierung, wobei die Basisadresse für jedes Segment im Deskriptor steht. Segmente dürfen sich übrigens überlappen, wahrscheinlich weil es zu aufwändig wäre, jedes Mal zu prüfen, ob sie disjunkt sind.

Wenn Paging dagegen eingeschaltet ist, wird die lineare Adresse als virtuelle Adresse interpretiert und mithilfe von Seitentabellen auf eine physische Adresse abgebildet, genau wie in den früheren Beispielen. Die einzige Komplikation ist, dass ein Segment mit einem 32-Bit-Adressraum bis zu einer Million 4-KB-Seiten enthalten kann. Entsprechend benutzt der Pentium zweistufige Seitentabellen, um deren Größe für kleine Segmente zu reduzieren.

Jeder Prozess hat ein **Seitenverzeichnis** (*page directory*), das aus 1.024 32-Bit-Einträgen besteht. Es liegt an einer Adresse, die in einem globalen Register gespeichert wird. Jeder Eintrag in diesem Verzeichnis zeigt auf eine Seitentabelle, die auch 1.024 32-Bit-Einträge enthält. Die Einträge in der Seitentabelle zeigen auf Seitenrahmen. Das Schema wird in ▶Abbildung 3.42 dargestellt.

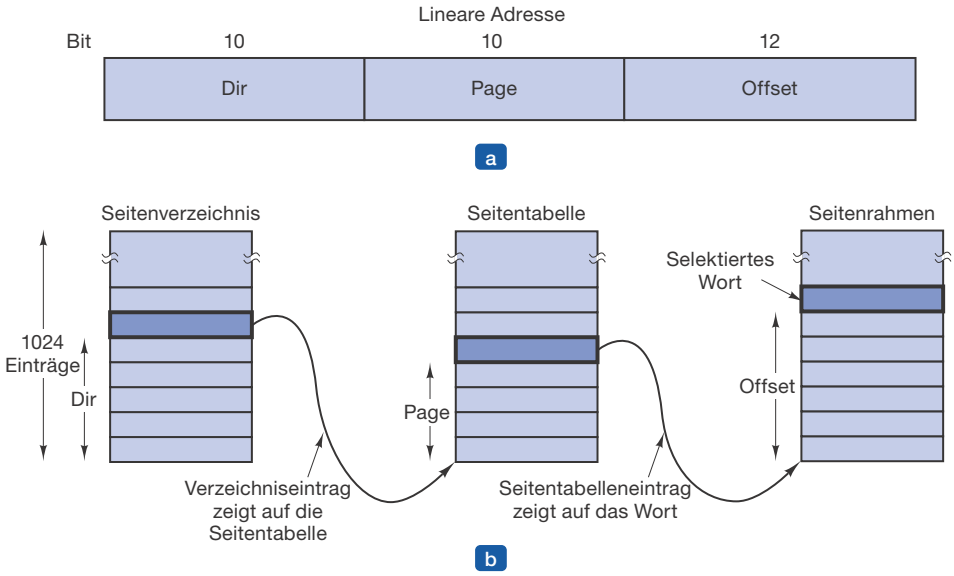

Abbildung 3.42: Abbildung einer linearen auf eine physische Adresse

▶Abbildung 3.42(a) zeigt eine lineare Adresse, die in drei Felder aufgeteilt ist, *Dir*, *Page* und *Offset*. Das *Dir*-Feld ist ein Index für das Seitenverzeichnis, um die richtige Seitentabelle zu finden. Als Nächstes wird mit dem *Page*-Feld als Index die Adresse des Seitenrahmens aus der Seitentabelle ausgelesen. Zu dieser Adresse wird schließlich das *Offset*-Feld addiert. Das Ergebnis ist die benötigte physische Adresse.

Die Einträge in der Seitentabelle sind jeweils 32 Bit lang, von denen 20 die Seitenrahmennummer enthalten. Die restlichen Bits sind die *R*- und *M*-Bits, die die Hardware für das Betriebssystem setzt, Schutzbits und einige Bits für andere Zwecke.

Jede Seitentabelle hat Einträge für 1.024 4-KB-Seiten, also ist jede Seitentabelle für 4 MB Speicher zuständig. Ein Segment, das kürzer ist als 4 MB, hat ein Seitenverzeichnis mit nur einem einzigen Eintrag, der einen Zeiger auf seine einzige Seitentabelle enthält. Der Aufwand für ein kleines Segment beläuft sich also nur auf zwei Seiten anstatt der Millionen Seiten, die bei einer einstufigen Seitentabelle nötig wären.

