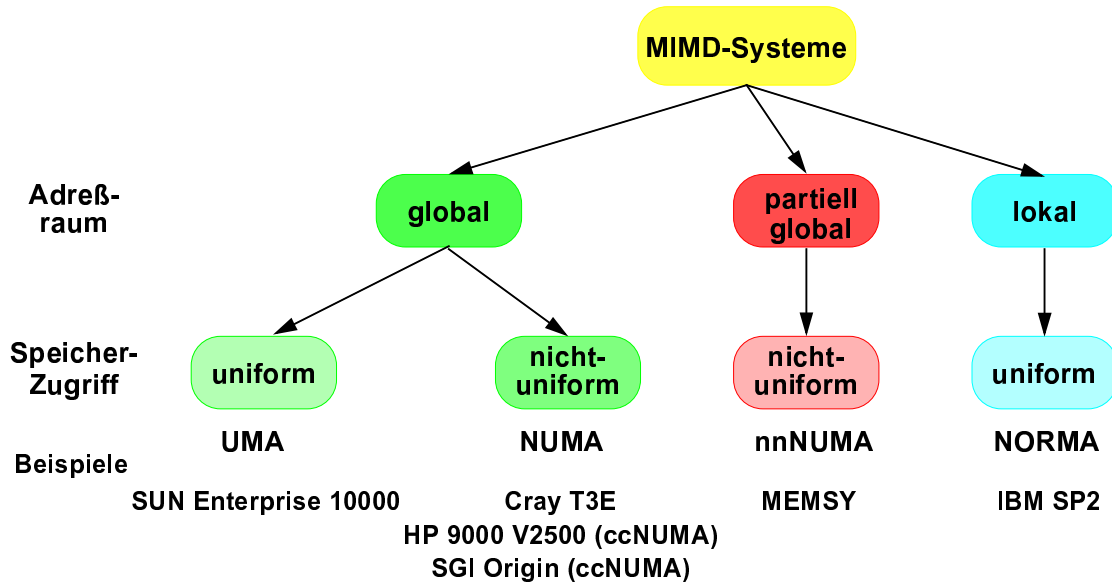


BP 2

2.2

2.2.1

2.2.1.1



2.2-1

BP 2

2.2.1.2

UMA: Sun Enterprise 10000; Prozessor UltraSparc II

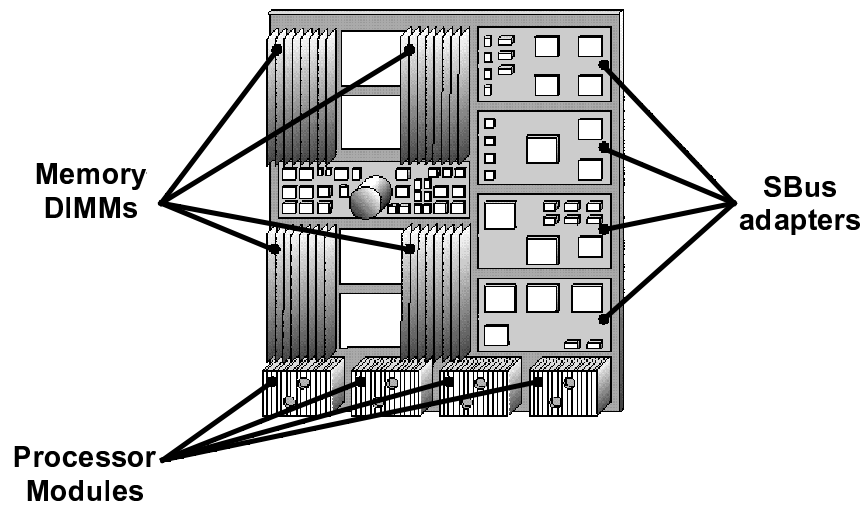
<http://www.sun.com/servers/highend/10000/>

Prozessor und Pufferspeicher

Ultra Sparc II			
	Ebene 1		Ebene 2
	Datenspeicher	Befehlsspeicher	
on chip	ja	ja	nein
Adressierung	virtuell	physikalisch	physikalisch
Markierung	physikalisch	physikalisch	physikalisch
Größe	16KB	16 KB	0,5 - 16 MB
Indexbreite	7 Bit	6 Bit	14 - 18 Bit
Zeilenzahl	256	256	8192 - 262144
Zeilenlänge	32 Byte	32 Byte	64 Byte
Assoziativität	1	2	1
write-on-hit	write-through	(nicht beschreibbar)	write-back
write-on-miss	nonallocating	(nicht beschreibbar)	allocating
Kohärenz	konsistent mit L2	konsistent mit L2	MOESI
	Keine Konsistenz zwischen Daten- und Befehlsspeicher		

