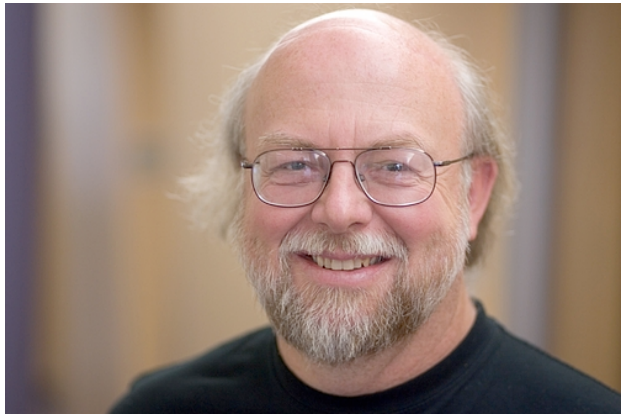
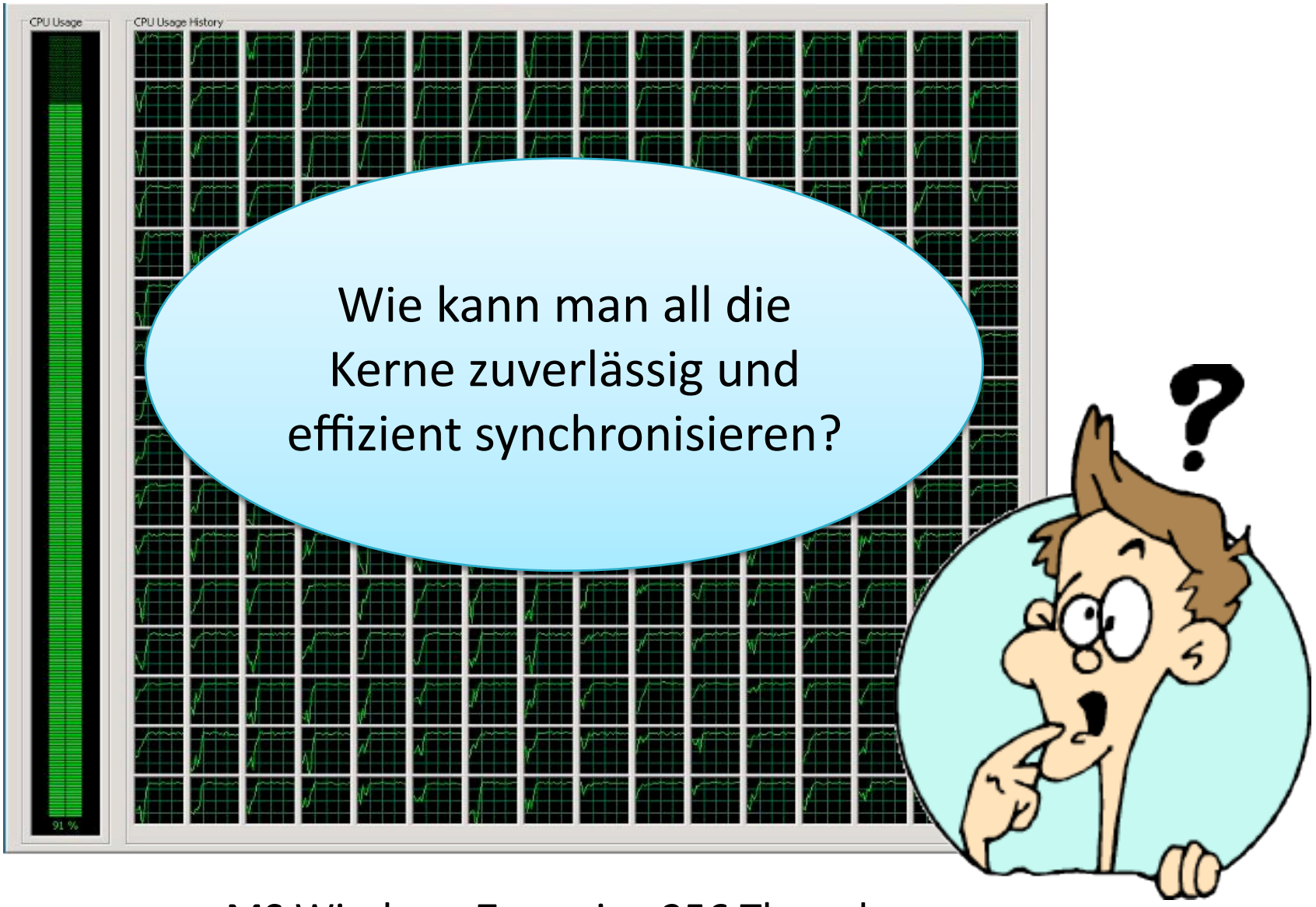
Martin Hoffmann

HARDWAREUNTERSTÜTZUNG FÜR NICHT-BLOCKIERENDE SYNCHRONISATION



„Es ist ziemlich eindeutig, dass Moore's Law nicht mehr die Taktrate, sondern die Zahl der Kerne misst. Es scheint so, als ob sich die Anzahl der Kerne alle paar Jahre verdoppelt. In zehn Jahren werden wir bei 128 CPUs liegen. In so einer Umgebung muss man anders über Programmierung an sich denken."