Um mehrfache unnötige Speicherzugriffe zu vermeiden, hat der Pentium genau wie MULTICS einen TLB, der die zuletzt benutzten *Dir-Page*-Kombinationen auf die physischen Adressen der zugehörigen Seitenrahmen abbildet. Nur bei Zugriffen auf lineare Adressen, deren *Dir-Page*-Kombination nicht im TLB steht, wird der Mechanismus aus ▶Abbildung 3.42 ausgeführt. Solange TLB-Fehler selten sind, ist die Leistung gut.

Manche Programme benötigen keine Segmente, sondern sind mit nur einem einzigen 32-Bit-Adressraum zufrieden. Solche Programme kann man schreiben, indem man einfach denselben Selektor in alle Segmentregister lädt. Der zugehörige Deskriptor enthält dann als *Basis* 0 und als *Limit* das Maximum. Die lineare Adresse ist dann einfach der Offset innerhalb des Segmentes und das Ergebnis ist normales Paging. Tatsächlich funktionieren so alle Betriebssysteme, die es momentan für den Pentium gibt. Das einzige Betriebssystem, das die Möglichkeiten des Pentium voll ausgeschöpft hat, war OS/2.

Insgesamt muss man den Entwicklern des Pentium ein Kompliment machen. Der Pentium sollte reines Paging, reine Segmentierung und Segmentierung mit Paging beherrschen, dabei zum 268er kompatibel und auch noch effizient sein. Wenn man diese widersprüchlichen Ziele bedenkt, ist das Design überraschend einfach und sauber.

Nachdem wir, wenn auch nur kurz, die gesamte Speicherarchitektur des Pentium behandelt haben, sollten wir noch ein paar Worte zum Speicherschutz sagen, weil dieses Thema eng mit dem virtuellen Speicher verbunden ist. Ebenso wie der virtuelle Speicher des Pentium ist auch sein Speicherschutz eng an MULTICS angelehnt. Der Pentium unterstützt vier Schutzebenen (*protection level*), wobei Ebene 0 am meisten und Ebene 3 am wenigsten privilegiert ist. ▶Abbildung 3.43 zeigt die vier Ebenen. Jedes Programm läuft zu jeder Zeit auf einer bestimmten Ebene, die durch ein 2-Bit-Feld im PSW codiert wird. Jedem Segment im System wird auch eine Ebene zugeordnet.

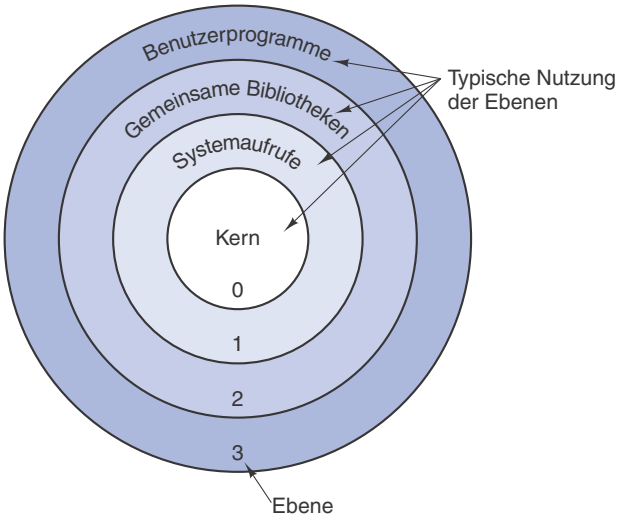

Abbildung 3.43: Schutzebenen des Pentium

Solange ein Programm sich auf die Segmente seiner eigenen Ebene beschränkt, gibt es keine Probleme. Zugriffe auf Daten höherer Ebenen sind erlaubt. Der Versuch, auf Daten niedrigerer Ebenen zuzugreifen, ist illegal und erzeugt einen Schutzfehler. Es ist möglich, Prozeduren auf anderen Ebenen aufzurufen, diese Aufrufe werden aber sehr sorgfältig kontrolliert. Der CALL-Befehl für einen ebenenübergreifenden Aufruf muss anstatt einer Adresse einen Selektor enthalten. Der Deskriptor zu diesem Selektor heißt **Call Gate** und enthält die Adresse der Prozedur, die aufgerufen werden soll. Es ist also unmöglich, einen beliebigen Punkt in einem Codesegment auf einer anderen Ebene anzuspringen. Nur die offiziellen Einsprungspunkte sind erlaubt. Das Konzept der Schutzebenen und **Call Gates** wurden von MULTICS eingeführt. Unter MULTICS hießen die Schutzebenen **Schutzringe** (*protection ring*).

