Verteilte Betriebssysteme

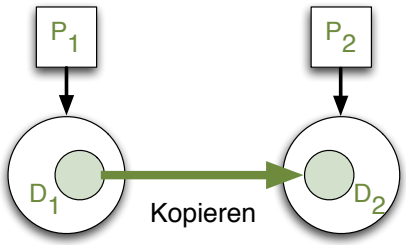
11. Kapitel
Speicher

Matthias Werner
Professur Betriebssysteme

11.1 Grundlagen
Wiederholung aus „Betriebssysteme“

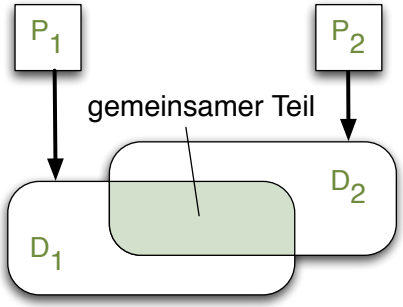
- Zur Interaktion nebenläufiger Prozesse gibt es zwei Programmiermodelle:

Nachrichtenversand
(=expliziter Datentransport)



gerichtete Beziehung

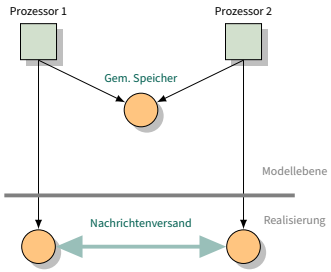
Gemeinsamer Speicher
(=impliziter Datentransport)



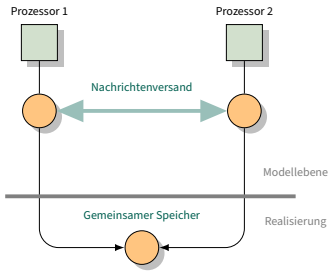
symmetrische Beziehung

Abbildbarkeit

- Die beiden Formen des Informationsaustauschs sind als **Konzepte** zu verstehen
- Sagen noch **nichts** über die Implementierung aus



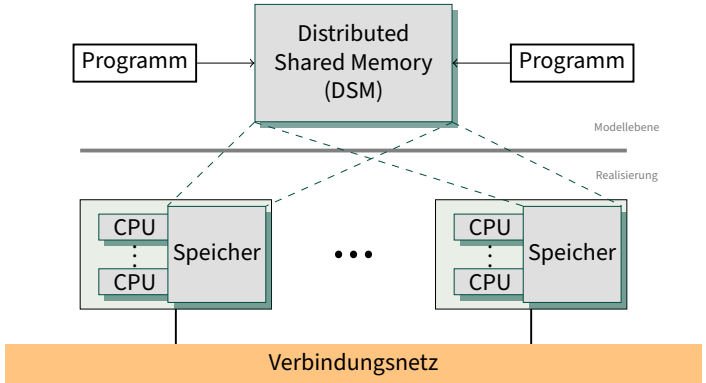
Gemeinsamer Speicher realisiert durch Nachrichtenversand



Nachrichtenversand realisiert durch gemeinsamen Speicher

Verteilter Speicher

- Im Folgenden Diskussion der Realisierung eines gemeinsamen Speichers im verteilten System



Verteilter Speicher (Forts.)

- Auch wenn gemeinsame Kooperation und Kommunikation (Speicher und Nachrichtenaustausch) gleichmächtig sind, wird von vielen Programmierern das Arbeiten mit Kooperation bevorzugt
  - Informationsaustausch muss nicht explizit gemacht werden
  - Arbeiten mit Speicher ist Standardfall (auch ohne Nebenläufigkeit)
- Allerdings ist auch das Programmieren von nebenläufigen Anwendungen mit gemeinsamen Speicher fehleranfällig
- **Beispiel**: Ist folgender Mutex-Mechanismus korrekt?

```
// Prozess A
while(1){
  while(var == 1);
  var = 1;
  /* Kritischer Abschnitt */
  var = 0;
}
```

```
// Prozess B
while(1){
  while(var == 1);
  var = 1;
  /* Kritischer Abschnitt */
  var = 0;
}
```


Verteilter Speicher (Forts.)

- Wir sind vom Speicher ein spezifischen Verhalten „gewöhnt“, das die Programmierer häufig intuitiv voraussetzen:
  Sepicherzugriffe erfolgen
  - sofort ➡ es kann immer das gelesen werden, was gerade geschrieben wurde
  - geordnet ➡ Zugriffe „überholen“ sich nicht
  - uniform ➡ alle Lesezugriffe zu einen Zeitpunkt ergeben stets das gleiche Ergebnis
- Diese Annahmen sind schon im nichtverteilten Fall heue kaum gegeben (Cache), Verteiltheit macht die Sache nicht besser
- Müssen uns wieder Gedanken über Konsistenz machen

11.2 Konsistenz

Konsistenzproblem

- Beispiel: Wird immer „ok“ ausgegeben?

