Kapitel 13: Virtueller Speicher

- Motivation und Einführung
- Virtueller Speicher und Lokalitätsprinzip
- Seitenbasierter Virtueller Speicher
- Implementierung von Seitentabellen
- Hardwarevorrausetzungen
    - MMU
    - TLB
- Strategien beim Seitenorientierten Virtuellen Speicher
- Lastkontrolle

13.1 Motivation und Einführung

Mit dem virtuellen Speicherprinzip setzt man den Anwenderwunsch um, dass stets nur die Applikationsadressraumabschnitte im Hauptspeicher präsent sind, an denen eine Applikation gerade arbeitet. Dabei werden folgende Fragestellungen gelöst, wenn man das virtuelle Speicherprinzip wirkungsvoll implementiert.

1. Vollständig (und eventuell sogar noch zusammenhängend) abgebildete logische Adressräume benötigen im physischen Adressraum, also im Hauptspeicher unnötig viel Platz, wodurch ohne Gewinn der Multiprogrammiergrad reduziert wird, was die Systemeffizienz reduziert.

2. Wie könnte man logische Adressräume zur Ausführung bringen, die sogar größer als der physische Hauptseicher sind?

3. Teilweise abgebildete Adressräume sind aus Anwendersicht jedoch recht umständlich und ohne Systemkenntnis nur schwer beherrschbar (siehe u.a. die Overlay Technik).

Beim virtuellen Speicherprinzip übernimmt das System die Aufgabe, die jeweils benötigten Adressraumteile automatisch so nachzuladen, dass hierdurch die Verweildauern der Task bzw. Prozesse nicht dramatisch erhöht werden. Durch geeignete Nachladestrategien kann man es erreichen, dass nicht mehr oder noch nicht benötigten Adressraumabschnitte auf der Platte und die aktuell benötigten Adressraumabschnitte der aktivierten Tasks bzw. Prozesse im Hauptspeicher geladen sind.

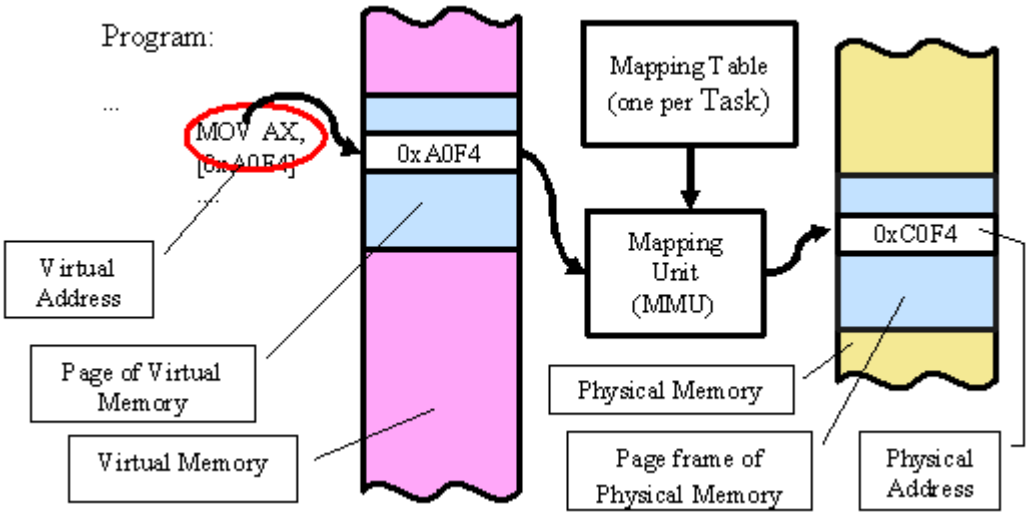
Der virtuelle Speicher erstreckt sich quasi über zwei Speicherstufen, den hinlänglich schnellen Hauptspeicher und den um Größenordnungen langsameren Plattenspeicher, der quasi als *Auffangbetriebsmittel* (*background oder secondary memory*) im Hintergrund all die Adressraumabschnitte speichert, die aktuell nicht oder nicht mehr im Hauptspeicher (*primary memory*) benötigt werden. Somit umfasst der virtuelle Speicher alle im System aktivierten logischen bzw. virtuellen Adressräume.

13.2 Virtueller Speicher und Lokalitätsprinzip

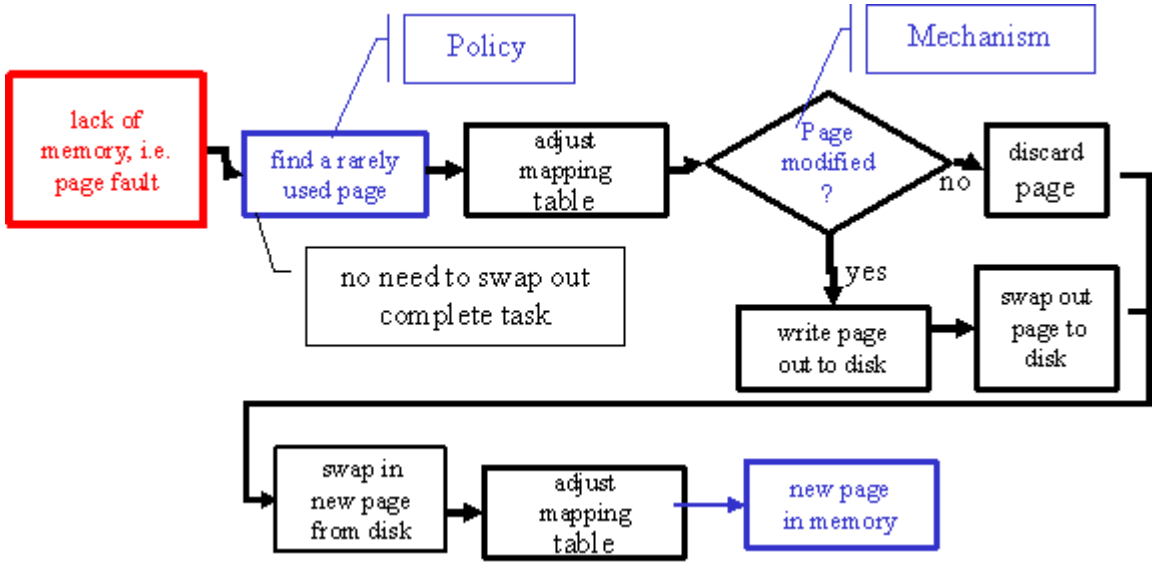
Das virtuelle Speicherkonzept sieht vor, dass entweder im Vorgriff (*pre paging*) oder auf Verlangen (*demand paging*) genau die Adressraumseiten in den Hauptspeicher nachgeladen werden, die auf Grund der Programmlokalität des Prozesses bzw. der Threads der Task gerade benötigt werden.

Alle im Programmcode in den einzelnen Regionen enthaltenen Adressen sind logische Adressen, beginnend bei der logischen Adresse 0 bis .... Je nachdem auf welche physischen Adressbereiche eine Region gerade abgebildet wird, wird dann der nächste Befehl bzw. der im Befehl angesprochene Operand aus der entsprechenden Hauptspeicherkachel geholt. Hierzu muss es einen Adressumsetzungseinheit geben, da ansonsten der Leistungsverslust zu groß wäre, diese Einheit ist die MMU.

