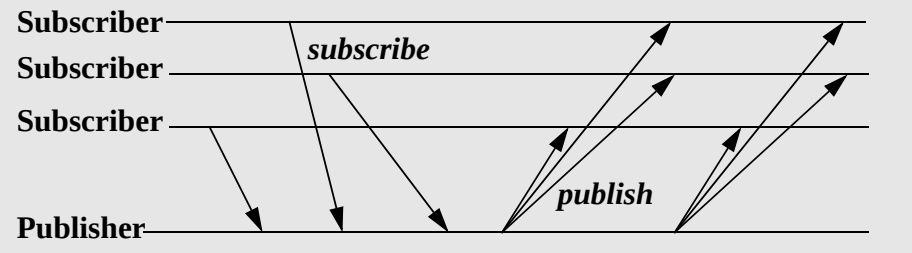
Push-Paradigma; Publish & Subscribe

- Im Unterschied zum klassischen “Pull-” bzw. “Request / Reply”-Paradigma
  - wo Clients die gewünschte Information aktiv anfordern müssen
  - sie aber nicht wissen, ob bzw. wann sich eine Information geändert hat
  - dadurch periodische Nachfragen beim Server notwendig sind (“polling”)

- Subscriber (= Client) meldet sich für den Empfang der gewünschten Information an
  - z.B. “Abonnement” eines Informationskanals (“channel”)
  - u.U. auch dynamische, virtuelle Kanäle (→ “subject-based addressing”)



- Subscriber erhält automatisch (aktualisierte) Information, sobald diese zur Verfügung steht
  - “callback” des Subscribers (= Client) durch den Publisher (= Server)
  - push: “event driven” ↔ pull: “demand driven”

- “Publish” entspricht (logischem) Multicast
  - “subscribe” entspricht dann einem “join” einer Multicast-Gruppe
  - Zeitaktualität, Stärke der Multicast-Semantik und Grad an Fehlertoleranz wird oft unscharf als “Quality of Service” bezeichnet

Tupelräume

- Gemeinsam genutzter (“virtuell globaler”) Speicher

- Blackboard- oder Marktplatz-Modell
  - Daten können von beliebigen Teilnehmern eingefügt, gelesen und entfernt werden
  - relativ starke Entkoppelung der Teilnehmer

- Tupel = geordnete Menge typisierter Datenwerte

- Entworfen 1985 von D. Gelernter (für die Sprache Linda)

- Operationen:
  - out (t): Einfügen eines Tupels t in den Tupelraum
  - in (t): Lesen und Löschen von t aus dem Tupelraum
  - read (t): Lesen von t im Tupelraum

- Inhaltsadressiert (“Assoziativspeicher”)
  - Vorgabe eines Zugriffsmusters (bzw. “Suchmaske”) beim Lesen, damit Ermittlung der restlichen Datenwerte eines Tupels (“wild cards”)
  - Beispiel: int i,j; in(“Buchung”, ?i, ?j) liefert *ein* “passendes” Tupel
  - analog zu einigen relationalen Datenbankabfragesprachen

- Synchrone und asynchrone Leseoperationen
  - ‘in’ und ‘read’ blockieren, bis ein passendes Tupel vorhanden ist
  - ‘inp’ und ‘readp’ blockieren nicht, sondern liefern als Prädikat (Daten vorhanden?) ‘wahr’ oder ‘falsch’ zurück

Tupelräume (2)

- Mit Tupelräumen sind natürlich die üblichen Kommunikationsmuster realisierbar, z.B. Client-Server:

```
/* Client */
...
out("Anfrage" client_Id, Parameterliste);
in("Antwort", client_Id, ?Ergebnisliste);
...
/* Server*/
...
while (true)
{  in("Anfrage", ?client_Id, ?Parameterliste);
   ...
   out("Antwort", client_Id, Ergebnisliste);
}
```

Beachte: Zuordnung des "richtigen" Clients über die client_Id

- Kanonische Erweiterungen des Modells
  - *Persistenz* (Tupel bleiben nach Programmende erhalten, z.B. in DB)
  - *Transaktionseigenschaft* (wichtig, wenn mehrere Prozesse parallel auf den Tupelraum bzw. gleiche Tupel zugreifen)
- Problem: effiziente, skalierbare Implementierung?
  - *zentrale Lösung*: Engpass
  - *replizierter Tupelraum* (jeder Rechner hat eine vollständige Kopie des Tupelraums; schnelle Zugriffe, jedoch hoher Synchronisationsaufwand)
  - *aufgeteilter Tupelraum* (jeder Rechner hat einen Teil des Tupelraums; 'out'-Operationen können z.B. lokal ausgeführt werden, 'in' evtl. mit Broadcast)
- Kritik: globaler Speicher ist der strukturierten Programmierung und der Verifikation abträglich
  - unüberschaubare potentielle Seiteneffekte

JavaSpaces

- "Tupelraum" für Java
  - gespeichert werden Objekte → neben Daten auch "Verhalten"
  - Tupel entspricht Gruppen von Objekten
- Teil der Jini-Infrastruktur für verteilte Java-Anwendungen
  - Kommunikation zwischen entfernten Objekten
  - Transport von Programmcode vom Sender zum Empfänger
  - gemeinsame Nutzung von Objekten
- Operationen
  - *write*: mehrfache Anwendung erzeugt verschiedene Kopien
  - *read*
  - *readifexists*: blockiert (im Gegensatz zu read) nicht; liefert u.U. 'null'
  - *takeifexists*
  - *notify*: Benachrichtigung (mittels eines Ereignisses), wenn ein passendes Objekt in den JavaSpace geschrieben wird
- Nutzen (neben Kommunikation)
  - atomarer Zugriff auf Objektgruppen
  - zuverlässiger verteilter Speicher
  - persistente Datenhaltung für Objekte

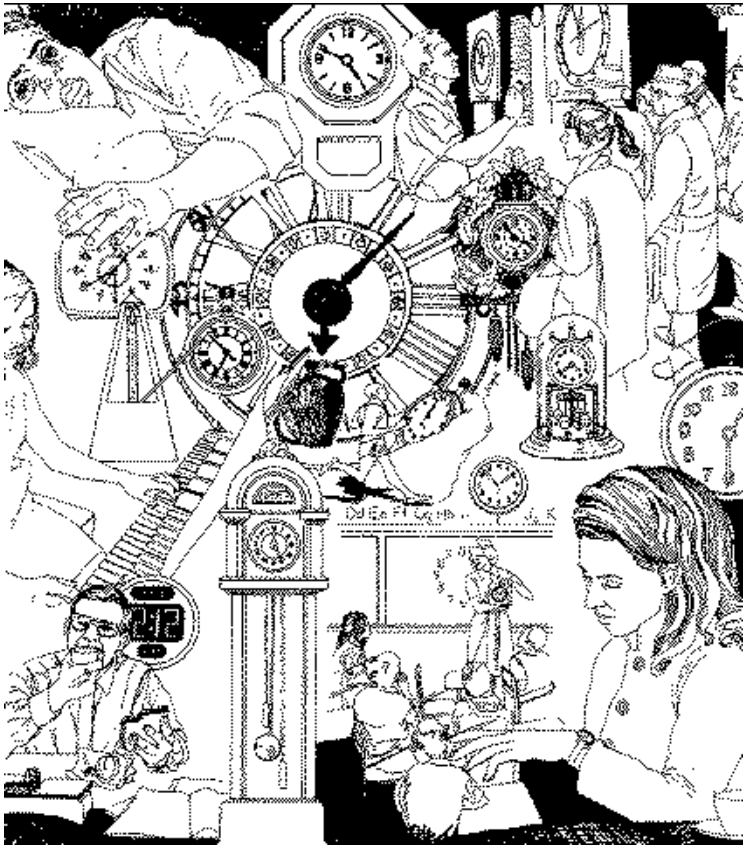
  } aber keine Festlegung, ob eine Implementierung fehlertolerant ist und einen Crash überlebt
- *Implementierung* könnte z.B. auf einer (relationalen / objektorientierten) Datenbank beruhen
- *Semantik*: Reihenfolge der Wirkung von Operationen verschiedener Threads ist nicht festgelegt
  - selbst wenn ein *write* vor einem *read* beendet wird, muss *read* nicht notwendigerweise das lesen, was *write* geschrieben hat

Logische Zeit und wechselseitiger Ausschluss

Zeit?

