Wenn der Arbeitsspeicher zu klein ist...

... dann muß man gleichsam außen anschuhen - soll heißen: die Daten und Programme auf Massenspeichern unterbringen.

Fachbegriff: virtueller Speicher.

Virtuell = scheinbar. Total scheinbar geht es aber nicht - die zu unterstützende Speicherkapazität muß irgendwo vorhanden sein.

Die Ideallösung:

- Verbund aus Arbeits- und Massenspeicher,
- Programmschnittstelle: Maschinenbefehle für Speicherzugriffe (Laden, Speichern, Operationen mit Speicheroperanden). Soll heißen: der Programmierer greift einfach auf den Speicher zu, ohne sich besonders darum zu kümmern.
- Zugriffszeit: wie Arbeitsspeicher,
- Speicherkapazität: gesamter unterstützter Adreßraum,
- Kosten: wie Massenspeicher.

In der Praxis geht es aber nicht ganz so ideal zu...

1. Einfachlösungen ohne Vorkehrungen in der Hardware

Wenn es in der Hardware nichts gibt, muß der Programmierer halt aufpassen.

Programmschnittstelle: Einteilung in handliche Häppchen, auf die explizit zugrgirffen wird (besondere Zugriffsfunktionen).

Naheliegend: Einteilung gemäß der gleichsam naturgegebenen Struktur der Anwendungen (in einzelne Programme, Datenbereiche usw.).

Ganz einfach: Transientbereiche (Overlay-Prinzip)
Der Arbeitsspeicher wird entsprechend aufgeteilt. Was unbedingt anwesend sein muß, wird fest eingelagert (residente Programme und Datenbereiche, Stack(s), Heap(s)). Alles andere wird in Form von Dateien auf den Massenspeichern gehalten (transiente Programme und Datenbereiche). Um die Programme ausführen und die Daten bearbeiten zu können, werden im Arbeitsspeicher feste Bereiche, die sog. Transientbereiche, reserviert. Größe: jeweils gemäß dem größten einzulagerndem Programm oder Datenbereich

Einfachste Nutzungsweise:
Zu Fuß. Der Programmierer holt ein Programm oder einen Datenbereich dann, wenn er es/ihn braucht.

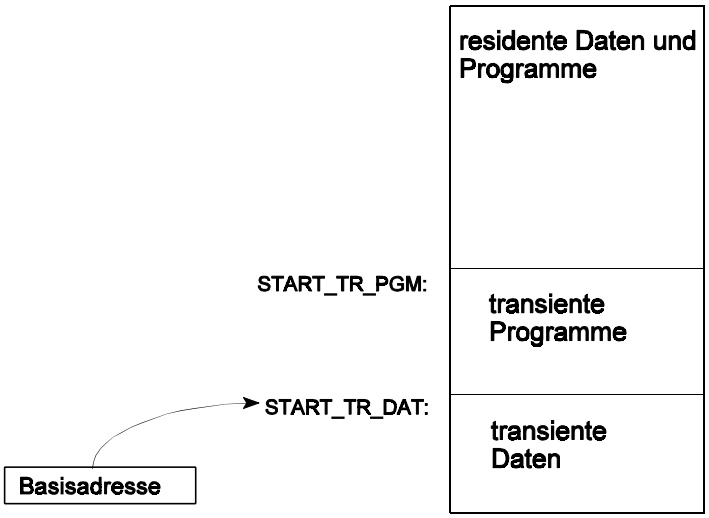

Abb. 1 Transientbereiche

Als Programmschnittstelle genügen 3 Funktionen:

- Holen Programm (FETCH),
- Holen Datenbereich (GET),
- Speichern Datenbereich (PUT).

Parameter:
Dateiname, ggf. (falls nicht von vornherein fest) Anfangsadresse des Transientbereichs. Die Funktionen können leicht selbst ausprogrammiert werden (werden aber typischerweise vom System bereitgestellt).

Ausführen eines transienten Programms:

FETCH	*Programmname*
GOTO	*Startadresse*

Besonderheiten transienter Programme:

Fall 1: Code ist verschieblich (relocatable). Kein Problem. Läuft überall, wo man ihn hinlädt.

Fall 2: Code ist nicht verschieblich (non-relocatable). Transientbereich mit fester Anfangsadresse. Programm mit dieser Anfangsadresse assemblieren. Fallbeispiel: diverse Mikrocontroller.

Fall 3: Code ist nicht verschieblich (non-relocatable). Transientbereich mit variabler Anfangsadresse. Programm mit Anfangsadresse 0 assemblieren und sich etwas einfallen lassen, nämlich:

- die nichtverschieblichen Adreßbezüge beim Laden auflösen. Erfordert fest formatierte Angaben, um diese Adressen aufzufinden. Fallbeispiel: .EXE- und DLL-Dateien (mit .EXE-Header)*).
- die nichtverschieblichen Adreßbezüge beim ersten Ausführen auflösen. Erfordert, daß transiente Programme entsprechend geschrieben werden (z. B. jeder Sprung, jeder Speicherzugriff als Macro). Beim Ausführen wird der Macro durch den jeweiligen Originalbefehl mit entsprechend berechneter Adresse ersetzt. Fallbeispiel: Tricksen in der Mikrocontroller-Szene**).

