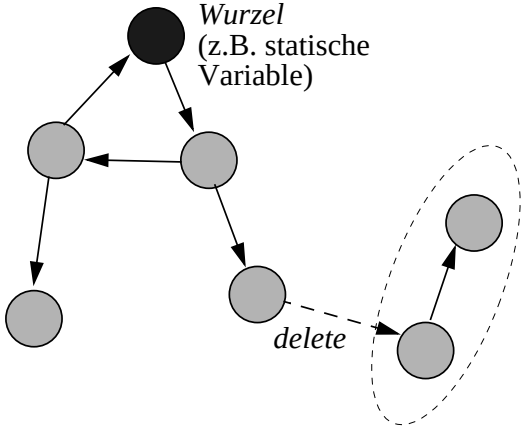
Garbage-Collecton in verteilten Systemen

Garbage-Collection

- "Verpointerte Objekte"



Diese beiden Objekte sind nicht mehr von der Wurzel aus erreichbar und werden zu "garbage"

Ein *Garbage-collector* soll solche Objekte identifizieren und deren Speicher wiederverwenden

- Wenn man diese Objekte als "aktiv" ansieht, hat man schon "fast" ein paralleles / verteiltes System

- Vgl. Puppentheater: Auch wenn eine einzige Person die Figuren im Zeitmultiplex bedient, kann man dies als ein paralleles System autonomer Objekte betrachten

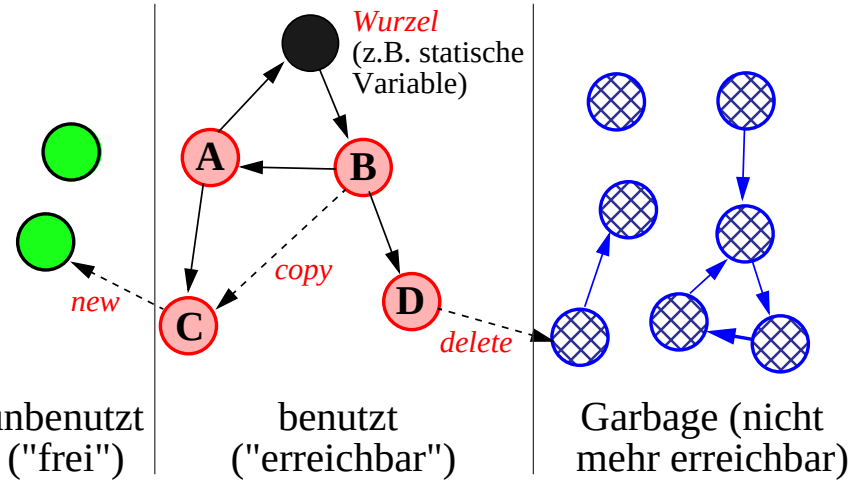
- Typischerweise läuft auch der Garbage-collector (echt oder im Zeitmultiplex) parallel zur Anwendung

==> Algorithmen zur Identifikation von Garbage-Objekten im parallelen (oder verteilten) Fall nützen auch bei sequentiellen objektorientierten Systemen

Garbage-Collection ist allerdings (insbesondere in einer verteilten Umgebung) nicht trivial!

Garbage-Collection-Modell

- Zweck: Recycling von "verbrauchtem", unbenutztem Speicher
- Bei Sprachen mit dynamischem Speicher und Zeigerstrukturen
  - historisch: LISP (bereits in den frühen 60er Jahren)
  - Interesse heute: objektorientierte Sprachen (+ ggf. parallele Implementierung)
  - statische Variablen und Variablen im Laufzeitkeller ("Stack") stets erreichbar, dynamische Variablen auf der Halde ("Heap") u.U. jedoch "abgehängt"



- *copy:* Füge Referenz zwischen 2 erreichbaren (!) Objekten hinzu
  z.B.: obj1.refvar := obj2.refvar

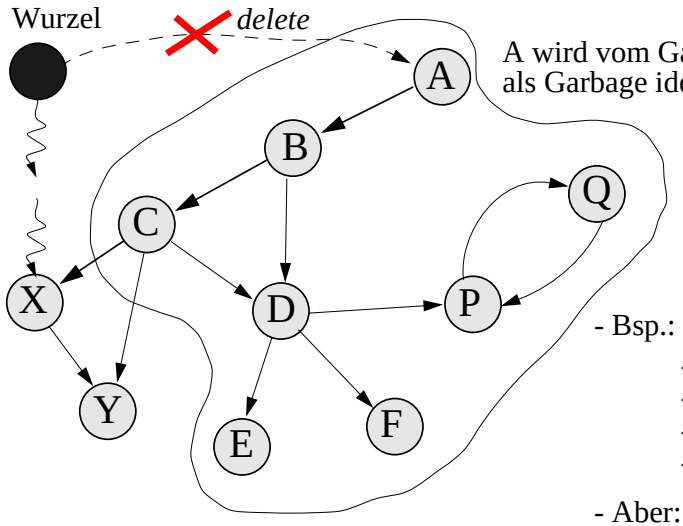
- Objekt *erreichbar* <==> ∃ Pfad von der Wurzel dorthin
- "Garbage sein" ist stabiles Prädikat (vgl. Terminierung!)
- *Mutator* <--> *Collector* spielen mit-/gegeneinander

Anwendungsprogramm	Kontrollprogramm
(manipuliert Zeiger zwischen Objekten)	identifiziert Garbage

Rekursives Freigeben

- Falls ein Objekt als Garbage erkannt wird:
  - sollten seine ausgehenden Referenzen gelöscht werden,
  - damit kann ggf. eine grössere daran "aufgehängte" Substruktur als Garbage erkannt werden!