Ich halte ja eine Uhr für überflüssig.
Sehen Sie, ich wohne ja ganz nah beim Rathaus. Und jeden Morgen, wenn ich ins Geschäft gehe, da schau ich auf die Rathausuhr hinauf, wieviel Uhr es ist, und da merke ich's mir gleich für den ganzen Tag und nütze meine Uhr nicht so ab.

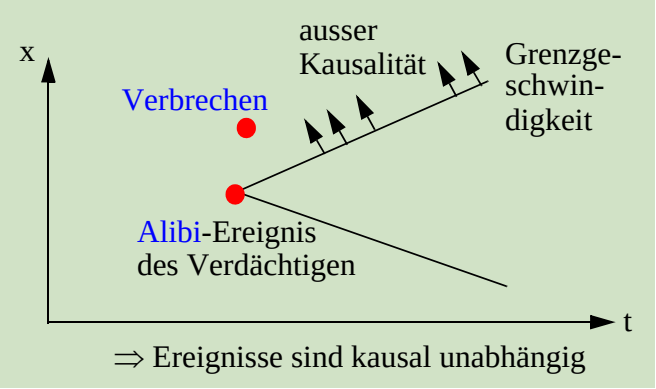
Karl Valentin

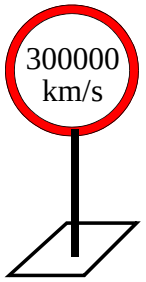


Kommt Zeit, kommt Rat

1. Volkszählung: Stichzeitpunkt in der Zukunft
   - liefert eine gleichzeitige, daher kausaltreue "Beobachtung"

2. Kausalitätsbeziehung zwischen Ereignissen ("Alibi-Prinzip")
   - wurde Y später als X geboren, dann kann Y unmöglich Vater von X sein
     → Testen verteilter Systeme: Fehlersuche / -ursache





"Speed limit
of causality"
(P. Langevin)

3. Fairer wechselseitiger Ausschluss
   - bedient wird, wer am längsten wartet

4. Viele weitere nützliche Anwendungen von "Zeit" in unserer "verteilten realen Welt"
   - z.B. kausaltreue Beobachtung durch "Zeitstempel" der Ereignisse

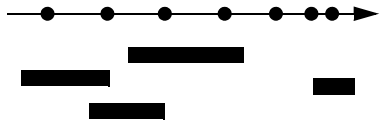
Eigenschaften der "Realzeit"

Formale Struktur eines Zeitpunktmodells:

- transitiv
- irreflexiv } → lineare Ordnung ("später")
- linear
- unbeschränkt ("Zeit ist ewig": Kein Anfang oder Ende)
- dicht (es gibt immer einen Zeitpunkt dazwischen)
- kontinuierlich
- metrisch
- vergeht "von selbst" → jeder Zeitpunkt wird schliesslich erreicht

---

Ist das Zeitpunktmodell adäquat? Oder sind Zeitintervalle besser?



- Wann tritt das Ereignis (?) "Sonne wird rot" am Abend ein?

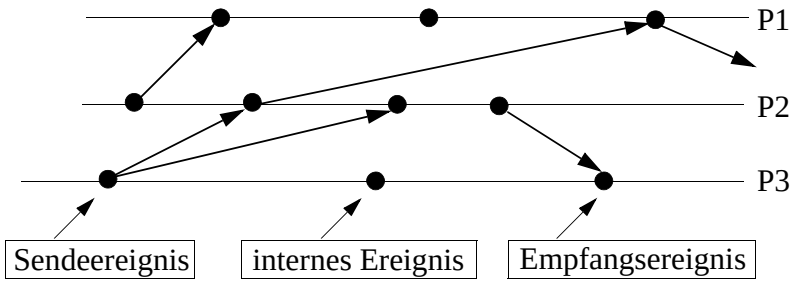
Welche Eigenschaften benötigen wir wirklich?

- dazu vorher klären: was wollen wir mit "Zeit" anfangen?

→ "billigeren" Ersatz für fehlende globale Realzeit!
(sind die rellen / rationalen / ganzen Zahlen gute Modelle?)

- wann genügt "logische" (statt "echter") Zeit? (Und was ist das genau??)

Raum-Zeitdiagramme und die Kausalrelation



- interessant: von links nach rechts verlaufende “Kausalitätspfade”

- Definiere eine *Kausalrelation* ‘<‘ auf der Menge E aller Ereignisse:

Es sei **x** < **y** genau dann, wenn:

1) **x** und **y** auf dem gleichen Prozess stattfinden und **x** vor **y** kommt, *oder*

2) **x** ist ein Sendeereignis und **y** ist das korrespondierende Empfangsereignis, *oder*

3) $\exists$ **z** mit **x** < **z** $\wedge$ **z** < **y**

- Relation wird oft als “*happened before*” bezeichnet
  - eingeführt von L. Lamport (1978)
  - aber Vorsicht: damit ist nicht direkt eine "zeitliche" Aussage getroffen!

Logische Zeitstempel von Ereignissen

- Zweck: Ereignissen eine Zeit geben (“dazwischen” egal)
- Gesucht: Abbildung C: E $\rightarrow$ **N**

Clock — natürliche Zahlen

- Für e $\in$ E heisst C(e) *Zeitstempel* von e
- C(e) bzw. e *früher* als C(e’) bzw. e’, wenn C(e) < C(e’)

- Sinnvolle Forderung:

Kausalrelation

Uhrenbedingung: e < e’ $\Rightarrow$ C(e) < C(e’)

Ordnungshomomorphismus — Zeitrelation “früher”

Interpretation (“Zeit ist kausaltreu”):

Wenn ein Ereignis e ein anderes Ereignis e’ beeinflussen kann, dann muss e einen kleineren Zeitstempel als e’ haben

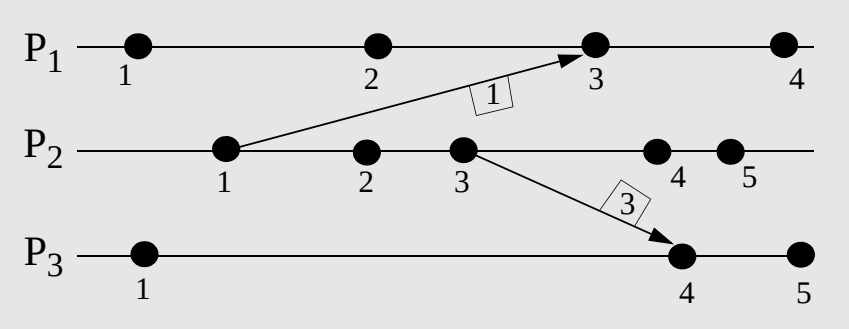
Logische Uhren von Lamport

Communications ACM 21(7), 558-565, 1978:
Time, Clocks, and the Ordering of Events in a Distributed System

$C: (E,<) \rightarrow (\mathbf{N},<)$ Zuordnung von Zeitstempeln

Kausal-relation

$e < e' \Rightarrow C(e) < C(e')$ Uhrenbedingung



Protokoll zur Implementierung der Uhrenbedingung:

- Lokale Uhr (= "Zähler") *tickt* bei *jedem* Ereignis
- Sendeereignis: Uhrwert mitsenden (*Zeitstempel*)
- Empfangsereignis: *max*(lokale Uhr, Zeitstempel)

zuerst, erst danach "ticken"!