▶Abbildung 3.43 zeigt eine typische Anwendung für diesen Mechanismus. Auf Ebene 0 befindet sich der Kern, der sich um Ein-/Ausgabe, Speicherverwaltung und andere kritische Aufgaben kümmert. Auf Ebene 1 sehen wir die Behandlungsroutinen für Systemaufrufe. Benutzerprogramme dürfen eine geschützte und genau definierte Liste dieser Prozeduren aufrufen. Ebene 2 enthält Bibliotheksfunktionen, die möglicherweise von vielen Programmen gleichzeitig benutzt werden. Benutzerprogramme können diese Prozeduren aufrufen und ihre Daten lesen, aber sie können sie nicht modifizieren. Auf Ebene 3, der Ebene mit dem wenigsten Schutz, befinden sich schließlich die Benutzerprogramme.

Interrupts und Sprünge in das Betriebssystem funktionieren so ähnlich wie Call Gates. Auch hier wird anstelle einer Adresse ein Deskriptor verwendet, der auf eine spezielle Prozedur zeigt, die ausgeführt werden soll. Das *Typ*-Feld in ▶Abbildung 3.40 unterscheidet zwischen Codesegmenten, Datensegmenten und den verschiedenen Arten von Call Gates.

3.8 Forschung zur Speicherverwaltung

Speicherverwaltung und besonders Paging-Algorithmen waren einmal ein fruchtbares Forschungsgebiet, das aber inzwischen größtenteils an Bedeutung verloren hat, zumindest für Systeme, die nicht für spezielle Anwendungen gebaut werden. Die meisten Systeme neigen zu einer Variante von Clock, weil es einfach zu implementieren und relativ effektiv ist. Eine Ausnahme in jüngster Zeit war der Neuentwurf der virtuellen Speicherverwaltung von 4.4 BSD (Cranor und Parulkar, 1999).

Immer noch geforscht wird dagegen auf dem Gebiet von Paging für spezielle Anwendungen und neue Arten von Systemen. Zum Beispiel Mobiltelefone und PDAs sind mittlerweile kleine PCs und viele lagern Seiten vom RAM auf die „Platte“ aus, nur dass hier die Platte ein Flash-Speicher ist, der andere Eigenschaften als eine rotierende Magnetplatte hat. Neuere Arbeiten dazu findet man bei (In et al., 2007; Joo et al., 2006; Park et al., 2004a). Park et al. (2004b) behandeln außerdem energiebewusstes Demand Paging bei mobilen Geräten.

Ein weiterer Forschungsbereich ist die Modellierung der Performanz von Paging-Strategien (Albers et al., 2002; Burton und Kelly, 2003; Cascaval et al., 2005; Panagiotou und Souza, 2006; Peserico, 2003). Ebenfalls von Interesse ist die Speicherverwaltung für Multimedia-Systeme (Dasigenis et al., 2001; Hand, 1999) und für Echtzeitsysteme (Pizlo und Vitek, 2006).

ZUSAMMENFASSUNG

In diesem Kapitel haben wir uns mit **Speicherverwaltung** befasst. Wir haben gesehen, dass die einfachsten Systeme weder Swapping noch Paging benutzen: Ein Programm, das geladen wird, bleibt bis zum Programmende im Speicher. Einige Betriebssysteme können jeweils nur ein Programm ausführen, andere unterstützen Multiprogrammierung.

Der nächste Schritt nach vorne ist **Swapping**. Swapping bedeutet, dass das Betriebssystem mehr Prozesse ausführen kann, als im Speicher Platz haben. Die Prozesse, für die kein Platz ist, werden auf die Festplatte ausgelagert. Der freie Platz im Speicher und auf der Platte kann mithilfe einer Bitmap oder einer Freibereichsliste verwaltet werden.

Moderne Computer unterstützen oft **virtuellen Speicher**. In der einfachsten Form wird der Adressraum eines Prozesses in gleich große Blöcke zerlegt, die Seiten genannt werden. Jede Seite kann in einen beliebigen Seitenrahmen im Speicher geladen werden. Es gibt viele verschiedene Seitenersetzungsalgorithmen; zu den besseren gehören Aging und WSClock.

Paging-Systeme können modelliert werden, indem man ein Programm als eine Folge von Speicherzugriffen betrachtet und diese Referenzkette mit verschiedenen Algorithmen benutzt. Mit diesen Modellen lassen sich Vorhersagen über das Paging-Verhalten machen.

Ein guter **Seitenersetzungsalgorithmus** macht noch kein gutes Paging-System. Dazu gehören auch Dinge wie eine gute Strategie für die Speicherzuteilung, eine Methode, den Arbeitsbereich eines Prozesses herauszufinden, und eine passende Seitengröße.