James Gosslin, Erfinder der Programmiersprache Java.
(Interview Java Magazin Februar 2008)



MS Windows 7 running 256 Threads
(64 Dual-Core Itanium + HT) WinHEC 11/2008

Überblick

- Nicht-blockierende Synchronisation
  - Atomare Synchronisationsbefehle
  - Hierarchie nach Herlihy
- Mehrprozessorsysteme
  - Architekturen
  - Cachekohärenz, -protokolle
- Zusammenfassung

Synchronisation

- Notwendig bei nebenläufigen Datenzugriffen
- Sicherstellung der Datenkonsistenz
- Blockierende oder nicht-blockierende Verfahren

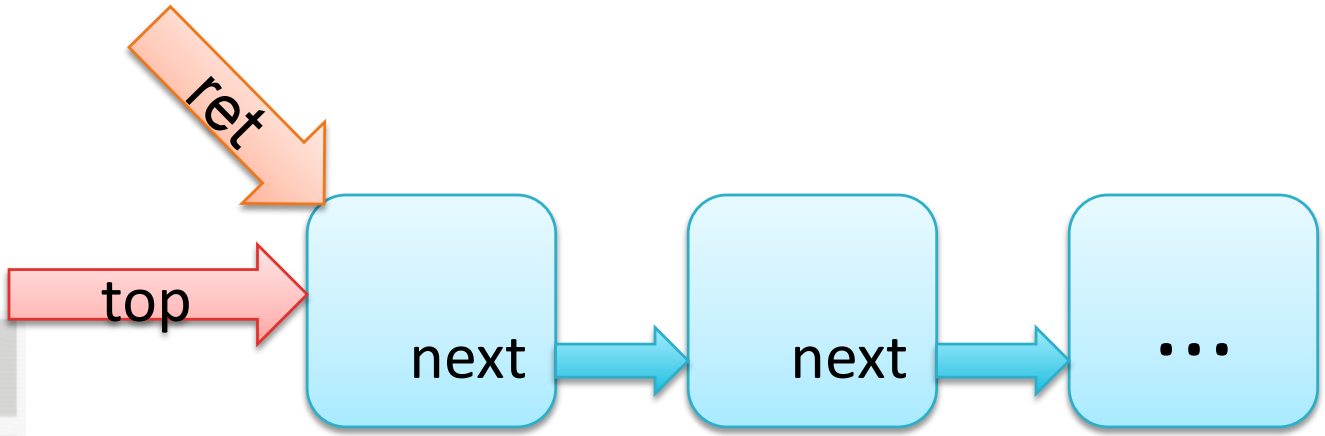
Blockierend

- Schlossvariablen sperren kritische Abschnitte
- Probleme:
  - Verklemmung (Philosophenproblem)
  - Effizienzverlust durch zu grob granulare Sperren
  - Prioritätsumkehrung
  - Zugriff auf Schlossvariable über Systemaufruf
    - Sehr langsam
  - Blockieren in ISR problematisch

Blockierender Stack

```
1   Obj* Pop(){
2      Obj* ret = top;
3      top = ret->next;
4      return ret;
5   }
```

```
1   Obj* Pop(){
2      LOCK();
3      Obj* ret = top;
4      top = ret->next;
5      UNLOCK()
6      return ret;
7   }
```



Nicht-blockierend

- Synchronisation ohne Schlossvariablen
- Anforderung: atomares Read-Modify-Write (RMW)
    1. Speicherstelle lesen
    2. Neuen Wert berechnen
    3. Falls Speicherstelle unverändert
        → neuen Wert schreiben, fertig.
        → sonst zurück zu 1.
- Punkt 3 prüft und setzt atomar

    Annahme: Wert ist seit 1. unverändert → Ablauf war atomar

    Falls nicht, noch mal versuchen

Atomare Synchronisationsbefehle

- Grundlage für nicht-blockierende Algorithmen
- RMW atomar ohne explizite Sperre

  → Hardwareunterstützung nötig

  - Compare-And-Swap
  - Load-Linked/Store-Conditional

Compare-And-Swap

- Atomare Vergleichs- und Schreibeoperation

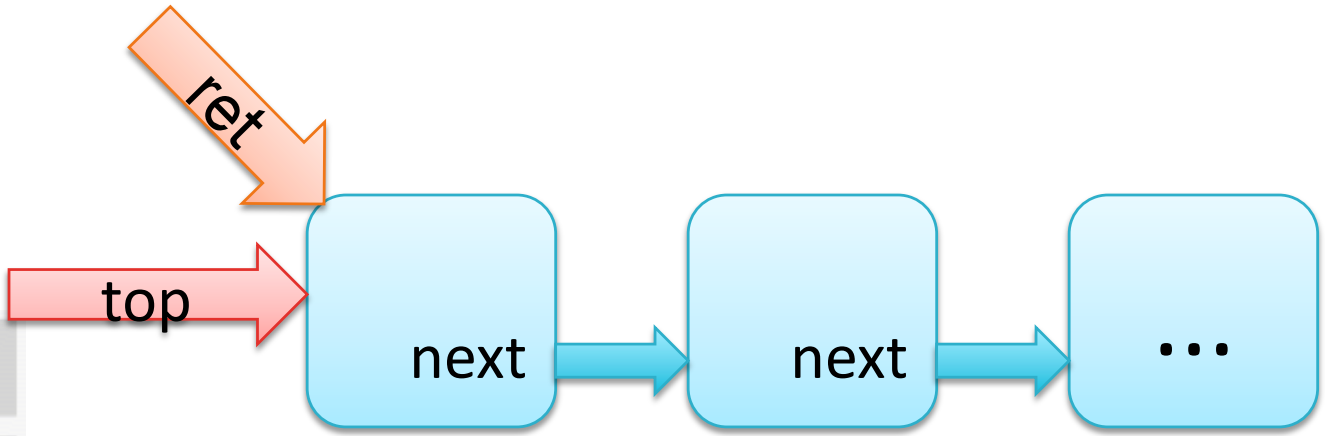
- Ablauf: (atomar in Hardware)

```
1 bool CAS(int *speicherstelle, int alt, int neu){
2   if (*speicherstelle != alt) return false;
3   *speicherstelle = neu;
4   return true;
5 }
```