*): das Laden von DOS- oder Windows-Programmen ist im Grunde nichts anderes als ein Laden in einen großen, sich dynamisch ändernden Transientbereich...

**): ob sich das Tricksen heutzutage noch lohnt? - Gelegentlich schon. Beispiel: kleiner Mikrocontroller (mit kleinem Adreßraum) + vergleichsweise kleiner SRAM + gigantischer NAND-Flash (Mega...Gigabytes, aber sequentieller Zugriff).

Datenadressierung:
Z. B. über Basisregister, das auf den Anfang des betreffenden Transientbereichs zeigt. Wurde ein Datenbereich geändert, ist er vor dem Überladen des Transientbereichs wieder zurückzuspeichern (PUT-Funktion).

Sind mehrfache Overlays möglich?
Soll heißen: Transientprogramm A ruft Transientprogramm B usw. - wir haben aber nur einen einzigen Transientbereich.

1. Fall: im gerufenem Programm B gibt es keine Rückkehr zur Aufrufstelle im Programm A: kein Problem.

2. Fall: Programm A ruft Programm B im Sinne eines Unterprogramms (mit Rückkehr). Auch das bekommt man zum Laufen, z. B. so:

```
        PUSH A-ident      -- Bezeichnung (identifier) des aktuellen Transient-
                          --  programms retten
        LDA R1, next      -- Fortsetzungsadresse retten
        PUSH R1
        FETCH B           -- transientes UP holen und ausführen
        GOTO b_start
next:                     -- Fortsetzung nach transientem UP-Ruf

        Rückkehr aus B nach A:
            POP A-ident       -- Bezeichnung des rufenden Transient-
                              -- programms wiederherstellen
            FETCH A
            POP R1
            BRA R1            -- Verzeigen mit geretteter Fortsetzungsadresse
```

Komfortabler: segmentbezogener oder objektorientierter Zugriff

Segmentierung
Ein Segment ist ein zusammenhängender Speicherbereich jeweils festgelegter Größe. Segmente werden als Ganzheiten verwaltet. Dem Programmierer erscheint jedes Segment als ein eigener Speicher, der mit Adresse 0 beginnt (Segmentanfang) und eine jeweils bestimmte Größe hat (Segmentlänge).

Einrichten von Segmenten für die einzelnen zu speichernden Informationsstrukturen
Erfolgt typischerweise von Hand (bzw. vom Laufzeitsytem) auf Grundlage der Nutzungsweise. So liegt es nahe, Datensegmente, Stacksegmente und Programmsegmente vorzusehen.

Jedes Segment für sich muß handhabbar sein. Soll heißen: es muß in den Arbeitsspeicher passen - und zwar zusammen mit den Programmen und Daten, die zum Betrieb des Systems benötigt werden.

Segmente werden mit Segmentdeskriptoren beschrieben, die in Segmenttabellen zusammengefaß sind.

Um auf ein Segment zuzugreifen, ist es über einen Segmentselektor anzusprechen. Hierbei muß sichergestellt werden (durch Systemsoftware oder Hardware/Mikroprogramm (IA-32)), daß das Segment im Arbeitsspeicher anwesend ist.

Objektorientierung
Alle Programme und Datenbereiche werden als Objekte aufgefaßt. Objekte sind Behälter für Information, die jeweils als Ganzheit behandelt werden. Jedes Objekt hat eine Namen bzw. - zur Laufzeit - eine binär codierte Ordinalzahl, die das jeweilige Objekt aus der Menge aller Objekte auswählt.

Die Programme beziehen sich auf Objekte.

Die Objekte werden durch Objektdeskriptoren beschrieben, die in Objekttabellen zusammengefaßt sind:

- ist das Objekt resident oder transient und anwesend, enthält der Deskriptor einen Zeiger auf den betreffenden Bereich im Arbeitsspeicher (Anfangsadresse + Länge),
- ist das Objekt nicht anwesend, enthält der Deskriptor eine Positionsangabe für den Massenspeicher. (Am besten: gleich einen Zeiger auf den ersten physischen Sektor (Beispiel: Burroughs 5500 u. folgende).)

Die Objektverwaltung sorgt automatisch dafür, daß die jeweiligen Objekte in den Speicher geschafft oder ausgelagert werden.

Segmentierter und objektorientierter virtueller Speicher
Beides ist im Grunde das gleiche, der Unterschied liegt lediglich in der Audrucksweise und in der Programmierphilosophie:

- Segmentierung: Gegenstand der Seicherverwaltung sind Speicherbereiche. Segmente sind ein Organisations-Hilfsmittel, um den Speicheradreßraum in handhabbare Stücke zu zerlegen. Variable (aus Sicht der Programme) werden auf herkömmliche Weise adressiert (Adreßauflösung zur Compilierzeit). Der Anwendungsprogrammierer hat mit der Segmentierung kaum (zumeist. gar nichts) zu tun. Beispiel: IA-32 (gemäß ursprünglicher Nutzungsabsicht).
- Objektorientierung: Variable werden als Objekte angesehen, also über ihre Ordinalzahlen angesprochen (Adreßauflösung zur Laufzeit). Beispiel: Burroughs 5500 und folgende. Könnte auf Grundlage IA-32 Segmentierung implementiert werden (macht aber niemand)...