Segmentierung bietet Vorteile bei Datenstrukturen, die ihre Größe ändern, und vereinfacht das Binden und die gemeinsame Benutzung von Code und Daten. Außerdem ermöglicht es, verschiedene Segmente vor verschiedenen Arten von Zugriffen zu schützen. Segmentierung und Paging werden manchmal zu einem zweidimensionalen virtuellen Speicher kombiniert. MULTICS und der Intel Pentium unterstützen beide Segmentierung mit Paging.

Lösungshinweise

Übungen

1. In ▶Abbildung 3.3 enthalten das Basis- und das Limitregister den gleichen Wert: 16.384. Ist dies reiner Zufall oder sind die Inhalte immer gleich? Wenn es nur Zufall ist, warum sind es dann in diesem Beispiel dieselben Werte?

2. Ein Swapping-System entfernt Lücken aus dem Speicher durch Verdichtung. Wenn viele Lücken und Segmente zufällig über den Speicher verteilt sind und das Lesen oder Schreiben eines 32-Bit-Wortes 10 ns dauert, wie lange dauert es dann ungefähr, 128 MB Speicher zu verdichten? Der Einfachheit halber nehmen wir an, dass das Wort 0 in einer Lücke liegt und dass das letzte Wort im Speicher Daten enthält.

3. In dieser Aufgabe vergleichen wir die Speicherkosten für die Verwaltung von freiem Speicher mit einer Bitmap und mit verketteten Listen. Der Speicher ist 128 MB groß und wird in Einheiten von *n* Byte zugeteilt. Für die verkettete Liste setzen wir voraus, dass der Speicher abwechselnd aus 64 MB großen Lücken und Segmenten besteht. Außerdem nehmen wir an, dass jeder Knoten in der Liste 32 Bit für die Adresse, 16 Bit für die Länge und 16 Bit für den Zeiger auf den nächsten Knoten benötigt. Wie viel Speicher ist für jede der beiden Methoden nötig? Welche ist besser?

4. In einem Swapping-System sind folgende Lücken im Speicher, nach aufsteigenden Adressen geordnet: 10 KB, 4 KB, 20 KB, 18 KB, 7 KB, 9 KB, 12 KB und 15 KB. Welche Lücken wählen First Fit, Best Fit, Worst Fit und Next Fit jeweils aus, wenn nacheinander Segmente von 12 KB, 10 KB und 9 KB angefordert werden?

5. Berechnen Sie die virtuelle Seitennummer und den Offset für die folgenden dezimalen virtuellen Adressen, falls die Seitengröße 4 KB bzw. 8 KB ist: 20.000, 32.768, 60.000.

6. Der Intel-8086-Prozessor unterstützt keinen virtuellen Speicher. Trotzdem wurden früher Systeme verkauft, die eine unmodifizierte 8086-CPU enthielten und mit Paging arbeiteten. Können Sie sich vorstellen, wie das funktioniert haben könnte? *Hinweis*: Denken Sie an den logischen Ort der MMU.

7. Betrachten Sie das folgende C-Programm:

```
int     X[N];
int step = M;       // M ist eine vordefinierte Konstante
for (int i = 0; i < N; i += step) X[i] = X[i] + 1;
```

a. Wenn das Programm auf einer Maschine mit Seitengrößen von 4 KB und 64 Einträgen in der TLB läuft, welche Werte von M und N werden dann einen TLB-Fehler bei jedem Durchlauf der inneren Schleife verursachen?

b. Würde Ihre Antwort in (a) anders ausfallen, wenn die Schleife oft wiederholt würde? Begründen Sie Ihre Behauptung.

8. Die Größe des Festplattenplatzes für ausgelagerte Seiten ist abhängig von der maximalen Zahl von Prozessen *n*, von der Anzahl der Bytes im virtuellen Adressraum *v* und von der Größe des physischen Speichers *r*. Geben Sie eine Formel für den Speicherbedarf im ungünstigsten Fall an. Ist diese Zahl realistisch?

9. Ein Rechner hat einen 32-Bit-Adressraum und 8-KB-Seiten. Die Seitentabelle liegt komplett in der Hardware, mit einem 32-Bit-Wort pro Eintrag. Wenn ein Prozess startet, wird die Seitentabelle aus dem Arbeitsspeicher in die Hardware kopiert, was für jedes Wort 100 ns dauert. Wie groß ist der Anteil der Zeit, in der die CPU Seitentabellen lädt, wenn ein Prozess immer für 100 ms läuft (einschließlich der Zeit, in der die Seitentabelle geladen wird)?