Behauptung:

Protokoll respektiert Uhrenbedingung

Beweis: Kausalitätspfade sind monoton...

Lamport-Zeit: Nicht-Injektivität

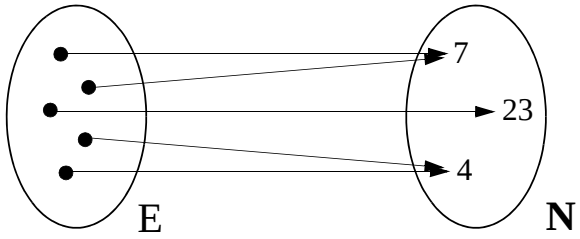


Abbildung ist nicht injektiv

- Wäre wichtig z.B. für: "Wer die kleinste Zeit hat, gewinnt"

- Lösung:

  Lexikographische Ordnung (C(e),i), wobei i die Prozessnummer bezeichnet, auf dem e stattfindet

  Ist injektiv, da alle lokalen Ereignisse (auf Prozess i) verschiedene Zeitstempel C(e) haben

  - *lin*. Ordnung $(a,b) < (a',b') \Leftrightarrow a<a' \vee a=a' \wedge b<b'$

  $\rightarrow$ alle Ereignisse haben *verschiedene* Zeitstempel

  $\rightarrow$ Kausalitätserhaltende Abb. $(E,<) \rightarrow (\mathbf{N}\times\mathbf{N}, <)$

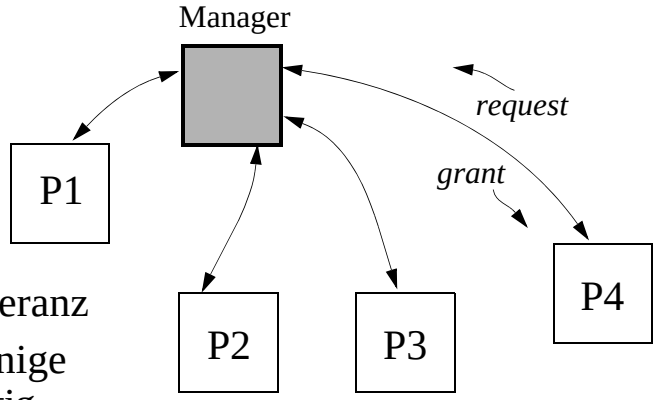
Jede (nicht-leere) Menge von Ereignissen hat so ein eindeutig "frühestes"!

Wechselseitiger Ausschluss

- “Streit” um exklusive Betriebsmittel
  - z.B. konkrete Ressourcen wie gemeinsamer Datenbus
  - oder abstrakte Ressourcen wie z.B. “Termin” in einem (verteilten) Terminkalendersystem
  - “kritischer Abschnitt” in einem (nebenläufigen) Programm
- Lösungen für Einprozessormaschinen, shared memory etc. nutzen typw. Semaphore oder ähnliche Mechanismen
  - ⇒ Betriebssystem- bzw. Concurrency-Theorie
  - ⇒ interessiert uns hier (bei verteilten Systemen) aber nicht

---

- Nachrichtenbasierte Lösung, die auch uninteressant ist, da stark asymmetrisch (“zentralisiert”): Manager-Prozess, der die Ressource (in fairer Weise) zuordnet:

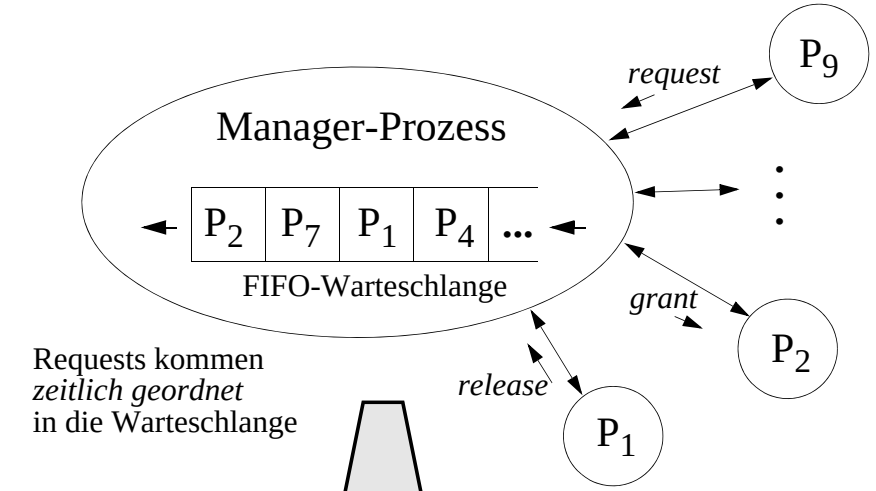


→ Engpass

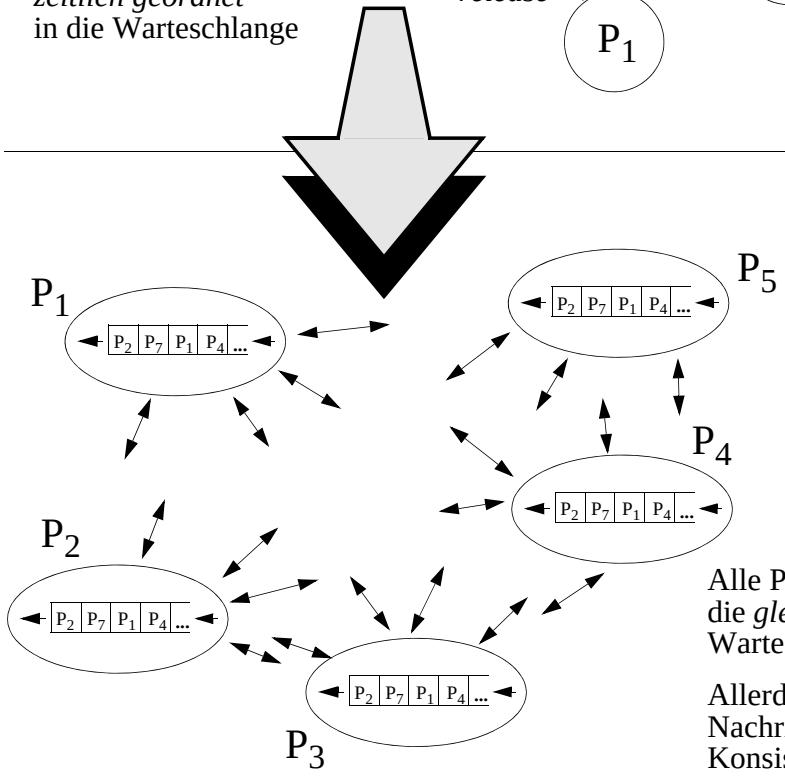
→ keine Fehlertoleranz

→ aber relativ wenige Nachrichten nötig

Replizierte Warteschlange?