2.2-2

Board

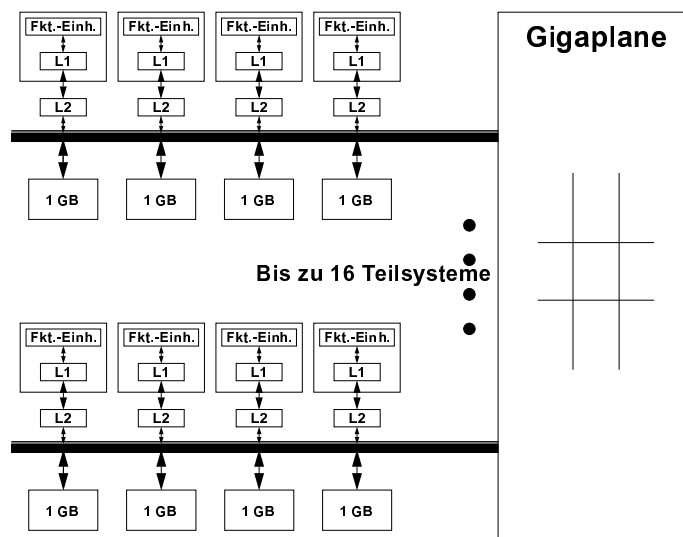


21.05.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-3

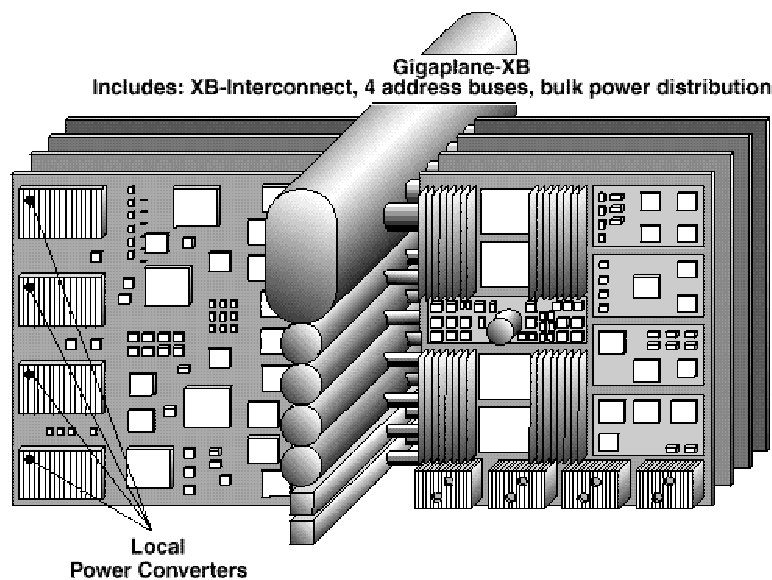
Gesamtstruktur
Vollständig kohärent



21.05.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-4



NUMA: Cray T3E; Prozessor Alpha 21264

<http://www.sgi.com/t3e/>

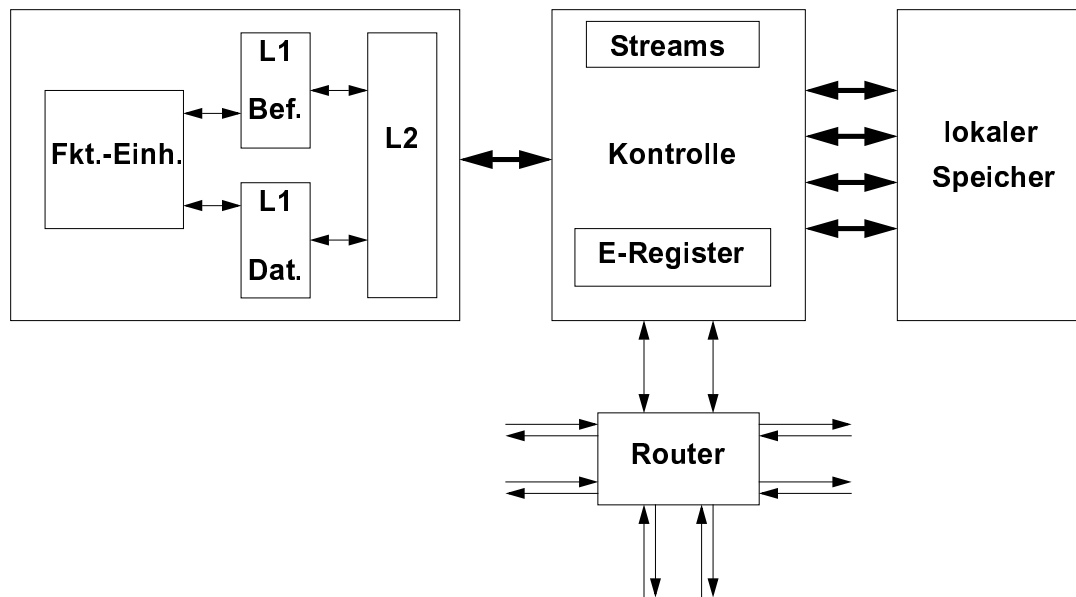
Prozessor und Pufferspeicher

Alpha 21264			
	Ebene 1		Ebene 2
	Datenspeicher	Befehlsspeicher	
on chip	ja	ja	ja
Adressierung	virtuell	virtuell	physikalisch
Markierung	physikalisch	physikalisch	physikalisch
Größe	8 KB	8 KB	96 KB
Indexbreite			
Zeilenzahl	256	256	512
Zeilenlänge	32 Byte	32 Byte	64 Byte
Assoziativität	1	1	3
write-on-hit	write-back	write-back	write-back
write-on-miss	allocating	allocating	allocating
Kohärenz	lokal	lokal	lokal

BP 2 Pufferspeicher bei Multiprozessoren: Cray T3E

Knoten

Alpha 21264



21.05.01

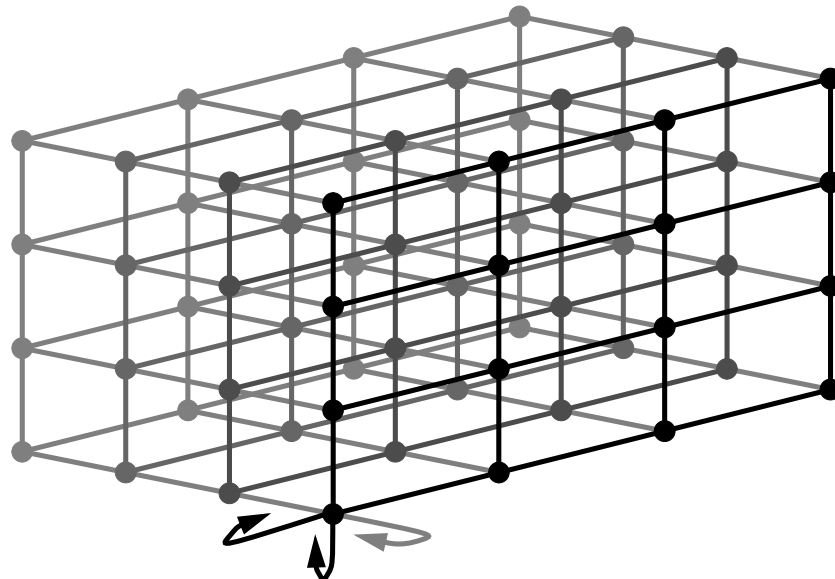
Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-7

BP 2 Pufferspeicher bei Multiprozessoren: Cray T3E

Gesamtstruktur

Lokale Kohärenz zwischen Prozessor-Cache und lokalem Speicher bei Lesen oder Schreiben mit Hilfe der E-Register



21.05.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-8

BP 2 Pufferspeicher bei Multiprozessoren: HP 9000 V

NUMA: HP 9000 V; Prozessor PA-8500

http://datacentersolutions.hp.com/vclass_servers_index_html

Prozessor und Pufferspeicher

PA 8500		
	Ebene 1	
	Datenspeicher	Befehlsspeicher
on chip	ja	ja
Adressierung	virtuell	virtuell
Markierung	---	---
Größe	1 MB	0,5 MB
Indexbreite	14	13
Zeilenzahl	16364	8192
Zeilenlänge	16 Byte	16 Byte
Assoziativität	4	4
write-on-hit	write-back/through	nicht beschreibbar
write-on-miss	nonallocating	nicht beschreibbar
Kohärenz	---	---
	Keine Konsistenz zwischen Daten- und Befehlsspeicher	

21.05.01

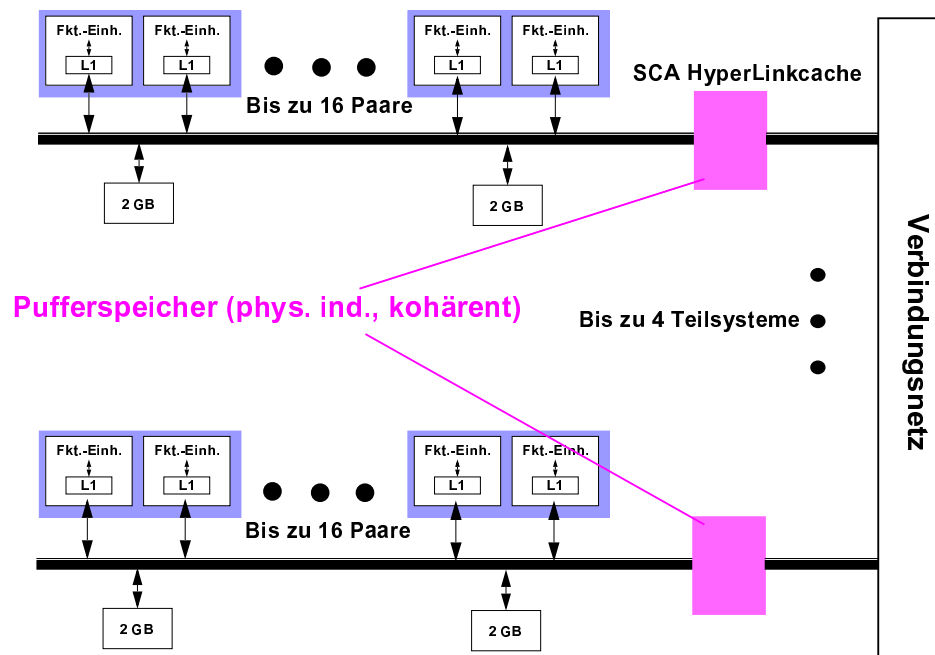
Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-9

BP 2 Pufferspeicher bei Multiprozessoren: HP 9000 V

Gesamtstruktur

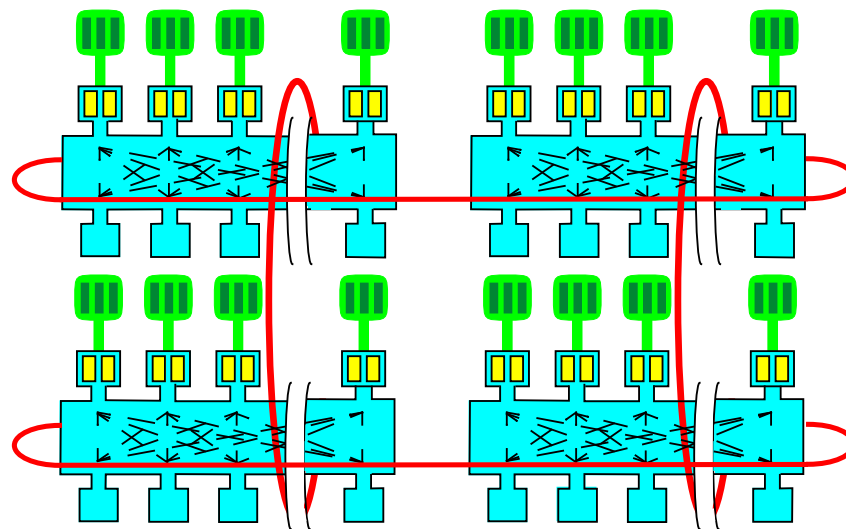
Vollständig kohärent; HyperLinkcache enthält Inhalte der lokalen L1-Caches



