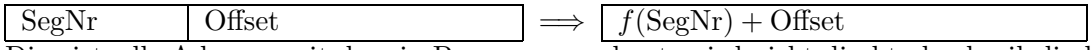
1 Virtueller Speicher

1.1 Einführung

In mordernen Rechnersystemen werden die Resourcen nicht mehr nur von einem Benutzer oder Programm gleichzeitig genutzt, sondern mehrere Programme laufen parallel und liegen in einem gemeinsamen Speicher. Da Hauptspeicher im Vergleich zu Festplattenplatz relativ teuer ist, lagert man Teile des Hauptspeichers aus, so daß dieser für andere Programme zur Verfügung steht. Neben dem Programmcode und den Daten benötigt ein Programm einen Laufzeitkeller und weiteren Speicherplatz für dynamische Daten. Der Speicher wird vom Betriebssystem zugeteilt und verwaltet. Um jedoch eine parallele Ausführung der Programme und eine Absicherung untereinander gegen überschreiben von Speicherbereiche zu ermöglichen, ist zusätzliche Hardware nötig.

1.2 Konzepte

1.2.1 Segmentierung

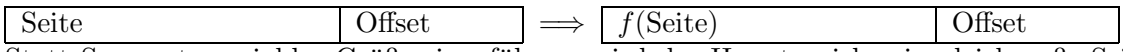


Die virtuelle Adresse, mit der ein Programm rechnet, wird nicht direkt als physikalische Adresse des Hauptspeichers verwendet, sondern wird als eine Tupel aus zwei Adressteilen betrachtet: Der höherwertige Anteil der Adresse wird als Segmentnummer angesehen, die über eine Umsetzungstabelle auf eine physikalische Adresse umgesetzt wird. Der restliche niederwertige Anteil der virtuellen Adresse wird als Offset auf diese Basisadresse betrachtet. Neben der Basisadresse für jedes Segment kann noch die Länge des Segments gespeichert werden, woduch eine automatische Überprüfung gegen Bereichsüberschreitung durchgeführt werden kann. Weitere Flags können zur Verhinderung von Schreib- oder Leseoperationen verwendet werden oder den Datenbereich gegen versehentliches Interpretieren als Programmcode verhindern. Neben der Aufteilung der virtuellen Adresse existiert noch eine Variante mit zusätzlichen Segmentregistern: Diese enthalten die Segmentnummer, so daß die komplette virtuelle Adresse als Offset benutzt werden kann.

Vorteil dieser Methode ist es, daß der physikalische Speicher vortlaufend belegt werden kann und Programm- bzw. Datensegmente an ausgerichteten Adressen beginnen.

Nachteilig ist allerdings, daß durch die Freigabe von Segmenten Löcher entstehen und diese unzusammenhängenden Bereiche nicht für neue Anforderungen als ein Bereich zur Verfügung stehen. Weiterhin können Segmente nur komplett auf Hintergrundspeicher ausgelagert werden, so daß sich alle Teile eines Programmes bzw. alle Daten immer komplett im Speicher befinden müssen.

1.2.2 Seitenverwaltung

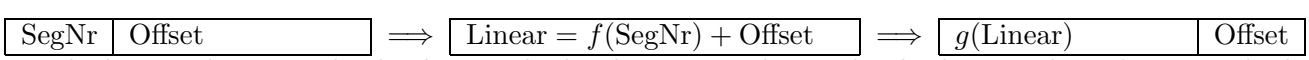


Statt Segmente variabler Größe einzuführen, wird der Hauptspeicher in gleichgroße Seiten unterteilt. Die virtuelle Adresse wird wieder als Konkatenation zweiter Werte interpretiert: Der höherwertige Anteil entspricht der Seitennummer, während der niederwertige Teil einem Offset innerhalb dieser Seite entspricht. Die Seitennummer wird über eine Tabelle in die reale Adresse übersetzt. Wegen ihrer Größe steht diese Tabelle selbst im Hauptspeicher und wird teilweise zur effizienteren Benutzung von der MMU im sogenannten *Translation lookaside Buffer* zwischengepuffert.

Vorteil dieser Methode ist es, daß virtuell aufeinanderfolgende Seiten physikalisch nicht aufeinanderfolgend sein müssen. Außerdem können unbenutzte Seiten eines Programmes ausgelagert werden, so daß nur die ständig benötigten Bereiche im Hauptspeicher gehalten werden brauchen. Da die Speicherverwaltung auf gleichgrossen Blöcken beruht, ist diese gegenüber der Segmentierung wesentlich einfacher.