Nicht-blockierender Stack

```
1   Obj* Pop(){
2      Obj* ret = top;
3      top = top->next;
4      return ret;
5   }
```

```
1   Obj* Pop(){
2   do{
3      Obj* ret = top;
4      Obj* n = ret->next;
5   }while(CAS(top,ret,n));
6   return ret;
7   }
```



Lock- und Wartefreiheit

- Herlihy 1991:

Eigenschaften einer nebenläufigen Datenstruktur

Lockfreiheit: „**Irgendeine Operation* wird in einer endlichen Anzahl an Schritten beendet.**“

Wartefreiheit: „**Jede Operation* wird in einer endlichen Anzahl an Schritten beendet.**“

*auf der nebenläufigen Datenstruktur

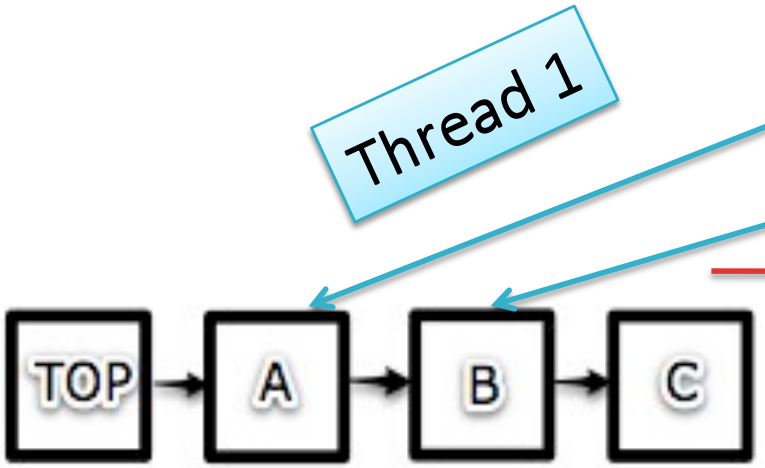
ABA-Problem

- Atomarer Abschnitt wird aufgespannt
  - Zwischen Lesen (Z. 3) und CAS (Z. 5)
  - Ist *top == ret* geht man von atomarem Ablauf aus:
    top wurde inzwischen nicht verändert

Stimmt das?

```
1   Obj* Pop(){
2   do{
3      Obj* ret = top;
4      Obj* n = ret->next;
5   }while(CAS(top,ret,n));
6   return ret;
7   }
```

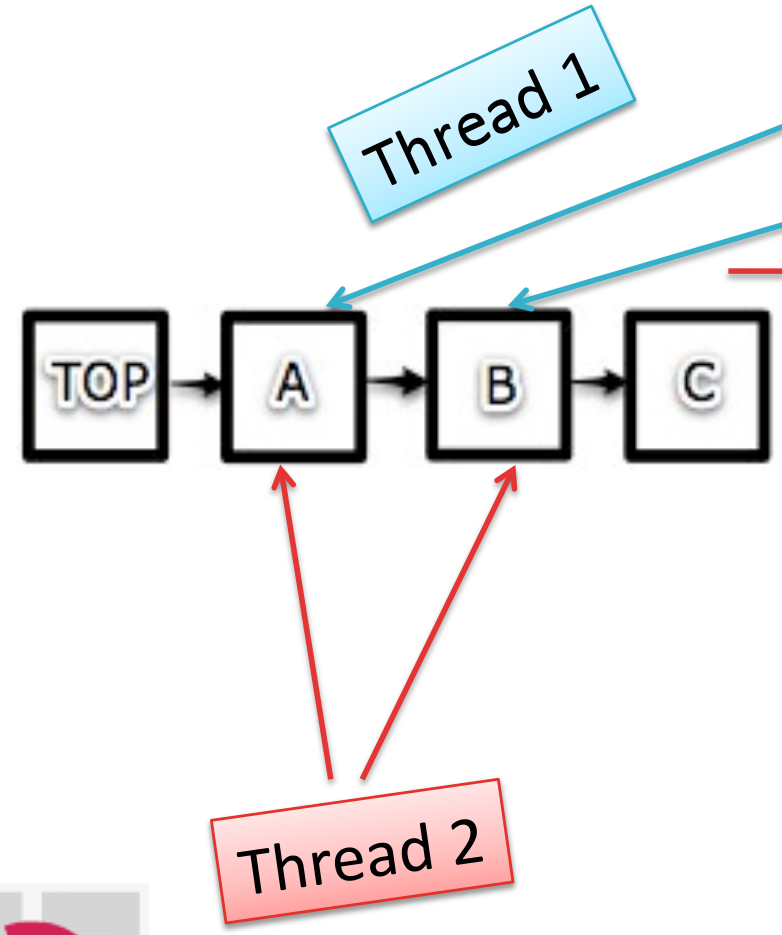
ABA-Problem 1




```
1  Obj* Pop(){
2  do{
3     Obj* ret = top;
4     Obj* n = ret->next;
5  }while(CAS(top,ret,n));
6  return ret;
7  }
```