Prozess 1

```
br:= b;
ar:= a;
if (ar>=br) then print("ok");
```

Prozess 2

```
a:=b:=0;
a:=a+1;
b:=b+1;
```

- Offensichtlich existiert ein unerwünschter Effekt durch Reihenfolge
- Mit Synchronisation kann solch ein Problem gelöst werden, aber:
  - Leistungsfähigkeit sinkt durch Synchronisationsaufwand
  - Aufwand wird mit steigender Knotenanzahl immer größer
- Konsensprotokolle zahlen diesen Preis
- Alternativ über Abstufungen der Konsistenz nachdenken

Notation

- Wir vergleichen verschiedene **Arten** von Konsistenz
- Letztendlich eine Anforderung an die Ordnung aus globaler Sicht (s. Kapitel 4)
  - Globale totale Ordnung aller Zugriffe
  - Globale totale Ordnung bzgl. bestimmer Daten
  - Globale kausale Ordnung
  - Totale Ordnung bzgl. der Zugriffe einer Anwendung
  - ...
- Schreibweisen:
  - r_2(y=23) ➡ Leseoperation von Knote 2 für Variable y, die den Wert 23 erkennt
  - w_1(x=42) ➡ Schreiboperation von Knoten 1 für x, die ihr den Wert 42 zuweist
  - Die Benennung des lesenden/schreibenden Knoten wird mitunter ausgelassen
- Wenn nichts anderes gesagt, wird davon ausgegangen, dass die Variablen anfangs den Wert 0 haben

Strikte Konsistenz

Definition 1 (Strikte Konsistenz)

Ein verteilter Speicher realisiert **strikte Konsistenz** (*strict consistency*), wenn jeder Lesezugriff den Wert der global letzten Schreiboperation liefert.

Anmerkungen:

- Strikte Konsistenz ist das, was man in einem nicht-verteilten System erwartet
- Andere Benennungen: „strenge Konsistenz“, „atomare Konsistenz“
- Sie setzt voraus, dass es Definition von „letzter Zugriff“ gibt
  ➡ globales Zeit-/Ordnungsmodell
- Problem ergab sich erstmalig Ende der 70er Jahre bei Datenbanken [Lin79]

Beispiele

- Folgendes Szenario erfüllt die strikte Konsistenz
  - Die Abszisse stellt die globale Zeit dar

P1:	w_1(x=1)		w_1(y=2)	
P2:		r_2(x=1)		r_2(y=2)

- Das folgende Szenario ist **nicht** strikt konsistent

P1:	w_1(x=1)	w_1(y=2)		
P2:		r_2(x=0)	r_2(x=1)	r_2(y=1)

Problem: Globale Zeit

- Lokal liegen keine Informationen über eine globale Zeit vor, zumindest i.d.R. nicht mit der notwendigen Granularität
- Daher muss bei jeder lokalen Zeitachse von einer anderen Zeitskala ausgegangen werden
- Die folgenden beiden Läufe sind dann (bei Wegfall der globalen Abzisse) gleich:

P1:	w_1(x=1)		w_1(y=2)
P2:		r_2(x=1)	r_2(y=2)

P1:		w_1(x=1)	w_1(y=2)
P2:	r_2(x=1)		r_2(y=2)

- Die EInführung eines Mechanismus zur Sicherstellung, dass jeder Leseoperation eine eindeutige globale Zeit zugeordnet werden kann (Lock-Step) würde das verteilte System extrem ausbremsen

Sequentielle Konsistenz

Definition 2 (Sequentielle Konsistenz)

Ein Speicher realisiert **sequentielle Konsistenz** (*sequential consistency*), wenn alle Prozesse dieselbe Reihenfolge der Speicherzugriffe wahrnehmen.

Anmerkungen:

- Definiert von Lamport zur Charakterisierung von Multiprozessorsystemen
- Jeder Prozess findet in der globalen Ordnung seine eigene Reihenfolge wieder
- Verschränkung über alle Prozesse ist aber beliebig
- Muss nicht der Wert der letzten globalen Schreiboperation geliefert werden
- Zwei Läufe desselben Programms können verschiedenen Ergebnisse liefern

P1:	w_1(x=1)			
P2:		w_2(y=2)		
P3:			r_3(y=2)	r_3(x=0) r_3(x=1)
P4:			r_4(x=0)	r_4(x=1)

Beispiel

- 3 Prozesse arbeiten auf den gemeinsamen Variablen a, b und c, die anfangs mit 0 initialisiert wurden

P1:
```
1 a=1;
2 print(b,c);
```

P2:
```
1 b=1;
2 print(a,c);
```

P3:
```
1 c=1;
2 print(a,b);
```

- Für 6 Instruktionen gibt es theoretisch $6! = 720$ Ausführungsfolgen
- Es zählt die **äußere** Wahrnehmung, Instruktionen können sich überholen
- 90 von 720 Möglichkeiten sind sequentiell konsistent, u.a.:

Folge 1:
```
a=1;
print(b,c);
b=1;
print(a,c);
c=1;
print(a,b);
```
Ergebnis: 001011

Folge 2:
```
a=1;
b=1;
print(b,c);
print(a,c);
c=1;
print(a,b);
```
Ergebnis: 101011

Folge 3:
```
b=1;
c=1;
print(a,b);
print(a,c);
a=1;
print(b,c);
```
Ergebnis: 010111

Folge 4:
```
a=1;
b=1;
c=1;
print(b,c);
print(a,c);
print(a,b);
```
Ergebnis: 111111

Kausale Konsistenz

Definition 3 (Kausale Konsistenz)

Ein Speicher realisiert kausale Konsistenz (*causal consistency*), wenn alle Schreiboperationen, die potentiell kausal voneinander abhängen, von allen Prozessen in derselben korrekten Reihenfolge wahrgenommen werden.

