— VS —

Verteilter gemeinsamer Speicher

Verteilte Systeme, ©Wolfgang Schröder-Preikschat

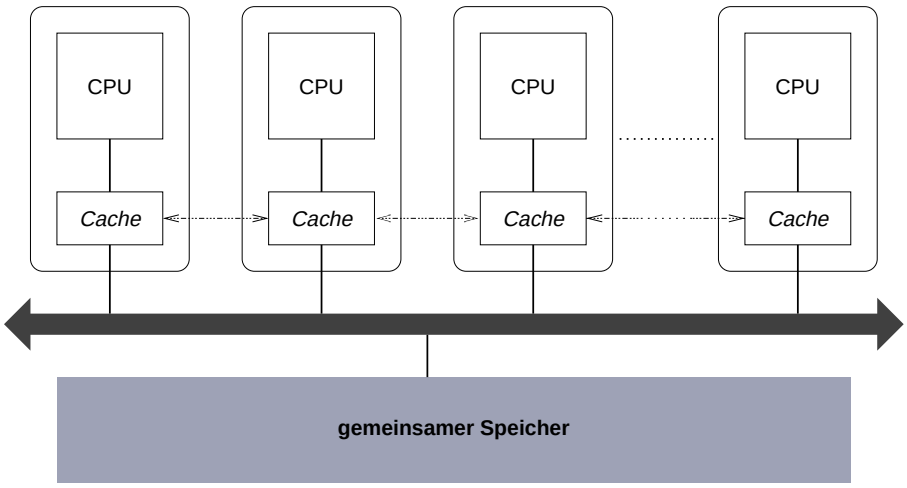
Überblick

Wat is. . . ?

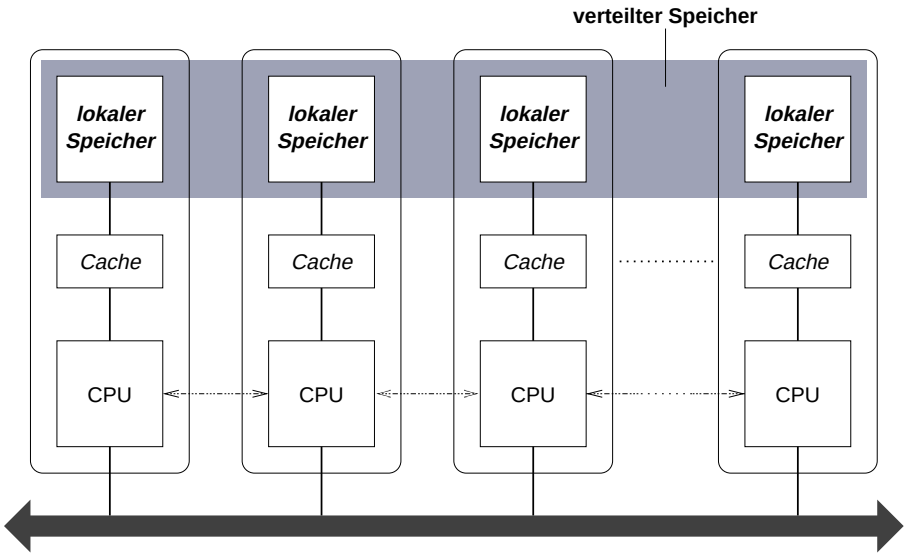
DSM *distributed shared memory* (nach [4]):

- verteilter gemeinsamer genutzter Speicher ist eine **Abstraktion**, die für die gemeinsame Nutzung von Daten zwischen Prozessen in Rechnern, die keinen gemeinsamen physischen Speicher besitzen, verwendet wird
- Prozesse greifen lesend/aktualisierend darauf zu und er erscheint ihnen wie ganz normaler Speicher innerhalb ihres **Adressraum**s
- ein **Laufzeitsystem** stellt sicher, dass Prozesse, die auf unterschiedlichen Rechnern ausgeführt werden, die auf anderen Rechnern vorgenommenen Aktualisierungen „sehen“ [Transparenz?]
- es ist, als ob die Prozesse auf einen einzelnen gemeinsam genutzten Speicher zugreifen, aber in Wirklichkeit ist der physische Speicher verteilt