Thread 1 wird vor der CAS Operation unterbrochen

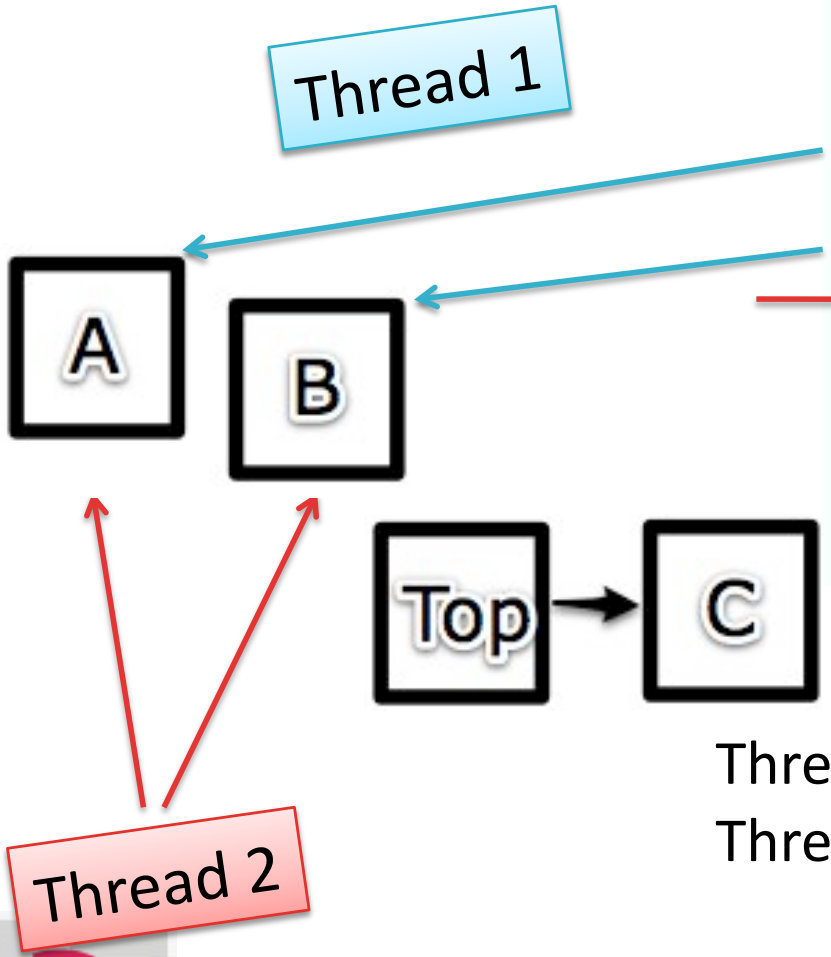
ABA-Problem 2




```
1  Obj* Pop(){
2  do{
3     Obj* ret = top;
4     Obj* n = ret->next;
5  }while(CAS(top,ret,n));
6  return ret;
7  }
```

Thread 2 wird eingelastet,
führt 2 erfolgreiche Pop()
Operationen aus

ABA-Problem 3

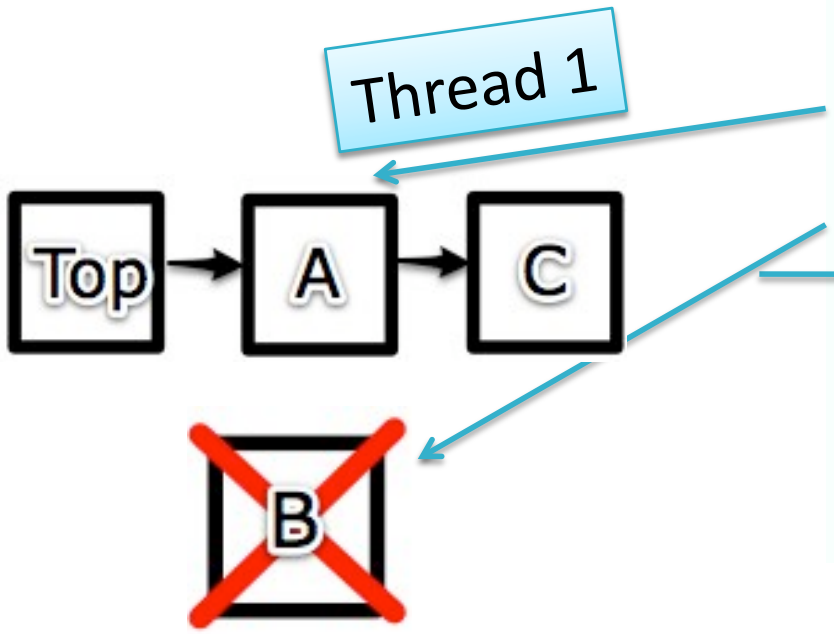



```
1  Obj* Pop(){
2  do{
3     Obj* ret = top;
4     Obj* n = ret->next;
5  }while(CAS(top,ret,n));
6  return ret;
7  }
```