10. Angenommen, eine Maschine hat virtuelle 48-Bit-Adressen und physische 32-Bit-Adressen.

a. Wenn eine Seite 4 KB groß ist, wie viele Einträge sind in der Seitentabelle, wenn sie nur eine Ebene hat? Warum?

b. Nehmen Sie nun an, dass dasselbe System ein TLB mit 32 Einträgen hat. Außerdem soll es ein Programm geben, dessen Befehle in eine Seite passen und das lange Festkommazahlen sequenziell aus einem Feld einliest, welches sich über Tausende von Seiten erstreckt. Wie effektiv ist der TLB in diesem Fall?

11. Betrachten Sie eine Maschine mit virtuellen 38-Bit-Adressen und physischen 32-Bit-Adressen.

a. Was ist der größte Vorteil einer mehrstufigen Seitentabelle gegenüber einer einstufigen?

b. Wie viele Bits sollten bei einer zweistufigen Seitentabelle mit 16-KB-Seiten und Einträgen der Länge 4 Byte für das obere Seitentabellenfeld reserviert werden? Und wie viele für das Feld der nächsten Ebene? Erläutern Sie Ihre Antwort.

12. Ein Computer mit 32-Bit-Adressen benutzt eine zweistufige Seitentabelle. Virtuelle Adressen werden in ein 9-Bit-Feld für die oberste Seitentabelle, 11 Bit für die zweite Seitentabelle und einen Offset unterteilt. Wie viele Seiten sind im Adressraum und wie groß sind sie?

13. Eine virtuelle 32-Bit-Adresse wird in vier Teile *a*, *b*, *c* und *d* aufgeteilt. Die ersten drei Felder sind die Indizes für eine dreistufige Seitentabelle. Das vierte Feld, *d*, ist der Offset. Hängt die Anzahl der Seiten von der Länge aller vier Felder ab? Wenn nein, welche Felder sind unwichtig?

14. Ein Computer hat 32-Bit-Adressen und 4-KB-Seiten. Das Programm und die Daten passen in die erste Seite (0–4.095). Der Stack passt in die letzte Seite. Wie viele Einträge werden für eine traditionelle (einstufige) Seitentabelle benötigt? Wie viele für eine zweistufige Tabelle, mit 10 Bit für jede Tabelle?

15. Ein Computer, dessen Prozesse 1.024 Seiten in ihrem Adressraum haben, hält seine Seitentabellen im Speicher. Der Aufwand, ein Wort aus einer Seitentabelle zu lesen, ist 5 ns. Um ihn zu verringern, hat der Computer einen TLB, der 32 Paare (virtuelle Seite, physischer Seitenrahmen) enthält und in 1 ns durchsucht werden kann. Welche Trefferrate ist nötig, um den durchschnittlichen Aufwand auf 2 ns zu reduzieren?

16. Der TLB einer VAX enthält kein *R*-Bit. Warum nicht?

17. Wie kann die Hardware für den Assoziativspeicher eines TLB implementiert werden und was sind die Folgen für die Erweiterbarkeit eines solchen Entwurfs?

18. Eine Maschine hat virtuelle 48-Bit-Adressen und physische 32-Bit-Adressen. Wie viele 8-KB-Seiten sind in der Seitentabelle?

19. Ein Computer mit 8-KB-Seiten, 256 MB Arbeitsspeicher und 64 GB virtuellem Adressraum benutzt invertierte Seitentabellen für seinen virtuellen Speicher. Wie groß sollte die Hashtabelle sein, damit die Listen in der Tabelle durchschnittlich weniger als einen Eintrag haben, vorausgesetzt, die Größe der Tabelle ist eine Zweierpotenz?

20. Ein Student in einem Compilerbau-Kurs schlägt als Projekt vor, einen Compiler zu schreiben, der eine Liste von Seitenreferenzen erzeugt, die benutzt werden kann, um den optimalen Seitenersetzungsalgorithmus zu implementieren. Ist das möglich? Begründen Sie Ihre Antwort. Gibt es eine Möglichkeit, das Paging zur Laufzeit effizienter zu machen?

21. Nehmen Sie an, die Folge der virtuellen Seitenzugriffe besteht aus einer langen Seitenreferenzkette, die immer wiederholt wird und der hin und wieder eine zufällige Seitenreferenz folgt. Zum Beispiel besteht die Folge 0, 1, ..., 511, 431, 0, 1, ..., 511, 332, 0, 1, ... aus Wiederholungen der Folge 0, 1, ..., 511, gefolgt von einer zufälligen Referenz auf die Seiten 431 und 332.

a. Warum sind die Standard-Seitenersetzungsalgorithmen (LRU, FIFO, Clock) bei der Bearbeitung dieser Sequenz nicht effizient, wenn die Seitenzuteilung kleiner als die Länge der Folge ist?

b. Nehmen Sie nun an, dass diesem Programm 500 Seitenrahmen zugeteilt sind. Beschreiben Sie ein Modell zur Seitenersetzung, das besser als LRU, FIFO oder Clock funktioniert.