Requests kommen *zeitlich geordnet* in die Warteschlange



- Lässt sich mit “logischer Zeit” realisieren, siehe nächste Seite!

Alle Prozesse sollen die *gleiche Sicht* der Warteschlange haben

Allerdings: viele Nachrichten, um Konsistenz zu gewähren; diese müssen ausserdem “geordnet” (=?) ankommen

Anwendung logischer Zeit für den wechselseitigen Ausschluss

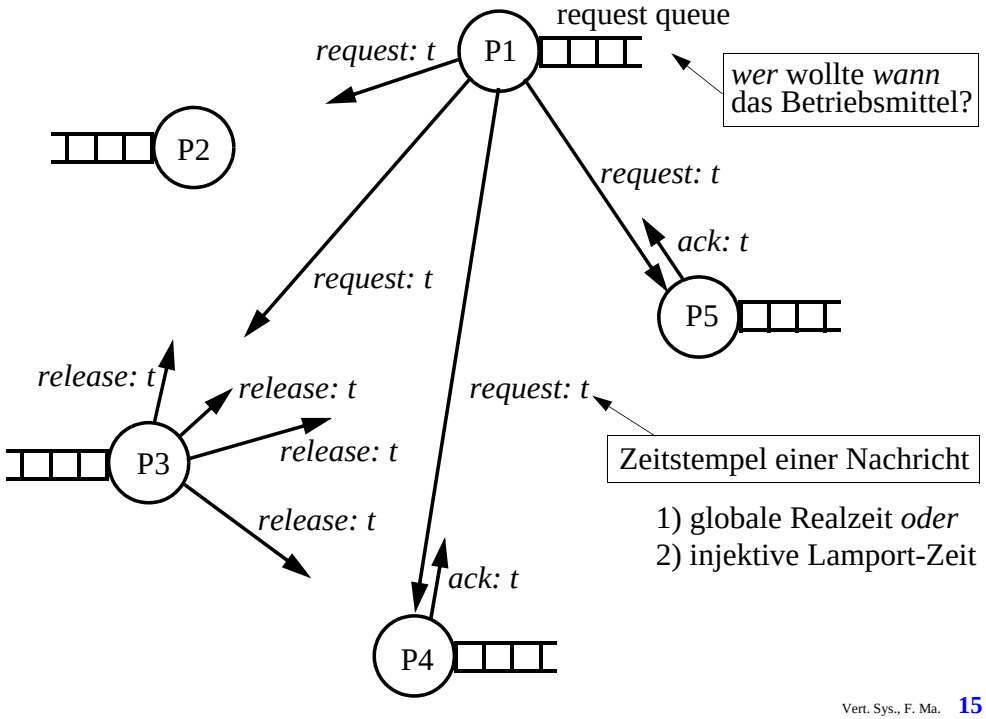
- Hier: Feste Anzahl von Prozessen; ein einziges exklusives Betriebsmittel
- Synchronisierung mit request- / release-Nachrichten

"request" / "release": → vor Betreten / bei Verlassen des *kritischen Abschnittes*

- Fairnessforderung: Jeder request wird "schliesslich" erfüllt

---

Idee: Replikation einer "virtuell globalen" request queue:



1) globale Realzeit *oder*
2) injektive Lamport-Zeit

Der Algorithmus (Lamport 1978):

- Voraussetzung: FIFO-Kommunikationskanäle
- *Alle* Nachrichten tragen (eindeutige!) Zeitstempel

z.B. logische Lamport-Zeit

- Request- und release-Nachrichten an *alle* senden

broadcast

1) Bei "request" des Betriebsmittels: Request mit Zeitstempel und Absender an alle versenden und in eigene queue einfügen.

2) Bei Empfang einer request-Nachricht: Request in eigene queue einfügen, ack versenden.

wieso notwendig?

3) Bei "release" des Betriebsmittels: Aus eigener queue entfernen, release-Nachricht an alle versenden.

4) Bei Empfang einer release-Nachricht: Zugehörigen request aus eigener queue entfernen.

5) Ein Prozess darf das *Betriebsmittel benutzen, wenn:*
   - eigener request ist frühester in seiner queue und
   - hat bereits von jedem anderen Prozess (irgendeine) spätere Nachricht bekommen.

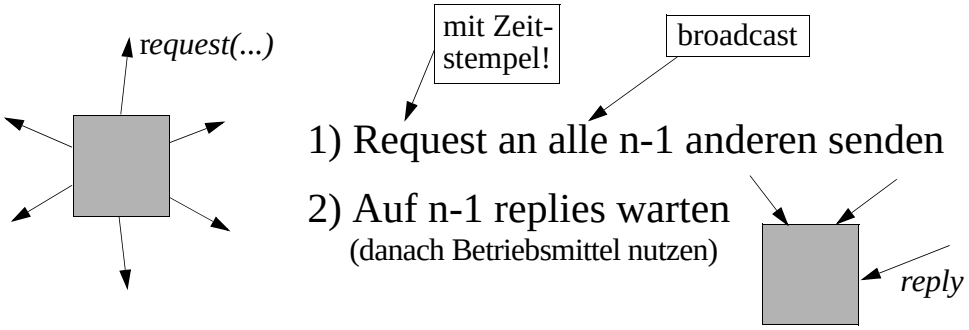
- Frühester request ist global eindeutig.
  ⇒ bei 5): sicher, dass kein früherer request mehr kommt (wieso?)
- 3(n-1) Nachrichten pro "request" (n = Zahl der Prozesse)

Denkübungen:
- wo geht die Uhrenbedingung / Kausaltreue der Lamport-Zeit ein?
- sind FIFO-Kanäle wirklich notwendig? (Szenario hierfür?)
- bei Broadcast: welche Semantik? (FIFO, kausal,...?)
- was könnte man bei / gegen Nachrichtenverlust tun? (→ Fehlertoleranz)

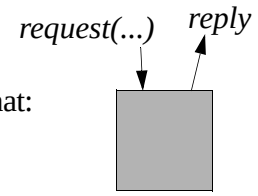
Ein anderer verteilter Algorithmus für den wechselseitigen Ausschluss

(Ricart / Agrawala, 1981)

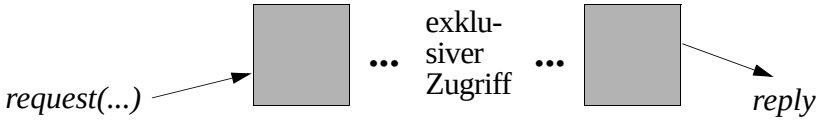
- 2(n-1) Nachrichten statt 3(n-1) wie bei obigem Verfahren
  (*reply-Nachricht* übernimmt Rolle von release und ack)



- Bei Eintreffen einer request-Nachricht:



  - reply sofort schicken, wenn nicht selbst beworben oder der Sender "ältere Rechte" (bzgl. logischer Zeit) hat:
  - ansonsten reply erst später schicken, nach Erfüllen des eigenen requests ("verzögern"):



- Älteste Bewerbung setzt sich durch (injektive Lamport-Zeit!)