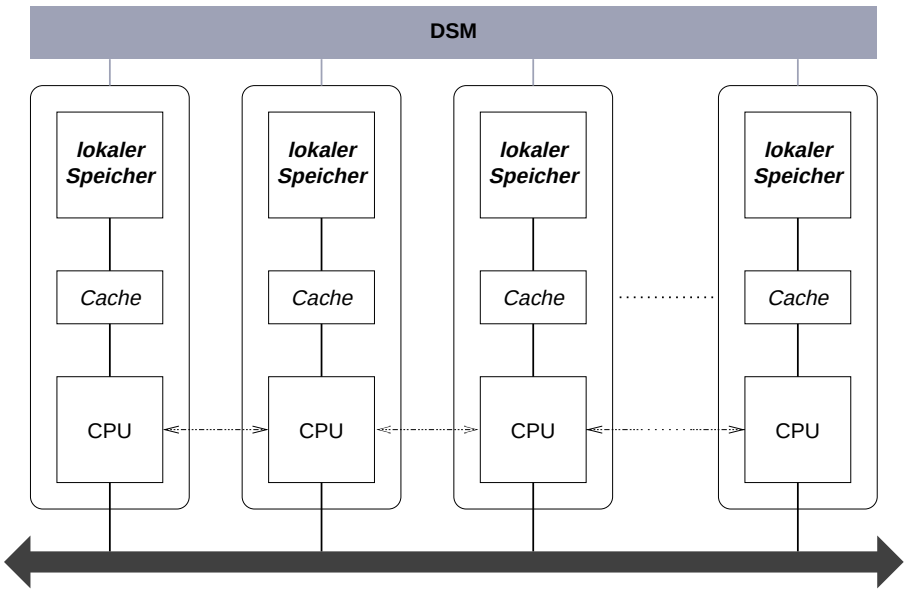
Konventioneller gemeinsamer Speicher



Verteilter (nicht-gemeinsamer) Speicher



Verteilter gemeinsamer Speicher



Implementierungsansätze

Hardware NUMA (*non-uniform memory access*) Rechnerarchitekturen

- die Adressierung von DSM-Operanden besitzt globale Signifikanz
- Lese-/Schreibbefehle können (spezialisierte) „Fernaufrufe“ bewirken

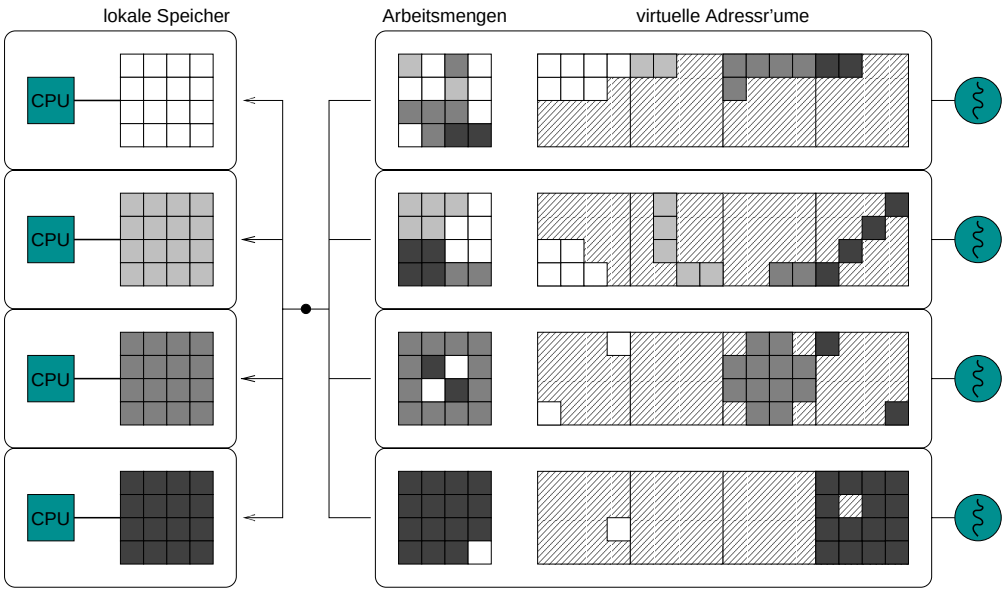
Betriebssystem gekachelter virtueller Speicher (*paged virtual memory*)

- DSM nimmt einen bestimmten Bereich im virtueller Speicher ein
- derselbe Adressbereich im Adressraum eines jeden teilnehmenden Prozesses

„communication support layer“ plattformunabhängiges Laufzeitsystem

- sprachgestützt und/oder als problemorientierte Middleware realisiert
- Abbildung von DSM-Datenelementen auf lokale Datenelemente

DSM als gekachelter virtueller Speicher



DSM $\Longleftrightarrow$ Virtueller Speicher

Platzierungsstrategie (*placement policy*) ***wohin*** anlegen?

- Lokalität und Lastausgleich sind weitere Kriterien zur Allokation der Rahmen
- die global eindeutige Lage der Kacheln[1] im Adressraum ist zu beachten

Ladestrategie (*fetch policy*) ***wann*** einlagern?

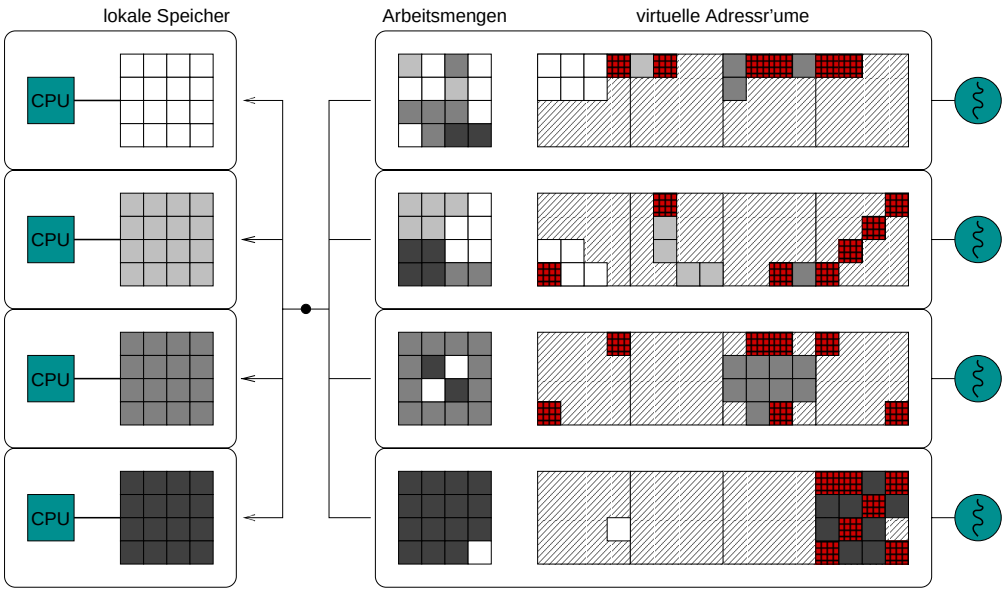
- Kacheln des DSM auf Anforderung, im Voraus oder kombiniert „fernladen“
- Zugriffsarten (lesen/schreiben) entscheiden über Aktualisierungsoptionen

Ersetzungsstrategie (*replacement policy*) ***was*** verdrängen?