  --> rekursives Erkennen von Garbage-Objekten, "pointer chasing"



- Bsp.: A Garbage
  --> B Garbage
  --> C Garbage
  --> D Garbage
  --> E, F Garbage
- Aber: X, Y kein Garbage!

- P, Q sind dann auch Garbage, manche Garbage-Collectoren können solchen "zyklischen Garbage" jedoch nicht erkennen!

Freezing: "Stop the World"-Prinzip

Mark & sweep-Algorithmus:

Das geschieht meist automatisch, weil Weiterarbeit unmöglich ist

1) Mutator anhalten ("out of memory")
2) Alle erreichbaren Objekte markieren (ausgehend von der Wurzel) --> Graph-Traversierung
3) Garbage = alle unmarkierten Objekte
4) Mutator wieder starten

"sweep" durch den Hauptspeicher und z.B. in eine Freispeicherliste einketten

- "Stop the world"-Paradigma --> schlechte Lösung für Realzeit- und interaktive Anwendungen!
- Erst recht untragbar in verteilten Systemen!

- Vergleiche dies mit dem folgenden (schlechten!) Terminierungs-Entdeckungverfahren:
  - friere alle Prozese ein
  - ermittle Gesamtzahl der versendeten und empfangenen Nachrichten
  - falls identisch und alle Prozesse passiv sind: terminiert
  - ansonsten: "auftauen" aller Prozesse
- Eingefrorener globaler Zustand ist konsistent!
  - "in aller Ruhe" ein Objekt nach dem anderen betrachten

"Behind the back copy"-Problem

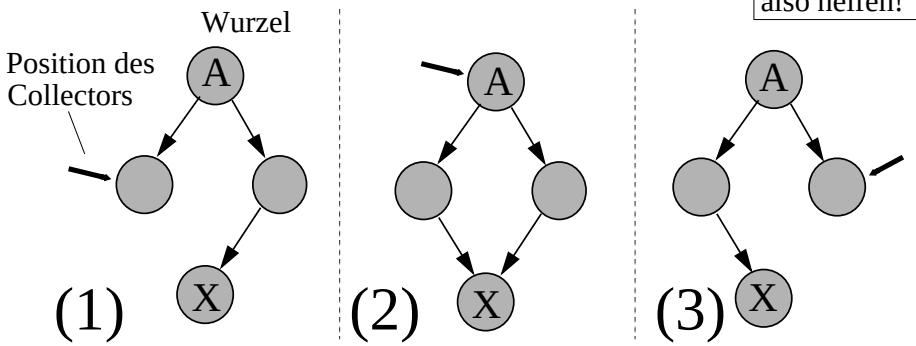
*Concurrent / parallel / on-the-fly-G*arbage-Collection:

- Collector versucht, Garbage-Objekte zu identifizieren, *während* der Mutator aktiv ist
- Verhindert somit lange Wartezeiten der Anwendung

Traversiert den Graphen und markiert erreichbare Objekte

Collector kann von gleichzeitig aktivem Mutator getäuscht werden --> Kooperation notwendig!

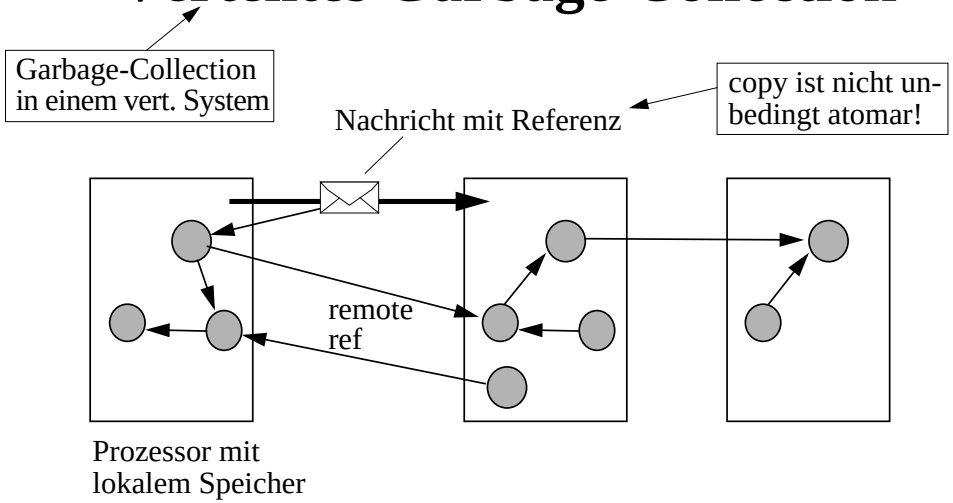
Mutator muss dem Collector also helfen!



- Knoten X wird als nicht erreichbar angesehen...
- ...obwohl es immer einen Weg von A zu X gibt!