22. Wie viele Seitenfehler erzeugt der FIFO-Algorithmus mit acht Seiten und vier Seitenrahmen für die Referenzkette 0172327103, wenn die Seitenrahmen zu Beginn leer sind? Wiederholen Sie die Aufgabe für LRU.

23. Betrachten Sie die Seitenfolge aus ▶Abbildung 3.15(b). Angenommen, die *R*-Bits für die Seiten *B* bis *A* sind 11011011. Welche Seite würde Second Chance entfernen?

24. Ein kleiner Computer hat vier Seitenrahmen. Im ersten Timerintervall sind die *R*-Bits 0111 (Seite 0 ist 0, der Rest ist 1). In den nächsten Intervallen sind die Werte 1011, 1010, 1101, 0010, 1010, 1100 und 0001. Geben Sie die Werte für die vier 8-Bit-Zähler eines Aging-Algorithmus nach dem letzten Intervall an.

25. Geben Sie ein einfaches Beispiel einer Seitenreferenzfolge an, bei dem die erste Seite, die zur Ersetzung ausgewählt wird, für Clock und LRU unterschiedlich ist. Nehmen Sie an, dass einem Prozess drei Rahmen zugeteilt sind und dass die Referenzkette Seitennummern aus der Menge 0, 1, 2, 3 enthält.

26. In ▶Abbildung 3.21(c) zeigt der Zeiger des WSClock-Algorithmus auf eine Seite mit $R = 0$. Wird die Seite entfernt, falls $\tau = 400$ ist? Was ist bei $\tau = 1.000$?

27. Eine Festplatte hat eine durchschnittliche Such- und Rotationszeit von jeweils 10 ms und eine Spurgröße von 32 KB. Wie lange dauert es, ein 64-KB-Programm zu laden, bei einer Seitengröße von

a. 2 KB?

b. 4 KB?

Die Seiten sind zufällig über die Platte verstreut und die Anzahl der Zylinder ist so groß, dass die Wahrscheinlichkeit, dass zwei Seiten auf demselben Zylinder liegen, vernachlässigbar ist.

28. Ein Computer hat vier Seitenrahmen. Die Tabelle zeigt für jede Seite die Ladezeit, die Zeit des letzten Zugriffes sowie die *R*- und *M*-Bits. Die Zeiten sind jeweils in Timerintervallen angegeben.

Seite	geladen	letzter Zugriff	R	M
0	126	280	1	0
1	230	265	0	1
2	140	270	0	0
3	110	285	1	1

a. Welche Seite ersetzt NRU?

b. Welche Seite ersetzt FIFO?

c. Welche Seite ersetzt LRO?

d. Welche Seite ersetzt Second Chance?

29. Gegeben sei das folgende zweidimensionale Feld:

```
int   X[64][64];
```

Nehmen Sie an, dass ein System vier Seitenrahmen hat und jeder Rahmen besteht aus 128 Wörtern (eine Zahl belegt ein Wort). Programme, die das *X*-Feld manipulieren, passen genau in eine Seite und besetzen immer Seite 0. Die Daten werden von den anderen drei Rahmen ein- und ausgelagert. Das *X*-Feld wird aufsteigend nach Zeilen gespeichert (d.h., *X*[0][1] kommt hinter *X*[0][0] im Speicher). Welches der beiden unten gezeigten Codefragmente wird die wenigsten Seitenfehler erzeugen? Begründen Sie Ihre Lösung und berechnen Sie die Gesamtzahl an Seitenfehlern.

Fragment A

```
for (int j = 0; j < 64; j++)
    for (int i = 0; i < 64; i++) X[i][j] = 0;
```

Fragment B

```
for (int i = 0; i < 64; i++)
    for (int j = 0; j < 64; j++) X[i][j] = 0;
```

30. Eine der ersten Timesharing-Maschinen, die PDP-1, hatte einen Speicher mit 4 KB 18-Bit-Wörtern. Es war jeweils nur ein Prozess im Speicher. Wenn der Scheduler einen anderen Prozess laufen lassen wollte, wurde der Prozess im Speicher auf eine Magnettrommel geschrieben, auf deren Oberfläche 4 KB 18-Bit-Wörter Platz hatten. Die Trommel konnte an jeder beliebigen Stelle anfangen zu lesen oder zu schreiben, nicht nur bei Wort 0. Können Sie sich vorstellen, warum diese Trommel benutzt wurde?