21.05.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-10



Globale E/A

Kreuzschienen-
verteiler

CPUs

Global gemeinsamer
SpeicherVerbindung des
global gemeinsamen
Speichers

ccNUMA: SGI Origin; Prozessor MIPS R10000

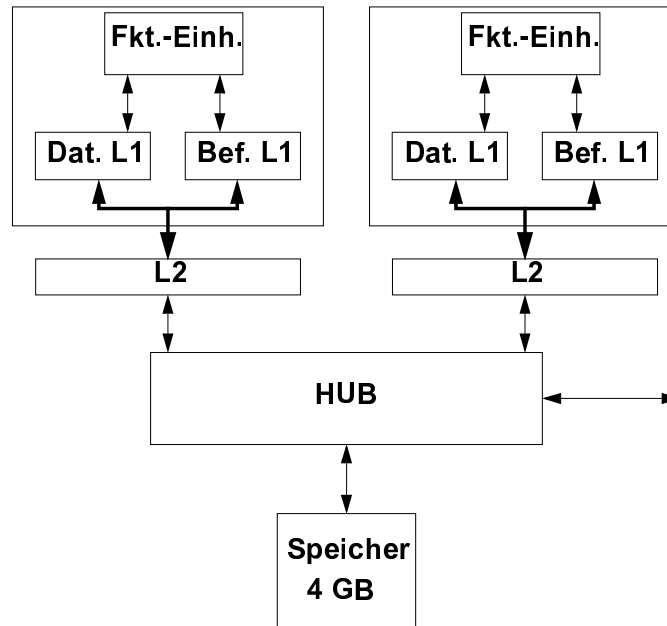
<http://www.sgi.com/origin/2000/index.html>

Prozessor und Pufferspeicher

MIPS R10000			
	Ebene 1		Ebene 2
	Datenspeicher	Befehlsspeicher	
on chip	ja	ja	nein
Adressierung	virtuell	virtuell	physikalisch
Markierung	physikalisch	physikalisch	physikalisch
Größe	32 KB	32 KB	0,5 - 16 MB
Indexbreite	9 Bit	8 Bit	12 - 17 Bit
Zeilenzahl	512	256	4096/2048 - 131072/65536
Zeilenlänge	32 Bytes	64 Bytes	64/128 Bytes
Assoziativität	2	2	2
write-on-hit	write-back	write-back	write-back
write-on-miss	allocating	allocating	allocating
Snooping	Systembus	Systembus	Systembus
Kohärenz	konsistent mit L2	konsistent mit L2	MESI-ähnlich

BP 2 Pufferspeicher bei Multiprozessoren: SGI Origin

Knoten



21.05.01

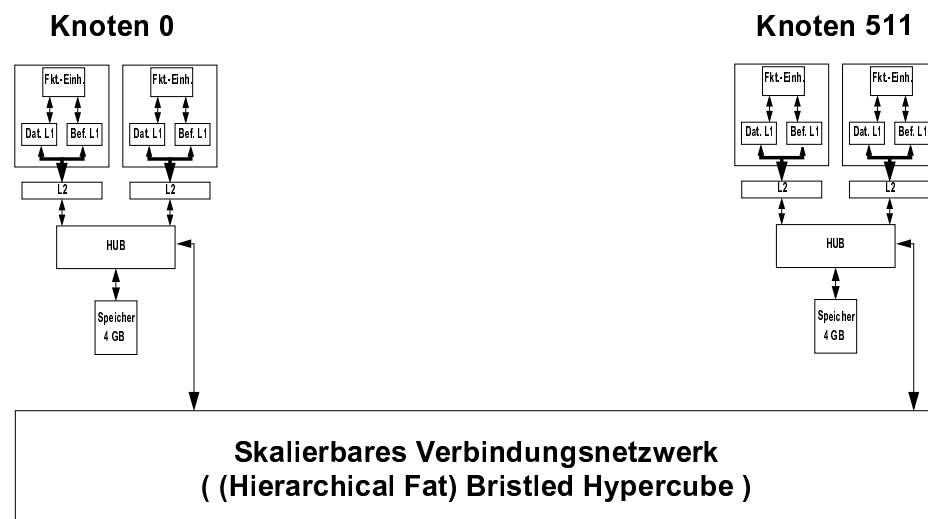
Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-13

BP 2 Pufferspeicher bei Multiprozessoren: SGI Origin

Gesamtstruktur

Vollständig kohärent

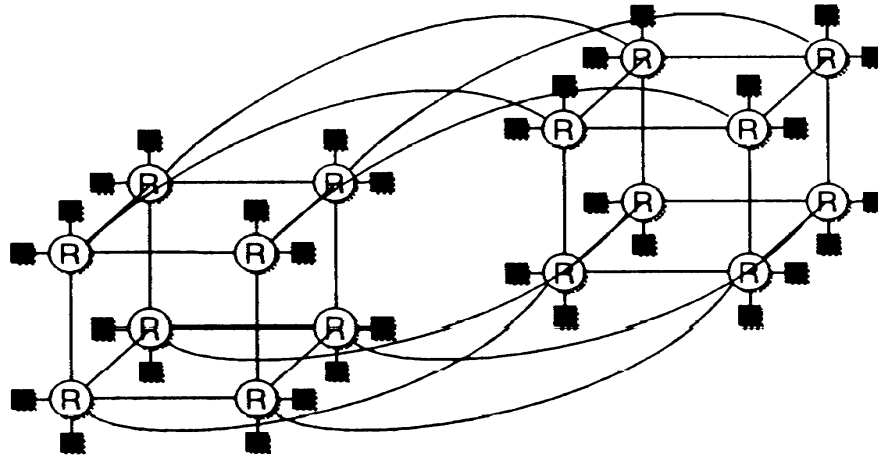


21.05.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-14

64 Processor System



21.05.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-15

nnNUMA: Memsy; Prozessor Motorola MC88100/MC88204

<http://www4.informatik.uni-erlangen.de/Projects/MEMSY/>

Prozessor und Pufferspeicher

MC88100/MC88204		
	Ebene 1	
	Datenspeicher	Befehlsspeicher
on chip	nein	nein
Adressierung	virtuell	virtuell
Markierung	physikalisch	physikalisch
Größe	64 KB	64 KB
Indexbreite	10 Bit	10 Bit
Zeilenzahl	1024	1024
Zeilenlänge	16 Byte	16 Byte
Assoziativität	4	4
write-on-hit	write-back/ through	write-back/ through
write-on-miss	allocating	allocating
Snooping	Systembus	Systembus

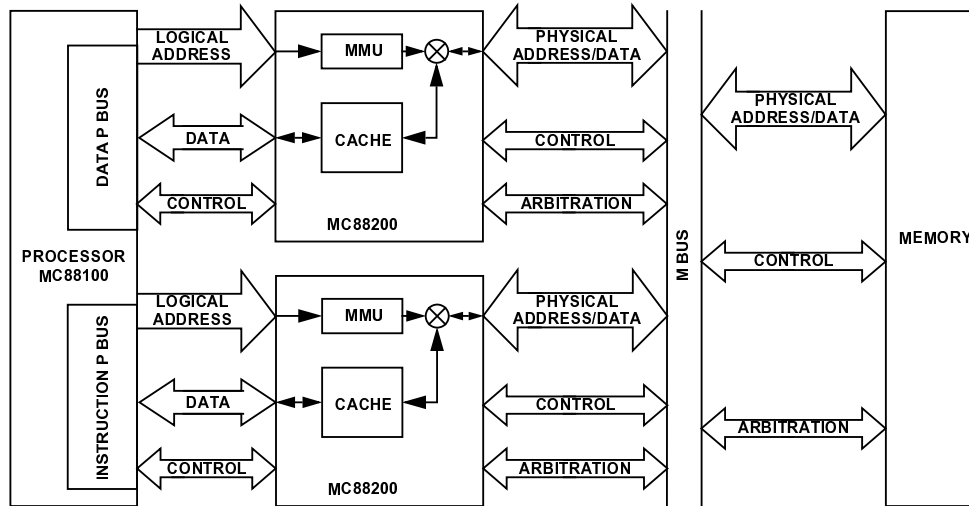
21.05.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-16