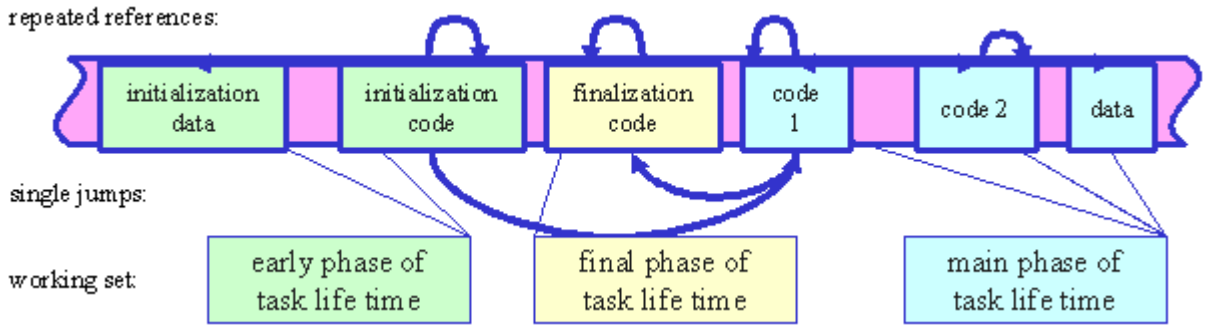
Zur Unterstützung dieser Aufgabe muss es demzufolge pro logischen Adressraum eine Umsetzungstabelle geben, aus der man diese Abbildung von logischen auf physische Adressen entnehmen bzw. berechnen kann. Diese Datenstruktur nennt man beim Seitenorientierten Virtuellen Speicher "Seitentabelle" (*page table*).



In der nächsten Skizze soll deutlich werden, wann welche Systemkomponenten sich wie bei einem Seitenfehler verhalten, wobei ein Seitenfehler entsteht, wenn ein Thread einer multi-threaded Task oder wenn ein Prozess auf eine logische Adresse zugreifen möchte, die zwar zum logischen Adressraum gehört, aber aktuell noch nicht im Hauptspeicher abgebildet ist (*currently not yet mapped*).



Man beachte insbesondere, dass man auch in dieser Skizze zum wiederholten Mal das Prinzip der Trennung von Strategie und Mechanismus erkennen kann. Es ist eine strategische Aufgabe, bei knappem Speicher aus der Menge der belegten Kacheln die Seite herauszufinden, die mit großer Wahrscheinlichkeit in der nächsten Zukunft nicht mehr benötigt wird. Dagegen gibt es die klar abgesetzte Aufgabe des Seitentauschmechanismus, dafür zu sorgen, dass wenn die ausgewählte Seite in der jüngsten Vergangenheit beschrieben worden ist, diesen Seiteninhalt zuvor auf die Platte zu retten, bevor der neue Seiteninhalt in die entsprechende Kachel nachgeladen wird. Man sieht wiederum sehr leicht, dass man auf dem gleichen Mechanismus beliebige Seitenersetzungsstrategien aufsetzen kann. Außerdem wird hier auch deutlich, dass jede Strategie den Mechanismus dominiert. (Analogie: In einem Unternehmen werden die Entscheidungen für die Unternehmensstrategie im Management getroffen, das Personal setzt die Strategie dann um.)

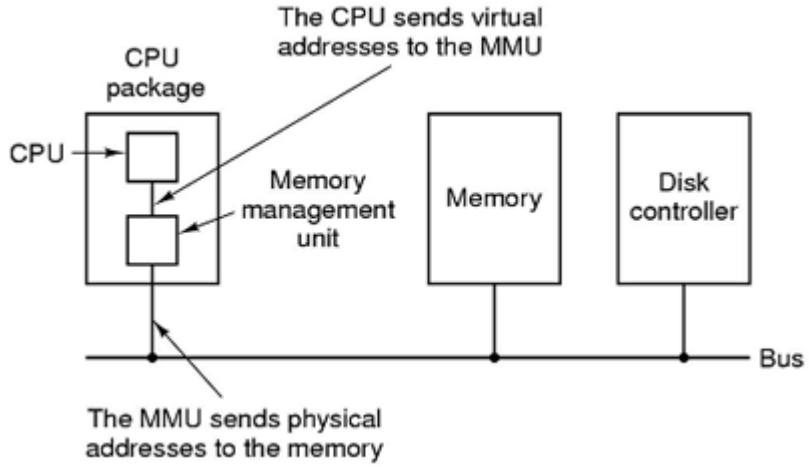


Das automatische Nachladen der benötigten Adressraumabschnitte wird durch das Programmverhalten in vielen Fällen unterstützt, wie in der obigen Skizze angedeutet. Das Prinzip der Lokalität in einer Task aus zwei Arbeiterthreads, die nach der Initialisierungsphase durch den Mainthread ihre Arbeit wahrnehmen, zeigt doch sehr deutlich, dass es in vielen Anwendungen, zumindest aber in den größeren Anwendungen durchaus Adressraumbereiche geben kann, die entweder gar nicht oder zumindest für längere Zeit nicht mehr benutzt werden. Beispielsweise würde diese Task nach einem Abbruch in einem der Arbeiterthreads überhaupt nicht bis in die finale Phase kommen.

13.3 Seitenbasierter Virtueller Speicher

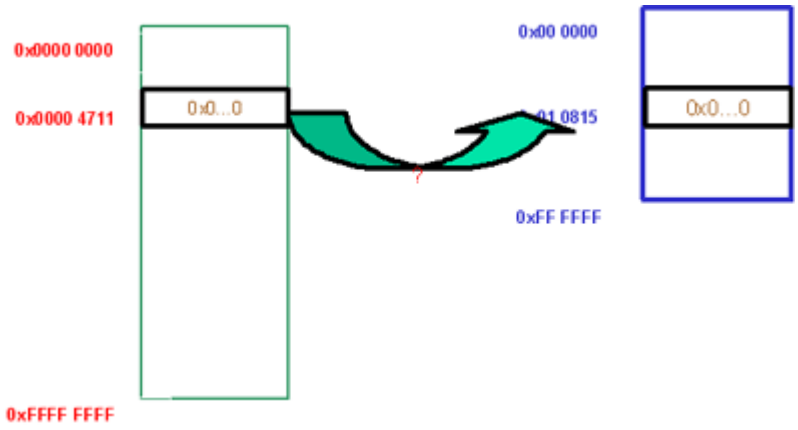
In einem eher traditionellen, seitenorientierten virtuellen Speichersystem geht man davon aus, dass man mit einer systemweit konstanten Seitengröße die Bedürfnisse des Systems und seiner Anwender zumindest im Mittel gut befriedigen kann. Typische Seiten- bzw. Kachelgrößen variieren zwischen [1 KB, 64 KB]. Modernere Pagingsysteme unterstützen auch verschieden große Seiten (siehe 11....).