Denkübungen:

- Argumente für die Korrektheit? (Exklusivität, Deadlockfreiheit)
- wie oft muss ein Prozess maximal "nachgeben"? (→ Fairness)
- sind FIFO-Kanäle notwendig?
- geht wechsels. Ausschluss vielleicht mit noch weniger Nachrichten?

Namens-verwaltung

Namen und Adressen

- Namen dienen der (eindeutigen) *Bezeichnung* und der *Identifikation* von Objekten, Diensten etc.

- Adressen ermöglichen die *direkte Lokalisierung* und damit den direkten Zugriff auf Objekte
  - sind gewissermassen “physische” Namen

- Adressen von Objekten können z.B. sein:
  - Speicherplatzadressen
  - Internetadressen (IP-Nummern)
  - Netzadressen
  - Port-Nummer bei TCP
  - ...

- Adressen sind innerhalb eines Kontextes (“Adressraum”) eindeutig

- Adresse eines Objektes ist u.U. *zeitabhängig*
  - mobile Objekte
  - “relocatable”
- *Dagegen*: Name eines Objektes ändert sich i.Allg. nicht

- Entkoppelung von Namen und Adressen unterstützt die *Ortstransparenz*

- Zuordnung Name → Adresse nötig
  - vgl. persönliches Adressbuch
  - “Binden” eines Namens an eine Adresse

Binden

- Binden = Zuordnung Name → Adresse
  - konzeptuell daher auch: Name → Objekt

- *Binden bei Programmiersprachen:*
  - Beim Übersetzen / Assemblieren
    → “relative” Adresse
  - Durch Binder (“linker”) oder Lader
    → “absolute” Adresse
  - Evtl. Indirektion durch das Laufzeitsystem
    - z.B. bei Polymorphie objektorientierter Systeme

- *Binden in verteilten Systemen*
  - Dienste entstehen dynamisch, werden evtl. verlagert
    - haben evtl. unterschiedliche Lebenszyklen und -dauer
  - Binden muss daher ebenfalls *dynamisch* erfolgen
    - “zur Laufzeit” bzw. beim Objektzugriff

  → Name Service; Lookup Service

Namensverwaltung ("Name Service")

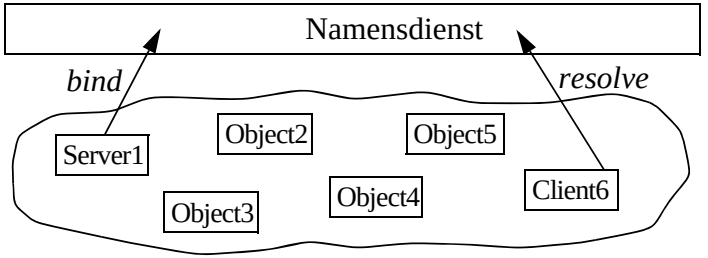
- Verwaltung der Zuordnung Name → Adresse
  - Eintragen: "bind (Name, Adresse)" sowie Ändern, Löschen etc.
  - Eindeutigkeit von Namen garantieren
  - Zusätzlich u.U. Verwaltung von Attributen der bezeichneten Objekte

- Auskünfte ("Finden" von Ressourcen, "lookup")
  - z.B. Adresse zu einem Namen ("resolve": Namensauflösung)
  - z.B. alle Dienste mit gewissen Attributen (etwa: alle Duplex-Drucker) "yellow pages" ↔ "white pages"

- Evtl. Unterstützung von Schutz- und Sicherheitsaspekten
  - Capability-Listen, Schutzbits, Autorisierungen,...
  - Dienst selbst soll hochverfügbar und sicher (z.B. bzgl. Authentizität) sein

- Evtl. Generierung neuer eindeutiger Namen
  - UUID (Universal Unique Identifier)
  - innerhalb eines Kontextes (z.B. mit Zeitstempel oder lfd. Nummer)
  - bzw. global eindeutig (z.B. eindeutigen Kontextnamen als Präfix vor eindeutiger Gerätenummer; evtl. auch lange Zufallsbitfolge)

Verteilte Namensverwaltung



Logisch ein einziger (zentraler) Dienst; *tatsächlich* verteilt realisiert

- Jeder Kontext (Teilnamensraum) wird (logisch) von einem dedizierten *Nameserver* verwaltet
  - evtl. ist ein Nameserver aber für mehrere Kontexte zuständig
  - evtl. Aufteilung des Namensraums / Replikation des Nameservers → höhere Effizienz, Ausfallsicherheit

- Typisch: *kooperierende* einzelne Nameserver, die zusammen den gesamten Verwaltungsdienst realisieren
  - hierzu geeignete Architektur der Server vorsehen
  - Protokoll zwischen den Nameservern (für Fehlertoleranz, update der Replikate etc.)
  - Dienstschnittstelle wird typw. durch lokale Nameserver realisiert

bzw. "user agent"

- Typischerweise *hierarchische Namensräume*
  - entsprechend strukturierte Namen und kanonische Aufteilung der Verwaltungsaufgaben
  - Zusammenfassung Namen gleichen Präfixes vereinfacht Verwaltung

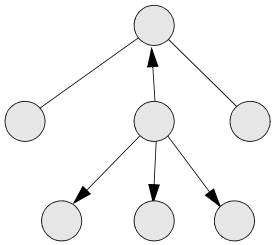
- *Annahmen*, die Realisierungen i.Allg. zugrundeliegen:
  - *lesende* Anfragen viel häufiger als schreibende ("Änderungen")
  - *lokale* Anfragen (bzgl. eigenem Kontext) dominieren
  - seltene, temporäre *Inkonsistenzen* können toleriert werden

ermöglicht effizientere Realisierungen (z.B. Caching, einfache Protokolle,...)

Namensinterpretation in verteilten Systemen

Hierarchische Architektur:

- Ein Nameserver kennt den Nameserver der *nächst höheren Stufe*



- Ein Nameserver kennt alle Nameserver der *untergeordneten* Kontexte (sowie deren Namensbereiche)
- Hierarchiestufen sind i.Allg. klein (typw. 3 oder 4)
- *Blätter* verwalten die eigentlichen Objektadressen und bilden die Schnittstelle für die Clients
- Nicht interpretierbare Namen werden an die nächst höhere Stufe weitergeleitet

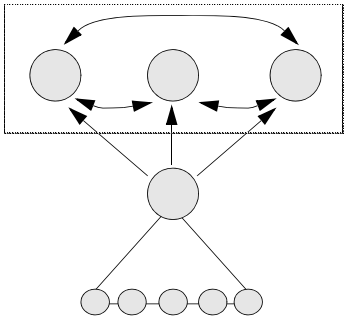
---

Verwendung von Broadcast

- falls zuständiger Nameserver unbekannt
  ("wer ist für XYZ zuständig?" oder: "wer ist hier der Nameserver?")
- ist aufwändig, falls nicht systemseitig effizient unterstützt (wie z.B. bei LAN oder Funknetzen)
- ist nur in begrenzten Kontexten anwendbar

Repliklation von Nameservern

- Zweck: Erhöhung von Effizienz und Fehlertoleranz
- Vor allem auf höherer Hierarchieebene relevant
  - dort viele Anfragen
  - Ausfall würde grösseren Teilbereich betreffen