BP 2

Pufferspeicher bei Multiprozessoren: Memsy



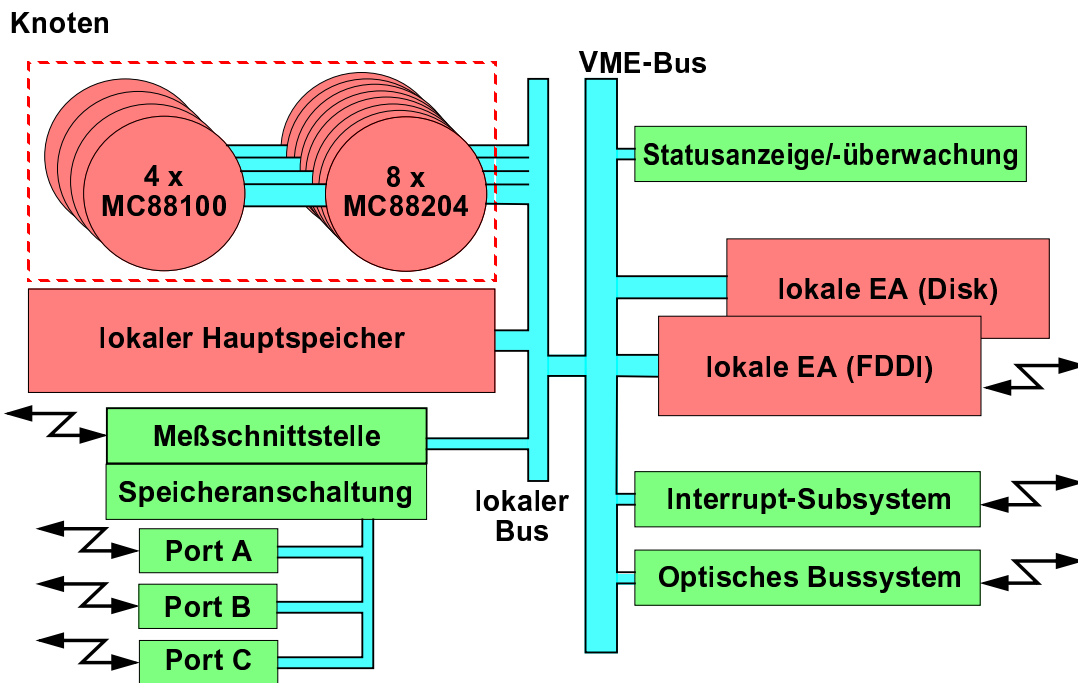
21.05.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-17

BP 2

Pufferspeicher bei Multiprozessoren: Memsy



21.05.01

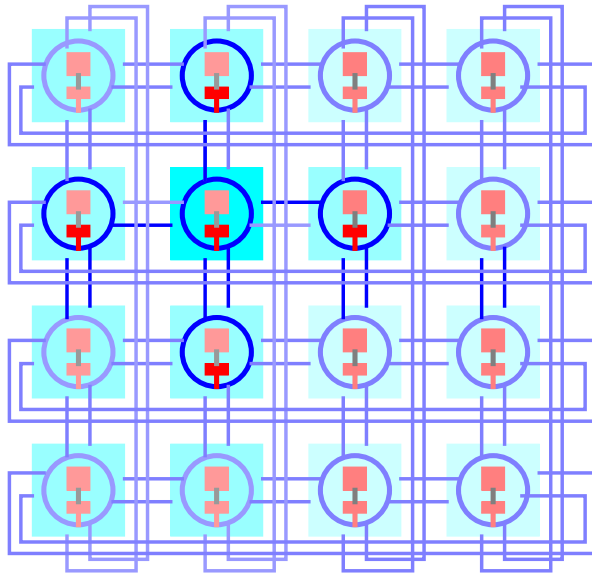
Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-18

BP 2 Pufferspeicher bei Multiprozessoren: Memsy

Gesamtstruktur

Cache-Kohärenz nur lokal!

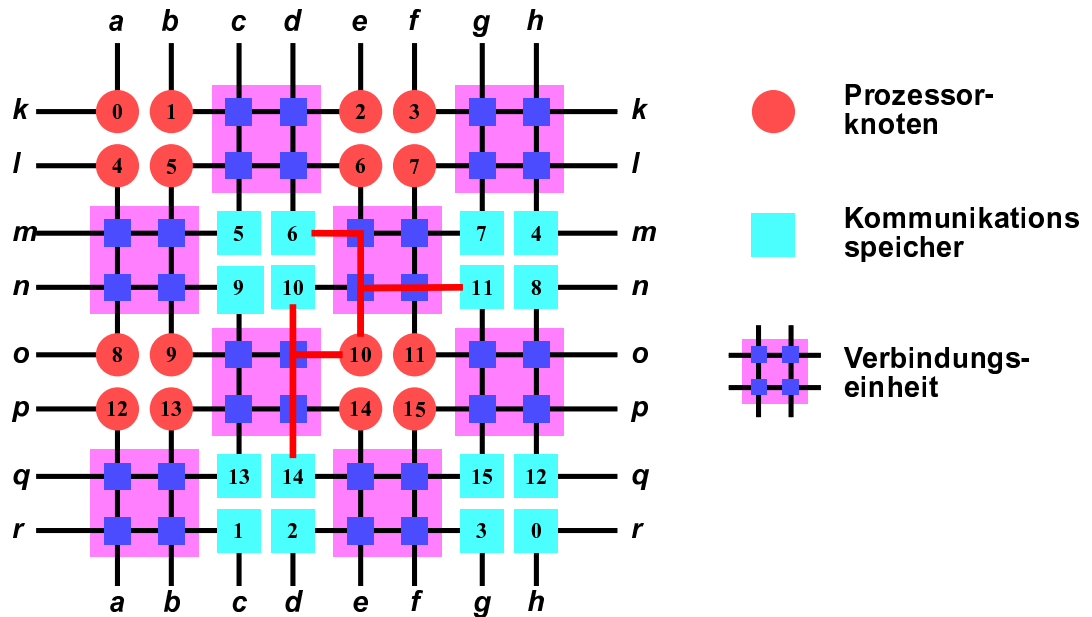


21.05.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-19

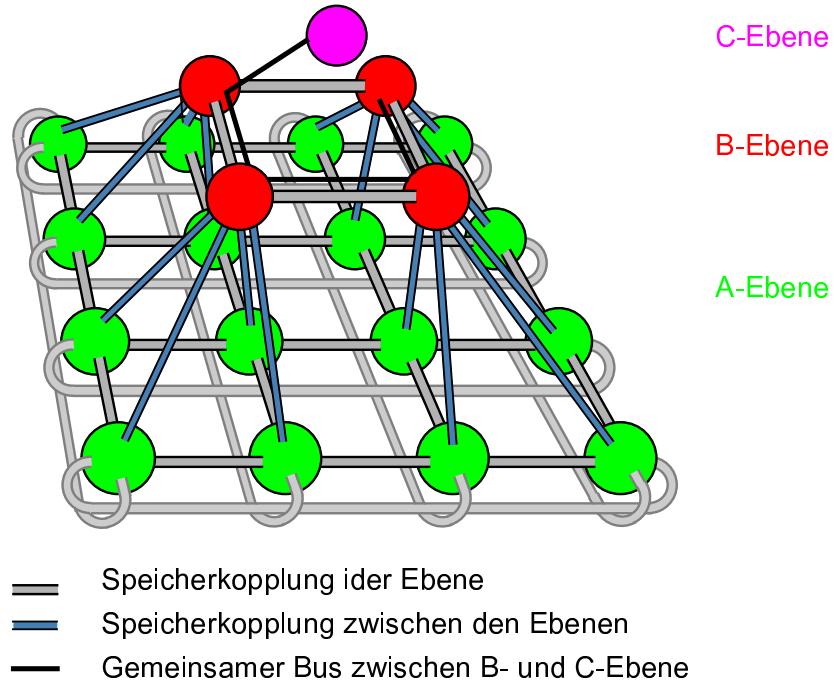
BP 2 Pufferspeicher bei Multiprozessoren: Memsy



21.05.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-20



2.2.2 Implementierung von Spinlocks und Barrieren

Literatur

Mellor-Crummey, J. M.; Scott, M. L.: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. ACM Transactions on Computer Systems, Vol. 9, No. 1, February 1991, pp.21-65.