Damit der virtuelle Speicher effizient implementiert werden kann, benötigen wir Unterstützung von der Hardware, die dafür sorgt, dass die logischen (virtuellen) Adressen der Task in die entsprechenden Hauptspeicheradressen umgerechnet werden. Der HW-Baustein, der für die automatische Umsetzung der logischen in die benötigten physischen Adressen verantwortlich ist, heißt MMU (*memory management unit*). Eine MMU befindet sich immer in unmittelbarer Nähe zum Prozessor, d.h. pro Prozessor gibt es eine eigene MMU. MMUs sind hinsichtlich ihrer prinzipiellen Aufbaus von Prozessortyp zu Prozessortyp verschieden, manche beinhalten einen zusätzlichen Beschleunigerpuffer (TLB, der die jeweils aktuellsten Adresstransformationsvorschriften enthält).

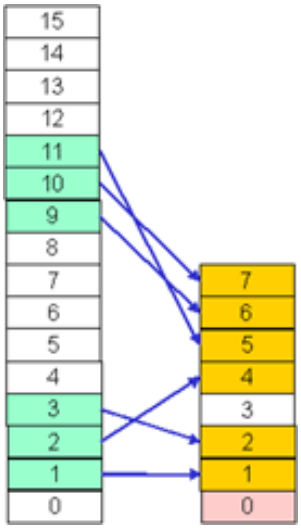


In einem System mit virtuellem Speicher werden also nach wie vor im Prozessor nur logische Adressen verwendet, die zur MMU weitergeleitet werden, ehe dann über den Systembus an den Hauptspeicher auf die Speicherzellen mit den umgerechneten physischen Adressen zugegriffen wird.

Wie wird nun diese Adresstransformation von virtueller zu physischer Adresse prinzipiell erfolgen?

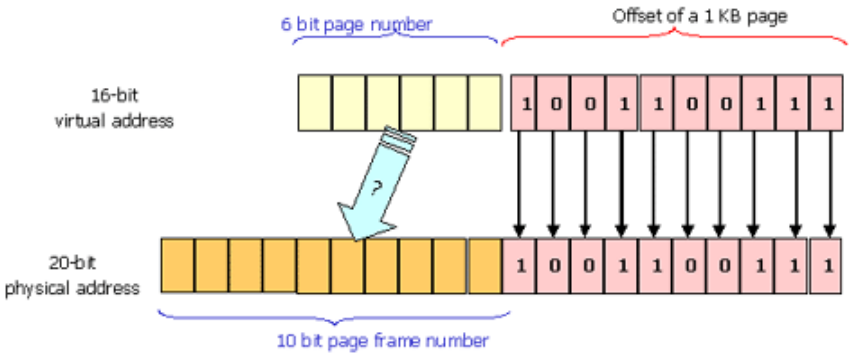


Hierzu wird sowohl der virtuelle als auch der physische Adressraum in gleich große Portionen unterteilt, den Seiten (*pages*) bzw. den Kacheln oder Seitenrahmen (*page frames*) wie in der folgenden Skizze angedeutet. In der Skizze sind die Seiten 1,2,3,9,10 und 11 aktuell im Hauptspeicher auf den Kacheln 1,4,2,6,7, und 5 abgebildet. Die Information, welche Seiten im Hauptspeicher abgebildet sind und welche auf dem Hintergrundspeicher liegen, wird pro logischem Adressraum in einer Seitentabelle gehalten. Die Seitentabellen der aktivierten Tasks bzw. Prozesse werden im Kern implementiert. Welche Information muss in einem Seitentabelleneintrag stehen? Nun offenbar die Kachelanfangsadresse bzw. die periphere Adresse, sowie weitere die Adressabbildung unterstützenden Kontrollbits (siehe später).

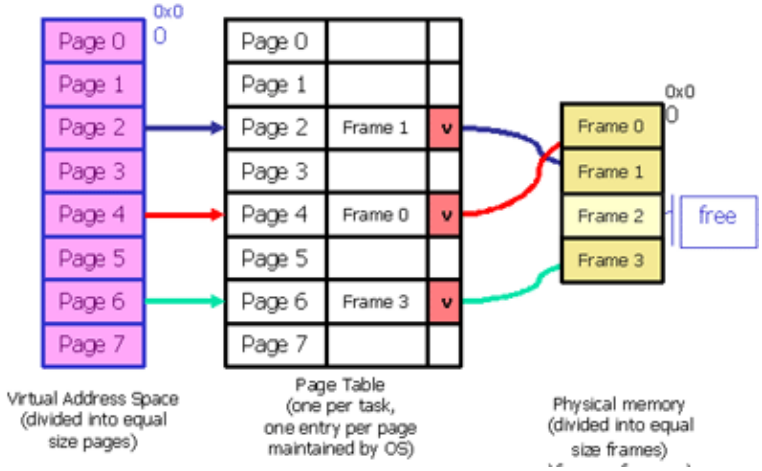


13.3.1 Adresstransformation

Jede im Rahmen der Abarbeitung einer Applikation im Prozessor errechnete virtuelle Adresse muss zur Laufzeit von der MMU in die richtige physische Hauptspeicheradresse transformiert werden, d.h. bei jedem `load` oder `store` Befehl oder bei einen `fetch` im Zuge der Befehlsfortschreibung muss diese Adresstransformation durchgeführt werden. In einem Pagingsystem ist diese Transformation relativ simpel, es muss die so genannte Seitennummer durch die Kachelnummer ersetzt werden (*page number substitution*), da der so genannte Versatz (*offset*) innerhalb der Seite bzw. Kachel identisch ist. Es gab bzw. gibt Systeme, in denen der physische Adressraum (d.h. der Hauptspeicher) größer als jeder virtuelle Adressraum ist (s.u.), aber typischerweise ist es gerade anders herum.



13.3.2 Adresstransformation mittels einer linearen Seitentabelle



Der Kern muss offenbar wissen, wo die Seitentabelle jedes aktivierten virtuellen Adressraums beginnt, insbesondere die des gerade rechnenden Adressraums. Nehmen wir dazu an, dass es im Prozessor wiederum ein Basisregister gibt, das uns die Anfangsadresse der Seitentabelle des rechenden Adressraum beinhaltet. Aus der Seitennummer, also aus den führenden Adressbits der virtuellen Adresse kann man den zugehörigen Seitentabelleneintrag leicht finden, denn die Länge eines Seitentabelleneintrags ist zumindest dem Kern bekannt. Wenn also beispielsweise eine virtuelle Adresse innerhalb der Seite 4 angesprochen wird, dann errechnet sich der Index des entsprechenden Seitentabelleneintrags (PTE) wie folgt: PT[i] = 4* Length(PTE) + (Basisregister). Dort findet man dann die gewünschte Kachelnummer, im obigen Beispiel die Kachelnummer 0.

Eine grobe Leistungsanalyse zeigt sofort, dass dieser Adresstransformationsschritt, der ja bei jeder Referenz durchzuführen ist, beschleunigt werden muss, ansonsten taugt das virtuelle Speicherkonzept nicht viel.

Das in der obigen Seitentabelle angegebene Valid-Bit soll der Hardware anzeigen, ob die entsprechende Seite aktuell im Hauptspeicher abgebildet ist, ob also im Seitentabelleneintrag die für die Adresstransformation nötige Information gültig ist oder nicht.