Anmerkungen:

- Schwächer als sequentielle Konsistenz, da sich die Definition sich nur auf kausal verknüpfte Schreiboperationen bezieht
- Zwei Schreiboperationen sind potentiell kausal abhängig, wenn vor der zweiten Operation eine Leseoperation stattgefunden hat, die den Wert der zweiten Schreiboperation beeinflusst haben **kann**
- Wenn $w_2()$ potentiell kausal von $w_1()$ abhängt, dann ist die Reihenfolge $w_1()$ $w_2()$ korrekt

Beispiel

- Die folgende Ausführung ist kausal konsistent, aber nicht sequentiell konsistent
- w_1(x=1) und w_2(x=2) sind kausal abhängig, da P2 die Schreiboperation wahrgenommen hat
- P3 und P4 sehen die Updates in unterschiedlicher Reihenfolge, jedoch sind die dazugehörigen Leseoperationen nicht kausal abhängig
- Es gibt hier keine globale Zeitachse mehr, nur noch Reihenfolge pro Prozess!

P1:	w_1(x=1)		w_1(x=3)	
P2:		r_2(x=1) w_2(x=2)		
P3:		r_3(x=1)	r_3(x=3)	r_3(x=2)
P4:		r_4(x=1)	r_4(x=2)	r_4(x=3)

FIFO-Konsistenz

Definition 4 (FIFO-Konsistenz)

Ein Speicher realisiert **FIFO-Konsistenz** (*FIFO consistency*), wenn alle **Schreib**operationen **jeweils** jedes einzelnen Prozesses von allen anderen Prozessen in derselben Reihenfolge wahrgenommen werden, während Schreiboperationen von **verschiedenen** Prozessen in verschiedenen Reihenfolgen wahrgenommen werden können.

Beispiel:

- Ein Prozess P1 führt w_{11} und w_{12} durch, ein Prozess P2 w_{21} und w_{22}
- ➡ Ausführung genügt aus globaler Sicht der FIFO-Konsistenz, wenn P1 und P2 jeweils irgendeine (auch unterschiedliche) dieser Sequenzen wahrnehmen:
  (w_{11} w_{12} w_{21} w_{22}), (w_{11} w_{21} w_{12} w_{22}), (w_{11} w_{21} w_{22} w_{12}), (w_{21} w_{11} w_{12} w_{22}), (w_{21} w_{11} w_{22} w_{12}), (w_{21} w_{22} w_{11} w_{12})
- Berücksichtigt unterschiedliche Latenzen bei Kommunikation

FIFO-Konsistenz (Forts.)

- Schreibzugriffe verschiedener Prozesse können von unterschiedlichen Prozessen in unterschiedlicher Reihenfolge gesehen werden
- Die Bezeichnung FIFO-Konsistenz wird so nur von TANENBAUM/VAN STEEN benutzt; andere Bezeichnungen sind **PRAM-Konsistenz** oder **Prozessorkonsistenz**[1]
- FIFO-Konsistenz kann zu unerwarteten Ergebnissen führen:

P1

```
1 a=1;
2 if (b==0) kill(P2);
```

P2

```
1 b=1;
2 if (a==0) kill(P1);
```

Unter FIFO-Konsistenz können (auch) **beide** Prozesse beendet werden!

[1] **Achtung:** In der Literatur wird der Begriff „Prozessorkonsistenz“ (*processor consistency*) mit (mindestens) drei verschiedenen Semantiken belegt, weshalb in dieser Vorlesung der (seltenere) Begriff „FIFO-Konsistenz“ bevorzugt wird.

Beispiel

- Von P3 und P4 werden die beiden Schreiboperationen in unterschiedlicher Reihenfolge gesehen
- Da sie aber von unterschiedlichen Prozessen durchgeführt wurden, genügt die Ausführung der FIFO-Konsistenz, aber nicht der kausalen Konsistenz, da die beiden Schreiboperationen (potentiell) kausal abhängig sind
- Es gibt hier keine globale Zeitachse mehr, nur noch Reihenfolge pro Prozess!