- betrifft weiterhin nur die lokale Arbeitsmenge (*working set*) des Prozesses

[1] Der Begriff „Seite“ ist synonym zu „Kachel“, er bezieht sich auf den allen am DSM teiln. Prozessen gleichen virtuellen Adressraum. Dagegen bezieht sich der Begriff „Rahmen“ auf den jeweils lokalen physikalischen Adressraum.

Kacheln sind Gegenstand der Replikation



DSM ist. . .

pros

hauptsächlich ein Werkzeug für **parallele Anwendungen** oder für beliebig verteilte Applikationen bzw. Applikationsgruppen, für deren einzelne gemeinsam genutzte Datenelemente ein direkter Zugriff möglich ist

cons

allgemein weniger gut geeignet für Klient/Anbieter-Systeme (*client/server systems*), wo die Klienten normalerweise die vom Anbieter bereitgestellten Betriebsmittel als abstrakte Daten betrachten und unter Verwendung von Anforderungen bzw. Fernaufrufe darauf zugreifen (aus Gründen der Modularität und der Sicherheit)

Verteilter gemeinsamer Speicher als Dienst

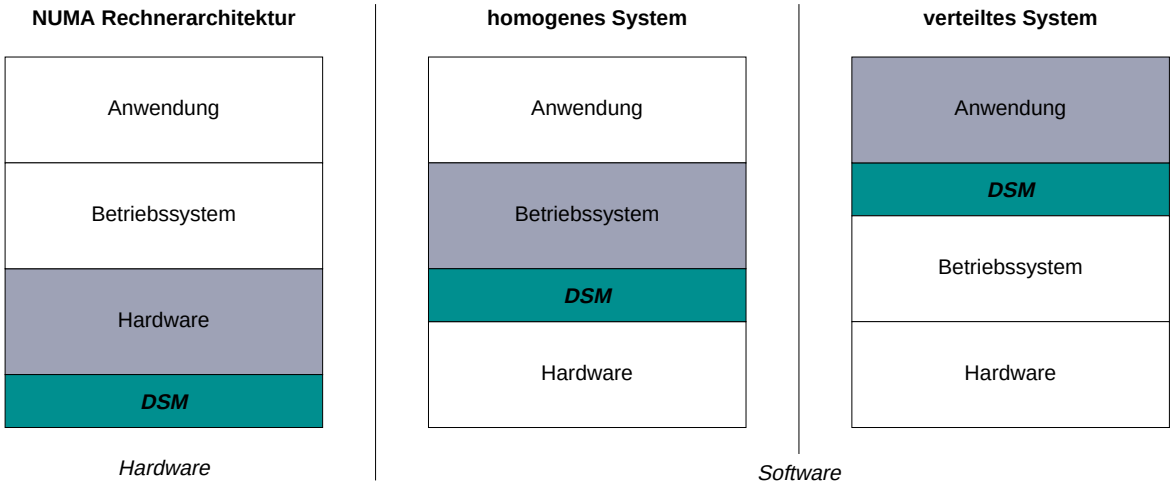
DSM Anbieter stellen gemeinsam nutzbaren Speicher bereit

- den Adressraum des Anbieters z.B. als „*RAM disk*“ verfügbar machen
  – genauer: den physikalischen, logischen oder virtuellen Adressraum
- eher: **speicherabgebildete Dateien** (*memory-mapped files*) à la Multics [9]

DSM Klienten greifen auf den zur Verfügung gestellten Speicher zu

- ferne Speicherbereiche wurden lokal in virtuelle Adressräume eingeblendet
  – die Ladestrategie der (lokalen) Speicherverwaltung setzt Fernaufrufe ab
- die beteiligten Betriebssysteme realisieren ein gewisses **Konsistenzmaß**

Ebenen der Implementierung



Replikationssystem „DSM“

Klientenprogramme können Operationen auf Objekte ausführen, als handelte es sich um eine einzige Kopie jedes Objektes, wobei aber in Wirklichkeit (z.B. aus Gründen höherer Leistung oder Fehlertoleranz) mehrere physische Repliken eben dieser Objekte vorliegen können:

- Applikationsprozesse stellen die *Abstraktion einer Objektauflistung* dar
  - im DSM-Fall bezieht sich die Auflistung (mehr oder weniger) auf Speicher
- die Ansätze variieren hinsichtlich *Objektbegriff* und *Objektadressierung*
  - fortlaufende Bytes, Objekte auf Sprachebene oder unveränderliche Daten

Zusammensetzung von DSM (1)

fortlaufende Bytes der Zugriff erfolgt wie auf normalen virtuellen Speicher [7, 3]

- gemeinsam benutzte Objekte sind direkt adressierbare Speicherpositionen
  - einzelne {8,16,32,64}-Bit Einheiten bzw. Verbünde davon
- SMP[2]-Programme sind mit geringer oder keiner Anpassung übertragbar