13.3.3 Weitere Kontrollbits im Seitentabelleneintrag

Neben dem Validbit kann es je nach Architektur weitere nützliche Kontrollbits im Seitentabelleneintrag geben. Die folgende Tabelle gibt einige der gebräuchlichsten Kontrollbits wieder.

R-Bit: Zeigt an, ob von der Seite gelesen werden darf.
W-Bit zeigt an, ob auf die Seite geschrieben werden darf.
X-Bit zeigt an, ob der Seiteinhalt als Code interpretiert werden darf.
V-Bit zeigt an, ob die Seite im Hauptspeicher geladen ist.
Re-Bit zeigt an, ob auf die Seite in der Vergangenheit zugegriffen wurde.

M-Bit: Modified-Bit zeigt an, ob die Seite verändert worden ist.
D-Bit zeigt an, ob diese Seite zum virtuellen Adressraum gehört.
S-Bit zeigt an, ob diese Seite von mehreren Adressräumen gemeinsame benutzt wird.
C-Bit zeigt an, ob diese Seite in die Caches geladen werden darf.
SU-Bit zeigt an, ob es sich um eine Superseite handelt.
P-Bit zeigt an, ob diese Seite bis auf Widerruf im Hauptspeicher festgenagelt ist.

Hinweis: Es wird erwartet, dass man pro Kontrollbit weiß, wer wie und wann damit umgeht, insbesondere setzt der Prozessor, die MMU oder ein Systemprogramm diese Kontrollbits.

13.4 Implementierung von Seitentabellen

Es gibt eine Reihe von Vorschlägen, wie man Seitentabellen implementieren könnte, so dass sowohl wenig Platz pro Seitentabelle benötigt wird, als auch wenig Aufwand bei einem Adressraumwechsel getrieben werden muss.

Die einfachste, gleichwohl höchst ineffiziente Methode besteht darin, sich im Kernadressraum einen zusammenhängenden Platz zur Aufnahme einer linearen Seitentabelle pro virtuellen Adressraum zu beschaffen. Jeder Seitentabelleneintrag besteht zumindest aus der Kachelnummer sowie einigen zusätzliche Kontrollbits, die den Umgang mit der Seitentabelle erleichtern sollen. Man kann sich leicht vorstellen, dass insbesondere die ersten Programme im Leben einer(s) Informatikstudierenden i.d.R. eher sehr kleine Programme sein werden. Trotzdem erhalten auch diese Miniprogramme einen eigenen Adressraum, der andererseits aber auch wieder ausreichen muss, um evtl. sehr große Programme aufzunehmen, d.h. wir müssen im Kern jedes Mal den Speicherbedarf einer total gefüllten Seitentabelle anfordern.

Nehmen wir ferner an, dass wir einen 32-Bit Mikroprozessor besitzen, dessen Anwenderadressraum 2 GByte groß sein soll (der Rest ist für den Systemkern reserviert). Nehmen wir an, dass eine Seite 4KByte groß sein soll und dass ein Seitentabelleneintrag 4 Byte groß ist.

Wie viel Platz würde dann eine Seitentabelle im Kernadressraum benötigen?

Zunächst ist zu klären, wie viele Seitentabelleneinträge benötigt würden, offensichtlich 2^{19} à 4 Byte. Daraus folgt, jede Seitentabelle würde 2^{21} Byte Platz benötigen, das sind insgesamt **2 MByte** pro Seitentabelle.

Wenn wir weiter davon ausgehen, dass die Mehrzahl der Anwendungen nur sehr wenige Seitentabelleneinträge überhaupt benötigt (vielleicht nur einige hundert Seitentabelleneinträge), dann sieht man sehr schnell, dass diese Lösung höchst ineffizient bezüglich des Platzbedarfs im Kern ist. Diese Ineffizienz würde noch ins Absurde gesteigert, wenn man zu 64 Bit Prozessoren übergehen würde.

Welche Hardwareunterstützung nötig ist, um einen effizienten Adressraumwechsel durchzuführen, wollen wir auf spätere Abschnitte verschieben.

Ein Ausweg aus obigem Platzbedarfsdilemma sind mehrstufige Seitentabellen, zwei bis vierstufige Schemata wurden bereits vorgeschlagen.

Im 32-Bit Bereich wendet man beispielsweise oft folgendes Schema an: Die führenden 10 Adressbits der logischen (virtuellen) Adresse sind der Index in die Pagedirectorytabelle, die nächsten 10 Adressbits bestimmen den Index in der eigentlichen Seitentabelle, die restlichen 12 Adressbits sind der Versatz (*offset*) innerhalb der Seite.

Sowohl die Pagedirectorytabelle als auch jede der benötigten Seitentabellen benötigen 2^{10} Seitentabelleneinträge, passen also jeweils gerade in eine Kachel hinein.

Welche Konsequenzen für den Platzbedarf ergeben sich für die Mehrzahl der kleineren Programme und damit für nur spärlich gefüllte logische (virtuelle) Adressräume?

Kleinere Programme benötigen für ihre drei Regionen Code, Daten und Stapel gerade mal drei Seitentabelleneinträge in der Pagedirectorytabelle und weitere drei Seitentabellen für jede der angegebenen Regionen, der Platzbedarf insgesamt beträgt in diesem Fall: 4*1024*4 Byte = 16 KByte statt der satten 2 MB bei einer linearen Seitentabelle.

Ein weiterer Vorteil von mehrstufigen Seitentabellen besteht darin, dass sie dafür prädestiniert sind, verschiedene Seitengrößen zu unterstützen. Beispielsweise wird durch obiges dreistufiges Schema es ermöglicht, durch ein Kontrollbit in der Pagedirectorytabelle Standardseiten von den so genannten "Superseiten" (à 4 MByte) zu unterscheiden.

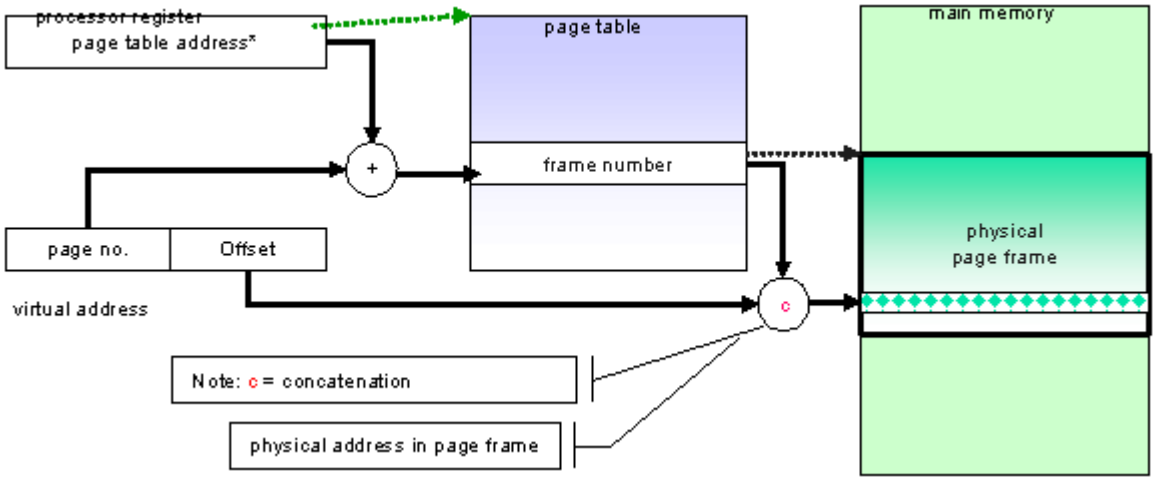
Im letzteren Fall steht dann in der Pagedirectorytabelle nicht die Kachelnummer der Seitentabelle sondern bereits die Kachelnummer der Superseite. Manche der modernen Mikroprozessoren unterstützen eine ganze Palette von unterschiedlichen Seitengrößen.