Für UMA und NUMA Architekturen sind zwei Klassen von Koordinierungsmechanismen bedeutsam:

Blockierende Koordinierung

Koordinierung durch aktives Warten

Bedeutsam bei kurzen kritischen Abschnitten, zwei Ausprägungen

- **Spinlocks**
Zur Erzielung gegenseitigen Ausschlusses
- **Barrieren (barriers)**
Zur Koordinierung des Übergangs von parallelen in serielle Phasen

Probleme

- Bei großen Prozessorzahlen entstehen leicht 'hot spots'

BP 2 Pufferspeicher bei Multiprozessoren: Spinlocks

Untersuchungen von Agarwal und Cherian an Benchmarkprogrammen:

- In Systemen mit Cache-Kohärenz ist mehr als die Hälfte der Fehlzugriffe durch Spinlocks bedingt
- Werden Spinlock-Variable nur im Arbeitsspeicher geführt, erzeugen die Spinlocks nahezu die Hälfte des Datentransports zwischen CPU und Arbeitsspeicher

Folgerung: Entwicklung geeigneter Algorithmen

2.2.3 Spinlocks

- Teste-und-setze als atomare Aktion verfügbar
- Bei alleiniger Führung der Variablen im Arbeitsspeicher Belastung der Übertragungswege durch wartende Fäden
- Bei Caches mit Kohärenz häufige Invalidierungen der Variablen und damit Rückfall auf das vorige Problem
- Bei Caches ohne automatische Kohärenz zusätzlicher Aufwand um für die Spinlock-Variable Kohärenz zu sichern
- Verbesserung durch Erhöhen der Zeit bis zur nächsten Überprüfung, falls letzte Überprüfung negativ

21.05.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-23

BP 2 Pufferspeicher bei Multiprozessoren: Spinlocks

```
import java.awt.*;

class Lock {
    Scheduler scheduler;
    int value;

    public Lock(Scheduler scheduler) {
        value = 0; this.scheduler = scheduler;
    }

    private int testAndSet() {
        scheduler.beginOfAtomic();
        int old = value; value = 1;
        scheduler.endOfAtomic();
        return old;
    }
}
```

21.05.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-24

```

public void acquire() {
    int delay = 1;
    while (testAndSet() != 0) {
        scheduler.system.updateSyncState
            (scheduler.myPid(), new String("D: " + delay), Color.red);
        scheduler.sleep(delay);
        delay = 2 * delay;
    }
    scheduler.system.updateSyncState
        (scheduler.myPid(), Color.green);
}

public void release() {
    value = 0;
    scheduler.system.updateSyncState
        (scheduler.myPid(), Color.white);
}
}

```

```

import java.awt.*;

public class UserThread extends SimProcess {
    String name = null; Lock lock;
    public UserThread(String f_name, Scheduler f_scheduler,
        int f_nodeNumber) {
        super(f_name, f_scheduler, f_nodeNumber);
        name = f_name;
        lock = ((Spinlock)(scheduler.system)).lock;
    }
    public void runProcess() {
        for (int i = 0; i < 5; i++) {
            lock.acquire();
            scheduler.system.updateSyncState
                (scheduler.myPid()
                scheduler.sleep(2000);
            lock.release();
            scheduler.sleep(5000);
        }
    }
}
}

```

BP 2 Pufferspeicher bei Multiprozessoren: Ticket-Lock

Weitere Überlegungen

Verlängerung der Verzögerung muß begrenzt werden

`test-and-test_and_set`

Dadurch im wesentlichen während des aktiven Wartens nur Leseaufrufe

Ticket-Lock

Idee: Thread, der kritischen Abschnitt betreten will, bekommt ein Ticket.

Tickets werden fortlaufend nummeriert.

Es wird jeweils nächste Nummer aufgerufen (durch Setzen eines entsprechenden Zählers).

Vorteil: Während des aktiven Wartens keine Schreibaufrufe und damit keine durch Konsistenzerhaltung bedingten Invalidierungen. FIFO-Abarbeitung (kein Aushungern).

Voraussetzung: Atomare `fetch_and_increment`-Operation

Schwierigkeiten: Hot Spots durch Pollen der Anzeige für das aufgerufene Ticket

21.05.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-27

BP 2 Pufferspeicher bei Multiprozessoren: Ticket-Lock

```
class TicketLock implements Lock {
    Scheduler scheduler;
    int nextTicket, nowServing;

    public TicketLock(Scheduler scheduler) {
        nextTicket = 0;
        nowServing = 0;
        this.scheduler = scheduler;
    }

    private int fetchAndIncrement(){
        scheduler.beginOfAtomic();
        int old;
        old = nextTicket++;
        scheduler.endOfAtomic();
        return old;
    }
}
```

21.05.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-28

BP 2**Pufferspeicher bei Multiprozessoren: Ticket-Lock**

```
public void acquire() {
    int myTicket = fetchAndIncrement();
    while (myTicket != nowServing) {
        scheduler.system.updateSyncState
            (scheduler.myPid(), Color.red);
        scheduler.sleep((myTicket - nowServing) * 100);
    }
    scheduler.system.updateSyncState(scheduler.myPid(),
                                    Color.green);
}

public void release() {
    nowServing++;
    scheduler.system.updateSyncState(scheduler.myPid(),
                                    Color.white);
}
}
```

21.05.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-29

BP 2**Pufferspeicher bei Multiprozessoren: Array-Based Queuing Lock****Array-Based Queuing Lock**

Idee: Jedem Thread eine eigene Spinvariable zuordnen, z. B. aus einem Vektor von Spinvariablen.

Vorteil: Skaliert deutlich besser als die beiden vorangehenden. Bei **Cache-Kohärenz** auch besser als TicketLock, ohne Cache-Kohärenz ist bei NUMA-Architekturen TicketLock vorzuziehen. Erzwingt FIFO-Abarbeitung.

Voraussetzung: Atomare fetchAndIncrement-Operation zur Zuteilung der Spinvariablen.

Atomare fetchAndAdd-Operation zur Vermeidung von Überlaufproblemen bei dem Zähler, der für die Zuordnung der Spinvariablen benötigt wird.

Schwierigkeiten: Pro Lockvariable für ihre Verwaltung benötigter Speicherplatz proportional zur Zahl der Threads.