Objekte auf Sprachebene linguistische Unterstützung zur parallelen/verteilten Programmierung [1]

- Objektsemantik kann genutzt werden, um Konsistenz zu erzwingen
  - Serialisierung der Operationen ist objektspezifisch (nicht seitenspezifisch)
- allgemein sind die Objekte von höherer, anwendungsbezogener Semantik

[2] *shared-memory processor/processing*

Zusammensetzung von DSM (2.1)

unveränderliche Daten Prozesse können Datenelemente lediglich hinzufügen, lesen und entfernen; DSM als **Tupelraum** [2]

- **Tupel** gleich Folge typisierter Datenfelder: `<"fh",2003> <"wosch",2021>`
  - beliebige derartiger (und anderer) *Tupeltypen* bilden den Tupelraum
- gemeinsame Nutzung von Daten durch Zugriff auf denselben Tupelraum
  - Tupel werden durch

`write()`	hinzugefügt	
`read()`	ausgelesen	(invariant)
`take()`	entfernt	

- es gibt keine Möglichkeiten, Tupel (im Tupelraum) zu verändern

Zusammensetzung von DSM (2.2)

unveränderliche Daten (Forts.) der direkte Zugriff auf und Änderung von Tupel im Tupelraum ist nicht möglich

- lesen/entfernen erfolgt durch Angabe von *Tupelspezifikationen*
  - der Tupelraum liefert alle mit der Spezifikation übereinstimmenden Tupel
  - die Operation kehrt erst zurück, wenn mind. ein Tupel gefunden wurde
- Tupel sind zu ersetzen, um sie verändert im Tupelraum vorliegen zu haben:

```
<string, count> := fooTS.take(<"Counter", integer>);
fooTS.write(<"Counter", count + 1>);
```

- der Ansatz vermeidet das Auftreten von „*race conditions*“ [warum?]

Synchronisierungsmodell

- um DSM nutzen zu können, ist „*verteilte Synchronisation*“ zu realisieren
  - wenn z.B. immer $a = b$ gelten soll, sind folgende Sequenzen kritisch:

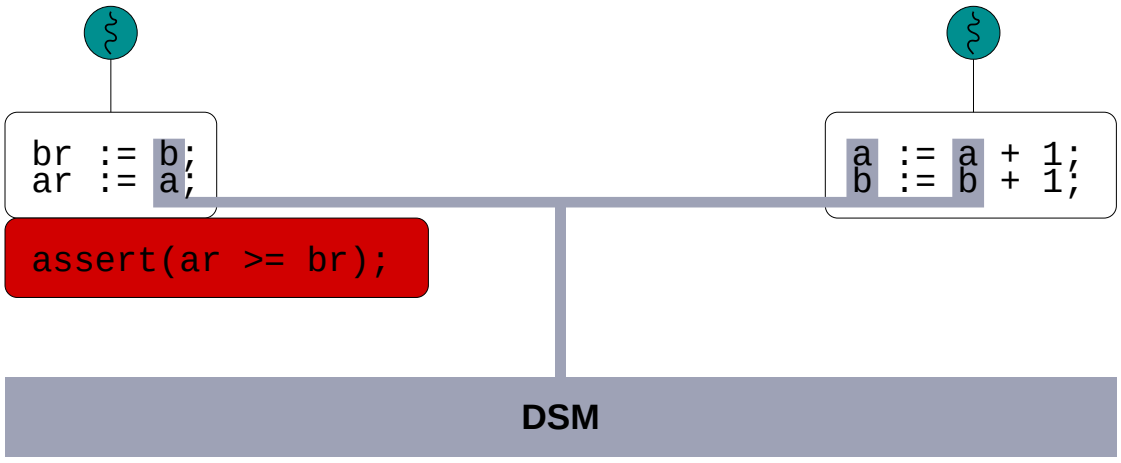
```
a := a + 1;
b := b + 1;
```

  - gegenseitiger Ausschluss (*mutual exclusion*) vermeidet mögliche Inkonsistenz
- Synchronisierungskonstrukte sind auf Basis von Botschaftenaustausch realisiert
  - konventionelle `TAS`- oder `CAS`-Befehle sind im gekachelten DSM ineffizient
  - bedingt durch die darauf aufsetzenden ({,nicht-}blockierenden) Verfahren
- Synchronisation/Koordination erfolgt explizit auf Anwendungsebene

Konsistenzproblem

- besteht bei Systemen, die den Inhalt gemeinsam genutzten Speichers replizieren
  - ein bekanntes Problem der „Geheimlager“ (*caches*) konventioneller Systeme
  - bei DSM übernehmen lokale Speicher (u.a.) „Geheimlagerungsaufgaben“
  - die **Aktualisierungen** geheim gelagerter Variablen sind zu propagieren
- in DSM werden Aktualisierungen ggf. gepuffert, d.h. gebündelt ausgeführt
  - dadurch können *Kommunikationskosten* amortisiert werden
  - Konsequenz: Aktualisierungen wirken ggf. (erheblich) zeitlich verzögert
- das *Konsistenzmodell* unterscheidet sich von dem konventionellen Speichers

Gemeinsam genutzte (DSM) Variablen



Konsistenz $\Longleftrightarrow$ Aktualisierungsreihenfolge

Intuitiv sollte die Zusicherung `assert(ar >= br)` des Beispiels ($\rightarrow$ S. 19) nach Zugriff auf die beiden DSM-Variablen a und b eigentlich jederzeit gelten:

konventioneller Speicher a wird immer vor b aktualisiert

- Wertekombinationen für ar und br sind: $\{0, 0\}$, $\{1, 0\}$, $\{1, 1\}$ $+$

DSM b kann jedoch durchaus auch vor a aktualisiert werden[3]

- damit ist auch als Wertekombination für ar und br gültig: $\{0, 1\}$ $-$
- sequentielle Ausführung heisst nicht „gleiche sequentielle Aktualisierung“

[3] Als Folge von Optimierungen, Netzausfälle oder einfach nur des gegenwärtigen Laufzeitverhaltens.