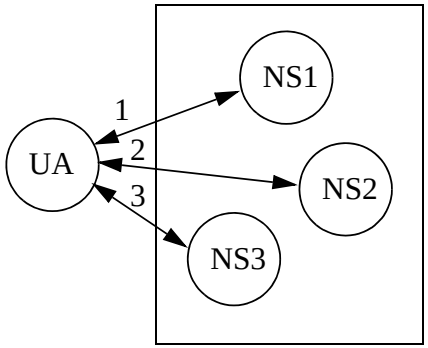


- Nameserver kennt mehrere übergeordnete Nameserver
- Broadcast an ganze Servergruppe oder Einzelnachricht an "nächsten" Server; anderen Server erst nach Ablauf eines Timeouts befragen

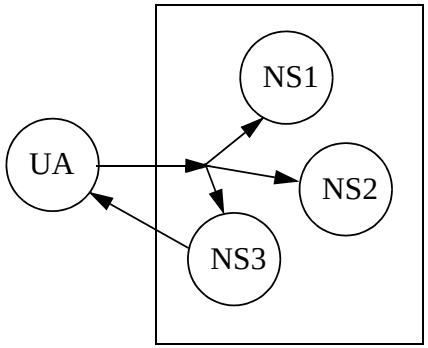
- Replizierte Server konsistent halten
  - evtl. nur von Zeit zu Zeit gegenseitig aktualisieren (falls veraltete Information tolerierbar)
  - Update auch dann sicherstellen, wenn einige Server zeitweise nicht erreichbar sind (periodisches Wiederholen von update-Nachrichten)
  - Einträge mit Zeitstempel versehen → jeweils neuester Eintrag dominiert (global synchronisierte Zeitbasis notwendig!)
- Symmetrische Server / Primärserver-Konzept:
  - *symmetrische Server:* jeder Server einer Gruppe kann updates initiieren
  - *Primärserver:* nur dieser nimmt updates entgegen
    - Primärserver aktualisiert gelegentlich "read only" Sekundärserver
    - Rolle des Primärservers muss im Fehlerfall von einem anderen Server der Gruppe übernommen werden

Strukturen zur Namensauflösung

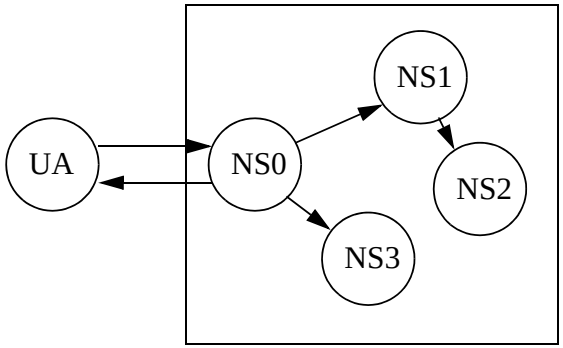
- User Agent (UA) auf Client-Seite
  - hinzugebundene Schnittstelle aus Bibliothek, oder
  - eigener lokaler Service-Prozess



Iterative Navigation: NS1 liefert Adresse eines anderen Nameservers zurück bzw. UA probiert einige (vermutlich) zuständige Nameserver nacheinander aus



Multicast-Navigation: Es antwortet derjenige, der den Namen auflösen kann (u.U. auch mehrere)



"Rekursive" Namensauflösung, wenn ein Nameserver den Dienst einer anderen Ebene in Anspruch nimmt

Serverkontrollierte Navigation: Der Namensdienst selbst (in Form des Serververbundes) kümmert sich um die Suche nach Zuständigkeit

Caching von Bindungsinformation

- Zweck: Leistungsverbesserung, insbesondere wenn nichtlokale Anfragen häufig sind

- Teile der Zuordnung "Name → Adresse" wird lokal im Cache gehalten
- vor Aufruf eines Nameservers überprüfen, ob Information im Cache
- Cache-Eintrag u.U. allerdings veraltet (evtl. Lebensdauer beschränken)
- Platz der Tabelle ist beschränkt → unwichtige / alte Einträge verdrängen
- neue Information wird als Seiteneffekt einer Anfrage im Cache eingetragen

(a) Abbildung Name → Adresse des *Objektes* oder:

(b) Abbildung Name → Adresse des *Nameservers* der tiefsten Hierarchiestufe, der für das Objekt zuständig ist

- Vorteil von (b): Inkonsistenz aufgrund veralteter Information kann vom Nameservice entdeckt werden
  - veralteter Cache-Eintrag kann transparent für den Client durch eine automatisch abgesetzte volle Anfrage ersetzt werden
- Bei (a) muss der *Client* selbst falsche Adressen *beim Zugriff* auf das Objekt erkennen und behandeln

---

- Caching kann bei den Clients stattfinden (z.B. im Web-Browser) und / oder bei den Nameservern

Internet Domain Name System (DNS)

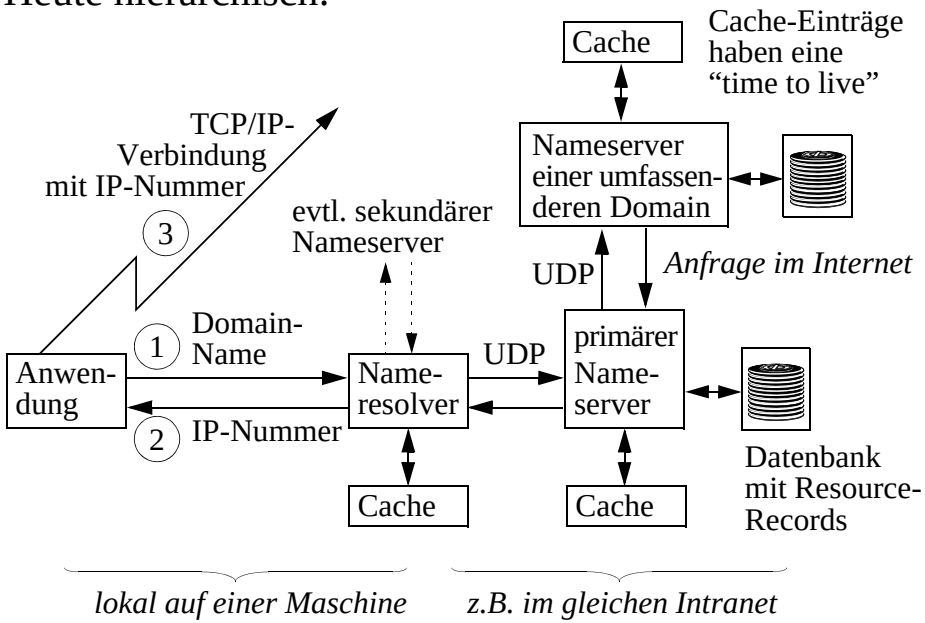
- Jeder Rechner im Internet hat eine IP-Adresse
  - bei IPv4: 32 Bit lang, typw. als 4 Dezimalzahlen geschrieben
  - Bsp.: 192.130.10.121 (= 11000000.10000010.00001010.01111001)
- Symbolische Namen sind für Menschen eher geeignet
  - z.B. Domain-Namen wie www.nanocomp.uni-cooltown.eu
  - gut zu merken; relativ unabhängig von spezifischer Maschine
  - muss *vor* Verwendung bei Internet-Diensten (WWW, E-Mail, ssh, ftp,...) in eine IP-Adresse umgesetzt werden
  - Umsetzung in IP-Adresse geschieht im Internet mit DNS
- Domains
  - hierarchischer Namensraum der symbolischen Namen im Internet
  - “Toplevel domains” com, de, fr, ch, edu,...
  - Domains (meist rekursiv) gegliedert in Subdomains, z.B.

eu
uni-cooltown.eu
informatik.uni-cooltown.eu
nano.informatik.uni-cooltown.eu
pc6.nano.informatik.uni-cooltown.eu

- Für einzelne (Sub)domains (bzw. einer Zusammenfassung einiger Subdomains) ist jeweils ein Domain-Nameserver zuständig
  - primärer Nameserver (www.switch.ch für die Domains .ch und .li)
  - optional zusätzlich einige weitere sekundäre Nameserver
  - oft sind Primärserver gleichzeitig Sekundärserver für andere Domains
  - Nameserver haben also nur eine Teilsicht!