Hinweis: Ihr könnt euch ja mal überlegen, welche Vorteile solch ein Schema mit mehreren Seitengrößen mit sich bringen kann. Andererseits muss natürlich auch der zusätzliche Aufwand ins Kalkül gezogen werden, der mit der entsprechenden Speicherverwaltung von verschiedenen Kachelgrößen einhergeht.

Andere Vorschläge verwenden virtuelle Seitentabellen bzw. invertierte Seitentabellen, die in dieser Zusammenfassung nicht behandeln werden. Stattdessen wollen wir die Hardwarevoraussetzungen für eine schnelle Adresstransformation und für einen zügigen Adressraumwechsel studieren.

13.5 Hardwarevoraussetzungen

Die folgende Skizze gibt nochmals einen Überblick über das Adresstransformationsschema bei einer linearen Seitentabelle.



Man beachte dabei, dass die Seitennummer, also die führenden Adressbits im Beispiel als Index in die Seitentabelle verwendet werden. Die entsprechenden führenden Adressbits der Kachel stehen in der Seitentabelle, sofern die Seite bereits im Hauptspeicher auf eine Kachel abgebildet ist. Für jede Task gibt es genau eine Adresse im Kernadressraum, an der ihre eigene Seitentabelle(n) anfängt, diese Seitentabellenanfangsadresse wird beim Wechsel von einer Task zur nächsten in ein entsprechende Basisregisters des Prozessors geladen. Diese Realisierungsvariante kommt noch ohne MMU aus, allerdings müsste dann in der Tat bei jeder Referenz der Prozessor zunächst einen Hauptspeicherzugriff tätigen, nur um auf die so genannte Softwareseitentabelle zuzugreifen um herauszufinden, in welcher Kachel die eigentliche Information liegt. Andererseits hat diese reine Softwarelösung den Vorteil, dass der Adressraumwechsel sehr fix geht, man muss nur den Basisregisterinhalt austauschen.

Als Alternative zur Softwareseitentabelle wurde auch schon vorgeschlagen, diese als so genannte Hardware-Seitentabelle komplett in die MMU beim Adressraumwechsel hinzukopieren.

Welche Konsequenzen hat dieses Vorgehen und für welche Systeme wäre dies eine realisierbare Alternative?

13.5.1 Translation Lookaside Buffer

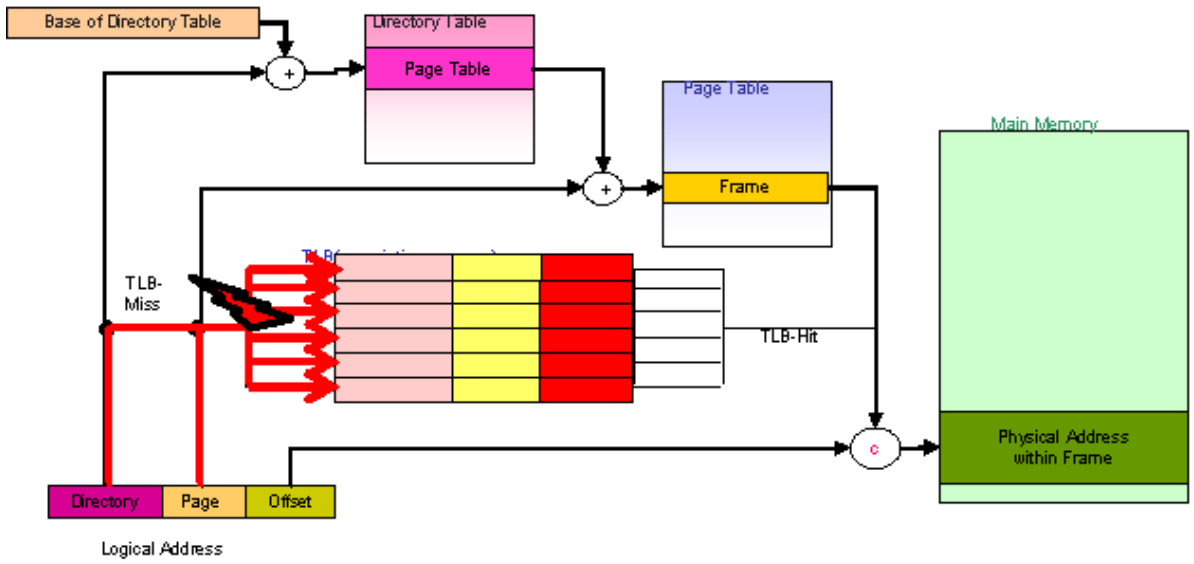
Die beiden oben vorgestellten Muster stellen quasi die reine Software- bzw. die reine Hardwarelösung dar. Wie so oft im Leben setzt sich letztendlich ein gesunder Kompromiss durch, der in unserem Umfeld TLB oder *Translation Lookaside Buffer* heißt und nichts anderes darstellt, als einen Pufferspeicher für aktuell verwendete Adressumsetzungen.

Sofern im TLB bereits auf Grund von früheren Referenzen das Paar Seitennummer/Kachelnummer enthalten ist (TLB-Hit) kann sofort die Hauptspeicheradresse auf den Systembus geschickt werden (bzw. wird auf die entsprechende Cacheline zugegriffen). Welche zusätzlichen Kontrollbits nun ebenfalls im TLB enthalten sind, richtet sich ganz nach der Rechnerarchitektur. Langrechnende Großapplikationen, die konkurrenzlos ihre Arbeit auf dem Prozessor verrichten können und die demzufolge auch viele unterschiedliche Paare von Seiten/Kachelnummern benötigen, können durchaus auch den TLB komplett erschöpfen. Hardware kontrollierte TLB sind solche, bei denen die Hardware selbständig die Seitentabelle nach dem neuen Seitentabelleneintrag durchläuft und den voraussichtlich nicht mehr benötigten TLB-Eintrag nach einem TLB-Miss durch den neuen TLB-Eintrag ersetzt (z.B. Pentium), während beim MIPS die Software das Nachladen des gefüllten TLB in Form einer Ausnahmebehandlung übernimmt.