Thread 1 hält weiterhin Zeiger auf A und B

Thread 2 löscht Objekt B, schiebt A zurück

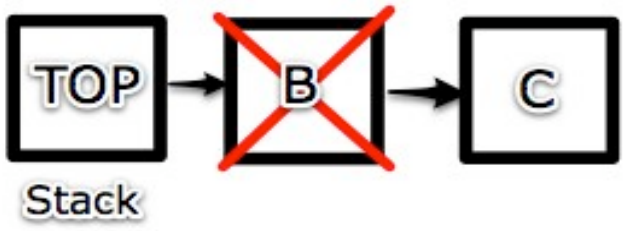
ABA-Problem 4




```
1  Obj* Pop(){
2  do{
3     Obj* ret = top;
4     Obj* n = ret->next;
5  }while(CAS(top,ret,n));
6  return ret;
7  }
```

Thread 1 hält weiterhin Zeiger auf A und B
Thread 1 läuft weiter → CAS erfolgreich

ABA-Problem 5




```
1  Obj* Pop(){
2  do{
3     Obj* ret = top;
4     Obj* n = ret->next;
5  }while(CAS(top,ret,n));
6  return ret;
7  }
```

CAS Operation ist erfolgreich, da A==A
Vorher gelöschtes Objekt B wird eingehängt
→ „top“ zeigt auf freigegebene Speicherstelle

ABA-Problemlösung

- Referenzzähler
  - Muss atomar mit Zeiger modifiziert werden
  - Benötigt Multi-Word-CAS
    - Nicht in jeder in Hardware implementiert
    - Mittels CAS nachbildbar
- Keine endgültige Lösung
  - Wahrscheinlichkeit wird nur verkleinert

Load-Linked/Store-Conditional

- Instruktionspaar zum Aufspannen eines atomaren RMW-Abschnitts:
  1. Load-Linked (LL) liest Datum, setzt Reservierung
  2. Wert kann modifiziert werden
  3. Store-Conditional (SC) schreibt Ergebnis zurück, falls Reservierung noch gesetzt

Reservierung wird aufgehoben falls die reservierte Speicherstelle vor dem SC Aufruf geändert wird.

LL/SC praktisch

- Behebt ABA-Problem
- Problem: ungewollte Invalidierungen
  - Exception (DEC Alpha)
  - False-Sharing im Cache

  → Schwaches LL/SC

Starkes LL/SC:

Invalidierung dann, und nur dann, wenn Speicherstelle tatsächlich geändert wurde

Hierarchie nach Herlihy (1991)

Eine Operation einer höheren Ebene kann eine nicht-blockierende Implementierung einer niedrigeren liefern

LL/SC und CAS sind universelle Primitiven

Starkes LL/SC kann CAS nachbilden

∞ Compare-And-Swap Load-Linked/Store-Conditional

2. test-and-set fetch-and-store fetch-and-Φ

1. Atomares Lesen/Schreiben

Zusammenfassung

- CAS und LL/SC theoretisch universelle Primitiven
- Praktisch:
  - ABA-Problem bei CAS
  - Ungewollte Invalidierung bei LL/SC
    (bisher keine echte starke LL/SC Implementierung)
- atomare Maschinenbefehle
  - müssen RMW-Semantik zuverlässig bereitstellen
  - auch in Mehrprozessorsystemen(!)

Überblick

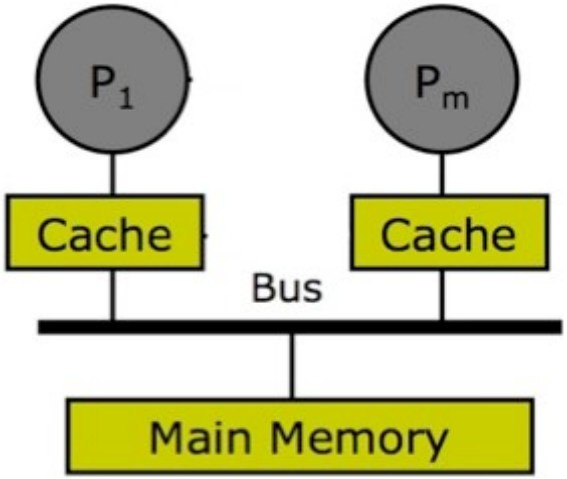
- Nicht-blockierende Synchronisation
  - Atomare Synchronisationsbefehle
  - Hierarchie nach Herlihy
- Mehrprozessorsysteme
  - Architekturen
  - Cachekohärenz, -protokolle
- Zusammenfassung

Mehrprozessorsysteme

- Integration mehrerer Prozessorkerne auf einem Chip
- Schrumpfung von SMP-Systemen
- Gemeinsame Nutzung von Ressourcen
  - Speicherhierarche (L2, L3, Hauptspeicher)
  - I/O Busse