Namensauflösung im Internet

- Historisch: Jeder Rechner hatte eine Datei hosts.txt, die jede Nacht von zentraler Stelle aus verteilt wurde
- Heute hierarchisch:



- Sicherheit und Verfügbarkeit sind wichtige Aspekte
  - Verlust von Nachrichten, Ausfall von Komponenten etc. tolerieren
  - absichtliche Verfälschung, denial of service etc. verhindern (→ DNSSEC)

Interaktive DNS-Anfrage

nslookup - query name servers interactively

nslookup is an interactive program to query Internet domain name servers. The user can contact servers to request information about a specific host, or print a list of hosts in the domain.

```
> pc20
Name:    pc20.nanocomp.inf.ethz.ch
Address:  129.132.33.79
Aliases:  ftp.nanocomp.inf.ethz.ch

> google.com
Name:    google.com
Addresses:  74.125.57.104,
74.125.59.104, 74.125.39.104

> google.com
Name:    google.com
Addresses:  74.125.59.104,
74.125.39.104, 74.125.57.104

> cs.uni-sb.de
Name:    cs.uni-sb.de
Addresses:  134.96.254.254, 134.96.252.31
```

Dies deutet auf einen "round robin"-Eintrag hin: Der Nameserver von google.com ändert alle paar Minuten die Reihenfolge der Einträge, die bei anderen Nameservern auch nur einige Minuten lang gespeichert bleiben dürfen. Da Anwendungen i.Allg. den ersten Eintrag nehmen, wird so eine Lastverteilung auf mehrere google-Server vorgenommen; stellt gleichzeitige eine rudimentäre Fehlertoleranz bereit.

Router an zwei Netzen

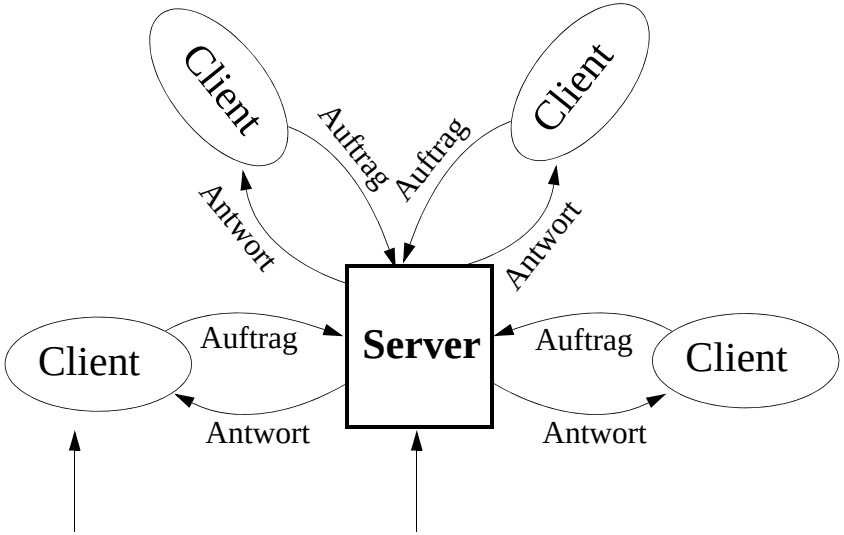
---

dig - DNS lookup utility

dig (domain information groper) is a flexible tool for interrogating DNS name servers. It performs DNS lookups and displays the answers that are returned from the name server(s) that were queried. Most DNS administrators use dig to troubleshoot DNS problems because of its flexibility, ease of use...

Client/Server-Modell

Das Client/Server-Modell

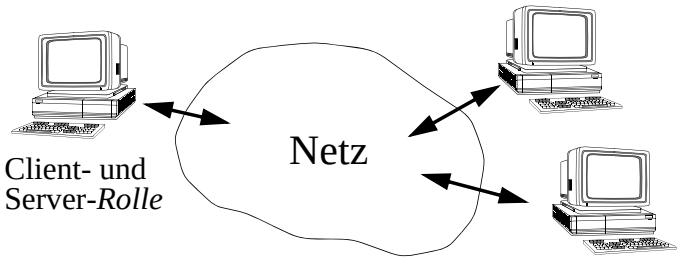


- Aufgabenteilung und asymmetrische Struktur
  - *Clients*: typischerweise Anwendungsprogramme und Benutzungsschnittstelle ("front end") für einen Nutzer
  - *Server*: zuständig für Dienstleistungen für viele Clients
- Populär wegen des eingängigen Modells
  - entspricht Geschäftsvorgängen in unserer Dienstleistungsgesellschaft
  - gewohntes Muster → intuitive Struktur, gute Überschaubarkeit
- Modell ist jedoch nicht für alle Zwecke geeignet
  - z.B.: Pipelines, asynchrone Mitteilung, peer-to-peer

Peer-to-Peer-Strukturen

"Gleichrangiger"

- Im Gegensatz zum asymmetrischen Client/Server-Modell



- Ein Client fungiert zugleich als Server für seine Partner
  → keine (teuren) dedizierten Server notwendig
- In der Idealform keine zentralisierten Elemente
  - dies wird gelegentlich in "politischer" Weise artikuliert (vgl. Tauschbörsen)

- *Nachteile*:
  - "Anarchischer" als Client/Server-Architektur
  - Computer müssen leistungsfähig genug sein (cpu-Leistung, Speicherausbau), um für den "Besitzer" leistungstransparent zu sein
  - geringere Stabilität (Peers typw. weniger redundant als echte Server)
  - Datensicherung i.Allg. problematischer als bei zentralen Servern
  - Sicherheit und Schutz kritisch: Lizenzen, Viren, Integrität,...

Zu vielen Aspekten von Peer-to-Peer-Systemen: R. Steinmetz, K. Wehrle (Eds): *Peer-to-Peer Systems and Applications*, Springer-Verlag, 2005

Zustandsändernde /-invariante Dienste

- Verändern Aufträge den Zustand des Servers wesentlich?

- Typische *zustandsinvariante* Dienste:
  - Auskunftsdienste (z.B. Name-Service)
  - Zeitservice

- Typische *zustandsändernde* Dienste:
  - Datei-Server

Idempotente Dienste / Aufträge

- Wiederholung eines Auftrags liefert gleiches Ergebnis

nicht zustandsinvariant!

- Beispiel: “Schreibe in Position 317 von Datei XYZ den Wert W”
- Gegenbeispiel: “Schreibe ans Ende der Datei XYZ den Wert W”
- Gegenbeispiel: “Wie spät ist es?” — aber zustandsinvariant!