21.05.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-30

```
class ArrayBasedQueuingLock implements Lock {
    static int MUSTWAIT = 1;
    static int HASLOCK = 2;
    Scheduler scheduler;

    int slots[] = new int[Spinlock.NUMBEROFPROCESSES];
    // each element of slots should lie in a different memory module
    // or cache line

    int myPlace[] = new int[Spinlock.NUMBEROFPROCESSES];
    // each element of myPlace is private to a different process

    int nextSlot;

    public ArrayBasedQueuingLock(Scheduler scheduler) {
        nextSlot = 0; slots[0] = HASLOCK;
        for (int i = 1; i < Spinlock.NUMBEROFPROCESSES; i++) {
            slots[i] = MUSTWAIT;
        }
        this.scheduler = scheduler;
    }
}
```

```
private int fetchAndIncrement(){
    scheduler.beginOfAtomic();
    int old = nextSlot++;
    scheduler.endOfAtomic();
    return old;
}

private void atomicAdd(int x) {
    scheduler.beginOfAtomic();
    nextSlot += x;
    scheduler.endOfAtomic();
    return;
}
```

```
public void acquire() {
    myPlace[scheduler.myPid()] = fetchAndIncrement();
    if (myPlace[scheduler.myPid()] == Spinlock.NUMBEROFPROCESSES){
        atomicAdd(-Spinlock.NUMBEROFPROCESSES);
    }
    myPlace[scheduler.myPid()]
        = myPlace[scheduler.myPid()] % Spinlock.NUMBEROFPROCESSES;
    while (slots[myPlace[scheduler.myPid()]] == MUSTWAIT) {
        scheduler.system.updateSyncState
            (scheduler.myPid(), "Slot: " + myPlace[scheduler.myPid()],
            Color.red);
        scheduler.schedule();
    }
    scheduler.system.updateSyncState
        (scheduler.myPid(), Color.green);
    slots[myPlace[scheduler.myPid()]] = MUSTWAIT;
}
```

```
public void release() {
    slots[(myPlace[scheduler.myPid()] + 1)
        % Spinlock.NUMBEROFPROCESSES]
        = HASLOCK;
    scheduler.system.updateSyncState
        (scheduler.myPid(), Color.white);
}
}
```

List-Based Queuing Lock

Idee: Anforderungen werden in verketteter Liste geführt. Für blockierte Prozesse wird jeweils ihre zugeordnete Zustandsinformation in einem Listenelement geführt, das in seinem lokalen Speicher liegt.

Bei Freigabe wird jeweils der eigene Zustand und der des nächsten Wartenden verändert.

Vorteil:

- Garantiert FIFO-Abarbeitung
- Wartet aktiv an lokalen Variablen
- Beansprucht wenig Speicherplatz (1 Wort pro Knoten unabhängig von der Zahl der Lockvariablen)
- Arbeitet gleich gut für Systeme mit und ohne Cachekohärenz

Voraussetzung: Atomare `fetch_and_store`- sowie `compare_and_swap`-Operation

Vorteile: Bestes bekanntes Verfahren

Anmerkung: Durch etwas kompliziertere Realisierung der nachfolgenden `release`-Operation kann auf atomare `compare_and_swap`-Operation verzichtet werden.

```
import java.awt.*;

class QueueNode {
    int next; boolean locked;
    public QueueNode() {
        next = -1; locked = false;
    }
}

class ListBasedQueuingLock implements Lock {
    Scheduler scheduler;
    QueueNode[] queueNodes
        = new QueueNode[Spinlock.NUMBEROFPROCESSES];
    QueueNode tail;
    public ListBasedQueuingLock(Scheduler scheduler) {
        tail = new QueueNode();
        for (int i = 0; i < Spinlock.NUMBEROFPROCESSES; i++) {
            queueNodes[i] = new QueueNode();
            // for each i private to process i
        }
        this.scheduler = scheduler;
    }
}
```

```
private int fetchAndStore(QueueNode memory, int value){
    scheduler.beginOfAtomic();
    int old = memory.next;
    memory.next = value;
    scheduler.endOfAtomic();
    return old;
}

private boolean compareAndSwap(QueueNode memory,
                               int value1, int value2) {
    boolean result = false;
    scheduler.beginOfAtomic();
    if (result = (memory.next == value1)) {
        memory.next = value2;
    }
    scheduler.endOfAtomic();
    return result;
}
```

```
public void acquire() {
    queueNodes[scheduler.myPid()].next = -1;
    int predecessor = fetchAndStore(tail, scheduler.myPid());
    if (predecessor >= 0) {
        queueNodes[scheduler.myPid()].locked = true;
        queueNodes[predecessor].next = scheduler.myPid();
        while (queueNodes[scheduler.myPid()].locked) {
            scheduler.system.updateSyncState
                (scheduler.myPid(), getQueue(), Color.red);
            scheduler.schedule();
        }
    }
    scheduler.system.updateSyncState
        (scheduler.myPid(), getQueue(), Color.green);
}
```

```

public void release() {
    if (queueNodes[scheduler.myPid()].next < 0) {
        if (compareAndSwap(tail, scheduler.myPid(), -1)) {
            return;
        }
        while (queueNodes[scheduler.myPid()].next < 0)
            scheduler.schedule();
    }
    queueNodes[queueNodes[scheduler.myPid()].next]
        .locked = false;
    queueNodes[scheduler.myPid()].next = -1;
    scheduler.system.updateSyncState
        (scheduler.myPid(), Color.white);
}

```

```

String getQueue() {
    int i, root;

    String result2 = new String();
    root = -1;
    for (i = 0; i < Spinlock.NUMBEROFPROCESSES; i++) {
        if (!queueNodes[i].locked && queueNodes[i].next >= 0) {
            root = i;
            break;
        }
    }
    if (root < 0) root = tail.next;
    while (root >= 0) {
        result2 = result2 + root + " ";
        root = queueNodes[root].next;
    }
    return result2;
}
}

```

Messungen an einer 'Sequent Symmetry' nach Mellor-Crummey und Scott

UMA-Multiprozessor mit bis zu 30 Prozessoren

Prozessor: Intel 80386 (16 MHz)

64 kB Cache, 2-fach assoziativ

Kohärenz durch snooping

write-through, falls Inhalt der Pufferzeile auch in anderen
Pufferspeichern,

sonst write-back

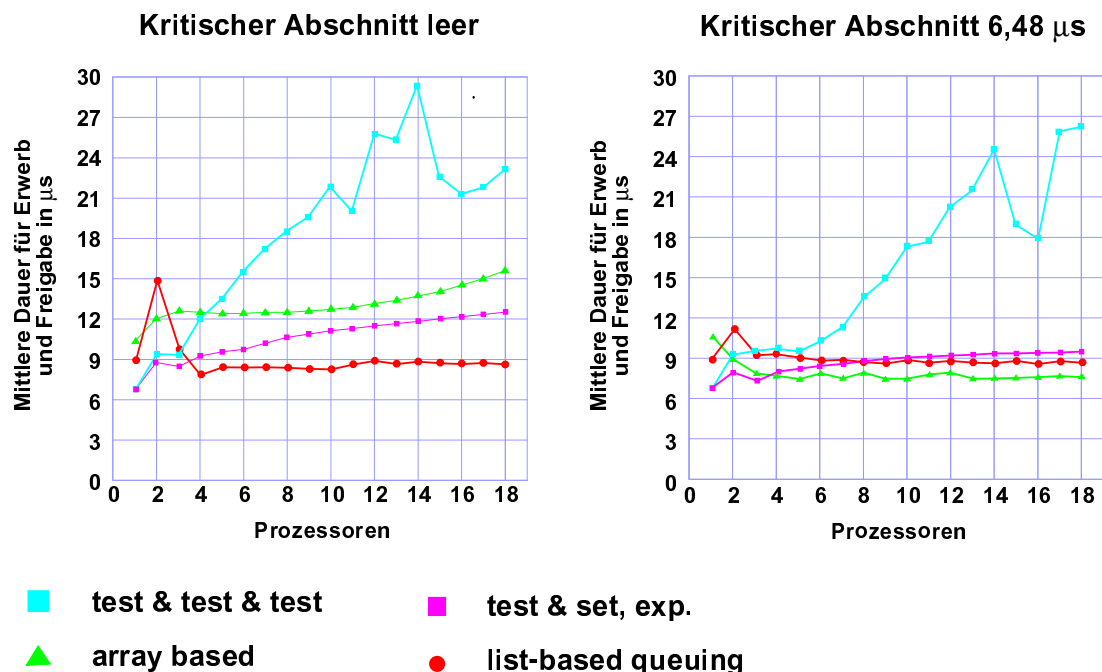
Atomarer Befehl: fetch_and_store mit 1, 2 oder 4 Byte

21.05.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-41

Ausführungszeiten verschiedener Spinlock-Implementierungen



21.05.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-42

2.2.4 Barrieren**Zentralisierte Barrieren**

An einer zentralen Variablen, die die noch ausstehenden Prozesse zählt, warten ankommende Threads aktiv bis der letzte ankommt.