31. Ein Computer stellt jedem Prozess einen Adressraum von 64 KB zur Verfügung, aufgeteilt in 4-KB-Seiten. Ein Programm hat 32.768 Byte Programmcode, 16.386 Byte Daten und 15.870 Byte Stack. Passt dieses Programm in den Adressraum? Würde es passen, wenn die Seiten 512 Byte groß wären? Denken Sie daran, dass eine Seite nicht zwei Teile von verschiedenen Segmenten enthalten kann.

32. Kann eine Seite gleichzeitig zu zwei verschiedenen Arbeitsbereichen gehören? Warum (nicht)?

33. Es wurde beobachtet, dass die Zahl der ausgeführten Befehle zwischen zwei Seitenfehlern direkt proportional zur Anzahl der Seitenrahmen ist, die das Programm belegt. Wenn sich der verfügbare Speicher verdoppelt, verdoppelt sich auch die Zeit zwischen den Seitenfehlern. Angenommen, ein normaler Befehl dauert eine Mikrosekunde, aber wenn ein Seitenfehler auftritt, dauert er 2.001 Mikrosekunden (d.h. 2 Millisekunden für den Seitenfehler). Wie lange würde ein Programm laufen, das nach 60 Sekunden beendet ist und während dieser Zeit 15.000 Seitenfehler erzeugt, wenn es doppelt so viel Speicher zur Verfügung hätte?

34. Ein Team von Betriebssystementwicklern der Spar Computer GmbH will in ihrem neuen Betriebssystem Hintergrundspeicher einsparen. Der Oberguru macht den Vorschlag, den Programmtext gar nicht erst im Swapping-Segment zu speichern, sondern ihn bei Bedarf direkt aus der Programmdatei einzulagern. Ist das für den Programmtext möglich und wenn ja, unter welchen Bedingungen? Wie sieht es mit den Programmdaten aus?

35. Ein Maschinenbefehl, der ein 32-Bit-Wort aus dem Speicher in ein Register lädt, enthält die Speicheradresse des Wortes, das geladen werden soll. Wie viele Seitenfehler kann der Befehl maximal auslösen?

36. Wenn, wie in MULTICS, Segmentierung und Paging benutzt werden, muss zunächst der Segmentdeskriptor geladen werden, dann der Seitendeskriptor. Funktioniert der TLB so auch mit einer zweistufigen Suche?

37. Wir betrachten ein Programm, das aus zwei Segmenten besteht (siehe Tabelle). Die Befehle befinden sich in Segment 0 und die Daten zum Lesen und Schreiben sind in Segment 1. Segment 0 ist vor dem Lesen und Ausführen geschützt, Segment 1 vor Lese-/Schreibzugriffen. Das Speichersystem ist ein virtueller Speicher mit Demand Paging und virtuellen Adressen, die eine 4-Bit-Seitennummer und einen 10-Bit-Offset haben. Die Seitentabellen und Schutz sind wie folgt (alle Zahlen in der Tabelle sind Dezimalzahlen):

Segment 0		Segment 1	
Lesen/Ausführen		Lesen/Schreiben	
Virtuelle Seite	Seitenrahmen	Virtuelle Seite	Seitenrahmen
0	2	0	auf der Platte
1	auf der Platte	1	14
2	11	2	9
3	5	3	6
4	auf der Platte	4	auf der Platte
5	auf der Platte	5	13
6	4	6	8
7	3	7	12

Geben Sie für jeden der folgenden Fälle entweder eine reale (aktuelle) Speicheradresse an, die aus der dynamischen Adressenübersetzung resultiert, oder identifizieren Sie den Typ des aufgetretenen Fehlers (entweder Seitenfehler oder Schutzverletzung).

a. Hole von Segment 1, Seite 1, Offset 3.

b. Speichere in Segment 0, Seite 0, Offset 16.

c. Hole von Segment 1, Seite 4, Offset 28.

d. Springe zu Segment 1, Seite 3, Offset 32.

38. Können Sie sich eine Situation vorstellen, in der die Unterstützung von virtuellem Speicher eine schlechte Idee ist? Was könnte damit gewonnen werden, wenn man auf virtuellen Speicher verzichtet? Warum?