Das Hauptproblem: wegen der unterschiedlichen Größe der Objekte bzw. Segmente kommt es zur Fragmentierung des Speichers. Diese Form der Speicherverwaltung funktioniert letzten Endes nur mit Garbage Collection (die Zeit frißt...).

Einstufiges Zugriffsschema
Die Variablen im Programm bezeichnen unmittelbar die jeweiligen Objekte. Das entspricht einer Programmumgebung mit ausschließlich globalen Variablen.

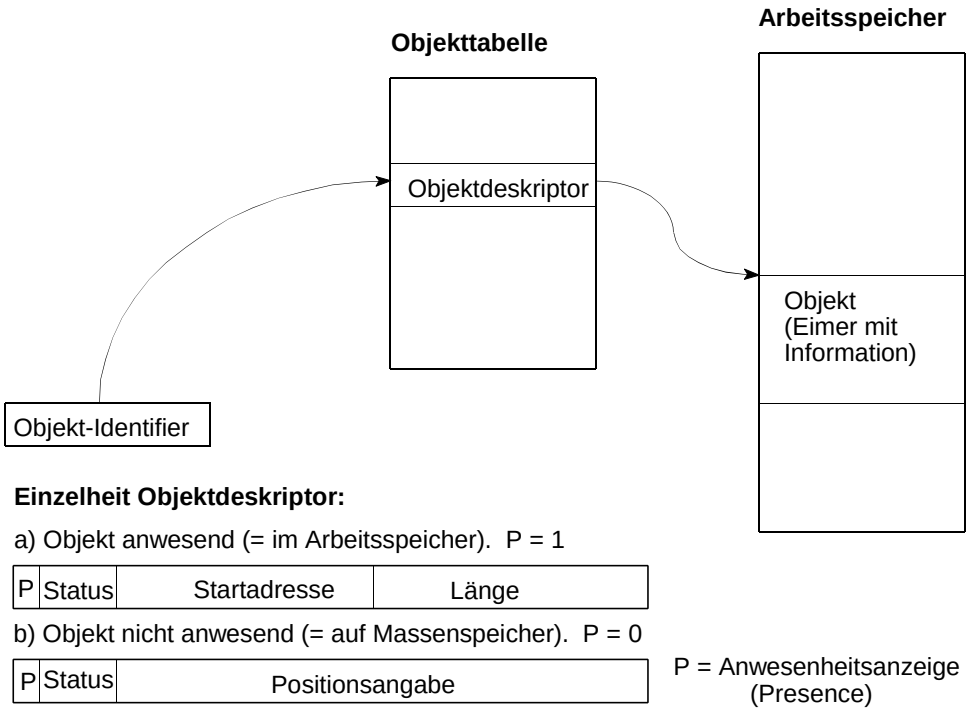

Abb.2 Objektorientierte Zugriffsweise (1). Einstufiges Zugriffsschema

Verfeinerungen:

- Unterscheidung zwischen globalen und lokalen Objekten,
- Unterteilung der Objektmenge in mehrere Regionen.

Beispiel: Segmentierung bei IA-32. Kann zu dieser Zugriffsweise ausgenutzt werden.