- Manipulationen "hinter dem Rücken" des Collectors
- Collector rekonstruiert aus seinen zusammengesetzten lokalen Beobachtungen einen falschen (nie existenten) Graphen
- dabei ist Beobachtungszeitpunkt = Besuchszeitpunkt

Verteiltes Garbage-Collection

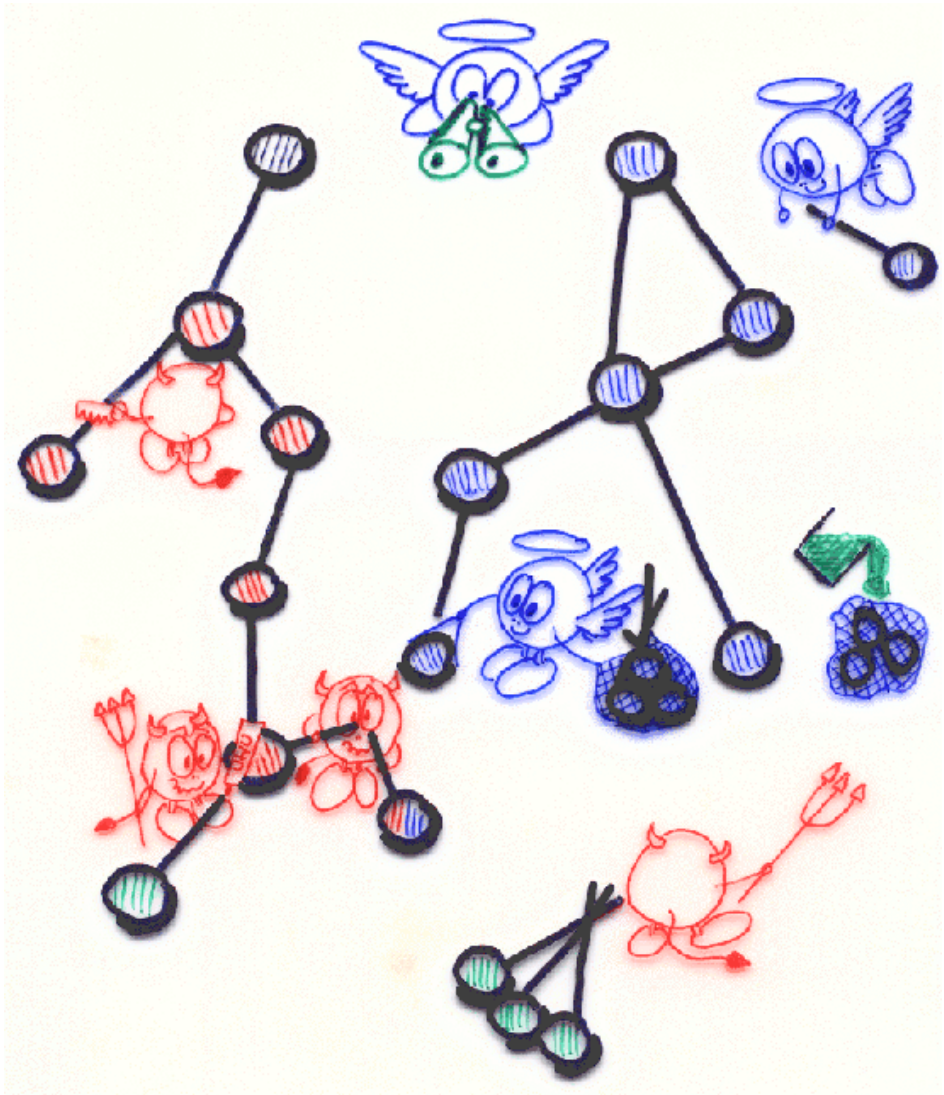


(1) Prozessorüberschreitende Referenzen ("*remote ref*")

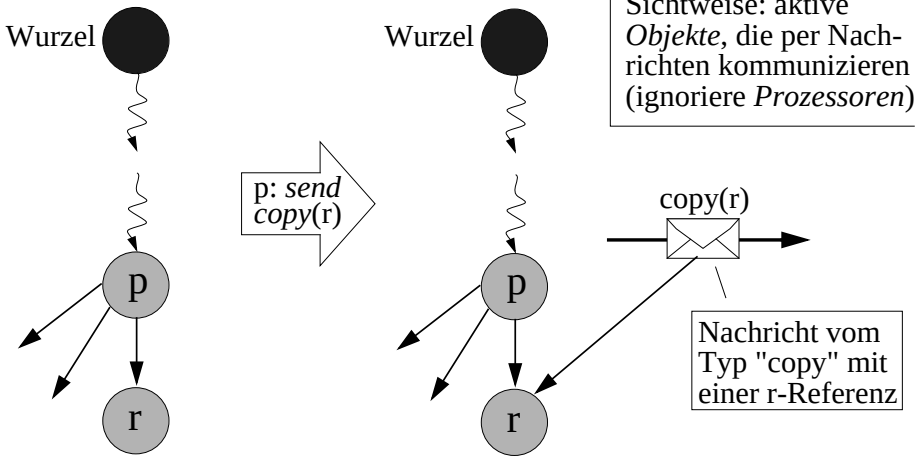
(2) Nachrichten enthalten (Objekte mit) Referenzen auf andere Objekte ("*remote copy*")

- Referenzen in Nachrichten dürfen nicht übersehen werden!