Der letzte leitet durch Umsetzen einer Booleschen Variablen die nächste Runde ein.

Schwierigkeiten: Bei Parallelrechnern ohne Cachekohärenz entsteht starke Busbelastung durch die vorzeitig angekommenen Threads, da dann die Zählvariable nicht im Pufferspeicher eingelagert werden darf.

```
class Process extends Thread {  
    private Scheduler scheduler;  
    private Barrier barrier;  
    private int round = 0;  
    public Process(Scheduler scheduler, String name,  
                  Barrier barrier) {  
        this.scheduler = scheduler; setName(name);  
        this.barrier = barrier;  
    }  
    private void println(String string) {  
        scheduler.system.tmp.println(string);  
        System.out.println(string);  
    }  
}
```

```
public void run () {
    scheduler.schedule();
    println("Process " + getName() + " started");
    scheduler.sleep(1000);
    for (round = 0; round < 3; round++) {
        println(getName() + ": begin of round " + round);
        scheduler.sleep((scheduler.myPid() + 1) * 10000);
        barrier.join();
    }
    scheduler.eliminateThread(this);
}
}
```

```
class Barrier {
    Scheduler scheduler;
    boolean localSense[] = new boolean[Scheduler.NUMBEROFTHREADS];
    // component i of localSense is local to process/thread i
    boolean sense = true;
    int numberOfThreads;
    int count;

    public Barrier(Scheduler scheduler, int numberOfThreads) {
        count = this.numberOfThreads = numberOfThreads;
        for (int i = 0; i < Scheduler.NUMBEROFTHREADS; i++) {
            localSense[i] = true;
        }
        this.scheduler = scheduler;
    }
}
```

```

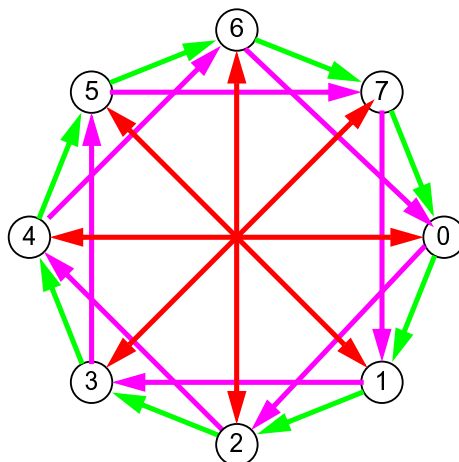
private int fetchAndDecrement(){
    scheduler.beginOfAtomic();
    int old = count--;
    scheduler.endOfAtomic();
    return old;
}

public void join() {
    localSense[scheduler.myPid()] = !localSense[scheduler.myPid()];
    System.out.println(scheduler.myPid() + ": count = " + count);
    if (fetchAndDecrement() == 1) {
        count = numberOfThreads;
        sense = localSense[scheduler.myPid()];
    } else {
        scheduler.system.updateSyncState(scheduler.myPid(), Color.red);
        while (sense != localSense[scheduler.myPid()])
            scheduler.schedule();
    }
    scheduler.system.updateSyncState(scheduler.myPid(), Color.green);
}
}

```

Dissemination Barrier

Idee: Baumartige Anordnung der Koordinierung. In der k -ten Runde (Zählung mit 0 beginnend) koordinieren sich Prozessor i mit Prozessor $(i + 2^k) \% P$ (P ist die Zahl der beteiligten Threads)



Vorteil: Reduktion der Zugriffe zu nicht-lokalen Speicherorten; jeder Prozessor wartet aktiv an eigenen, statisch festlegbaren Variablen.

```
class Flag {
    private boolean flag;
    public Flag(boolean f) {
        flag = f;
    }
    public boolean getValue() {
        return flag;
    }
    public void setValue(boolean f) {
        flag = f;
    }
}
```

```
class Flags {
    public Flag myFlags[][], partnerFlags[][];
    public Flags(int rounds) {
        myFlags = new Flag[2][rounds];
        partnerFlags = new Flag[2][rounds];
        for (int r = 0; r < 2; r++)
            for (int k = 0; k < rounds; k++)
                myFlags[r][k] = new Flag(false);
    }
}
```

```
class Barrier {
    Scheduler scheduler;
    int parity[], numberOfThreads, rounds = 0;
    boolean sense[];
    Flags localFlags[], allnodes[];
```

```

public Barrier(Scheduler scheduler, int numberOfThreads) {
    numberOfThreads = numberOfThreads;
    parity          = new int[numberOfThreads];
    sense           = new boolean[numberOfThreads];
    localFlags      = new Flags[numberOfThreads];
    allnodes        = new Flags[numberOfThreads];
    int tmp = numberOfThreads * 2 - 1;
    while ((tmp = tmp / 2) > 0) rounds++;
    for (int i = 0; i < numberOfThreads; i++) {
        allnodes[i] = new Flags();    parity[i] = 0;
        sense[i] = true;              localFlags[i] = allnodes[i];
    }
    for (int i = 0; i < numberOfThreads; i++)
        for (int j = 0; j < numberOfThreads; j++)
            for (int k = 0; k < rounds; k++)
                if (j == (i+(int)(Math.pow(2,k)+0.1)) % numberOfThreads)
                    for (int r = 0; r < 2; r++)
                        allnodes[i].partnerFlags[r][k]
                            = allnodes[j].myFlags[r][k];
    this.scheduler = scheduler;
}

```

```

public void join() {
    int myPid = scheduler.myPid();
    for (int round = 0; round < rounds; round++) {
        localFlags[myPid]
            .partnerFlags[parity[myPid]][round]
            .setValue(sense[myPid]);
        do {
            scheduler.schedule();
        } while (localFlags[myPid]
            .myFlags[parity[myPid]][round]
            .getValue()
            != sense[myPid]);
    }
    if (parity[myPid] == 1) {
        sense[myPid] = !sense[myPid];
    }
    parity[myPid] = 1 - parity[myPid];
}
}

```

Tree-Based Barrier

Idee

1. Warten auf Rundenende in Warte(acquire)-Baum
 2. Start der neuen Runde signalisieren über Aufweck(wakeup)-Baum
- d. h. die Informationen über das Erreichen bzw. Verlassen der Barriere werden in baumartigen Strukturen gesammelt bzw. verbreitet.

Vorteile

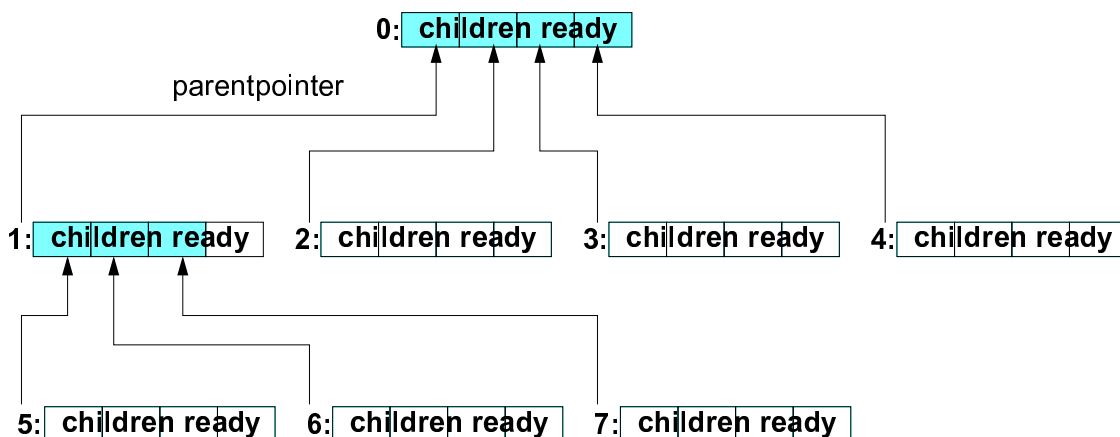
1. Aktives Warten nur an knotenlokalen Variablen
2. Benötigter Speicherplatz proportional zur Zahl der Knoten
3. Erzeugt an Maschinen ohne Broadcast das theoretische Minimum an Nachrichten
4. Hat im kritischen Pfad eine Nachrichtenzahl proportional zu $\log(\#Threads)$

21.05.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-53

Anhand eines Ankunftsbaums mit Fanin 4 wird die Ankunft aller Prozesse an der Barriere abgewartet.