Nachteilig hingegen ist die Größe der Seitantabelle. Deshalb geht man an dieser Stelle dazu über, die Auflösung der Adresse in mehrenen Stufen durchzuführen: Die Seitenadresse wird z.B. als eine Konkatenation von drei Offsets betrachtet, so daß die Auflösung der Adresse über drei Ebenen erfolgt. Unbenutzte Tabellenreferenzen werden als nicht vorhanden markiert. Ein weiterer Nachteil besteht darin, daß die Informationen über die Seiten wie Zugriffsrechte an mehrenen Stellen verteilt abgelegt sind.

1.2.3 Virtueller Speicher



Durch die Kombination der beiden Methoden kann man die Nachteile der jeweils anderen Methode weitestgehend umschiffen: Zunäechst wird die virtuelle Adresse über die Segmentierung in eine lineare Adresse übersetzt, bevor diese dann über die Seitentabelle auf physikalische Hauptspeicheradressen abgebildet wird.

Um durch die zweistufige Umsetzung keine Zeit zu verlieren, werden diese beiden Funktionen oft in einem einzigen Schritt durchgeführt.

1.3 Fazit

Virtueller Speicher ist für heutige Multiuser-Multitasking Systeme unerlässlich: Neben der Absicherung der Programme untereinander sorgt dieses Konzept dafür, daß Speicherresourcen optimal ausgenutzt werden können: Durch das Auslagern von Informationen auf Hintergrundspeicher kann auf eine teure Hauptspeichererweiterung verzichtet werden.

2 Cache

2.1 Einführung

Warum Caches:

Bandspeicher	TByte		Kapazität groß
Plattenspeicher	GByte	10ms	
Hauptspeicher	MByte	50ns	
Cache	KByte	15ns	
Register	Byte	3ns	Zugriffszeit klein

- Wahlfreier, Sequentieller, Assoziativer Zugriff
- Parallele Transfers zwischen Hirarchien
- Ausnutzung von Blocktransfers
- Entlastung von Bussen in MP-Systemen: Zugriffskonflikte auf gemeinsamen Speicher, Multiport-Speicher aufwendig und teuer, verschränkter Speicher beseitigt nicht das Problem, lokaler Cache bringt Coherenz-Problem
- Ausgewogene Bandbreite: $B_{Mem} = B_{CPU} + B_{IO}$
- Scheinbar schneller Speicher soll transparent sein
- Zukunft unbekannt, Vergangenheit extrapolieren
- Hit Rate h, Miss Rate $1 - h$, Zugriffszeit bei Hit t_c, Zugriffszeit des HS t_M, effektive Zugriffszeit $t_c \leq h\dot{t}_c + (1-h)\dot{t}_M \leq t_M$, aber zusätzliche Verzögerung bei Miss und Write-Back führt zu $0.3 \ldots 0.7 t_M$.

Daten Konstanten, Variablen(ro), Variablen(r/w)

Code ro, Sprünge(Schleifen, Unterprogramme), Prozeßwechsel

2.2 Konzepte

Look-through Die Anfrage wird zunächst an den Cache gestellt. Wenn die Daten dort nicht vorhanden sind, wird die Anfrage danach an den Hauptspeicher weitergereicht. Diese Strategie sorgt für eine Entlastung des Busses, verzögert aber Zugriffe bei einem Cache-Miss.

Look-aside Anfragen werden parallel an Speicher und Cache gestellt. Wenn der Eintrag im Cache vorhanden ist, wird der Speicherzugriff angebrochen und die Anfrage aus dem Cache beantwortet. Diese Variante ist schneller, benötigt aber jedesmal den Bus.

Unified Cache Daten und Code teilen sich einen Cache gemeinsam. Da Code und Daten unterschiedliche Lokalitätseigenschaften haben, können diese nicht berücksichtig werden.

Split-Cache Daten und Code haben jeweils einen eigenen Cache. Nachteilig daran ist, daß die Ausnutzung der beiden Caches sehr unterschiedlich sein kann und deshalb weniger effizient sind.