P1:	w_1(x=1)			
P2:		r_2(x=1)	w_2(x=2)	
P3:			r_3(x=1)	r_3(x=2)
P4:			r_4(x=2)	r_4(x=1)

Schwache Konsistenz

- In der Regel benötigen Prozesse nur in bestimmten Situationen eine konsistente Sicht auf den Speicher
- Schwache Konsistenz unterscheidet daher **normale** Variablen von **Synchronisationsvariablen**

Definition 5 (Schwache Konsistenz)

Ein Speicher realisiert **schwache Konsistenz** (*weak consistency*) , wenn die folgenden Bedingungen gelten:

- Alle Zugriffe auf Synchronisationsvariablen genügen der sequentiellen Konsistenz.
- Alle vorangegangenen Schreiboperationen sind überall abgeschlossen, bevor ein Zugriff auf eine Synchronisationsvariable erlaubt ist.
- Alle vorangegangenen Zugriffe auf Synchronisationsvariable sind abgeschlossen, bevor ein Zugriff auf eine normale Variable stattfindet.

Beispiel

- Die Synchronisation erzwingt eine Aktualisierung der Werte im Speicher
- Wer seine Schreibzugriffe sichtbar machen möchte für andere Prozesse muss synchronisieren
- Wer auf die aktuellsten Daten zugreifen möchte muss ebenfalls synchronisieren
- Wer vor dem Lesezugriff nicht synchronisiert **kann** veraltete Werte sehen

P1:	w_1(x=1)	w_1(x=2)	S		
P2:		r_2(x=1)	S	r_2(x=2)	
P3:		r_3(x=2)	r_3(x=1)	S	

Diskussion

- Die hier vorgestellten Konsistenzmodelle sind nicht erschöpfend
- Andere Modelle sind z.B. lokale Konsistenz (*local consistency*), langsame Konsistenz (*slow consistency*), Freigabe-Konsistenz (*release consistency*), Kohärenz (*memory coherence*) oder Eingangskonsistenz (*entry consistency*)
- Modelle haben Abhängigkeiten, bspw.:
  - Sequentielle Konsistenz impliziert kausale Konsistenz.
  - Kausale Konsistenz impliziert FIFO-Konsistenz.
- Auch sind die hier dargestellten Modelle etwas vereinfacht: eigentlich müsste genauer zwischen den Zugriff auf eine und dieselbe Variable (**Koheränz**) und auf unterschiedliche Variablen unterschieden werden
  - Unter diesem Gesichtspunkt gibt es eine ganze Reihe von Modellen, die im Bereich FIFO/Prozessor/PRAM angesiedelt sind
- Ein Programm ist unter einem Modell **nur** dann korrekt, wenn es für **alle** möglichen Abläufe ein hinnehmbares Ergebnis liefert
- Allgemein gilt: Je „näher“ ein Konsistenzmodell am nichtverteilten Fall ist, desto teurer ist die Implementation

11.3 Linda

- Konsistenz ist **nur** ein Problem, da der Mensch im Speichermodell des von-Neumann-Rechners verhaftet ist
  - Speicher: Wert, die über Adresse angesprochen wird
- Alternativer Ansatz Linda: **assoziativer Speicher**
  - Wert und Adresse werden nicht mehr getrennt ➡ Tupel
  - Klassische Speicherzelle kann als Zweiertupel (Adresse, Wert) aufgefasst werden
  - Linda-Tupel ist allgemeines n-Tupel
  - Bekannt von Caches, z.B. TLB
- **Tupelspace:**
  - Prozesse schreiben und lesen Tupel in/aus einem Tupelspace
  - Tupel hat keinen (dedizierten) **Ort** ➡ mehrere identische Tupels können **nebeneinander** bestehen
- Eigenschaft: Konsistenz tritt **nicht** als Problem auf, da ‘Überholen’ als Problem nicht mehr existiert

Speicherprimitiven

- Primitiven in Linda:
  - `in` liest und entfernt ein Tupel aus dem Tupelspace (atomar)
  - `rd` liest Tupel nichtzerstörend
  - `out` schreibt ein Tupel in den Tupelspace
  - `eval` erzeugt neue Prozesse um Tupels zu evaluieren und schreibt das Ergebnis in den Tupelspace
- Bei Leseoperationen wird durch angegebene Tupelkomponenten ein logischer Name gebildet
- Beispiel: `in(42,?,'a',?)` gibt *ein* Tupel zurück, dessen erste Komponente 42 und dessen dritte Komponente ’a’ ist
  - Gibt es kein solches Tupel, blockiert die Operation bis ein entsprechendes Tupel im Tupelspace erscheint

Beispiel

- Tokenpassing-Algorithmen in C-Linda [CaGe93]

```
#define NPROC 4

ring(id){
  if (id) in("token", id);
  out("token",(id+1)%NPROC);
  if (!id) in("token",id);
}

real_main(){
  int i;
  for(i=1;i<NPROC;i++) eval("ring node", ring(i));
  ring(0):
 }
```