Zentrale Frage zum Speicherkonsistenzmodell

Wenn ein Lesezugriff auf eine Speicherstelle erfolgt, sollen welche Werte von Schreibzugriffen auf diese Stelle dem Lesevorgang bereitgestellt werden? Antwort:

schwächstes Extrem von jedem Schreibvorgang, der vor dem Lesen erfolgte

- prinzipiell können Aktualisierungen unendlich lange verzögert werden
- damit ist das Modell zu schwach, um eigentlich noch sinnvoll zu sein

stärkstes Extrem den aktuellsten, der vor dem Lesen geschrieben wurde

- die Bedeutung von „aktuellsten Wert“ ist dabei höchst problematisch
  - Lese-/Schreiboperationen treten nicht zu genau einem Zeitpunkt auf
  - Eintrittszeitpunkte von Ereignissen sind nur bedingt eindeutig bestimmbar
- damit ist das Modell zu stark, um noch umsetzbar zu sein

Zum „aktuellsten Wert“. . .

Das Problem basiert einfach auf unserer beschränkten Fähigkeit, Ereignissen auf verschiedenen Knoten ausreichend genaue **Zeitstempel** zuzuweisen, um die **Reihenfolge** festzustellen, in der zwei Ereignisse stattgefunden haben, oder um festzustellen, ob sie *gleichzeitig* stattgefunden haben. Es gibt keine absolute globale Zeit, die wir heranziehen können. [4]

Uhrauflösung aufeinander folgende Ereignisse weisen nur dann unterschiedliche Zeitstempel auf, wenn die *Zeitspanne zwischen den Aktualisierungen des Uhrwerts* kleiner als das Zeitinterval zwischen den aufeinander folgenden Ereignissen ist.

Grundlegende Konsistenzmodelle

Atomare Konsistenz

Sei $R(x)a$ eine Leseoperation, die den Wert a von der Speicheradresse x liest, und $W(x)b$ eine Schreiboperation, die den Wert b an die Speicheradresse x schreibt.

Definition 1. [AC1] *Wenn $R(x)a$ in einer verzahnten Abfolge von Operationen vorkommt, dann ist $W(x)b$ die letzte davor auftretende Schreiboperation, oder es tritt keine Schreiboperation vor ihr auf und a ist der Ausgangswert von x.*

D. h., eine Variable kann nur durch eine Schreiboperation geändert werden.

Definition 2. [AC2] *Die Reihenfolge der Operationen in der Verzahnung ist konsistent zu den Echtzeiten, zu denen die Operationen bei der tatsächlichen Ausführung auftreten.*

In Analogie zur ***Linearisierbarkeit*** von Operationen auf replizierte Objekte [4].

Sequentielle Konsistenz

Ein DSM-System ist dann „sequentiell konsistent“ [6], wenn es *für jede Ausführung* eine Verzahnung der von allen Prozessen abgesetzten Operationsfolgen gibt, die die beiden folgenden Kriterien erfüllt:

Definition 3. [SC1] *Wie Definition 1. der atomaren Konsistenz.* → S. 24

Definition 4. [SC2] *Die Reihenfolge der Operationen in der Verzahnung ist konsistent zu der Progammreihenfolge, in der die einzelnen Klienten sie ausgeführt hat.*

Im Gegensatz zu Definition 2. der atomaren Konsistenz bezieht sich Kriterium SC2 nicht auf eine zeitliche Reihenfolge der Operationen, wodurch sequentielle Konsistenz implementierbar ist — und relativ hohe Kosten verursacht → S. 35

Abgeschwächte sequentielle Konsistenz

Kohärenz eines der schwächeren Modelle mit exakt formulierten Eigenschaften, um die (relativ hohen) Kosten sequentieller Konsistenz zu vermeiden

- alle Prozesse einigen sich auf die *Reihenfolge von Schreiboperationen auf dieselbe Speicherstelle* (nicht unbedingt auf unterschiedliche Speicherstellen)
- das *Protokoll* wirkt separat auf jede Einheit replizierter Daten — z. B. auf jede einzelne DSM-Seite
- die Ersparnis resultiert aus der Tatsache, dass Zugriffe auf zwei unterschiedliche Seiten unabhängig voneinander sind und sich durch das separat auf sie angewandte Protokoll gegenseitig nicht verzögern müssen

Schwache Konsistenz

Verfeinerter Ansatz zur weiteren Kostensenkung sequentieller Konsistenz [5]:

- Wissen über *kritische Abschnitte* wird genutzt, um die Speicherkonsistenz zu lockern, während der Anschein erweckt wird, sequentielle Konsistenz zu implementieren
  - *wechselseitiger Ausschluss* impliziert, dass nur noch ein einziger Prozess auf DSM-Datenelemente zugreifen kann
  - demzufolge ist es redundant, Aktualisierungen an diese Elemente weiterzugeben, bis der Prozess den kritischen Abschnitt verlässt
- obwohl die Werte der Elemente eine gewisse Zeit lang „inkonsistent“ sind, kann während Phasen wechselseitigen Ausschlusses kein Zugriff darauf erfolgen

Weitere. . . (1) — Einheitliche Modelle

kausale Konsistenz Lese- und Schreiboperationen werden über die „Geschehen-vor"-Beziehung verknüpft

Prozessorkonsistenz ausgehend vom kohärenten Speicher einigen sich alle Prozesse auf die Reihenfolge von zwei Schreibzugriffen, die vom selben Prozess ausgehen

pipelined RAM alle Prozesse einigen sich auf die Reihenfolge, in der die Schreiboperationen von einem bestimmten Prozessor abgesetzt werden

Die Gemeinsamkeit der Modelle besteht darin, nicht zwischen verschiedenen Speicherzugriffsarten zu unterscheiden.

Weitere. . . (2) — Hybride Modelle

Freigabekonsistenz kritische Lese-/Schreiboperationen sind als *release*-, *acquire*- oder *non-synchronization*-Zugriffe zu kennzeichnen

Eintrittskonsistenz jede gemeinsam genutzte Variable ist zu einem Synchronisationsobjekt gebunden und ein Prozess, der die „Sperre" als erster anfordert, erhält unter Garantie den neuesten Variablenwert

Bereichskonsistenz vereinfacht das Programmiermodell der Eintrittskonsistenz: die Variablen werden automatisch Synchronisationsobjekten zugeordnet

Die Gemeinsamkeit der Modelle besteht darin, zwischen normalen und synchronisierenden Zugriffen (sowie anderen Zugriffstypen) zu unterscheiden.

Zugriffsmuster

multiple-reader/single-writer (MRSW)

- lesender Zugriff legt eine lokale Kopie mit nur Lesezugriffsrecht an und entfernt das Schreibzugriffsrecht vom „Referenzobjekt“
- schreibender Zugriff verwirft all vorhandenen Kopien und definiert ein neues „Referenzobjekt“ mit Lese-/Schreibzugriffsrecht

multiple-reader/multiple-writer (MRMW)

- lesender Zugriff legt eine lokale Kopie mit nur Lesezugriffsrecht an, ändert jedoch nichts an den ggf. bestehenden Schreibzugriffsrechten
- schreibender Zugriff legt ggf. eine lokale Kopie von dem „Referenzobjekt“ [welches?] mit Lese-/Schreibzugriffsrecht an

SRSW (keine Replikation) und SRMW (ohne Relevanz) sind weniger bedeutsam

Aktualisierungsoptionen

Der Schreibvorgang auf ein Replik eines DSM-Objektes muss die Aktualisierung aller anderen Exemplare desselben Repliks nach sich führen. Zur Weitergabe der Aktualisierungen von einem Prozess an die anderen gibt es zwei Ansätze:

Schreiben-Aktualisieren (*write-update*) — die Aktualisierungen erfolgen lokal, sie werden an alle Prozesse, die eine Kopie des Datenelements besitzen, kommuniziert; unterstützt MRSW- und MRMW-Zugriffsmuster

Schreiben-Entwerten (*write-invalidate*) — die Aktualisierung erfolgt nur an einer Stelle und führt dazu, dass alle Prozesse, die eine Kopie des Datenelements besitzen, aufgefordert werden, diese zu verwerfen; unterstützt SRSW- und MRSW-Zugriffsmuster

Schreiben-Aktualisieren (*write-update*)

- wichtigster Faktor (von vielen) ist die **Reihenfolgeneigenschaft** der *Multicasts*
  - sie müssen *vollständig sortiert* sein und dürfen erst dann zurückkehren, nachdem die jeweilige Aktualisierungsnachricht lokal ausgeliefert wurde
  - anschließend müssen sich alle beteiligten Prozesse auf die Reihenfolge der Aktualisierungen einigen

- Lesezugriffe erweisen sich als „billig“ im Vergleich zu Schreiboperationen
  - Prozesse lesen von lokalen Kopien, ohne dass dafür Kommunikation anfällt
  - sortierte Multicasts sind aber kostspielig, auch mit Hardware-Unterstützung

Schreiben-Entwerten (*write-invalidate*)

- ein Datenelement wird zu einem Zeitpunkt immer nur in einem Modus genutzt:

  lesender Modus Zugriff erlaubt durch einen oder mehrere Prozesse
  - das Datenelement kann „unendlich“ oft kopiert vorliegen
  - ein Schreibzugriff führt zum Multicast einer Invalidierungsnachricht
  - nach Eingang der Multicast-Bestätigung erfolgt der *Moduswechsel*

  lesender/schreibender Modus Zugriff erlaubt nur durch einen Prozess
  - andere lesende Prozesse werden bei ihrem Zugriffsversuch blockiert
  - mit Aufgabe des Schreibzugriffsrechts erfolgt der *Moduswechsel*
  - Aktualisierungen werden nur weitergegeben, wenn Daten gelesen werden

Schreibzugriffe werden nach dem Prinzip „wer zuerst kommt, der mahlt zuerst“ (*first come*, *first served*) ausgeführt, um sequentielle Konsistenz zu erreichen

„Seitenflattern“

- mögliches Problem beim „Schreiben-Entwerten“ Aktualisierungsverfahren:

  Thrashing die „Dresche“, „Tracht Prügel“; „Niederlage“
  - tritt auf, wenn die Laufzeitumgebung einen übermäßigen Zeitanteil damit verbringt, gemeinsam genutzte Daten zu entwerten und zu übertragen
  - ist jedoch immer in Relation zu der Zeit zu setzen, die Anwendungsprozesse damit verbringen, sinnvolle Arbeit zu leisten
  - liest ein Prozess wiederholt ein Datum, das ein anderer laufend aktualisiert, wird es ständig vom Schreiber übertragen und beim Leser invalidiert
  - den Datenelementen wird verschiedentlich eine „Mindestverweildauer“ gebilligt, um das Problem abzuschwächen

- „Schreiben-Aktualisieren“ kann daher (je nach Anwendungsfall) günstiger sein

Sequentielle Konsistenz im gekachelten DSM

Seitenschutz dient als Grundlage, um Datenkonsistenz durchzusetzen

- in der Praxis wird *write-update* nur dann eingesetzt, wenn Schreibzugriffe gepuffert werden können[4]
- stattdessen kommt *write-invalidate* zum Einsatz, aber nur dann, wenn die Seite auch nach dem ersten Seitenfehler schreibungeschützt bleibt und mehrere Schreibzugriffe stattfinden können, bevor die aktualisierte Seite weitergegeben wird

[4] Seitenfehlerbehandlung ist ungeeignet, jede einzelne Aktualisierung auf einer Seite zu verarbeiten. Die Behandlungsroutine wird lediglich den ersten von einer Folge von Schreibzugriffen erkennen können, es sei denn, die Seite ist bleibend schreibgeschützt, was jedoch aus Performanzgründen höchst unzweckmäßig ist.

Eigentümer und Kopienmenge

owner(p) der Prozess/Eigentümer mit der aktuellsten Version einer Seite p

- seine Adresse liefert ein Nachschlagedienst (*Manager*) oder Broadcast
  - zentraler oder verteilter Ansatz, um den Manager zu implementieren
  - er ist selbst Teil der Anwendung oder ein spezieller Anbieter
- der Manager empfängt „Seitenfehler“ und leitet diese entsprechend weiter
  - Fehlerverursacher und Eigentümer wickeln das weitere Protokoll selbst ab

copyset(p) die Menge der Prozesse, die eine Kopie der Seite p haben

- wird beim Eigentümer gespeichert und beim Schreibfehler abgeliefert
  - enthält Identifikationen/Transportadressen der teilnehmenden Prozesse
- der den Schreibfehler verursachende Prozess entwertet per Multicast

Konsistenzwahrung (1)

Lesefehler (*page fault*) tritt auf, wenn ein Prozess P_R versucht von einer Seite p zu lesen, für die er keine Zugriffsberechtigung hat

- Seite p wird von $owner(p)$ nach P_R kopiert und mit Leseberechtigung versehen im Adressraum von P_R platziert
- handelt es sich bei $owner(p)$ um einen einzelnen Schreiber, wird ihm das Schreib- nicht jedoch das Eigentumsrecht entzogen[5]
- $copyset(p) := copyset(p) \cup \{P_R\}$
- die Fehlerbehandlung schließt mit Neustart des fehlerverursachenden Befehls

[5] Damit kann der Eigentümer weiterhin fehlerfrei von der Seite p lesen. Allerdings muss er künftige Anforderungen für diese Seite auch dann verarbeiten, wenn er nicht mehr auf die Seite zugreift. In dem Fall wäre es besser, wenn das Eigentumsrecht an P_R vergeben worden wäre. Dazu muss aber das zukünftige Zugriffsprofil präzise feststellbar sein.

Konsistenzwahrung (2)

Schreibfehler (*page fault*) tritt auf, wenn ein Prozess P_W versucht auf eine Seite p zu schreiben, für die er keine Zugriffs- oder nur Leseberechtigung besitzt

- Seite p wird an P_W übertragen (falls dieser noch keine aktuelle schreibgeschützte Kopie besitzt) und mit Lese-/Schreibberechtigung versehen im Adressraum von P_W platziert
- alle anderen Kopien werden entwertet, indem den in $copyset(p)$ enthaltenen Prozessen jeweils die Zugriffsberechtigung auf die Seite p entzogen wird
- $copyset(p) := \{P_W\}$ und $owner(p) := P_W$
- die Fehlerbehandlung schließt mit Neustart des fehlerverursachenden Befehls

Granularität der gemeinsamen Nutzung

- grundsätzlich nutzen alle Prozesse den gesamten Inhalt eines DSM gemeinsam
  - gleichwohl wird jedoch immer nur eine bestimmte Teilmenge wirklich gemeinsam genutzt und die auch nur für eine bestimmte Zeitdauer
  - demnach ist nur die jeweilige, prozessbezogene **Arbeitsmenge** (*working set*) des DSM bei gemeinsamer Nutzung konsistent zu halten
- zentrale Fragestellung bleibt somit die *Größe* der zu übertragenden Einheiten:

  Bytefolgen feingranular; kleiner Übertragungs-, hoher Verwaltungsaufwand
  Seiten grobgranular; hoher Übertragungs-, kleiner Verwaltungsaufwand
- Phänomen gekachtelten DSM ist die mögliche „falsche gemeinsame Nutzung“