On-/Off-Chip Der Cache kann sich entweder direkt auf dem gleichen Chip wie die CPU befinden oder extern als zusätzlicher Baustein ausgelegt sein. Die On-Chip-Variante ermöglicht den Zugriff auf den Cache mit vollem Prozessortakt, wohingegen die zweite Variante leichter erweiterbar ist. Eine Mischform von beiden Varianten existiert zum Beispiel beim *Pentium Pro*, bei dem ein Cache zwar auf einem eigenen Chip realisiert ist, welcher sich aber im gleichen Gehäuse wie die CPU befindet und deshalb mit dem vollen Prozessortakt betrieben wird.

elle/Physikalisch On-Chip-Caches können mit den virtuellen Adressen vor der MMU oder mit physikalischen Adressen nach der Adressumsetzung arbeiten. Erstere sparen sich die zeitaufwendige Adressumsetzung, haben allerdings Nachteile beim Taskwechsel, bei dem die Caches invalidiert werdne müssen.

L1/L2-Cache Mehrere Caches können in Reihe geschaltet werden, um von den unterschiedlichen Vorteilen von Look-through und Look-aside gebrauch zu machen. Da der Platz auf der CPU-Chip begrenzt ist, ist so auch eine Erweiterung der Cachegröße möglich.

Backside Mit follter CPU-Tankfrequenz betrieberner Cache. Kann über einen eigenen *Cache-Bus* an die CPU angebunden sein.

2.2.1 Einlagerungsstrategien

Zunächst ist der Cache abgeschaltet. Durch entsprechende Prozessorbefehle wird der Cache zunächste geleert bevor er aktiviert wird. Um den Cache mit Einträgen zu füllen, gibt es zwei Strategien:

Prefetch Der Cache wird durch extra Befehle gefüllt. Dazu muß das Programm bzw. das Betriebssystem natrlich wissen, welche Seiten in Zukunft benötigt werden. Da in den seltensten Fällen eine Aussage über die Zukunft eines Programmes gemacht werden kann, ist diese Strategie nur selten verwirklicht. Sie bietet sich aber zum Beispiel für den *Translation-Lookaside-Buffer* der MMu an, wenn dort Segmentregister benutzt werden..

Demand Bei einem Cache-Miss wird die Zeile eingelagert, in der Hoffnung, daß diese demnächst nocheinmal benötigt wird. Setzt man eine gewisse Lokalität des Programmes voraus, d.h. Daten und Code werden mehrfach in kurzer Folge verwendet, so ist diese Strategie oft erfolgreich. Diese Strategie ist die meißtverbreiteste Variante, da sie relativ einfach zu verwirklichen ist und gute Ergebnisse erzielt.

	Hit	Miss
Read	Aus dem Cache	In Cache einlesen
Write-back	Cache aktualisieren, dirty-bit	Einlagern und aktualisieren
Write-trought with write-allocation	Beide aktualisieren	Einlagern und aktualisieren
Write-through without write-allocation	Cache/Memory aktualisieren	Memory aktualisieren

Wird keine Write-Back-Stategie eingesetzt, so kann die Verzögerung beim Schreiben durch einen Write-Buffer abgefabgeb werden, so daß die CPU schon mit dem nächsten Befehl fortfahren kann.

Cachezeilen werden nach unten hin immer größer: Während die CPU Byteweise auf den Speicher zugreift, lagert der L1-Cache normalerweise 4 Worte ein, da jeder erste Speicherzugriff durch den *Lead-Off Zyklus* langsamer ist als die Folgezugriffe. Um die Anfrage der CPU bei einem Cache-Miss schnellstmöglich zu beantworten, wird zuerst die angeforderte Speicherzelle zurückgegeben bevor die restlichen Einträge der Zeile geladen werden. Hierbei macht man sich den *Wrap-Around* zu nutze, indem man lediglich die niederwertigen Bits der Adresse zyklisch weiterzählt und die effektive Cache-Adresse beibehält.

2.2.2 Auslagerungsstrategien

Wenn alle Cache-Zeilen gefüllt sind und eine neue Zeile eingelagert werden soll, muß eine alte überschrieben werden. Dazu gibt es mehrere Strategien:

OPT Es wird die Seite entfernt, auf die am längsten nicht mehr zugegriffen werden wird. Da die Zukunf des Programmes aber im Allgemeinen unbekannt ist, ist diese Strategie zwar optimal, aber nicht zu verwirklichen.

FIFO Die Seiten werden in ihrem Set jeweils zyklisch ausgetauscht. Diese Strategie ist einfach zu implementieren, ist aber nicht gut.

Random Es wird eine beliebige Seite ersetzt.