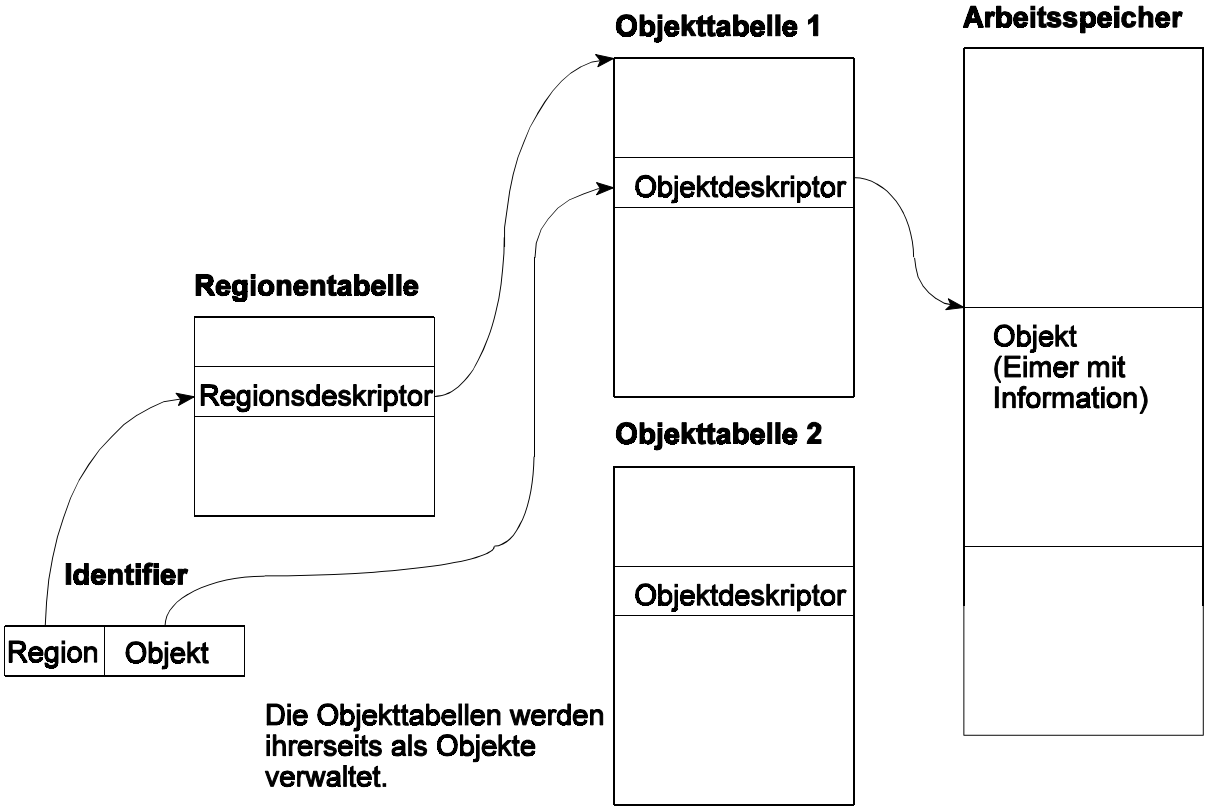

Abb. 3 Objektorientierte Zugriffsweise (2). Über eine weitere Beschreibungsebene (die Regionen) wird die Objektbeschreibung beherrschbar (Zerlegung in mehrere Objekttabellen, die ihrerseits als Objekte behandelt werden)

Zweistufiges Zugriffsschema (Capability Based Addressing)
Die Variablen im Programm bezeichnen Eintrittspunkte in eine zur Laufzeit aktuelle zugriffstabelle (Capability Table, Access Reference Table), die die Objekt-Identfier enthält, mit denen wiederum die Objekttabelle aufgesucht wird. Ermöglicht es, alle Bezüge zur Laufzeit aufzulösen. Isoliert die Objektmenge von den Programmen und erlaubt feinfühlige Zuweisung von Schutzrechten (bis auf die einzelne Variable genau). Beispiele: iAPX-432, Ada-Laufzeitsysteme.

Die Zugriffstabelle entspricht praktisch einem Stack Frame mit den Objekt-Identifiern der aktuellen lokalen Variablen. Anders herum: man braucht das zweistufige Schema, um bei Objektorientierung bis hertunter auf die einzelne Variable (jede Variable ist ein Objekt; Beispiele: iAPX-432, AS/400) auch lokale Variable (also ein C/Unix entsprechendes Aufrufschema) zu unterstützen.

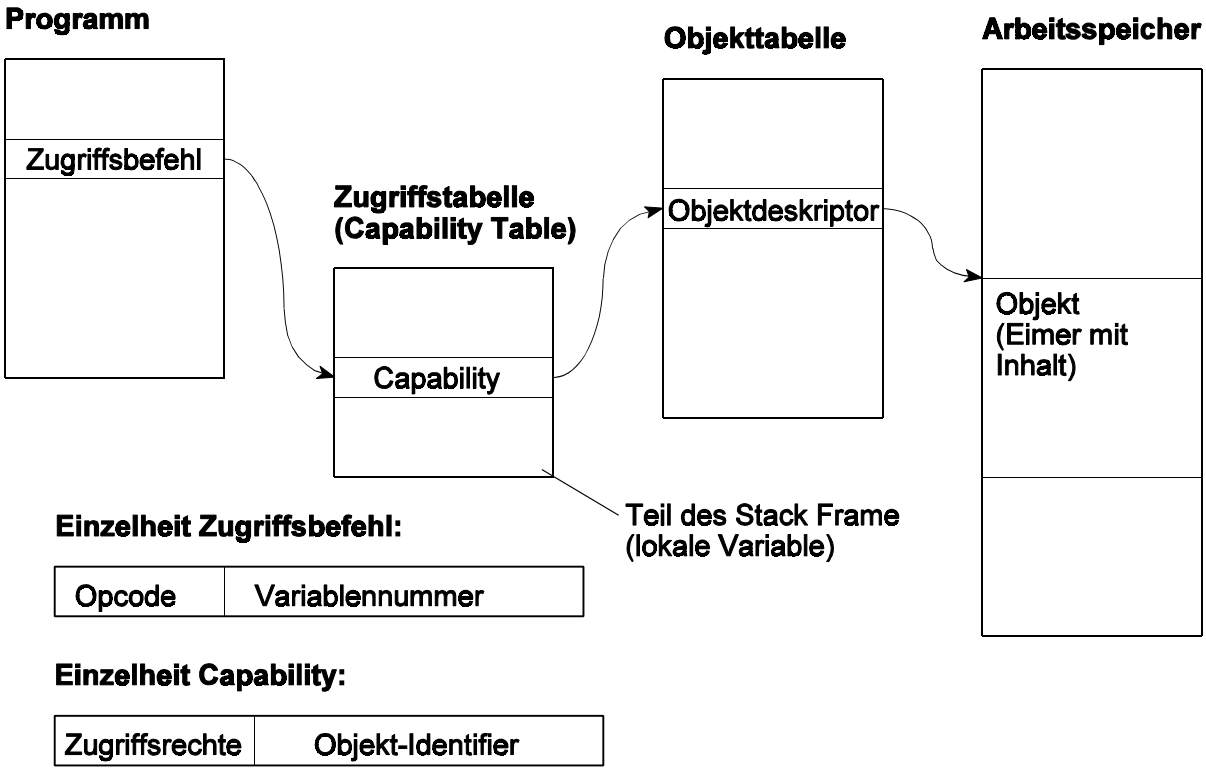

Abb. 4 Zweistufiger objektorienterter Zugriff (Capability Based Addressing)

Unterstützung in der Hardware:

- Präsenzbits in den Deskriptoren,
- Unterstützung/Beschleunigung der Tabellenzugriffe (entsprechende Datenwege, Rechenschaltungen (Addierer, Shifter, Vergleicher), hardwareseitiges Sequencing, mikroprogrammseitige Unterstützung, Spezialbefehle),
- Deskriptor-Caches.