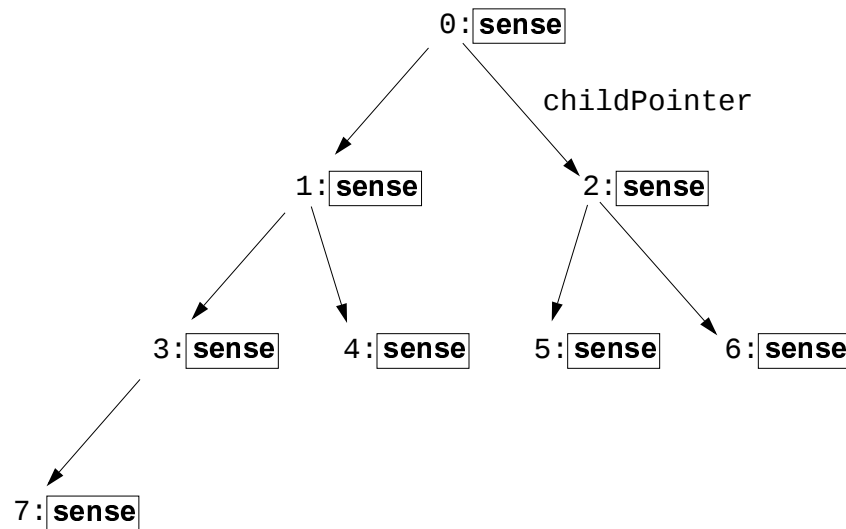


21.05.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-54

Mit Hilfe eines Aufweckbaumes mit Fanout 2 wird die Erlaubnis zur Weiterarbeit erteilt.



```

class Flag {
    public boolean flag;
    public Flag(boolean f) {
        flag = f;
    }
    public boolean getValue() {
        return flag;
    }
    public void setValue(boolean f) {
        flag = f;
    }
}

class TreeNode {
    public Flag parentSense;
    public Flag parentPointer;
    public Flag childPointers[] = new Flag[2];
    public Flag haveChild[] = new Flag[4];
    public Flag childNotReady[] = new Flag[4];
    public Flag dummy;
};
  
```

```

class Barrier {
    Scheduler scheduler;
    TreeNode nodes[];
    Flag sense[];          // sense[i] is local to node i
    public Barrier(Scheduler scheduler, int numberOfThreads) {
        int i, j;
        nodes = new TreeNode[numberOfThreads];
        sense = new Flag[numberOfThreads];
        for (i = 0; i < numberOfThreads; i++) {
            sense[i] = new Flag(true);
            nodes[i] = new TreeNode();
            nodes[i].parentSense = new Flag(false);
            nodes[i].dummy = new Flag(false);
        }
    }
}

```

```

// building acquire and wakeup tree
for (i = 0; i < numberOfThreads; i++) {
    // build part of acquire tree
    for (j = 0; j < 4; j++) {
        nodes[i].childNotReady[j]
            = (4*i+j+1 < numberOfThreads ? new Flag(true)
                                                : new Flag(false));
        nodes[i].haveChild[j]
            = (4*i+j+1 < numberOfThreads ? new Flag(true)
                                                : new Flag(false));
    }
    nodes[i].parentPointer
        = (i == 0 ? nodes[i].dummy
                : nodes[(i - 1) / 4].childNotReady[(i - 1) % 4]);
}

```

```
// build part of wakeup tree
if (i != 0)
    nodes[i].childPointers[0]
        = (2 * i + 1 >= numberOfThreads ? nodes[i].dummy
            : nodes[2 * i + 1].parentSense);
    nodes[i].childPointers[1]
        = (2 * i + 2 >= numberOfThreads ? nodes[i].dummy
            : nodes[2 * i + 2].parentSense);
}
this.scheduler = scheduler;
}

public void join() {
    int i, vpid = scheduler.myPid();
    while (nodes[vpid].childNotReady[0].flag
        || nodes[vpid].childNotReady[1].flag
        || nodes[vpid].childNotReady[2].flag
        || nodes[vpid].childNotReady[3].flag)
        scheduler.schedule();
}
```

```
// prepare for next barrier
for (i = 0; i < 4; i++) {
    nodes[vpid].childNotReady[i].flag
        = nodes[vpid].haveChild[i].flag;
}
// let parent know I'm ready
nodes[vpid].parentPointer.flag = false;
// if not root, wait until my parent signals wakeup
if (vpid != 0) {
    while (nodes[vpid].parentSense.flag != sense[vpid].flag)
        scheduler.schedule();
}
// signal children in wakeup tree
nodes[vpid].childPointers[0].flag = sense[vpid].flag;
nodes[vpid].childPointers[1].flag = sense[vpid].flag;
sense[vpid].flag = !sense[vpid].flag;
}
```

Messungen an einer 'Sequent Symmetry' und einer 'Butterfly' nach Mellor-Crummey und Scott

Butterfly (Hersteller: BBN)

NUMA-Multiprozessor mit bis zu 256 Prozessoren

Prozessor: Motorola MC68000 (8 MHz)

Atomarer Befehl: `fetch_and_clear_then_add`

`fetch_and_clear_then_xor` beide mit 2 Byte

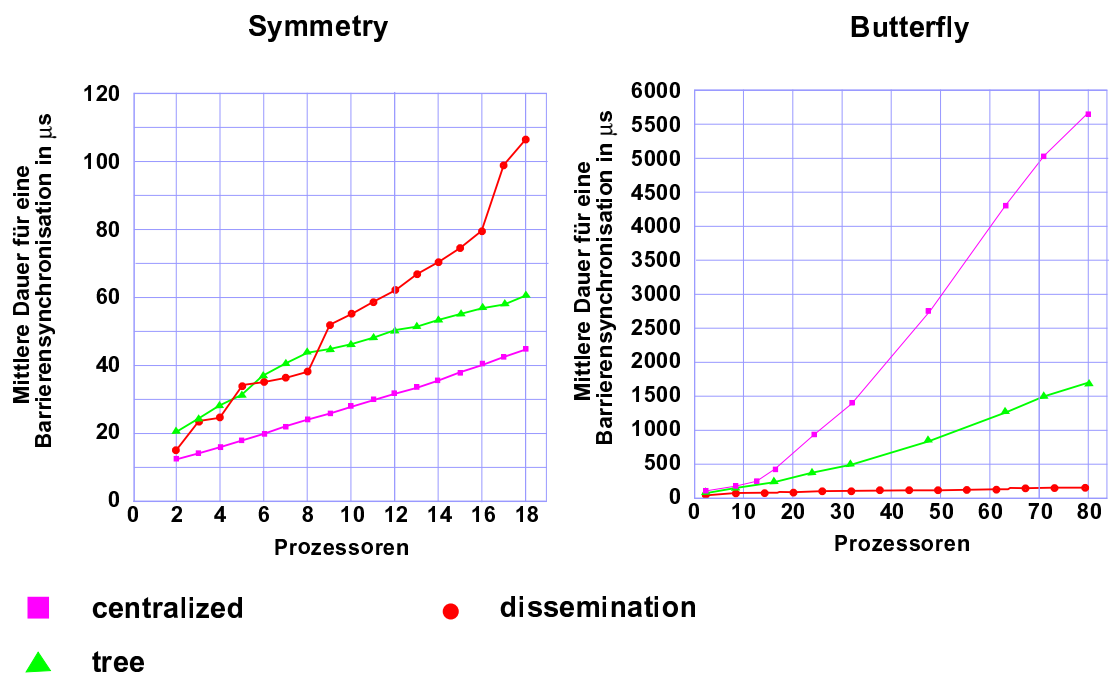
```
int fca(int* destination, int mask, int* source) {
    int tmp = destination;
    *destination = (*destination & mask) + *source;
    return tmp;
}
```

21.05.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-61

Ausführungszeiten verschiedener Barrieren-Implementierungen



21.05.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg
ist ohne Genehmigung des Autors unzulässig

2.2-62