11.4 Speicherbereinigung

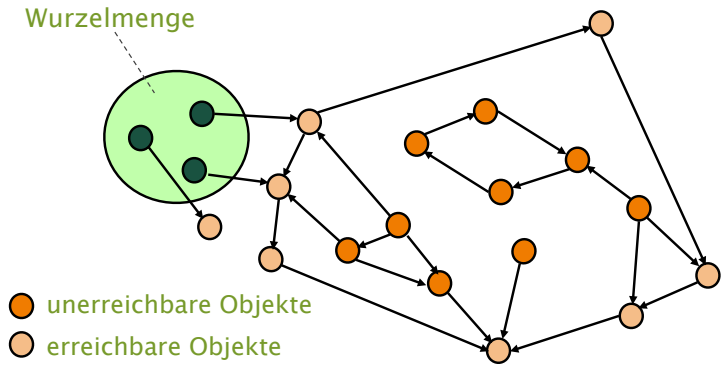
- Ursprünglich wurde die Speicherverwaltung manuell vom Programmierer durchgeführt
  - `new` und `delete` in C++
  - `malloc` und `free` in C
- Manuelle Speicherverwaltung ist schon in zentralisierten Systemen äußerst komplex!
  - **Programmfehler** durch fehlerhaftes Durchführen der Speicherverwaltung (z.B. Nutzung von freigegebenem Speicher)
    ➡ **dangling pointer**
  - **Speicherlöcher** durch vergessene Freigabe von alloziertem Speicher
    ➡ **memory leak**

Speicherbereinigung (Forts.)

- Wegen der Fehleranfälligkeit wird die Speicherverwaltung zunehmend im Laufzeitsystem automatisiert (z.B. in Java, Go, C#)
  ➡ **transparente** Speicherbereinigung (*garbage collection*)
- **Ziel**: Freigabe von Speicherblöcken (z.B. Objekten), auf die nicht mehr zugegriffen werden kann
- Betrachten/wiederholen zunächst den nichtverteilten Fall

Graph der Objektreferenzen

- Alle von der **Wurzelmenge** aus erreichbaren Objekte werden noch gebraucht; alle anderen können freigegeben werden



Anforderung: Sicherheit und Lebendigkeit

- Wie bei anderen Algorithmen gibt es Sicherheits- und Lebendigkeitsanforderungen
  - **Sicherheit**
    Es werden nur Objekte bereinigt, auf die nicht mehr zugegriffen werden kann
  - **Lebendigkeit**
    Ein Objekt, auf welches nicht mehr zugegriffen werden kann, wird nach endlicher Zeit bereinigt

Konzepte

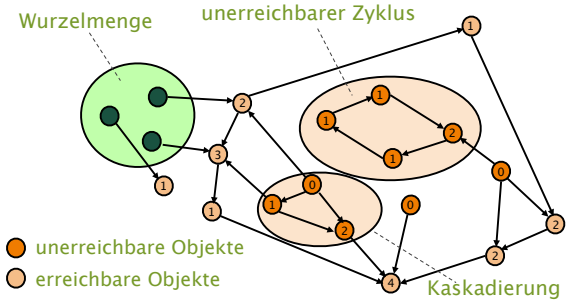
- Es werden in diesem Kontext zwei Arten von Algorithmen unterschieden:
  - **Mutator**: Manipuliert Variablen, die Referenzen auf Objekte enthalten
    ➡ Anwendungsprogramm (Basisalgorithmus)
  - **Kollektor**: Sucht Objekte, die bereinigt werden können und bereinigt diese
    ➡ Steuerungsalgorithmus
- Teilweise sind Mutator und Kollektor miteinander verschränkt, z. T. echt nebenläufig (auch im nichtverteilten Fall)
- Prinzipiell gibt es zwei Klassen von Kollektor-Ansätzen:
  - **Verfolgungsalgorithmen** (*tracing algorithmen*)
  - **Referenzzählung** (*reference counting*)
- Beginnen mit letzterem

11.4.1 Referenzzählung

- Jedes Objekt erhält einen Referenzzähler
  - Wird eine neue Referenz kreïert, wird der Referenzzähler des referenzierten Objektes inkrementiert
  - Wird eine Referenz vernichtet (das referenzierende Objekt freigegeben), wird der Referenzzähler des referenzierten Objektes dekrementiert
  - Erreicht ein Referenzzähler eines Objekts den Wert Null, existiert keine Referenz auf dieses Objekt, und es wird gelöscht
- **Einsatzbeispiele**: Apple (Objective-C, Swift), Perl, COM

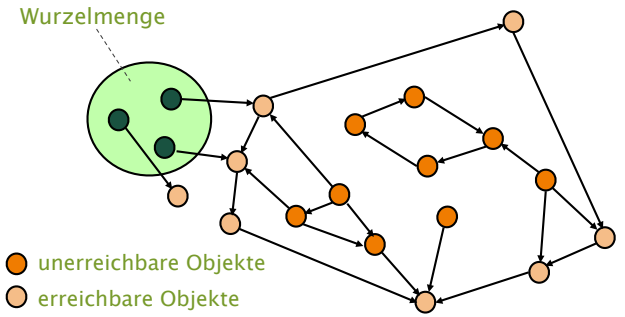
Diskussion

- **Vorteile**:
  - Aufwand für Bereinigung wird über die Laufzeit verteilt
  - Kein Stop-die-Welt-Ansatz nötig
- **Nachteile**:
  - Kaskadeneffekte möglich
  - Kein Auffinden von Referenz-Zyklen