Das mögliche Layout eines TLBs ist in folgender Skizze festgehalten:

Valid	Virtual page	Modified	Protection	Page frame
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

Man könnte eruieren, dass die beiden letzten Einträge den aktuellen Benutzerstapel enthalten, während die Codeseiten als Zugriffskontrollbits RX enthalten, die aktuellen globalen Daten könnten auf den Seiten 129 und 130 abgebildet ein, während die Halde (*heap*) ab der Seite 140 beginnt. Bei mehrstufigen Seitentabellen werden dann die vorderen Adressbits zu einem einzigen Eintrag zusammengefasst, wie dies in der folgende Abbildung skizziert ist, das im übrigen einen TLB-Miss veranschaulichen soll.



Im obigen Bild ist nun allerdings angedeutet, dass der TLB noch keine Information enthält, dass also für die erste vom Prozessor gelieferte Seitennummer die entsprechende Kachelnummer erst noch mittels Seiten-

tabelleinspektion gefunden werden muss, wobei so ein TLB-Miss auf Grund des Lokalitätsprinzips in der Praxis während der Ausführung eines Threads gar nicht so häufig ist. Allerdings ist ein TLB-Miss nach einem Adressraumwechsel geradezu unvermeidlich, in welchem der Threadumschalter i.d.R. den ganzen TLB löscht (*TLB flushing*). Denn da ja jede Task ihren eigenen logischen Adressraum besitzt, wäre es geradezu fatal, wenn man diese TLB-Einträge nicht löschen würde, denn dann könnte man ganz gezielt die Daten des Konkurrenten überschreiben.

Gängige TLBs können z.Z. zwischen 64 und 265 TLB-Einträge besitzen.

Überlegen Sie sich, worin der Hauptaufwand beim Adressraumwechsel besteht, entweder beim Flushen des TLBs oder beim Auffüllen des TLBs.

Eine Alternative zu diesem Zusatzaufwand beim Adressraumwechsel bieten so genannte Tagged-TLBs, die als Tag eine Adressraumidentifikation zusätzlich enthalten. Somit wird als Einstieg in den TLB nicht nur die virtuelle Adresse sondern zusätzlich noch die Adressraumidentifikation mit verwendet.

13.5.2 Effekte mittels TLBs

Welche Effekte kann man mittels TLBs erzeugen?

Ohne TLB und mit einer reinen Softwareseitentabelle benötigt man pro Referenz mindestens zwei Hauptspeicherzugriffe, den auf die Seitentabelle und den auf die gewünschte Kachel. Nimmt man dagegen an, dass im Mittel 99% TLB-Treffer (TLB-Hits) vorkommen, dann benötigt man im Mittel bei Verwendung des TLBs nur .99*1 + 0.01*2 = 1.01 Hauptspeicherzugriffe. Leider ist eine 99%ige Trefferrate etwas optimistisch.

13.5.3 Seitengrößenwahl

Welche Seitengröße wird man nun als Standardgröße für eine Kachel bzw. Seite wählen bzw. von welchen Einflussgrößen hängt die Seitengröße ab?

Zum einen spielt hierbei die Übertragungsrate beim Transport von Blöcken zwischen Platte und Hauptspeicher eine Rolle, zum anderen bestimmen die typischen Regionengrößen und deren Lokalitätsverhalten die Wahl der "idealen" Seitengröße.

Analyse:

1. Übertragungsrate zwischen Platte und Hauptspeicher

   Abhängig von der Plattengeometrie und den zwischengeschalteten Bussystemen mag es Leistungsunterschiede beim Transport einer bestimmten Blockgröße geben.

   Aus Kompatibilitätsgründen mit sehr alten Datei- und Plattensystemen wird auch auf modernen Platten manchmal eine minimale Blockgröße von 0.5 KByte angeboten. Üblicherweise sind die heutigen typischen physischen Plattenblöcke eher größer, so zwischen 4 KB und 16 KB groß.

   Als Daumenregel kann man davon ausgehen, dass ein zusammenhängend abgespeicherter Datenblock von p KByte auf der Platte schneller in den Hauptspeicher transportiert wird, als wenn diese Daten auf p Blöcken der Blockgröße 1/p auf der Platte abgespeichert würden. Dies liegt in erster Linie daran, dass viel Positionierungszeit vergeht, bis der Plattenkopf über der richtigen Plattenspur angekommen ist. Zudem muss man im Mittel noch die halbe Umdrehung abwarten, ehe mit dem eigentlichen Zugriff begonnen werden kann.

2. Einlagern von überflüssiger Information und interner Verschnitt

   Je größer andererseits die Seiten werden, desto mehr Information wird jedes Mal in den Hauptspeicher geladen, obwohl wegen mangelnder Lokalität nur ein Bruchteil der großen Seiten gebraucht

wird. Je größer die Seite gewählt wird, desto größer ist auch der interne Verschnitt (*internal fragmentation*) pro Seite am Ende einer Region.

Moderne virtuelle Speichersysteme bieten entweder variable Seitengrößen an, die teilweise sogar von der Hardware unterstützt werden, z.B. bieten die Pentiumprozessoren 4 KByte "Standard"- und 4MByte große "Super"-Seiten an. Größere Seiten werden u.a. dafür benötigt, bestimmte Geräte effizient zu unterstützen. Beispielsweise kann man in einer Superpage den Videobuffer des Monitors unterbringen, der stets zusammenhängend gespeichert sein muss, damit die Refresh-Algorithmen auch schnell arbeiten können. Würde man stattdessen 1014 4 KByte Standardseiten wählen, dann hätte man erstens entsprechend viele Einträge in den Seitentabellen und zudem noch entsprechend viele Einträge im TLB bzw. dieser würde i.d.R. kapazitätsmäßig gar nicht groß genug sein, um alle benötigten Einträge für die 1024 Adressumsetzungsvorschriften halten zu können. Größere Seiten kann man auch dann verwenden, wenn bestimmte Daten, wie z.B. große Arrays entweder immer zusammen oder gar nicht benutzt werden.

Rechnerarchitekturen (u.a. DEC Alpha oder Itanium) offerieren dem Benutzer bzw. dem System eine ganze Palette unterschiedlich großer Seiten an (siehe u.a. den Konferenzbeitrag, "Transparent OS-Support for Superpages", J. Navarro, S. Iyer, P. Druschel, A.- Cox, 5. Symposium on OSDI, Boston, December 2002)

Hinweis: Studieren Sie diesen Artikel sorgfältig!

13.6 Pagingstrategien

Während wir schon früher festgestellt haben, dass es neben den Pagingstrategien auch den Seitentauschmechanismus gibt, wollen wir uns nun mögliche Strategien im Umfeld des Seitentausches studieren. Wir können folgende prinzipielle Strategien unterscheiden:

- Einlagerungsstrategie (*fetch policy*)
- Platzierungsstrategie (*placement policy*)
- Ersetzungsstrategie (*replacement policy*)
- Reinigungsstrategie (*cleanings policy*)

13.6.1 Einlagerungsstrategie

Bei dieser Strategie geht es darum, wann eine Seite eines virtuellen Adressraums in den Hauptspeicher geladen werden soll. Zwei prinzipielle Vorgehensweisen können unterschieden werden:

- Einlagern auf Verlangen (*demand paging*)
- Vorausschauendes Einlagern (*pre paging*)

Beim Demandpaging geht man davon aus, dass man solange zuwarten kann, bis ein Thread der gerade rechnenden Task einen Seitenfehler (*page fault*) produziert, d.h. die MMU stellt fest, dass diese Seite aktuell gar nicht im Hauptspeicher vorhanden ist. In diesem Fall wird eine Ausnahmebehandlung durchgeführt, in deren Verlauf die bislang fehlende Seite nachgeladen wird, solange muss der Thread warten, der den Seitenfehler verursacht hat. Wenn der Thread ein brauchbares Lokalitätsverhalten zeigt, dann werden Seitenfehler eher selten sein.