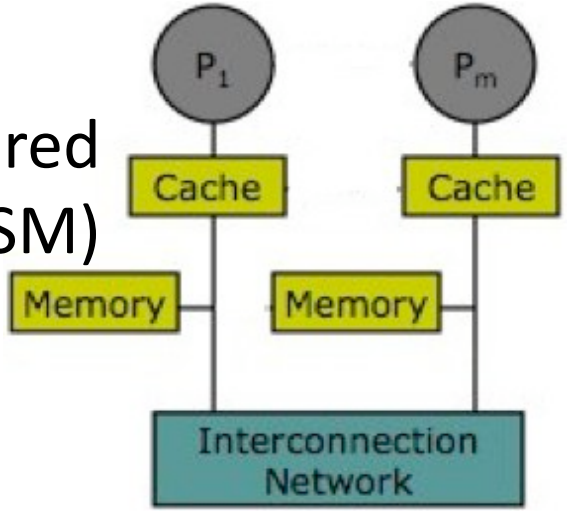
- Weiterentwicklung existierender Architekturen
  - IA32 → Core 2 Duo, Core 2 Quad, AMD Opteron
  - Sparc → Niagara, Niagara 2

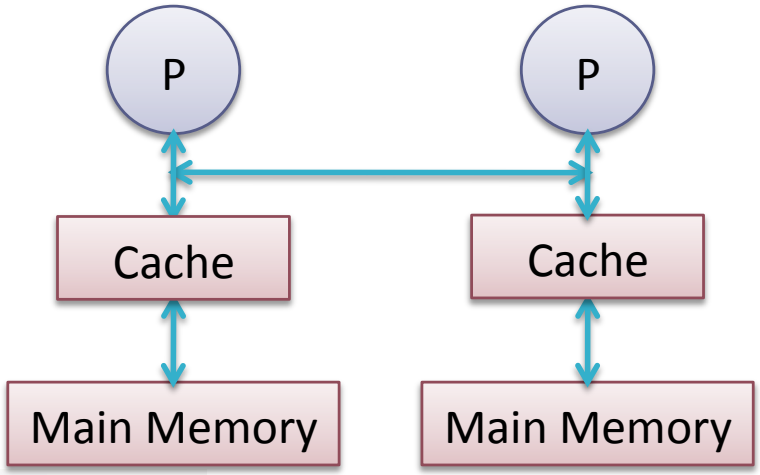
MP Architekturen



Symmetrischer Multiprozessor (SMP)
Unified Memory Access (UMA)

Distributed Shared Memory (DSM)





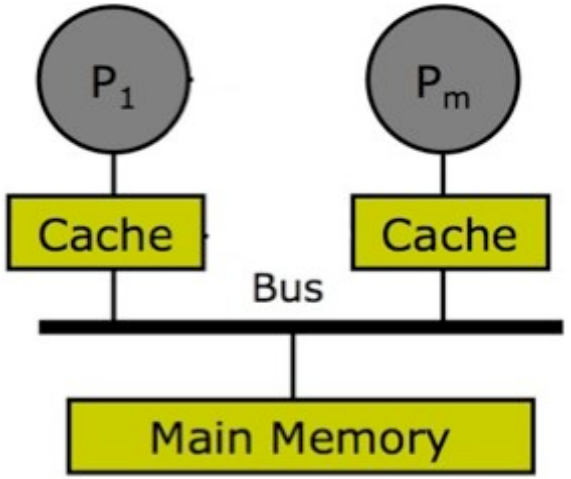
Symmetrischer Multiprozessor (SMP)
Non Unified Memory Access (NUMA)

Cachekohärenz

- Kohärenz
  - jeder Speicherzugriff liefert aktuell gültigen Wert
- Notwendig bei gemeinsam genutzten Daten in MP-Systemen (verteilte Caches)
- Sicherstellung mittels
  - Bus Snooping („Bus schnüffeln“)
  - Verzeichnisbasiert

Bus Snooping

- Kohärenzherstellung in MP-Systemen mit zentralem Kommunikationsmedium
- Ausschließlich Broadcastnachrichten
- „Bus Schnüffeln“:
  - Jeder Cachecontroller verfolgt alle Zugriffe
  - Ermöglicht Kohärenzherstellung



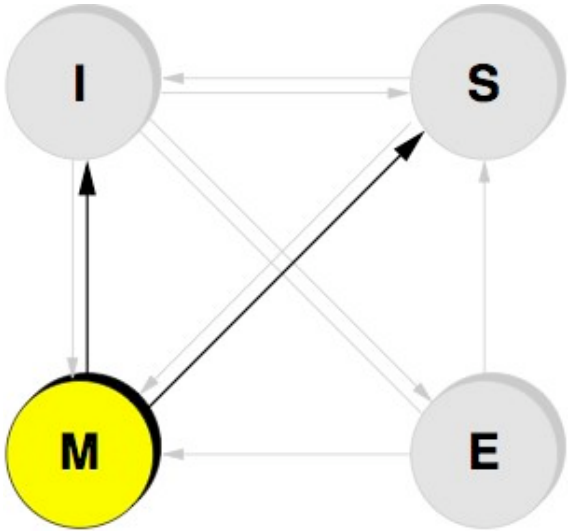
MESI Kohärenzprotokoll

- Zustandsautomat mit 4 Zuständen
- Cache-Zeile befindet sich in einem der Zustände
  - Modified (M), Exclusive (E), Shared (S), Invalid (I)
- Zustandsübergang ausgelöst durch lokale und entfernte Zugriffe (durch Snooping festgestellt)