11.4.2 Verfolgung

- **Grundidee**:
  - „Verfolgung“ der Erreichbarkeit von der Wurzelmenge aus
- **Klassischer Algorithmus**: Mark & Sweep
- **Einsatzbeispiele**: JVM-Sprachen (Java, Scala, Conjure), Smalltalk, Python



Mark & Sweep

- Das Verfahren verwendet eine Markierung für jedes Objekt und läuft in 2 Phasen ab
- 1. Phase: **Markieren** (Mark)
  - Das System wird angehalten
  - Die Markierung aller Objekte wird gelöscht
  - Ausgehend von der Wurzelmenge werden alle ausgehenden Referenzen verfolgt und die referenzierten Objekte markiert
  - Dieser Vorgang wird rekursiv fortgesetzt, bis keine unmarkierten Objekte mehr erreicht werden (Baumsuche)
- 2. Phase: **Ausfegen** (Sweep)
  - Alle unmarkierten Objekte werden bereinigt
  - Das System wird fortgesetzt

Diskussion

- **Vorteile:**
  - Alle zu bereinigenden Objekte werden sicher identifiziert
  - Kein / sehr geringer Overhead bei den Objekten (1 Bit)
- **Nachteile:**
  - Einfrieren des Systems (*Stop the world*)
  - Der gesamte (benutzte) Speicher muss inspiziert werden
    ➡ ungünstig in Verbindung mit **Demand Paging**
- Mark & Sweep wird in dieser reinen Form selten genutzt ➡ Vielzahl von Verbesserungen

Diskussion – Verbesserungen

- **Tricolor**:
  - Unterscheidung von drei Klassen: „sicher zubereinigen“, „sicher nicht zu bereinigen“, „evtl. zubereinigen“
  - nur letzte Klasse muss evaluiert werden
  - ➡ Performancegewinn
- **Schnappschuss**:
  - Beobachtung: einmal unerreichbare Objekte bleiben unerreichbar
  - Vorgehen: nebenläufig (konsistenten) Schnappschuss anlegen und auf dieser Grundlage Bereinigung durchführen
  - In der Zwischenzeit erzeugt der Mutator neue unerreichbare Objekte, die aber beim nächsten Lauf bereinigt werden
  - ➡ besseres Laufzeitverhalten, aber erhöhter Speicherbedarf
- **Generationelle Algorithmen**:
  - Beobachtung: viele Objekte „leben“ sehr kurz, andere länger
  - Vorgehen: Unterscheidung nach Lebensdauer und Anpassung der Strategien
  - ➡ besserer Trade-off

11.4.3 Speicherbereinigung in Verteilten Systemen

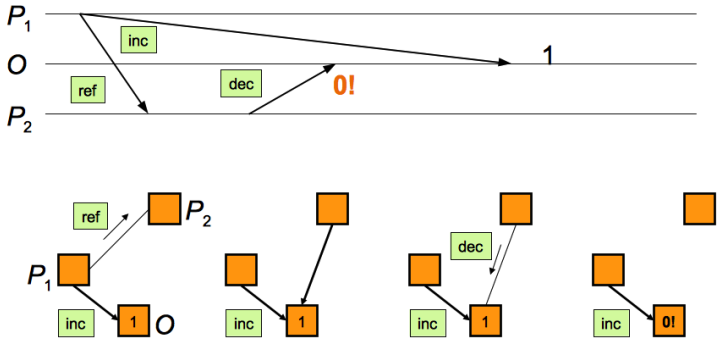
- Betrachten verteilte (Betriebs-)Systeme
- Schwieriger als in zentralisierten Systemen
- Remote-Referenzen erlauben den Zugriff auf Objekte, die auf anderen Rechnern liegen
- **Wichtig**: Remote-Referenzen können in Nachrichten **unterwegs** sein
- Koordination (wie immer) ausschließlich durch Nachrichten

Bereinigung vs. Terminierung

- Das Bereinigungsproblem ähnelt offensichtlich dem Terminierungsproblem:
  - Bei beiden Problemen wird ein Kontrollalgorithmus den Basisalgorithmen überlagert und nebenläufig ausgeführt
  - „Globale Terminierung“ und „Objekt kann bereinigt werden“ sind beides stabile Prädikate
- Kontrollalgorithmus soll das stabile Prädikat entdecken
- **Problem**: Aktionen hinter dem Rücken des Kontrollalgorithmus
  - **Terminierung**: Nachricht senden, die einen anderen Prozesses reaktiviert
  - **Bereinigung**: Referenz kopieren und verschicken, die weiterhin den Zugriff auf das Objekt ermöglicht
- Können Lösungen des einen Problems auf das andere Problem übertragen werden? (Spoiler: z. T. ja)

Fehlinterpretation

- Betrachten zunächst Referenzzählung
- Inkrement- und Dekrement-Nachrichten können sich gegenseitig direkt oder indirekt überholen
- Dies kann zu einer inkonsistenten Sicht und damit zu einer Fehlinterpretation führen