LFU Fuer jede Zeile wird mitgezählt, wie oft auf diese Zeile zugegriffen wurde. Es wird die Zeile ersetzt, die am wenigsten oft benutzt wurde. Danach werden alle Zähler zurückgesetzt. Diese Strategie ist besser, aber die Zähler sind zu aufwendig.

LRU Es wird die Seite aus dem Set ersetzt, auf die am längsten nicht mehr zugegriffen wurde. Diese Strategie wird am häufigsten eingesetzt, da die Realisierung relativ einfach ist und gute Ergebnisse liefert. Zur Bestimmung des ältesten Eintrags wird eine Methode benutzt, die unter dem Namen *Alterungsmatrix* bekannt ist.

RNU Variante der LRU, der nur eine Seite auswählt, die zuletzt nicht mehr benutzt wurde.

2nd Chance Variante der LRU, der nur mit wenigen zusätzlichen Bits auskommt: Ein *Reference-Bit* markiert einen Zugriff seit der letzten Auslagerung, ein *Dirty-Bit* markiert einen Schreibzugriff. Der Algorithmus durchläuft zyklisch alle Zeilen und ersetzt Zeilen, deren Referenze-Bit unmarkiert ist. Markierte Zeilen werden demarkiert. Modifizierte Zeilen müssten zurückgeschrieben werden, weswegen der Aufwand teurer ist. Diese Zeilen werden deshalb benachteiligt und erst bei einem zweiten Durchlauf in Betracht gezogen.

Clock Variante der FIFO und LRU: Nichtmarkiete Zeilen werden ersetzt, markierte Zeilen werden demarkiert. Zyklische Auswahl der Seiten.

Jenachdem an welcher Stelle im System Zwischenspeicher eingesetzt werden, sind unterschiedliche Varianten sinnvoll. In Hardware-Caches gibt man sich meißtens mit *RNU* Varianten zu frieden, da diese sehr schnell sind. Bei langsameren Hintergrundmedien lohnen sich dann schon serielle Varianten wie *2nd Chance* und *Clock*, die dann auch mit größeren Blöcken arbeiten.

2.2.3 Organisation

Cache besteht aus Zeilen. Jede Zeie enthält üblicherweise mehrere (4) Worte. LSBs werden zur Wortauswahl (2,3) innerhalb einer Zeile bzw. Byteauswahl (0,1) innerhalb eines Wortes benutzt.

- Voll-assoziativ

Tag	Wort	Byte

Jede Cachezeile kann jeden Speicherblock enthalten. Dazu muß für jede Cachezeile mitgeführt werden, welcher höherwertigen Adressanteil (4-31) der Zeile zugehört. Jede Zeile erfordert einen eigenen Vergleicher, der diesen Tag mit der gesuchten Adresse vergleicht. Der Aufwand ist sehr groß, so daß sich nur relativ kleine Caches in dieser Bauform realisieren lassen. Diese Form wird z.B. in der MMU des Prozessors verwendet, um schnell bei jedem Speicherzugriff den Segmentdeskriptor zu finden.

- Direct Mapping

Tag	Line	Wort	Byte

Jede Cachzeile kann nur auf bestimmte Speicherblöcke zugreifen: Dazu wird der Hauptspeicher fortlaufend in Blöcke der Cachegrö ß eingeteilt. Jede Cachzeile kann dann nur relativ zu einem Offset auf vielfache der Cachegröße zugreifen. Auch hier erfordert jede Zeile eine Markierung, die den höherwertigen Adressanteil enthält. Allerdings ist dieser Tag wesentlich schmaler und benötigt deshalb weniger Speicher. Außerdem ist nur noch ein Vergleicher nötig, der den höherwertigen Adressanteil mit dem Tag der einen Cachezeile vergleicht.

Diese From der Organisation hat den Nachteil, daß Adressen, die um vielfache der Cachegröße auseinanderliegen, sich gegenseitig ausschließsen. Bei ungünstiger Anordnung der Daten im Speicher kann also der Cache seine Wirkung verlieren.

- Set-assoziativ

Tag	Set	Wort	Byte

Diese From ist eine Mischung der beiden vorangegangenen Organisationen: Mehrere Direct-Mapping-Caches e werden parallel geschaltet. Dadurch wird bei vorgegebener Cachegröße zwar die Anzahl der Zeilen verringert, aber jede Zeile kann dafür mehrere Speicherblöcke aufnehmen, die um vielfache der reduzierten Cachegröße auseinanderliegen. Die Anzahl der benötigten Vergleicher hängt von der Anzahl der parallel geschaltenen Blöcke e ab. Da die Anzahl der Zeilen mit steigendem e abnimmt, vergößert sich dadurch die Bitbreite des zu speichernden Tags.