- GC typischerweise dezentral, parallel und hierarchisch
  - *lokale* GC (Annahme: alle eingehenden remote refs kommen von einem erreichbaren Objekt)
  - *globale* GC (Suchen prozessorübergreifender "Garbage-Ketten")
- GC aufwendiger als im nicht-verteilten Fall
- Viele Mutator / Collector (pro Prozessor einen?)
  - Synchronisation zwischen den vielen Prozessen notwendig
- Pragmatisches:
  - "Stop the world" ist hier schon gar nicht angemessen
  - Kontrollkommunikation minimieren (Kosten, Effizienz)

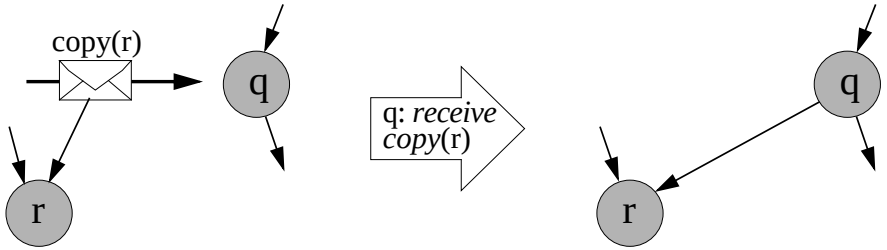


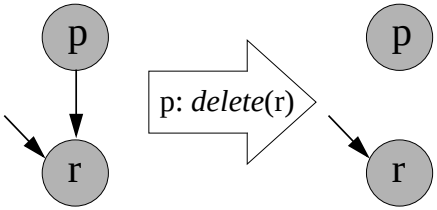
Wirkung der Mutator-Operationen



Sichtweise: aktive *Objekte*, die per Nachrichten kommunizieren (ignoriere *Prozessoren*)

Beachte: Copy-Operation ist nicht atomar ==> Aufspalten in zwei Aktionen *send / receive copy*





- "new" hier nicht relevant
- jede Aktion ändert den globalen Graphen "etwas"
- Folge solcher Änderungen -->" Berechnung"
- hier: "Interleaving-Modell": Operationen sind atomar ("zeitlos") --> es gibt keine gleichzeitige Aktionen --> verschränkte Ausführung

Formalisierung des verteilten Garbage-Collection-Problems

Aktionen des Mutators (lokal zu jedem Objekt p):

beachte: p kennt q (hat also "in gewissem Sinne" eine Referenz auf q)

guard

C_p: {p ist erreichbar und hat eine r-Referenz}
send copy(r) **to** q

Parameter Name } der Nachricht (vgl. Prozeduraufruf)

R_p: {Empfang einer copy(r)-Nachricht}
Installiere eine r-Referenz in p

D_p: {p hat eine r-Referenz}
Lösche die r-Referenz

"new" auf copy zurückführen: freie Objekte sind stets erreichbar (z.B. über eine an der Wurzel verankerte Freispeicherliste)

- So "sieht" der Collector die Basisberechnung
- Vergleiche dies mit Basisaktionen beim Problem der verteilten Terminierung!
- Aufgabe: Überlagerung mit Aktionen des Collectors:
  - zusätzliche atomare Aktionen
  - Ergänzung der 3 Aktionen mit weiteren Statements, um die notwendige Kooperation mit dem Collector zu erreichen
- Bedingungen an eine korrekte Lösung:
  - *Safety*: Wenn ein Objekt eingesammelt wird, dann ist es Garbage
  - *Liveness*: Wenn ein Objekt Garbage ist, dann wird es *schliesslich* eingesammelt

Verteilte Terminierung und Garbage-Collection

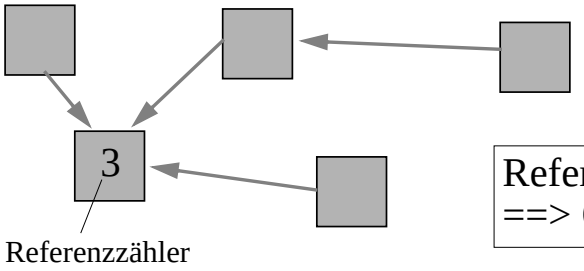
- Interessante Analogie zwischen beiden Problemen:

Verteilte Terminierung	*Garbage-Collection*
Überlagerter Kontrollalgorithmus	
Globale *Terminierung* ist ein stabiles Prädikat	"Objekt ist *Garbage*"
Kontrollalgorithmus soll das globale Prädikat entdecken	
Kontrollalgorithmus nebenläufig zur Anwendung	
Senden einer *Nachricht*	Kopieren einer *Referenz*
Problem: Behind the back	
reactivation	*copy*

- Können Lösungen des einen Bereiches auf den anderen Problembereich angewendet werden?

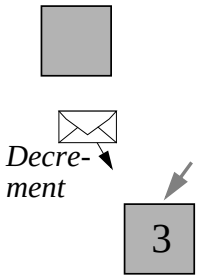
Referenzzähler-Verfahren

- Idee: Jedes Objekt weiss, wie oft es referenziert wird
  - wird dieser Referenzzähler 0 --> Garbage
- Nachteil: Zyklischer Garbage wird nicht entdeckt
- Zugehörigen Referenzzähler "atomar" zusammen mit der copy- oder delete-Operation aktualisieren
  - relativ einfach in einem nicht-verteilten System
- In einem verteilten System:
  increment / *decrement* Nachrichten



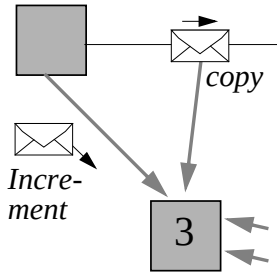
Referenzähler = 0
==> Objekt ist Garbage

Löschen einer Referenz:



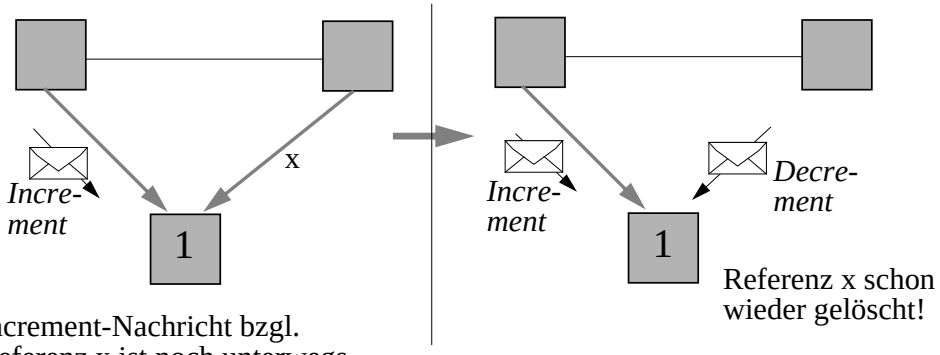
3 ==> 2

Kopieren einer Referenz:



3 ==> 4

Kopieren kann zu *Fehlinterpretationen* führen!



Decrement schneller als vorangehendes Increment
==> Referenzzähler wird 0, jedoch kein Garbage!

Korrektheitsbedingung:
Empfang von *Inc* früher als alle kausal abhängigen *Dec*

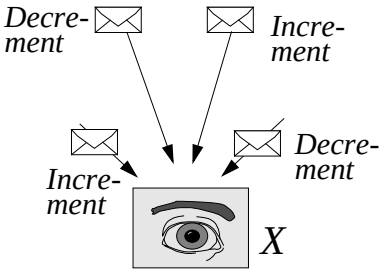
Wie dies garantieren?

1) Synchrone Kommunikation (--> "atomare" Operation)
2) Acknowledge von Inc-Nachrichten (+ warten)
3) "Causal Order" realisieren...

Objekt hat eine *kausaltreue Sicht*
der Ereignisse (die es betreffen)

--> (indirekte) Überholungen vermeiden

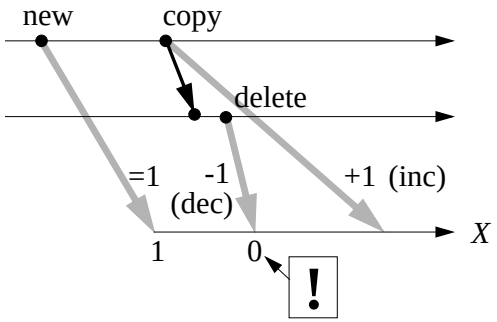
Garbage-Identifikation als konsistentes Beobachtungsproblem



Objekt X "beobachtet" die copy- und delete-Operationen, die es selbst betreffen

- die increment- und decrement-Nachrichten dienen der Beobachtung
- damit feststellen, ob *alle* copy / new durch delete kompensiert wurden

- Diese Beobachtung sollte kausaltreu sein!
  - zumindest darf ein dec nicht vor "seinem" inc empfangen werden

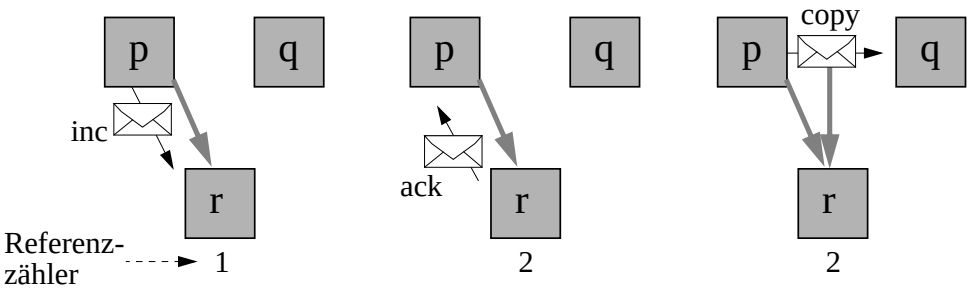


- *Jedes* Objekt beobachtet jedes andere
  - "Causal Order" (Globalisierung von FIFO) bzgl. Kommunikation gefordert

- Wie realisiert man kausaltreue Beobachtungen?
  - in diesem konkreten Fall diverse Möglichkeiten (--> viele Algorithmen!), z.B. copy solange verzögern, bis Bestätigung für inc-Nachricht eingetroffen
  - oder: kausaltreue Beobachter / "Causal Order"-Kommunikation allgemein implementieren?

Verteiltes Reference-Counting: Lösungen

--> Jede increment-Nachricht bestätigen (warten auf ack):



- Korrektheitskriterium erfüllt:

  inc wird vor einem kausal abhängigen dec empfangen

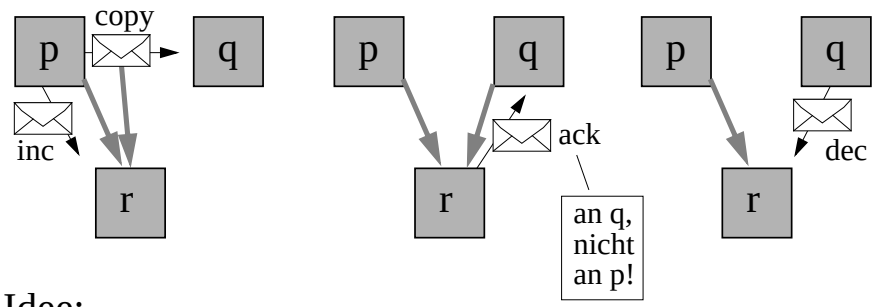
  (d.h. Objekt r erfährt über die Existenz einer Kopie einer r-Referenz, bevor es vom Löschen dieser oder einer "solchen" Referenz erfährt)