Problemlösung

- Ansätze zur Beseitigung möglicher Fehlinterpretationen
- **Synchrone** Kommunikation
  - Die Referenz wird nach der Inkrement-Nachricht abgeschickt
  - Da die Kommunikation synchron ist, kann die Referenz erst verschickt werden, nachdem die Inkrement-Nachricht angekommen ist
- **Bestätigung** der Inkrement-Nachricht
  - Die Inkrement-Nachricht wird abgeschickt
  - Die Inkrement-Nachricht wird vom Empfänger bestätigt
  - Erst wenn die Bestätigung empfangen wurde, wird die Referenz abgeschickt
- Erzwingen der **kausalen Ordnung** beim Empfang von Nachrichten
  - Aus Sicht des Empfänger können sich dann Nachrichten weder direkt noch indirekt überholen

Gewichtete Referenzzählung – Kreditmethode

- Beruht auf einem Algorithmus von Shing-Tsaan Huang (1989), der ursrünglich für die Terminierungserkennung gedacht war
- **Idee**:
  - Jedes neu (außerhalb der Wurzelmenge) kreïerte Objekt besitzt einen **Kreditwert** 1
  - Bei Versenden der Objektreferenz enthält die Nachricht die Hälfte des aktuellen Kredits, der Kredit des referenzierten Objekts vermindert sich entsprechend
  - Wird eine Referenz gelöscht, wird die entsprechende Kreditsumme an das referenzierte Objekt gesendet
  - Empfängt ein aktiver Prozess eine Nachricht, so erhöht sich sein Kredit um den Kredit der Nachricht

Gewichtete Referenzzählung – Kreditmethode (Forts.)

- **Invariante**
  - Die Kreditsumme der Referenzen eines Objekts ist stets 1
  - Referenznachrichten führen stets einen Kredit > 0 mit sich
  - Referenzierte Objekte haben stets einen Kredit > 0
- **Bereinigung**
  - Wenn die Kreditsumme bei einem Objekt wieder 1 ist, existieren keine Referenzen mehr, und es kann gelöscht werden

Optimierung: Darstellung der Kreditanteile

- Gleitkommazahlen ungeeignet ➡ Rundungsfehler
- **Besser**: Speicherung des negativen dualen Logarithmus des Kreditanteils $c = -\log_2 k$
  - $k = 1$ ➡ $c = 0$
  - $k = \frac{1}{2}$ ➡ $c = 1$
  - $k = \frac{1}{4}$ ➡ $c = 2$
  - ...
- Teilen und Verdoppeln entspricht In-/Dekrement von c
  - $c' = c \pm 1$
- Referenziertes Objekt hält Menge von Kreditwerten (Bitvektor)
- Zwei gleiche Kreditwerte werden zu nächstgrößeren vereinigt
- **Problem**: Bei Nutzung einer Integer-/Bitvektordarstellung können Kredite nicht beliebig oft geteilt werden

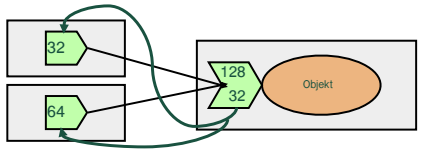
Gewichtete Referenzzählung –- Variante

Folgender Ansatz löst das Problem der Gewichtsteilung

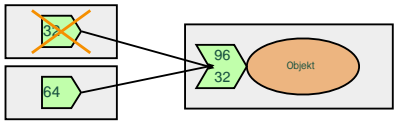
- Objekt hat **zwei** Gewichte: **Teilgewicht** und **Gesamtgewicht**
- Beide sind initial gleich und haben als Wert eine Zweierpotenz $g = 2^c$
- Referenzen haben nur ein Teilgewicht
- Das Gewicht wird als ganze Zahl in normaler Form gespeichert (also nicht logarithmisch)
  ➡ c-malige Teilung möglich

Gewichtete Referenzzählung –- Variante (Forts.)

- Ausgabe einer Remote-Referenz oder Kopieren einer Remote-Referenz wie gehabt (Teilgewicht wird halbiert)



- Vernichten einer Remote-Referenz: Teilgewicht wird zum Objekt geschickt und dort vom Gesamtgewicht **abgezogen**



Gewichtete Referenzzählung –- Variante (Forts.)

- **Invariante**
  Summe aller Teilgewichte = Gesamtgewicht des Objekts
- Objekt kann bereinigt werden, wenn Teilgewicht des Objekts = Gesamtgewicht des Objekts
- Was passiert, wenn ein Teilgewicht 1 ist und es halbiert werden soll?
  - Gesamtgewicht des Objekts und Teilgewicht des Objekts bzw. der Referenz um $g - 1$ (z.B. um 127) erhöhen
  - Invariante **bleibt erhalten**, da beide Seiten der Gleichung um die gleiche Zahl erhöht werden

Verteiltes Mark & Sweep

- Da das System eingefroren wird, kann der Algorithmus auch im verteilten Fall eingesetzt werden
- Einfrieren und auftauen erfolgt z.B. über Wellenalgorithmen
- Allerdings stellt Einfrieren im verteilten Fall wegen der größeren Latenz einen noch schwereren Eingriff dar, als im lokalen (nur nebenläufigen) Fall
- Lösungsansätze:
  - **Logically Centralized Reference Service** (Ladin und Liskov)
  - **Nutzung von Zeitstempeln** (Hughes)