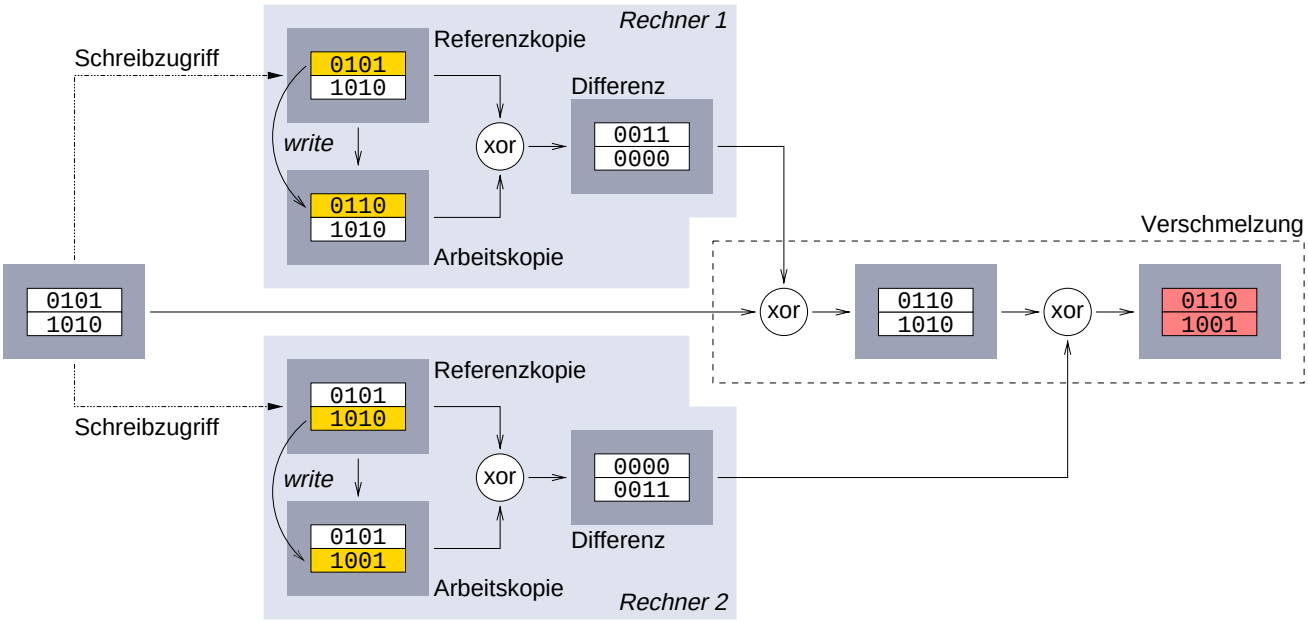
„Falsche gemeinsame Nutzung“

false sharing ein Prozess greift schreibend nur auf Variable A, ein anderer schreibend nur auf Variable B zu, beide Variablen liegen auf derselben Seite

- Prozesse tendieren dazu, mehr um große Seiten zu konkurrieren, da mit Zunahme der Seitengröße auch die Wahrscheinlichkeit wächst, dass die Daten, auf die sie zugreifen wollen, innerhalb derselben Seite liegen
- gekachelter DSM besitzt kein Wissen darüber, welche Speicherstellen einer Seite wie von den Prozessen gemeinsam genutzt werden, weshalb immer die gesamte Seite zwischen den Prozessen zu übertragen ist

Fließbandorientierte (verteilte) Verschmelzung auf Basis lokaler Differenzmengen toleriert falsche gemeinsame Nutzung und schwächt Seitenflattern ab

Verteilte Verschmelzung bei falscher gemeinsamer Nutzung



Zusammenfassung

- DSM als *Abstraktion* zur gemeinsamen Nutzung von Daten zwischen Prozessen
  - hauptsächlich gedacht als „Werkzeug“ für parallele Anwendungen
- Grundlage ist ein *Replikationssystem* mit verschiedenen Lösungsansätzen
  - fortlaufende Bytes, Objekte auf Sprachebene oder unveränderliche Daten
- Kernproblem besteht in der *Konsistenzwahrung* der physischen Repliken
  - sequentielle Konsistenz, Kohärenz, schwache Konsistenz
- *Aktualisierungsoptionen*: „Schreiben-Aktualisieren“ und „Schreiben-Entwerten“
  - Granularität, „falsche gemeinsame Nutzung“ (*false sharing*)

Referenzen

[1] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Experience with Distributed Programming in Orca. In *Proceedings of the International Conference on Computer Languages '90*, pages 79–89, New Orleans, USA, Mar. 1990. IEEE.

[2] N. Carriero and D. Gelernter. The S/Net's Linda kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, May 1986.

[3] J. Cordsen. *Virtuell gemeinsamer Speicher*. PhD thesis, Technische Universität Berlin, 1996.

[4] G. Coulouris, J. Dollimore, and T. Kimberg. *Verteilte Systeme: Konzepte und Design*. Pearson Education, 2002. ISBN 3-8273-7022-1.

[5] M. Dubois, C. Scheurich, and F. Briggs. Synchronization, Coherence and Event Ordering in Multiprocessors. *IEEE Computer*, 21(2):9–21, 1988.

[6] L. Lamport. How to make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs. *IEEE Transactions on Computers*, 28(9):241–248, 1979.

[7] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, 1986.

[8] D. Mosberger. Memory Consistency Models. Technical Report 93/11, University of Arizona, Nov. 1993.

[9] E. Organick. *The Multics System: An Examination of its Structure*. MIT Press, 1972.