- Nachteile der Methode:

  - copy wird verzögert
  - zusätzliche Nachricht (ack)

  --> insgesamt 3 Nachrichten pro copy-Operation

Variante von Lermen und Maurer



Idee:

Senden einer dec-Nachricht (bei Löschen der Referenz) erst dann, wenn das Objekt bereits eine zugehöriges ack (bzgl. inc) vom Zielobjekt der Referenz empfangen hat

==> Korrektheitskriterium erfüllt

- Beachte: Referenz kann stets gelöscht werden, nur das Senden von dec muss verzögert werden

---

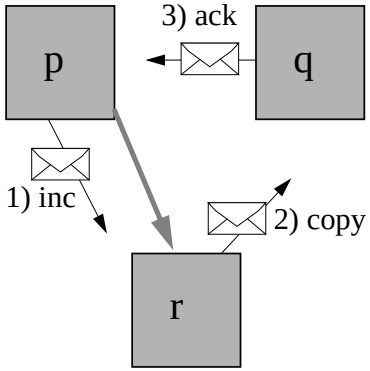
- Implementierungsskizze:
  - Zählen von empfangenen copy und ack-Nachrichten
  - dec-Nachricht erst senden, wenn Zähler ACK und COPY übereinstimmen
    - dann die Zähler ACK und COPY beide dekrementieren
    - "individuelle" Zuordnung von copy zu ack nicht notwendig!
    - Abschwächung $|ACK|>0 \wedge |COPY|>0$ möglich? Konsequenzen?

- *Vorteil*: kein Verzögern von Basisaktionen
- *Nachteil*: Ack-Nachricht und FIFO-Kanäle notwendig

wieso?

Varianten von Rudalics

1) 3-Nachrichten-Protokoll ("zyklisch"):



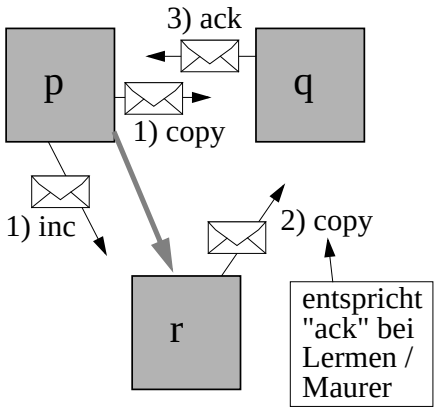
Idee: q bekommt Referenz auf r von r selbst; nachdem r seinen Zähler inkrementiert hat (veranlasst durch inc).

Bedingung: p darf seine r-Referenz erst löschen, wenn alle erwarteten ack-Nachrichten eingetroffen sind.

Frage: Wäre es auch möglich, dass das ack an p von r (statt q) gesendet wird?

- *Vorteil*: kein FIFO notwendig (falls FIFO garantiert ist: ack-Nachricht einsparen ==> nur zwei Nachrichten pro copy!)
- *Nachteil*: Kopieren dauert länger (2 Nachrichten)

2) 4-Nachrichten-Protokoll:



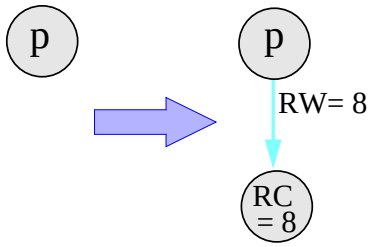
- Idee: ack erst senden, wenn *beide* copy-Nachrichten empfangen wurden
- q installiert die r-Referenz bei Empfang der *ersten* copy-Nachricht
- Unter welchen Bedingungen dürfen p bzw. q dec-Nachrichten senden?

- Vorteil: kein FIFO notwendig; keine Verzögerung
- Nachteil: 4 Nachrichten (aber nur 3 "sequentiell")

Die Referenzgewichts-Methode

(WRC: "Weighted Reference Counting")

Neues Objekt generieren ("new"):

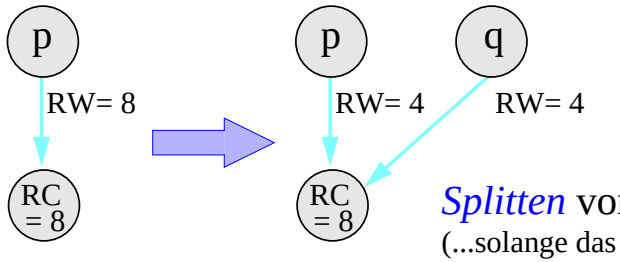


Reference *Weight*

Reference *Count*

(obwohl dadurch die Referenzen nicht tatsächlich "gezählt" werden)

Referenz kopieren ("copy "):



Splitten von RW

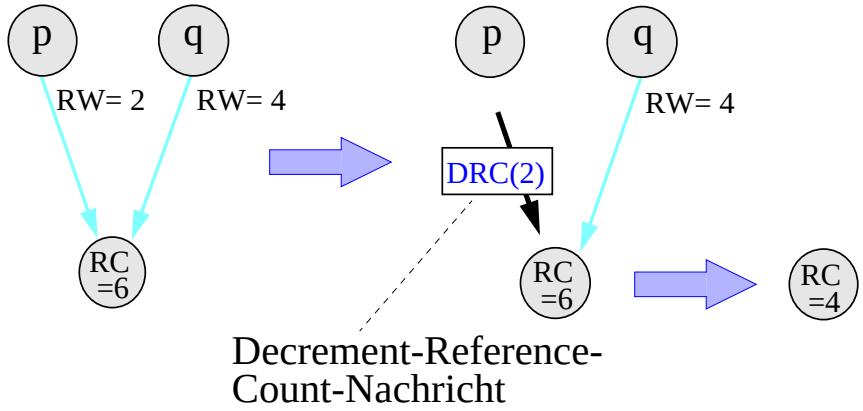
(...solange das geht!)

ansonsten (hier nicht genauer behandelt):

1) "Recharge"
2) Indirektionsobjekt

- Beachte: Es wird keine Increment-Nachricht benötigt!