Der Vorteil dieses Verfahrens liegt klar auf der Hand, es werden nur Seiten eingelagert, die auch tatsächlich benötigt werden. Andererseits wird dadurch auch der Start einer Task erheblich verzögert, da in der Anfangsphase sehr viele Seitenfehler verursacht werden, bis die Task die erforderliche Arbeitsmenge an Seiten im Hauptspeicher geladen hat, die sie benötigt um effizient arbeiten zu können. Es können sich dabei auch Seiteneffekte mit dem mittelfristigen Planer (*long term scheduler*) ergeben, wenn dieser von Zeit zu Zeit aktivierte Tasks temporär wieder komplett deaktiviert, um so Platz für neue Tasks zu schaffen. Wenn die mittlere Aufenthaltsdauer im inneren Zirkel nicht wesentlich länger als die Einlagerungszeit für die komplette Arbeitsmenge ist, dann ist diese Einlagerungsstrategie kontraproduktiv.

Alternativ zum Einlagern auf Verlangen kann man auch zum eher spekulativen vorausschauenden Einlagern übergehen, wobei spekulativ auch Seiten eingelagert werden, auf die bislang noch nicht zugegriffen worden ist. Wie bei allen Spekulationen, kann der "Schuss auch nach hinten losgehen", denn jede unnütz eingelagerte Seite nimmt anderen Arbeitsmengen Kacheln weg, zu dem wird der Verkehr mit der Platte u.U. verlangsamt. Diese Vorgehensweise kann jedoch bei häufig wieder verwendeten Applikationen, die stets nach dem gleichen Muster ablaufen, durchaus zu positiven Effekten führen, sofern man in geeigneter Weise deren frühere Abläufe im Hinblick auf das Referenzmuster zur Verfügung hat.

13.6.2 Platzierungsstrategie (placement policy)

Diese Strategie ist zum einen bei den segmentorientierten virtuellen Speichersystemen von Bedeutung, zum anderen muss auch bei den modernen Pagingsystemen mit unterschiedlichen Seitengrößen hierauf geachtet werden. In Pagingsystemen mit nur einer Seitengröße ist hingegen die Platzierung einer Seite auf eine freie Kachel völlig irrelevant, sieht man davon ab, dass aus Hardwarekompatibilitätsgründen bestimmte DMAs nur auf bestimmte Kacheln zugreifen können, dies kann aber bereits im Rahmen des Urladevorgangs (booten) berücksichtigt werden.

13.6.3 Ersetzungsstrategie

Im Zusammenhang mit dem Seitentausch gibt es ein Reihe von Ersetzungsstrategien (*replacement policies*), von denen einige rein theoretischen Vergleichswertcharakter besitzen, andere dagegen in der Praxis eingesetzt werden. Außerdem gibt es zwei verschiedene Gültigkeitsbereiche von Seitenersetzung:

- Global, d.h. alle Seiten-/Kachelkombinationen werden bei einer Seitenersetzung mitberücksichtigt
- Lokal, nur die Seiten-/Kachelkombinationen der Anwendung werden beim Seitentausch mitberücksichtigt.

Im letzten Fall muss es eine Instanz geben, die beim Starten einer Task sich um die Reservierung von k Kacheln für diese Task kümmert. Im ersten Fall könnte im Prinzip eine Task für sich alle in Frage kommenden Kacheln des Systems nach und nach anfordern. In der Praxis tritt dies allerdings nur dann auf, wenn eben kein gleichzeitiger Konkurrent im Spiele wäre.

Die beiden Ersetzungsstrategien, die häufig angewendet werden, sind FIFO und Clock und zwar sowohl in der globalen als auch in der lokalen Variante. In der ersten Strategie werden die Seiten ersetzt, die schon am längsten im System (Anwendung) sind, in der zweiten Strategie werden die zum Gültigkeitsbereich des Seitentauschers gehörenden Seiten-/Kachelkombinationen in Form einer Rundliste zusammengefasst. Ein "Uhrzeiger" zeigt genau auf die Kombination, die beim letzten Seitenfehler ersetzt worden ist. Beim nächsten Seitenfehler beginnt man an genau dieser Stelle nach einer Seiten/Kachelkombination zu suchen, die in der jüngeren Vergangenheit nicht mehr referenziert worden ist, hierzu wird also ein weiteres Kontrollbit im Seitentabelleneintrag benötigt. Wird nun bei der Suche nach einer Kachel, die man ersetzen könnte, eine nicht referenzierte Seite gefunden, dann ersetzt man diese. Alle anderen Seiten-/Kachelkombinationen, die man bei dieser Suche kontrolliert hat, die aber in der jüngeren Vergangenheit referenziert worden sind, bei denen löscht man dieses Referenzbit, mit der Intention: Wenn sie noch weiter benötigt werden, dann werden sie bis zum nächsten Seitenfehler auch nochmals referenziert, wenn nicht, dann haben sie Pech gehabt.

Wie lange dauert nun so eine Seitenfehlerbehandlung aus der Sicht der Anwendung? Nehmen wir mal an, dass zum Zeitpunkt t_0 der Seitenfehler auftritt, d.h. der Prozessor verursacht eine Ausnahme (exception), in deren Verlauf entweder der Seitentauschalgorithmus selbst durchgeführt wird (nicht so günstig, da hierdurch der Kern lange gesperrt wäre) oder in der der Seitentauscherthread eine geeignete Seiten/Kachelkombination bestimmt. Anschließend schickt der Seitentauscher eine Nachricht an den Plattentreiber, worauf er so lange warten muss, bis die Platte das Einlagern der neuen Seite durchgeführt hat, wir nehmen mal zur Vereinfachung an, dass die überlagerte alte Seite mittlerweile nicht verändert worden war. Nachdem der Plattentreiber fertig ist, schickt er an den Seitentauscher die Nachricht, dass die Seite erfolgreich im Hauptspeicher eingelagert worden ist, der Seitentauscher wird darauf hin die Seitentabelle entsprechend modifizieren, um dann zum Schluss den wartenden Anwenderthread wieder zu deblockieren und in die

Bereitwarteschlange (ready queue) einzureihen. Das folgende Zeitdiagramm, das nicht maßstäblich ist, soll dazu anregen, sich zu überlegen, ob und wie man diese Situation aus der Sicht der Anwendung vielleicht etwas günstiger zu gestalten

$$t_0 + t_{exception} + t_{seitentausch} + 2*t_{seitentransfer} + t_{seitentabellenmodifikation}$$

Durch welche organisatorischen Maßnahmen könnte man die Gesamtzeit zur Behandlung eines Seitenfehlers reduzieren?