- Sector-Mapping

Tag	Block	Byte

Nicht mehr gebräuchliche Variante mit zweistufiger Umsetzung: In einer ersten Stufe wird der höherwertige Anteil über eine vollassoziative Tabelle in einen Tabellenindex übersetzt. Dieser wird mit weiteren Bits aus der virtuellen Adresse erweitert und dient als Index in den Cache, der Zeilen aus

dem Hauptspeicher enthält. Zeilen müssen jedoch nicht vollständig sein, so daß für eine Cachezeile mehrere *Valid-Bits* vorhanden sind.

Die Variante des Direct-Mapping-Cache ergibt sich für $e = 1$, die des Voll-assoziativen-Caches für $e = m$. Typische Größen für e liegen zwischen 2 und 8. Dies ist die gebräuchlichste Form der Cache-Organisation.

2.2.4 Probleme

Solange nur lesend auf den Speicher zugegriffen wird, führen Caches zu den gewünschten Ergebnissen wie zum Beispiel Busentlastung bei MP-Systemen. Bei schreibendem Zugriff tritt jedoch ein Cohärenzproblem auf: Die Caches enthalten unterschiedliche Kopien der selben Speicheradresse. Um dies zu verhindert, müssen bei Schreibzugriffen die Caches invalidiert bzw. aktualisiert werden. Dazu lauschen alle Caches auf Adresszugriffe auf dem gemeinsamen Bus, um bei Kollisionen die notwendigen Aktionen durchführen zu können. Das bekannteste Protokoll dazu ist das *MESI*-Protokoll, das zwischen den 4 Zuständen Modified, Exlusive, Shared und Invalid für jede Cachzeile unterscheidet.

Dadurch ist aber noch nicht das Problem beseitigt, daß jeder Schreibzugriff auf dem Bus sichtbar sein muß, wodurch dieser wieder zu einem gemeinsamen Engpass wird. Es ist daher sinnvoll zwischen Speicherbereichen zu unterscheiden, die von mehreren Prozessoren benutzt werden bzw. solchen, die nur von einzelnen Mastern beschrieben werden.

2.3 Fazit

Caches werden an vielen Stellen eingesetzt, nicht nur zwischen CPU und Hauptspeicher. Weitere Anwendungen findet dieses Konzept innerhalb der CPU zur Sprungvorhersage (BPC) bzw. Sprungzielspeicher (BTC), innerhalbt der MMU zur schnellen Addressumsetzung (TLB) sondern auch auf tieferen Ebenen der Speicherhirarchie: So werden Festplattenzugriffe im schnelleren Hauptspeicher zwischengepuffert oder Festplatten als Zwischenmedium für speicherhunrige Daten von CDs oder Bändern benutzt. In Multiprozessorsystemen sind die unerläßlich, um die benötigte Speicherbandbreite zur Verfügung zu stellen.

3 Busarbitrierung

3.1 Einführung

Das Rechnersystem bestzht nicht nur aus einem Prozessor, sondern aus vielen Bausteinen, die miteinander kommunizieren müssen. Diese lassen sich in zwei Kategorien unterteilen: Bausteine wie Haupspeicher und Grafikkarte werden von den aktiven Komponenten adressiert und übernehmen die Daten vom Bus. Die aktiven Komponenten wie CPU und DMA-Controller starten diese Buszyklen und schreiben die Daten auf den Bus. Der Bus dient somit als gemeinsames Kommunikationsmedium und wird sich von allen Einheiten geteilt. Es ist nun erforderlich, ein Protokoll aufzustellen, nach dem sich die Komponenten um die Zuteilung des Busses bemühen, damit immer nur eine Einheit die aktive Steuerung des Busses übernimmt. Alle nicht aktiven Komponenten deaktivieren ihre Ausgangstreiber (*Tri-state*) und beobachten nur die Steuersignale.

3.2 Konzepte

Die Arbitration des Busses kann auf verschiedene Art und Weise realisiert werden:

	Zentral	Dezentral
Local	Zentral	Daisychain
Global		ID

Local Jeder aktive Busteilnehmer entscheidet anhand der verschiedenen Signalen der Nachbarschaft, ob er das Recht für den Bus erhält.

Global Alle Bauelemente sehen die Anforderungen der übrigen Teilnehmer. Nachteilig an dieser Variante ist der hohe Verdrahtungsaufwand bei steigender Anzahl der Komponenten.