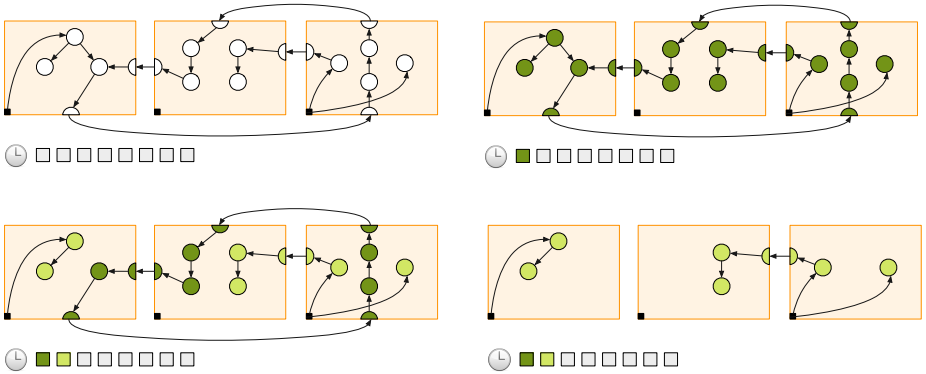
Ansatz von Ladin und Liskov

- Alle Knoten schicken ihre lokale Sicht zu einem zentralen Server
- Dort wird ein globaler Referenzgraph aufgebaut ➡ Schnappschuss
- Mark & Sweep auf zentralen Referenzgraph, Senden der Bereinigungsinformation an die Knoten
- **Diskussion**
  - Konsistenter Schnappschuss ist nichttrivial (vgl. Kapitel) 10
  - Bereinigungsmenge ist nicht aktuell
  - Zentraler Server stellt einen Flaschenhals dar

Ansatz von Hughes

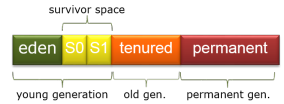
- Voraussetzung: globale Zeit ➡ Uhrensynchronisation
- Markierung erfolgt nicht mit einem Bit, sondern mit Zeitstempel
- Jeder Knoten führt lokal eine Markierungsphase durch und propagiert externe Referenzinformationen
- Empfangene Referenzinformationen werden aktualisiert
- Objekte mit Zeitstempeln kleiner als der kleinste globale Zeitstempel könnten bereinigt werden
- Diskussion
  - nicht robust ➡ langsamer oder ausgefallener Knoten wird zum Problem
  - Uhrensynchronisation notwendig

Ansatz von HUGHES (Forts.)



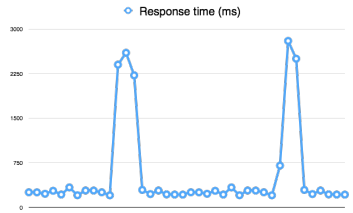
11.4.4 Diskussion

Fallbeispiel Java



Bildquelle: Wikipedia

- Java nutzt verschiedene Ansätze:
  - **Intern**: generationaler Verfolgungsalgorithmus
  - **RMI** (*remote message invokation* ➡ Java's RPC-Variante) einen verteilten Referenzzähler-Algorithmus
- **Problem verteilter Fall**:
  - Nichtentscheidbar, ob referenzhaltendes Objekt noch lebt oder gecrashed / Nachricht verloren gegangen ist
  - Lösung: dekrementieren des Referenzzählers nach vorgegebener Zeit ➡ Gefahr falscher Bereinigung
  - Alternativ (Standardfall): Von Zeit zu Zeit (1h ab Version 6, vorher 1min) wird „globale" GC durchgeführt ➡ Performanceproblem



Bildquelle: plumbr.io

11.4.4 Diskussion (Forts.)

- Es gibt keine perfekten Ansatz zur verteilten Speicherbereinigung
- Viele weitere Ansätze, inklusive hybrider Algorithmen
- Praktische Konsequenz: Aus performancekritischen Anwendungen sollten entfernte Referenzen (z.B. Java-RMI) vermieden werden (ggf. Speicherbereinigung überhaupt)
- Aktuelles Forschungsthema ➡ Potential für neue Lösungen

Literatur

[CDK05] Coulouris , G. , J. Dollimore und T. Kindberg: *Distributed Systems: Concepts and Design*. Prentice Hall, 2005, Kapitel 18

[StNu04] Steinke , R. C. und G. J. Nutt: „A Unified Theory of Shared Memory Consistency". *Journal of the ACM*, 51(5)2004, 800–849

[CaGe93] Carriero , N. und D. Gelernter: *Linda and Message Passing: What have we learned?*, technical report, Yale University, Department of Computer Science

[ACG86] Ahuja , S. , N. Carriero und D. Gelernter: „Linda and Friends". *IEEE Computer*, 19(8)1986, 26–34

[TaSt02] Tanenbaum , A. S. und M. van Steen: *Distributed Systems*. Prentice Hall, 2002 , Abschnitt 4.3

[JoLi96] Jones , R. und R. D. Lins: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. WileyJohn Wiley & Sons, 1996