1. Kann man evtl. sogar Seitenfehler vermeiden? Wenn man beim Start einer Task gleich mehrere Seiten einlagert und auch ansonsten bei einem Seitenfehler nicht nur die benötigte Seite einlagert, sondern auch gleich noch einige Seiten mehr, dann steigt mit Sicherheit die Wahrscheinlichkeit, dass diese Task weniger Seitenfehler produzieren wird (siehe Prepaging). Andererseits werden dann aber u.U. auch Seiten eingelagert, die beim aktuellen Lauf der Task gar nicht benötigt werden, was Verschwendung wäre, denn die hierfür verwendeten Kacheln könnten für andere Anwendungen sinnvoller verwendet werden. Diese Idee des "Prepaging" ist fast so alt wie die Idee des virtuellen Speichers, es gibt strikte Befürworter und strikte Gegner dieser Idee, es gibt dabei aber natürlich auch bestimmte Anwendungen, wo diese Idee sehr sinnvoll angewendet werden kann. Man denke dabei nur einmal an eine "in den Adressraum abgebildete sequentielle Datei" (memory mapped sequential file).

2. Wenn im obigen Zeitdiagramm auch noch auftauchen würde, dass man bevor die neue Seite einlagern darf, man erst noch die alte Seite auslagern muss, weil diese verändert worden ist, dann erkennt man eine weitere Leistungssteigerungsmöglichkeit, wenn man vom System aus ständig dafür sorgt, dass immer genügend freie Kacheln zur Verfügung stehen. D.h. immer dann, wenn der Vorrat an wirklich freien Kacheln unter einen bestimmten Wert fällt, dann wird man, quasi im Vorgriff den Clockalgorithmus wieder weiterlaufen lassen, bis er genügend viele freie, überlagerungsfähige Seiten/Kachelkombinationen gefunden hat (siehe Reinigungsstrategie). Wird dabei eine modifizierte Seiten/Kachelkombination entdeckt, dann wird deren Inhalt zur Sicherheit schon mal auf die Platte ausgelagert. Wenn man zusätzlich berücksichtigt, dass der eigentliche Datentransfer von und zur Platte um Größenordnungen höher ist, als der Seitentauschalgorithmus bzw. das Modifizieren der Seitentabelle, dann sieht man sehr leicht, dass diese Maßnahme durchaus erfolgreich die Systemeffizienz verbessern kann.

Jeder noch so gute Seitenersetzungsalgorithmus (paging policy) muss jedoch scheitern, wenn man den Zugang zum virtuellen Speicher nicht kontrolliert.

13.6.4 Reinigungsstrategie

Gereinigt werden muss eine Hauptspeicherkachel immer dann, wenn sie mittlerweile verändert worden ist, wenn sie also eine "schmutzige Seite" (*dirty page*) enthält, wenn auf sie schreibend zugegriffen worden ist. Analog zum Demandpaging wird beim Demandcleaning erst dann mit der Reinigung begonnen, wenn der Ersetzungsalgorithmus die Kachel für eine neue Seite benötigt. Dies hat zur Folge, dass der Seitentausch doppelt so lange dauert wie bei einer sauberen Seite.

Alternativ dazu könnte man auch im "Einklang mit der gewählten Ersetzungsstrategie" modifizierte Seiten, auf die schon längere Zeit nicht mehr zugegriffen worden ist "quasi in einem Aufwasch" auf Platte auslagern, um so den Vorrat an freien Kacheln wieder zu erhöhen.

Was macht man jedoch mit modifizierten Seiten, die bevor sie ausgelagert werden konnten, schon wieder referenziert worden sind?

13.6.5 Weitere Entwurfskriterien beim virtuellen Speicher

- Residente Menge an Seiten pro Adressraum (*resident set*)
- Ersetzungsspielraum (*replacement scope*)

Natürlich obliegt es dem System zu entscheiden, wie viele Seiten einem Adressraum maximal während dessen Abarbeitung zugebilligt werden können. Hierzu mag es statische oder auch dynamische Obergrenzen geben, letztere hängen sehr wohl von der aktuellen Systemlast ab. Verursacht eine Task permanent außergewöhnlich viele Seitenfehler, so liegt der Verdacht nahe, dass ihr das System einfach zu wenige Seiten zur Verfügung stellen will. Nicht jede Task braucht gleich viele Seiten, kleinere Programme kommen u.U. mit wenigen 4 KB Seiten zu recht, während größere numerische Kalkulationen allein schon wegen der enormen Datenmengen mindesten zig Seiten benötigen können. Bei ständig wiederholten Applikationen könnte man die mittlere Residentmenge vermessen und als weiterer Ladeparameter berücksichtigen.

Jede der bislang behandelten Ersetzungsstrategien kann entweder lokal oder global angewendet werden. Bei einer lokalen Ersetzung versorgt sich die Ersetzungsstrategie nur aus der aktuellen Residentmenge der Task, die den Seitenfehler verursacht hat (siehe Windows XP).

Bei einer globalen Ersetzung kann es dagegen auch zum Seitenklauen kommen, d.h. es wird das Seiten-Kachelpaar ausgewählt, das gemäß aller Seiten-Kachelpaare im gesamten System am längsten nicht mehr benutzt wurde, wenn die Ersetzung gemäß LRU erfolgen soll, wobei es völlig gleichgültig ist, ob dieses Seitenkachelpaar zum Adressraum des Threads gehört, der den Seitenfehler verursacht hat.

Der Vorteil der globalen Sichtweise ist der, dass sich die zeitlich unterschiedlichen Programmverhalten gegenseitig helfen können. Es mag einige aktivierte Adressräume geben, die sich gerade ausdehnen wollen (auf Kosten anderer), und einige, die gerade weniger Seiten benötigen (siehe Unix SV R4).

13.7 Lastkontrolle

Wie immer man auch die verschiedenen Strategievarianten miteinander kombiniert, es ist vermessen zu glauben, dass man mit Einführung des virtuellen Speicherkonzepts beliebig viele Applikationen gleichzeitig im System aktivieren kann. Wenn man beispielsweise gleichzeitig zu viele Task startet, nimmt jede Task der anderen ständig Seiten-/Kachelkombinationen weg, dieses Phänomen wird auch als THRASHING bezeichnet, da sich hierbei das Gesamtsystem nur noch schleppend vorwärts quält. Anstelle der Anwendungen ist permanent das Pagingsystem am arbeiten.

Zur Zugangskontrolle kann man entweder das Working-Set Modell verwenden, bei dem man neue Tasks nur startet, wenn für deren voraussichtliche Arbeitsmengen (working-set) genügend viele freie Kacheln zur Verfügung stehen. Man könnte andererseits auch messen, wie sich aktuell die Seitentauschrate entwickelt, falls diese eine obere Schranke überschreitet, müsste man eine oder sogar mehrere der aktivierten Tasks vorübergehend wieder aus dem Rennen um Hauptspeicher herausnehmen, indem diese dann in den Zustand ausgelagert oder swapped_out überführt werden.