Trotzdem ist es langsam... Der einzige Ausweg: auf der Hardwareseite Klotzen statt Kleckern (letzteres u. a. ein wesentlicher Grund für die Erfolglosigkeit des iAPX-432).

Als System erfolgreich: AS/400. Aber - sicherlich vor allem maßgebend: nur schlüsselfertige Lösungen für einen begrenzten Anwendungsbereich (niemand spielt COUNTERSTRIKE auf AS/400, AS/400 muß sich nicht den Tests der PC-Zeitschriften stellen usw.) + hohe technische Qualität.

2. Der seitenorientierte virtuelle Speicher

Die Speicherverwaltung nimmt auf den Speicherinhalt - also auf Programme, Daten usw. - überhaupt keine Rücksicht, sondern verwaltet den Adreßraum in Form gleichgroßer Seiten (Pages).

Wir haben zwei Adressen:

- die virtuelle Adresse. Adressiert den virtuellen Speicher aus Sicht des Programms: effektive Adresse + (ggf.) zusätzliche Anteile der Speicherverwaltung (Adreßraumauswahl, Segmentierung usw.).
- die physische Adresse. Adressiert den Arbeitsspeicher. Gelangt unmittelbar zu den Speicherschaltkreisen.

Adreßumsetzung (Address Translation):
bildet jede virtuelle Adresse auf eine physische ab.

Seiten und Seitenrahmen (Pages, Page Frames)
Beides sind Behälter für Speicherinhalte. Der virtuelle Speicher ist in Seiten eingeteilt, der physische Arbeitsspeicher in Seitenrahmen. Jeder Seitenrahmen kann eine Seite aufnehmen. Seite und Seitenrahmen sind jeweils gleich groß und an integralen Adressen angeordnet.

Die grundsätzlichen Aufgaben der Speicherverwaltung:

- die Adreßumsetzung (virtuelle Adresse auf physische Adrese),
- das Ein- und Auslagern von Seiten (Page Swapping)

Die Byteadresse innerhalb der Seite bzw. des Seitenrahmens ist gleich und wird deshalb bei der Adreßumsetzung nicht berücksichtigt. Das betrifft bei einer Seitengröße von n Bytes die ld n niedrigstwertigen Adreßbits. Umzusetzen ist also:

- der verbleibende höchstwertige Teil der virtuellen Adresse - die Seitenadresse (Ordinalzahl (laufende Nummer) der Seite; Virtual Page Number VPN)

in den

- verbleibenden höchstwertigen Teil der physischen Adresse - die Seitenrahmenadresse (Ordinalzahl (laufende Nummer) des Seitenrahmens: Page Frame Number PFN oder 4Physical Page Number PPN).

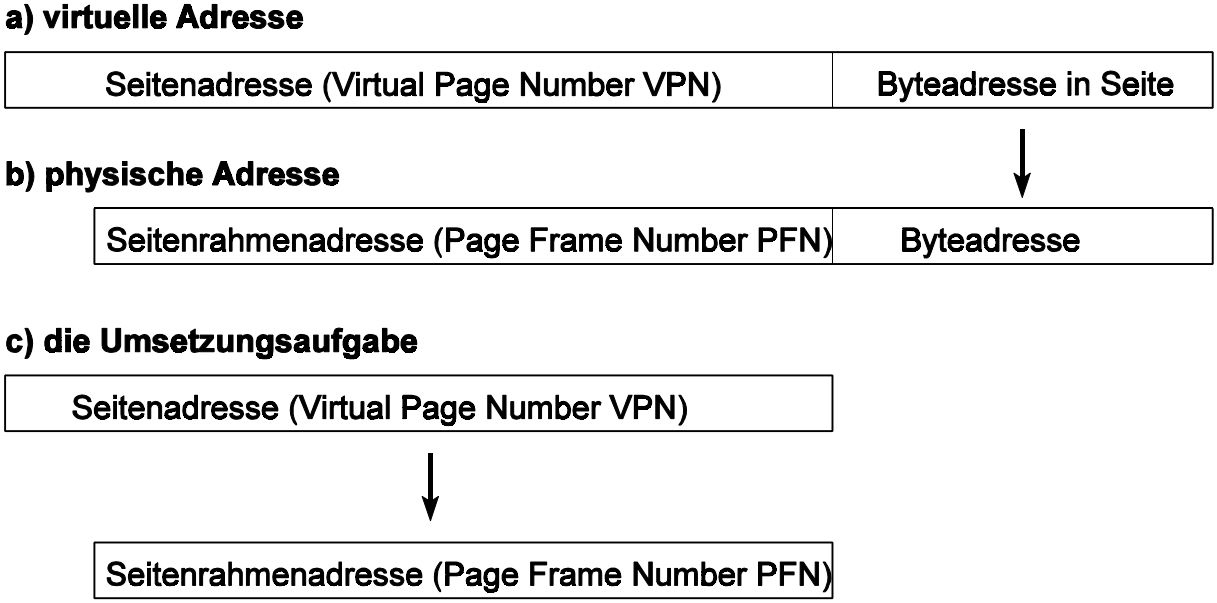

Abb. 5 Adressen und Adreßumsetzung

Prinzipien der Adreßumsetzung:

- über adressierbare Umsetzungstabellen (werden stufenweise mit Teilen der VPN adressiert),
- über assoziative Umsetzungstabellen (mit Wertepaaren VPN - PFN),
- über Umsetzungstabellen, die nach einem Hash-Verfahren adressiert werden.

Was noch in den Tabelleneinträgen steht
Im Rahmen der Tabelleneinträge werden typischerweise ergänzende Angaben gespeichert, die folgende funktionen haben:

- Speicherschutz,
- Unterstützung des Seitenaustauschs (Page Swapping),
- Unterstützung des Caching bzw. der Cache-Kohärenz.

Ohne Unterstützung durch die Hardware ist der seitenorientierte virtuelle Speicher nicht praktikabel.

Was sein muß, muß sein:
In letzter Konsequenz braucht jede Seite eine Umsetzungsangabe. Länge der virtuellen Adresse = V, Länge der Seite = P.

$$\text{Anzahl der Umsetzungsangaben} = 2^{V - ld\ P}$$

Beispiel: virtuelle Adresse 32 Bits, Seitenlänge 4k. Erforderlich sind 1 M

Umsetzungsangaben. Die - dem Prinzip nach - einfachste Realisierung ist eine Umsetzungstabelle mit 1 M Einträgen zu 4 Bytes (20 Bits für die Seitenrahmenadresse PFN, Rest für Steuer- und Schutzzwecke), also insgesamt 4 MBytes. Ds ist entschieden zuviel, um im Arbeitsspeicher gehalten zu werden (zumindest war es zu jener Zeit so, als solche Lösungen erfunden wurden).

Der Ausweg: ein mehrstufiges Tabellenschema, in dem jede Stufe einen Teil der VPN umsetzt.

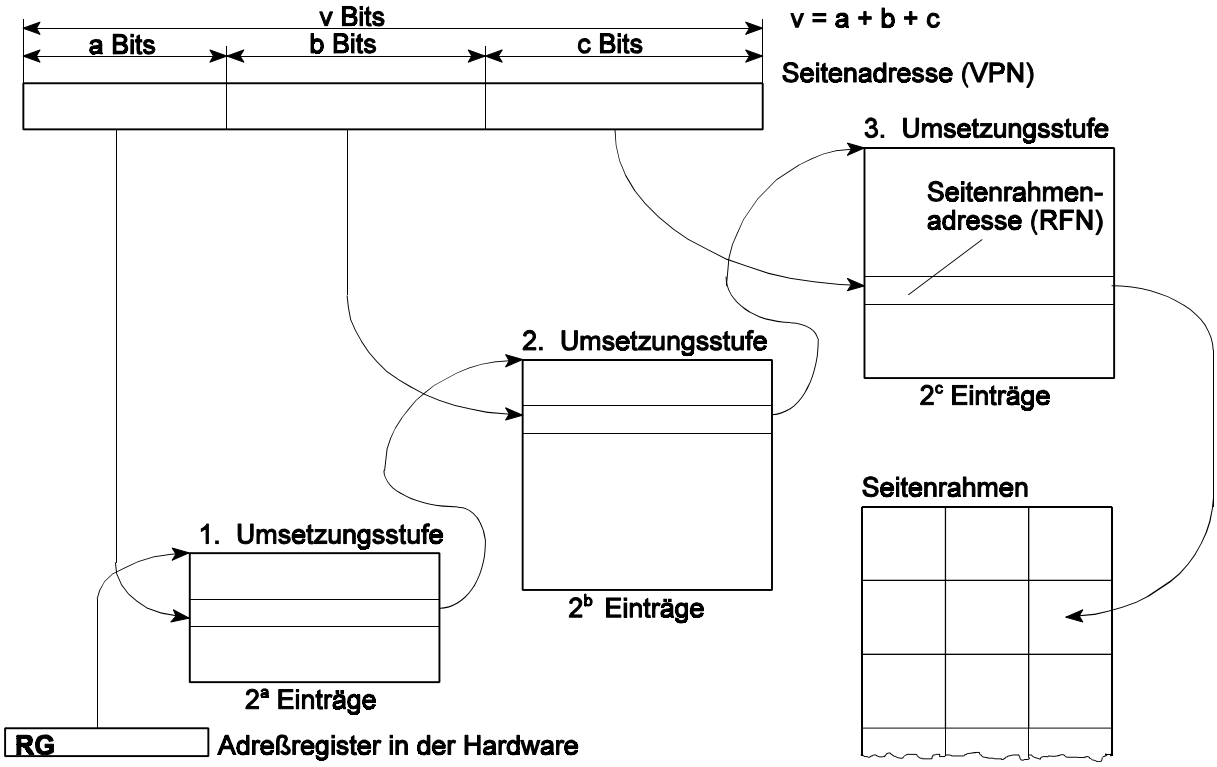

Abb. 6 Adreßumsetzung über adressierbare Umsetzungstabellen. Hier am Beispiel einer dreistufigen Umsetzung