39. Finden Sie die Größenverteilung der ausführbaren Dateien auf einem beliebigen Computer heraus. Berechnen Sie den Mittel- und den Medianwert. Auf einem Windows-System suchen Sie alle Dateien mit den Endungen .dll und .exe. Auf einem Unix-System verwenden Sie alle Dateien in den Verzeichnissen */bin*, */usr/bin* und */local/bin*, die keine Skripte sind (oder verwenden Sie das Programm *file*, um alle ausführbaren Dateien zu finden). Berechnen Sie die optimale Seitengröße für diesen Computer. Berücksichtigen Sie dabei nur die Größe des Programmcodes (nicht die Daten). Ziehen Sie die interne Fragmentierung und die Größe der Seitentabelle in Betracht. Machen Sie eine vernünftige Annahme über die Größe eines Eintrages in der Seitentabelle. Gehen Sie davon aus, dass alle Programme mit der gleichen Wahrscheinlichkeit benutzt werden und deshalb gleich gewichtet werden sollten.

40. Kleine Programme unter MS-DOS können zu *.COM*-Dateien übersetzt werden. Solche Dateien werden immer an Adresse 0x100 in ein einziges Segment geladen, das für Code, Daten und Stack gleichzeitig benutzt wird. Bei Befehlen zur Ablaufsteuerung (z.B. `JMP` oder `CALL`) oder Befehlen, die auf Daten an statischen Adressen zugreifen, wird die Adresse in den Objektcode hinein übersetzt. Schreiben Sie ein Programm, das eine *.COM*-Datei an jede beliebige Adresse relozieren kann. Das Programm muss den Programmcode nach den Opcodes von Befehlen durchsuchen, die feste Speicheradressen verwenden, und diese Befehle dann an den neuen Speicherbereich anpassen. Die benötigten Opcodes finden Sie in einem Lehrbuch über Assemblerprogrammierung. Diese Aufgabe ohne zusätzliche Informationen perfekt zu lösen, ist übrigens unmöglich, weil Daten und Operanden im Programm die Werte von Opcodes annehmen können.

41. Schreiben Sie ein Programm zur Simulation von Paging mithilfe des Aging-Algorithmus. Die Anzahl der Seitenrahmen soll als Parameter übergeben werden. Die Folge der Speicherzugriffe sollte aus einer Datei eingelesen werden. Stellen Sie für eine gegebene Eingabedatei die Anzahl der Seitenfehler pro 1.000 Speicherzugriffen als eine Funktion der Anzahl der verfügbaren Seitenrahmen dar.

42. Schreiben Sie ein Programm, das den Einfluss von TLB-Fehlern auf die tatsächliche Speicherzugriffszeit demonstriert, indem die Zeit pro Zugriff gemessen wird, die benötigt wird, um ein langes Feld zu durchlaufen.

a. Erläutern Sie die Hauptkonzepte Ihres Programms. Welche Aussagen erwarten Sie aus der Ausgabe für eine reale virtuelle Speicherarchitektur herleiten zu können?

b. Lassen Sie Ihr Programm auf einem Computer laufen und beschreiben Sie, wie gut die tatsächlichen Daten Ihren Erwartungen entsprechen.

c. Wiederholen Sie Teil b), diesmal aber mit einem älteren Computer, der eine andere Architektur besitzt. Erklären Sie die Hauptunterschiede in der Ausgabe.

43. Schreiben Sie ein Programm, das den Unterschied zwischen der Benutzung einer lokalen und einer globalen Seitenersetzungsstrategie für den einfachen Fall von zwei Prozessen demonstriert. Sie werden eine Routine benötigen, die eine Seitenreferenzfolge erzeugen kann, welche auf einem statistischen Modell basiert. Dieses Modell hat N Zustände, die von 0 bis $N-1$ durchnummeriert sind und die jeweils einen der möglichen Seitenzugriffe repräsentieren. Jedem Zustand i ist ein Wert p_i zugeordnet, der die Wahrscheinlichkeit darstellt, mit der der nächste Zugriff auf dieselbe Seite erfolgt. Andernfalls wird beim nächsten Seitenzugriff jede andere Seite mit gleich hoher Wahrscheinlichkeit angesprochen.

a. Zeigen Sie, dass sich die Routine zur Erzeugung der Seitenreferenzfolge für kleine N richtig verhält.

b. Berechnen Sie die Seitenfehlerrate für ein kleines Beispiel, in dem es einen Prozess und eine festgelegte Anzahl von Seitenrahmen gibt. Erklären Sie, warum dieses Verhalten korrekt ist.

c. Wiederholen Sie Teil b) mit zwei Prozessen mit unabhängigen Seitenreferenzfolgen und mit doppelt so vielen Seitenrahmen wie in Teil b).

d. Wiederholen Sie Teil c), aber benutzen Sie diesmal eine lokale Strategie anstelle einer globalen Strategie. Stellen Sie außerdem die Seitenfehlerrate pro Prozess der Fehlerrate der lokalen Strategie gegenüber.