- Ziel: Minimierung der Buszugriffe

MESI - Modified

- Exklusive, veränderte Kopie des Speicherblocks
- Schreib-/Lesezugriffe auf lokalem Cache
- Datum im Speicher veraltet
  - Zugriffe anderer Prozessoren abfangen
  - Zeile zurück schreiben
- Externer Lesezugriff → Shared
- Externer Schreibzugriff → Invalid



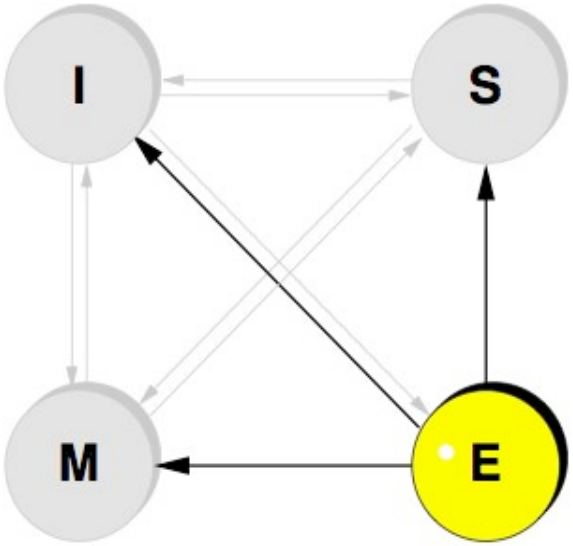
MESI - Exclusive

- Zeile enthält exklusive, unveränderte Kopie
- Lokale Schreib- oder Lesezugriffe direkt auf Cache, ohne Buszugriff
- Bei lokalem Schreibzugriff

  → Wechsel nach Modified
- Bei externem Schreibzugriff

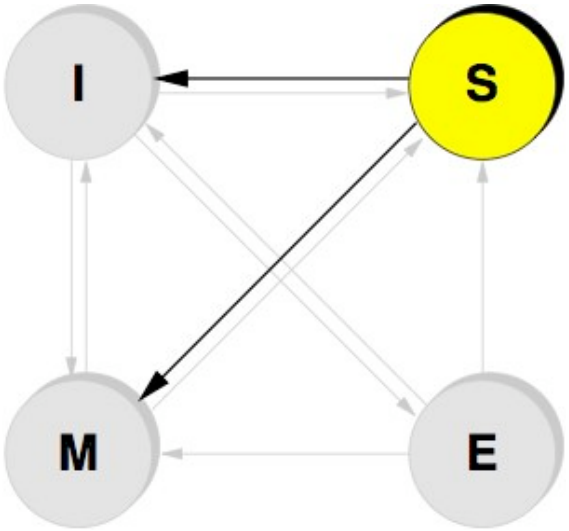
  → Wechsel nach Invalid
- Bei externem Lesezugriff

  → Wechsel nach Shared



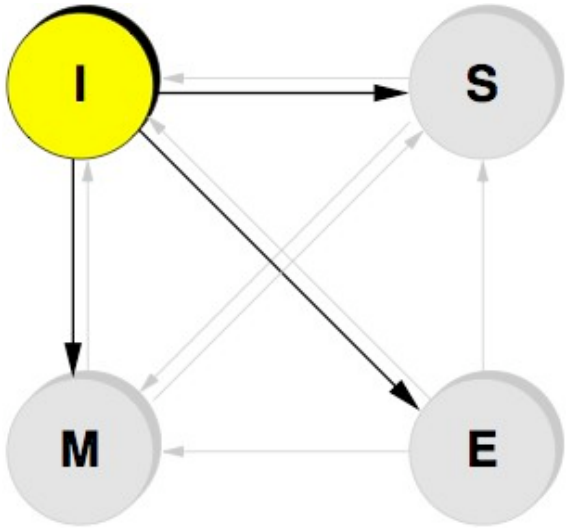
MESI - Shared

- Speicherblock existiert als unmodifizierte Kopie lokal und extern
- Bei Lesezugriff bleibt Zustand erhalten
- Bei Schreibzugriff
  - Wechsel nach Modified
  - Invalidierung der anderen Cache-Kopien
- Externer Schreibzugriff → Invalid



MESI - Invalid

- Zeile ist ungültig
- Schreib- oder Lesezugriffe veranlassen Einladen des Speicherblocks
- Anzeige durch andere Cache-Controller
  - bereits gespeichert → Shared
  - oder exklusiver Zugriff → Exclusive
- Mitteilung an andere Cache-Controller
  - bei Schreibzugriff →Modified



Bus Snooping

- Vorteil: Einfache Implementierung
  - Schnell und effizient bei geringer CPU Anzahl
  - Aktuell weit verbreitet (v.a. Multicore CPUs)

- Nachteil: schlechte Skalierbarkeit
  - Bei steigender CPU-Anzahl wird Bus zum Flaschenhals
  - Bus kann nicht beliebig vergrößert werden