Zentral Es existiert ein zentrales Bauelement, von dem aus Leitungen zu allen Busteilnehmern führen. Die Logik wählt aus mehreren Anforderungen dann ein Bauelement nach einer inneren Logik aus.

Dezentral Jeder Busteilnehmer hat seine eigene Logik, die lokal entscheidet, ob die Busbenutzung erteilt wird oder nicht.

3.2.1 Daisy-Chain

Die *Daisy-Chain* ist ein Beispiel für eine dezentrale, lokale Busarbitration. Die Bushoheit liegt bei einer CPU, von der der Bus per *wired-or*-verknüpften *Bus-Request* angefordert werden kann. Sobald die CPU ihren Buszyklus beendet hat, gibt sie den Bus per *Bus-grant* an einen Interessenten ab. Diese sind wie die Glieder einer Kette aufgereiht und geben das *Bus-grant* nur weiter, wenn sie selbst kein Interesse an dem Bus haben. Durch die Reihenfolge ist eine Priorisierung der Komponenten implizit vorgegeben: Komponenten am Anfang der Kette können Komponenten am Ende aushungern, indem sie das *Bus-grant*-Signal nicht weitergeben.

Verhinder läßt sich dies durch das Einführen eines weiteren Signals *Busy*: Ist dieses aktiv, dürfen keine weiteren (höherprioritisierten) Master den Bus übernehmen. Nachteilig daran ist, daß wichtige Transfers solange blockiert bleiben, bis alle Transaktionen den Zykluses abgearbeitet sind. Weiter verbessern läßt sich das Konzept dadurch, daß die Arbitration überlappend mit noch laufenden Übertragungen stattfindet. Dadurch wird die Zeit verringert, in der ein Bus nicht benutzt wird.

3.2.2 Zentraler Arbiter

Anstatt die Logik auf alle Busteilnehmer zu verteilen, existiert hier ein einziger Baustein, bei dem alle Interessierten über eine eigene Leitung den Bus anfordern können. Die Anforderung leitet der Baustein entweder an den Prozessor weiter oder entzieht bei einem symmetrischen System dem momentanen Benutzer das Recht oder wartet ab, bis der Bus frei wird und vergibt dann das Recht neu.

Nachteilig an dieser Lösung ist der erhöhte Verdrahtungsaufwand. Die Anzahl der Busteilnehmer ist durch die Anzahl der vorgesehenen Anschlüsse beim Arbiter begrenzt. Ist dennoch eine Erweiterung nötig, so können mehrere Arbiter kaskadiert werden.

Vorteilhaft ist bei dieser Lösung, daß die Logik in einem Chip liegt und dort beliebig realisiert werden kann.

3.2.3 ID Bus

Alle Komponenten erhalten bei der Initialisierung eine eindeutige Nummer, deren Bitmuster sie bei Interesse auf einen *ID-bus* legen. Der Bus fungiert als ein *Wired-or*. Jeder Teilnehmer vergleicht nun das Muster des Busses mit seiner eigenen Nummer. Beim höchsten Bit anfangend nimmt ein Teilnehmer seine Nummer vom *ID-bus*, wenn anstatt seiner 0 auf dem Bus eine 1 liegt. Nach einer Einschwingphase bleibt auf dem Bus nur noch die Nummer stehen, die am größten ist. Der Teilnehmer mit dieser Nummer erhält den Bus. Auch hier müssen noch weitere Leitungen eingeführt werden, um weitere Teilnahmen während eines Zykluses zu unterbinden und den Bus zu belegen. Diese Variante hat den Vorteil, daß sich dieser Bus leicht erweitern läßt und die Prioritäten frei programmierbar sind.

3.3 Fazit

Je nach Anwendungsgebiet und gewünschten Funktionsumfang existieren verschiedene Möglichkeiten, ein gemeinsam genutztes Medium einem Teilnehmer exklusiv zur Verfügung zu stellen. Neben den hier vorgestellten Verfahren, die hauptsächlich innerhalb eines Gerätes zum Einsatz kommen, gibt es für großflächigere Netzwerke weitere Verfahren wie *CSMA/CD* bei Ethernet. Bei steigener Anzahl von Teilnehmern entwickelt sich ein einzelner Bus jedoch als Engpaß, so daß man hier auf Mehrbussystme ausweichen muß, die über sogenannte *Bridges* miteinander verbunden werden.