Referenz löschen ("delete"):



Invariante: RC = ∑ RW + ∑ DRC

RC = 0 --> Objekt ist Garbage

--> alle Referenzen dieses Objektes auf andere Objekte löschen (DRC-Nachrichten senden)

- Logarithmische Kompression (2er-Potenzen!) von RW
  - mit nur 2 Bit pro Zeiger lassen sich so RW bis max. 8 darstellen
  - statt 8 kann ggf. auch ein (etwas?) grösserer Maximalwert gewählt werden
  - RC so nicht komprimierbar ==> int-Variable mit "vielen" Bits pro Objekt
- Keine Verzögerung bei copy / delete und bei copy keine zusätzlichen Nachrichten!
  - RW = 1 sollte ein eher seltenes Ereignis sein (--> Zusatzaufwand)
- Analogie zur *Kredit-Methode* bei vert. Terminierung!

Kredit-Methode und WRC

Terminierungserkennung mit der Kreditmethode		WRC-Garbage-Collection
- Senden einer Nachricht (Splitten des Kreditwertes)	⟷	- Kopieren einer Referenz (Splitten des RW)
- Passiv werden (Kredit an Urprozess zurückgeben)	⟷	- R-Referenz löschen (Decrement-Nachricht an R)
Invariante: ∑ Kredit =1	⟷	*Invariante*: RC = ∑ RW + ∑ DRC
- Gesamtkredit beim Urprozess = 2^0=1	⟷	- RC = 0 bei R
Terminierung	⟷	*R ist Garbage*

Also: Kreditmethode entspricht WRC-Garbage-Collection!

Garbage-Collection und Terminierung

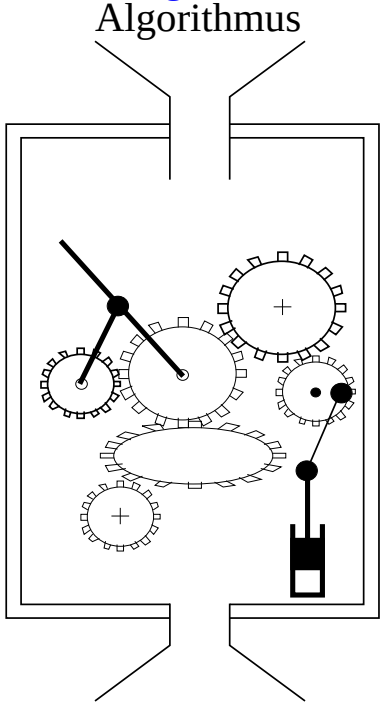
Theorem:

nicht notwendigerweise verteilte

Jeder *Garbage-Collection*-Algorithmus kann automatisch in einen Algorithmus zur Feststellung der *verteilten Terminierung* transformiert werden

Bemerkung: Für beide Probleme wurden viele nicht-triviale (und auch manche falsche!) Lösungen publiziert