Erklärung:
Um n Bits der Seitzenadresse (VPN) umzusetzen, braucht man Tabellen mit 2^n Einträgen. Die erste Stufe, die a Adreßbits umsetzt, umfaßt eine einzige Tabelle mit 2^a Einträgen, die zweite Stufe, die b Adreßbits umsetzt, umfaßt 2^a Tabellen mit jeweils 2^b Einträgen usw. Die letzte Stufe enthält in ihren Einträgen die jeweilige Seitenrahmenadresse (PFN). Es sind alle Bits der VPN umzusetzen. Stufenzahl und Tabellengrößen sind Erfahrungssache. Typischerweise richtet man es so ein, daß die einzelne Tabelle so groß ist wie eine Seite, so daß die Tabellen vion der zweiten Umsetzungsstufe an ebenso wie die gewöhnlichen Seiten der Auslagerung unterworfen und auf einem Massenspeicher gehalten werden können. Die Tabelle der ersten Umsetzungsstufe muß stets im Arbeitsspeicher stehen.

Adressierung der ersten Umsetzungsstufe:
Über ein Adreßregister in der Hardware (IA-32: CR03).

Was mindestens unterstützt werden muß:
Die Adreßumsetzung für wenigstens einen Teil der Seiten, die sich in Seitenrahmen des Arbeitsspeichers befinden. Man unterstützt naheliegenderweise jene Seiten, auf die jeweils am häufigsten zugegriffen wird.

Das technische Mittel heißt Translation Lookaside Buffer (TLB), Address Translation Cache (ATC) o. ä.

Was man der (System-) Software überläßt:
Den Seitenaustausch mit dem Massenspeicher (Page Swapping). Befindet sich die adressierte Seite nicht in einem Seitenrahmen des Arbeitsspeichers, so muß sie vom Massenspeicher geholt und in den Arbeitsspeicher geschafft werden. Ist dort kein Seitenrahmen frei, so muß einer freigemacht werden. Dazu kann es erforderlich sein, die bisher dort untergebrachte Seite auf den Massenspeicher auszulagern.

Was dazwischenliegt:
Der Zugriff auf Seiten, die zwar im Arbeitsspeicher anwesend sind, deren Umsetzungsangaben sich aber nicht im TLB befinden. Die Aufgabe: der TLB ist mit einer entsprechenden Umsetzungsangabe zu füllen. Hierzu gibt es verschiedene Lösungen:

- das Füllen erfolgt rein hardwareseitig durch automatisch ablaufende (bzw. mikroprogrammgesteuerte) Tabellenzugriffe. Beispiele: IA-32, Motorola 68k.
- auch zum Füllen des TLB wird die Software gerufen. Beispiel: MIPS R 4000.
- Kompromißlösungen. Z. B. wird zwar Software gerufen, es gibt aber Spezialbefehle zur Unterstützung der Tabellenzugriffe. Beispiel: IA-64.

Die Herausforderung:
Sehr lange virtuelle Adressen (vor allem: 64 Bits).

Bei 64 Bits Adreßlänge und 4 k großen Seiten braucht man 2^{52} = 4096 T = 4 P Umsetzungsangaben.

Manche 64-Bit-Maschinen unterstützen deshalb nur einen Teil des gesamten Adreßraums.

Was naheliegt, man aber tunlichst unterlassen sollte:
Mit den höherwertigen Adreßbits zu tricksen und den Adreßraum großzügig zu verwüsten (z. B. jedem Heap ein paar GBytes zu spendieren). Das mag aus akademischer Sicht elegant sein, ist aber nicht mehr zu verwalten. Manche Architekturen erzwingen deshalb feste Belegungen ungenutzter Adreßbits (Beispiel: AMD x86-64) und unterstützen mehrere Adreßräume auf eigene Weise (Beispiele: MIPS R 4000, IA-64).