Verzeichnisbasierte CC

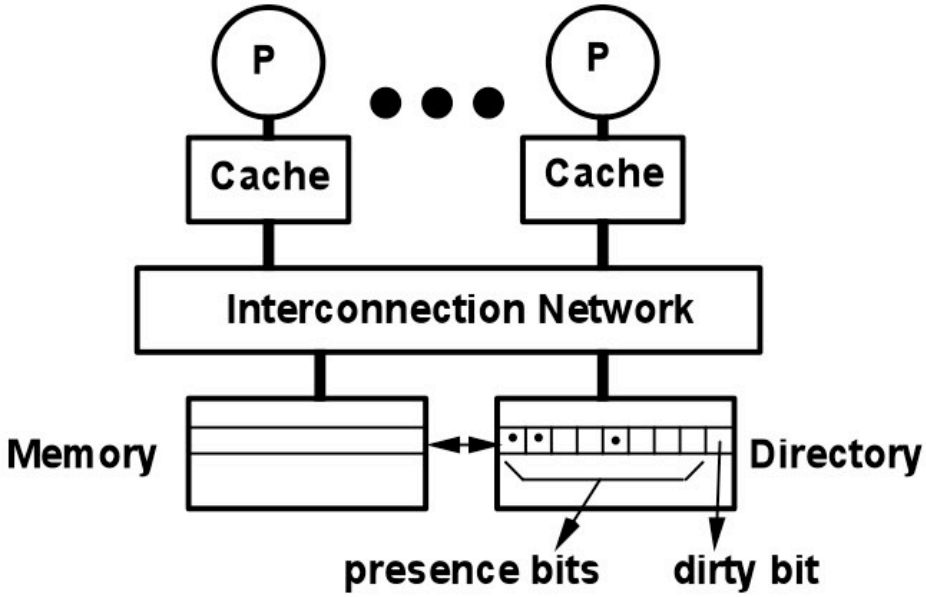
- Kohärenzherstellung bei MP-Systemen ohne zentrales Kommunikationsmedium
  - Snooping nicht möglich (kein Broadcastmedium)
- Lösung: Verzeichnis
  - Cache-Kohärenz über Verzeichnistabelle
  - Zentrale oder verteilte Verwaltung
  - Tabelle protokolliert für jeden Blockrahmen in welchen Caches Kopien liegen
  - Zustände Cache-intern ähnlich MESI-Protokoll

Verzeichnisbasierte CC

Prozessor i liest aus Hauptspeicher (HS):

Falls dirty bit = AUS → vom HS lesen; p[i] = AN

Falls dirty bit = AN → Wert vom Besitzer kopieren;
HS update; dirty bit = AUS;
p[i] = AN (cache status „Shared“)



Verzeichnisbasierte CC

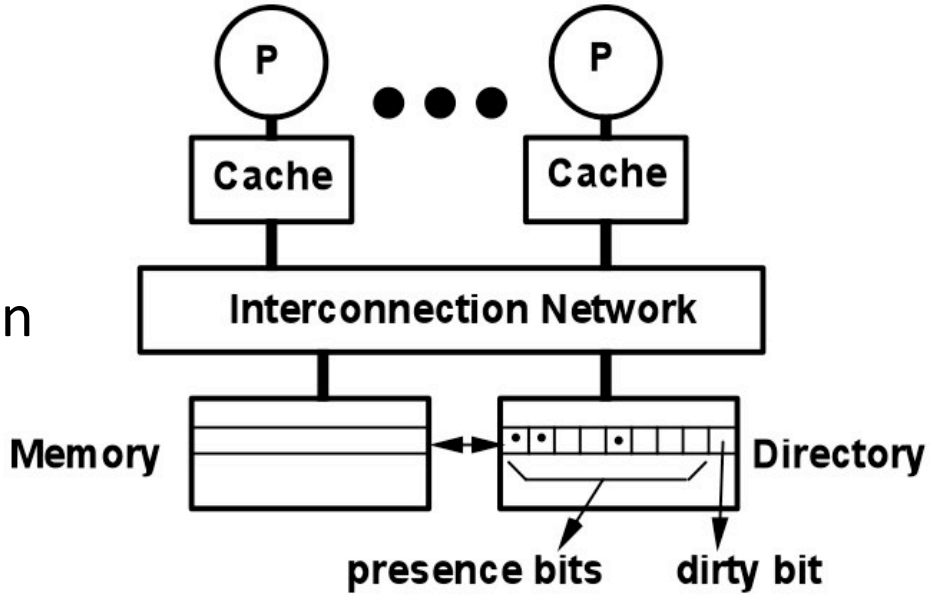
Schreibzugriff Prozessor i:

Falls dirty bit = AUS

→ Invalidierung aller Kopien;
dirty bit = AN; p[i] = AN
Cachezeile „Modified“

Falls dirty bit = AN:

→ Wert in HS zurückschreiben
dirty bit = AUS; weiter wie oben



Zusammenfassung

- Snooping weiterhin bei Desktop und Multicore
  - Einfache Implementierung
  - Nutzt Broadcastingeigenschaft des Speicherbusses
- Verzeichnis bei großen DSM Systemen
  - Skaliert sehr gut
  - Verzeichnis ist jedoch Flaschenhals

Quo vadis?

- Desktopmarkt:
  - Einzelne Multicore CPU mit gemeinsamen L3 Cache (Core 2 Duo/Quad)
- Servermarkt:
  - n Multicore CPUs in NUMA – System kombiniert
  - Intel Core i7, AMD Phenom
- High-Performance Computing:
  - n NUMA – Systeme zu DSM – System kombiniert

Ausblick

- Performancefaktor: Kohärenzherstellung
- Zukunft: Transaktionaler Speicher?

Vielen Dank für die Aufmerksamkeit!