Ein Garbage-Collection-Algorithmus

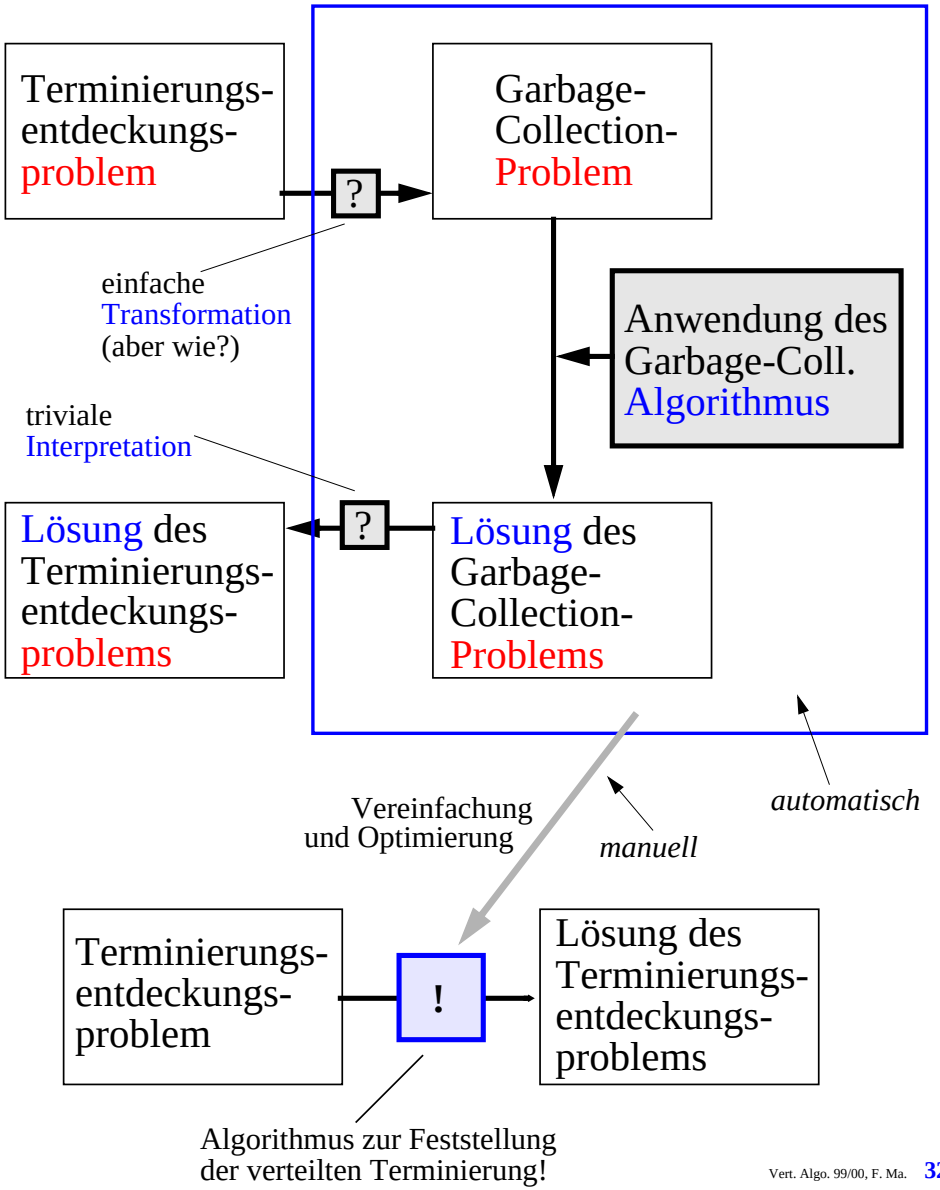


mechanische Transformation

Ein verteilter Terminierungsentdeckungs-Algorithmus

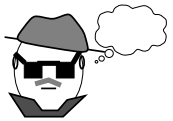
Problemtransformation

- Es wird das *Problem*, nicht der *Algortihmus* transformiert!

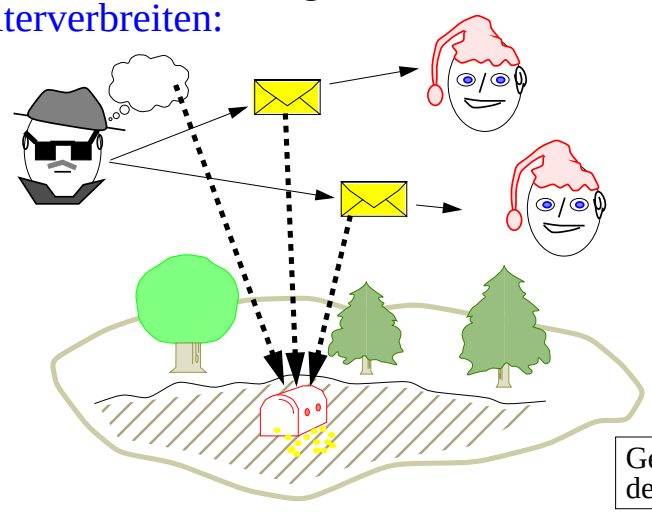


Vom Ende einer Geheimniskrämerei

Die vier goldenen Regeln der Geheimniskrämerei:

1) Es gibt Geheim-nisträger  und unein-geweihte Personen 

2) Nur ein Geheimnisträger kann das Geheimnis weiterverbreiten:



3) Wer das Geheimnis erfährt, wird zum Geheimnisträger

4) Ein Geheimnisträger kann das Geheimnis (endgültig) vergessen

---

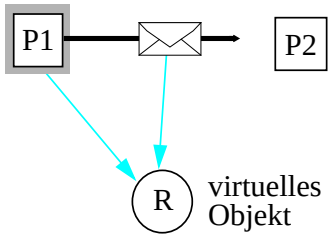
es gibt keine Geheimnisträger und keine Nachrichten mit dem Geheimnis

Geheimniskrämerei terminiert <==>
Schatz nicht mehr zugreifbar <==> *Schatz ist "Garbage"*

--> "watchdog" beim Schatz meldet Terminierung...

Die Transformation

- Jeder Prozess wird in ein Wurzelobjekt transformiert
- Ein zusätzliches virtuelles Objekt R wird hinzugefügt



1) Prozess P aktiv <==> P besitzt Referenz auf R

2) Jede Nachricht enthält Referenz auf R

- Die beiden Regeln lassen sich ("induktiv") erfüllen:
  - ein (aktives) Objekt / Prozess sendet eine Kopie seiner R-Referenz mit jeder Nachricht
  - ein reaktivierter Prozess erhält eine R-Referenz
  - ein Prozess, der passiv wird, löscht seine R-Referenz

R Garbage <==> Es gibt keine Referenz auf R <==> Alle Prozesse passiv und keine Nachricht unterwegs <==> Verteilte Berechnung terminiert

- Also: verwende *irgendeinen* GC-Algorithmus
  --> interpretiere Berechnung als GC-Problem
  --> melde Terminierung, wenn R als Garbage erkannt

- Übung: man mache dies für konkrete GC-Algorithmen aus der Literatur
- beachte: es entstehen keine Zyklen von Referenzen, daher sind auch Referenzzählverfahren anwendbar!