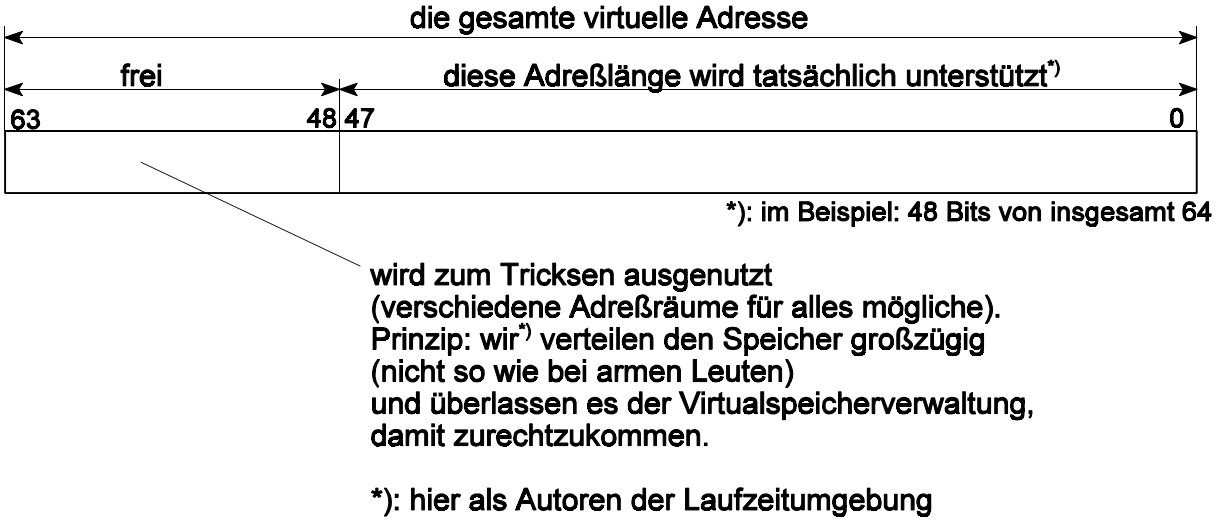

Abb. 7 Wer es so macht, schießt sich in den Fuß...

Mehrere Adreßräume
Ein virtueller Adreßraum ist eigentlich nichts anderes als ein Satz Seitenumsetzungssangaben + zugehöriger Speicherinhalt. Um mehrere Adreßräume bereitzustellen, genügt es, die Seitenumsetzungstabellen entsprechend auszuwechseln.

Der Vorteil:
Vom Adreßraum A aus kann man den Adreßaum B unmöglich erreichen - und mag man sich noch so sehr anstrengen -, also auch dort keinen Schaden anrichten. Deshalb liegt es nahe, in einer Multiprogramming-Umgebung je Prozeß einen eigenen Adreßraum vorzusehen. Beispiel: WIN32-API.

Andererseits hat ein einziger Adreßraum auch seine Vorteile - vor allem dann, wenn er sehr groß ist. Moderne Hochleistungsprozessoren unterstützen deshalb beide Adreßmodelle (Beispiele: MIPS R-4000; IA-64):

- MAS = Multiple Address Space. Die virtuelle Adresse wird hierbei um eine Adreßraum-Auswahlangabe (Region, Address Space Identifier ASI) ergänzt, die in die Umsetzung einbezogen wird.
- SAS = Single Addres Space.

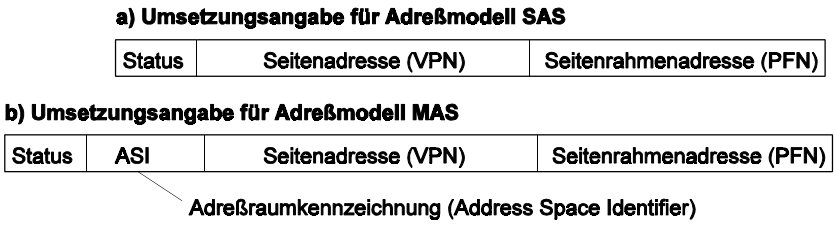

Abb. 8 Umsetzungsangaben für verschiedene Adreßmodelle

Seitengröße
Ist Erfahrungssache (Simulation, Versuch). Typischerweise werden eher kleine Seiten bevorzugt (Richwerte: 2, 4, 8, 16 kBytes). Sie sind schnell ein- und ausgelagert und erlauben einen feinstufige Zuteilung von Schutzrechten. Der Verwaltungsaufwand ist aber hoch. Gelegentlich braucht man größere zusammenhängende Speicherbereiche (Bildspeicher, Ein- und Ausgabe über den Speicheradreßraum). Deshalb hat man auch Seitengrößen im MBytes-Bereich vorgesehen (Richtwerte: 1, 2, 4 MBytes).

Eine Übungsaufgabe:
In einer neuartigen Rechnerarchitektur hat die effektive Adresse eine Länge von 43 Bits (Byteadressierung). Hierfür ist ein virtueller Speicher nach dem Prinzip der Seitenverwaltung zu implementieren. Die physische Adresse soll ebenfalls 43 Bits lang sein. Die Datenstrukturen der Adreßumsetzung werden im Prozessor über ein Zeigerregister ATP (Address Translation Pointer) adressiert. Die Seitengröße: 8 kBytes. Entwerfen Sie das Schema der Adreßumsetzung über eine mehrstufige Tabellenstruktur. Verwenden Sie bitte*) folgende Begriffe: Seitentabelle, Seitentabellenverzeichnis 1. Ordnung, Seitentabellenverzeichnis 2. Ordnung usw.

*): um das Korrigieren zu erleichtern...

Die Aufgabe sollte schrittweise angegangen werden. (Alle Teillösungen werden bewertet.)

a) wie sieht die integrale 43-Bit-Adresse einer solchen Seite aus?

b) diskutierten Sie kurz, ob wir mit Tabelleneinträgen von 32 Bits Länge auskommen.

c) nennen Sie kurz einen handgreiflichen Vorteil von 64 Bits langen Einträgen.

- Im folgenden bleiben wir bei 64 Bits -

d) diskutieren Sie kurz, wie groß die einzelne Tabelle sein sollte.

e) wieviele Umsetzungsstufen brauchen wir?

f) stellen Sie das Umsetzungsschema zeichnerisch dar.