Wiederholbarkeit von Aufträgen

- Bei Idempotenz oder Zustandsinvarianz kann bei Verlust des Auftrags (timeout beim Client) dieser erneut abgesetzt werden (→ einfache Fehlertoleranz)
  - vgl. auch frühere Diskussion bzgl. RPC-Fehlersemantik

Zustandslose / -behaftete Server

stateless — statefull

- Hält der Server Zustandsinformation über Aufträge hinweg? (“session”)
  - z.B. (Protokoll)zustand des Clients
  - z.B. Information über frühere damit zusammenhängende (Teil)aufträge

- Aufträge an zustandslose Server müssen autonom sein

- *Beispiel*: Datei-Server

```
open(“XYZ”);
read;
read;
close;
```

In klassischen Systemen hält sich das Betriebssystem Zustandsinformation, z.B. über die Position des Dateizeigers geöffneter Dateien

- bei zustandslosen Servern entfällt open/close; jeder Auftrag muss vollständig beschrieben sein (Position des Dateizeigers etc.)
- zustandsbehaftete Server daher i.Allg. effizienter — notw. Zustandsinformation ist beim Client
- Dateisperren sind bei echten zustandslosen Servern nicht einfach möglich
- zustandsbehaftete Server können wiederholte Aufträge erkennen (z.B. durch Speichern von Sequenznummern) → Idempotenz

- *Crash* eines Servers: Weniger Probleme im zustandslosen Fall (→ Fehlertoleranz)! — entscheidender Vorteil!

- Datei-Server wurden sowohl schon zustandslos (z.B. NFS) als auch zustandsbehaftet (z.B. RFS) realisiert

Sind Webserver zustandslos?

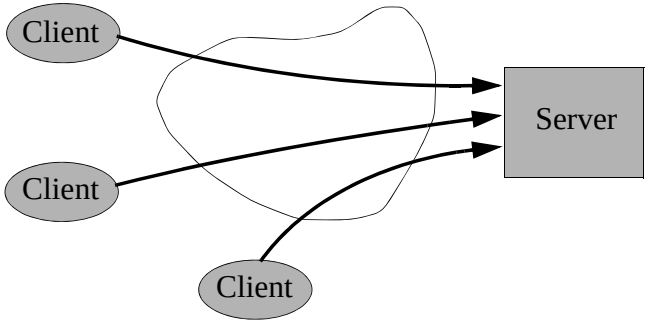
- Beim HTTP-Zugriffsprotokoll wird über den Auftrag hinweg keine Zustandsinformation gehalten
  - jeder link, den man anklickt, löst eine neue “Transaktion” aus
- Stellt z.B. ein Problem beim E-Commerce dar
  - gewünscht sind Transaktionen über mehrere Klicks hinweg und
  - Wiedererkennen von Kunden (beim nächsten Klick oder Tage später)
  - erforderlich z.B. für Realisierung von “Warenkörben” von Kunden
  - gewünscht vom Marketing (Verhaltensanalyse von Kunden)

Lösungsmöglichkeiten zur Wiedererkennung von Kunden:

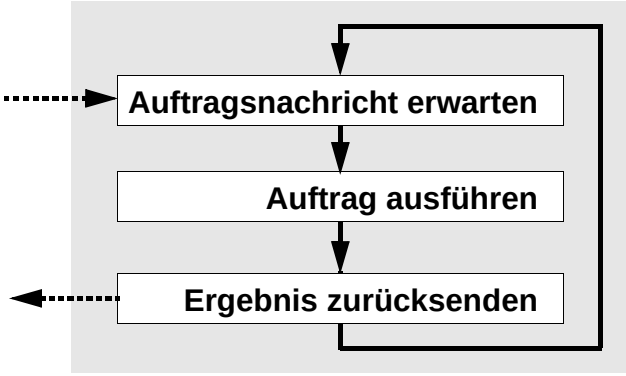
- IP-Adresse des Kunden an Auftrag anheften?
  - Problem: Proxy-Server → mehrere Kunden haben gleiche IP-Adresse
  - Problem: dynamische IP-Adressen → keine Langzeitwiedererkennung
- “URL rewriting” und dynamische Web-Seiten
  - Einstiegsseite eine eindeutige Nummer anheften, wenn der Kunde diese erstmalig aufruft
  - diese Nummer jedem link der Seite anheften und mit zurückübertragen
- Cookies
  - kleine Textdatei, die ein Server einem Browser (= Client) schickt und die im Browser gespeichert wird
  - der Server kann das Cookie später wieder lesen und damit den Kunden wiedererkennen

Gleichzeitige Server-Aufträge?

- Problem: Oft viele “gleichzeitige” Aufträge



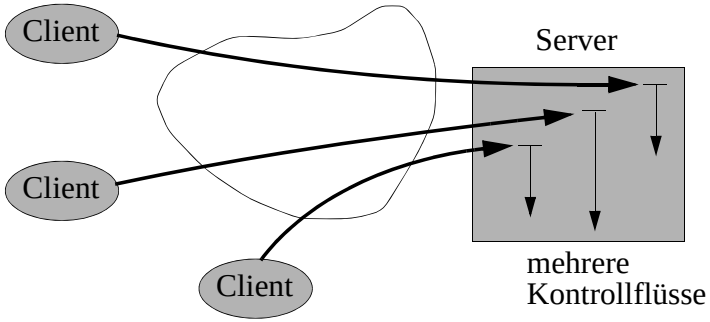
- *Iterative Server* bearbeiten nur einen Auftrag pro Zeit



- “single threaded” (nur ein einziger Kontrollfluss)
- eintreffende Anfragen während Auftragsbearbeitung: abweisen, puffern oder schlichtweg ignorieren
- einfach zu realisieren
- bei trivialen Diensten mit kurzer Bearbeitungszeit sinnvoll

Konkurrente ("nebenläufige") Server

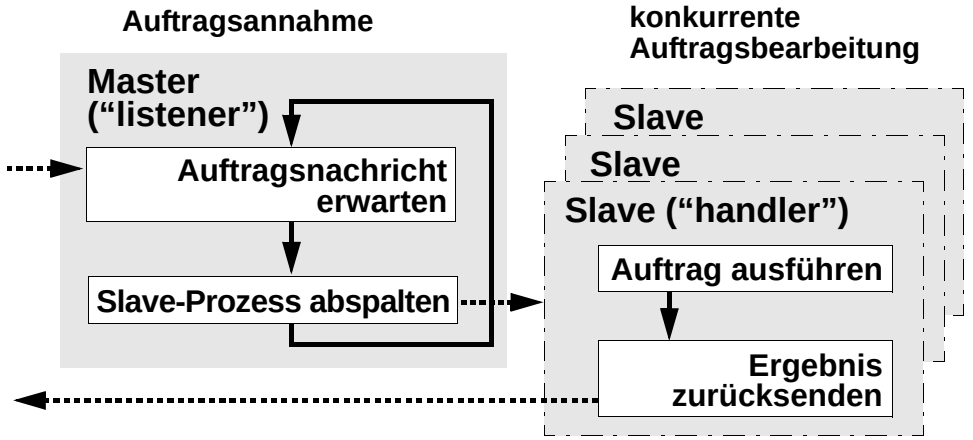
- Gleichzeitige Bearbeitung mehrerer Aufträge
  - sinnvoll (d.h. effizienter für Clients) bei längeren Aufträgen



- Ideal bei physischer Parallelität (z.B. multicore)
  - aber auch bei Monoprozessor-Systemen (vgl. Argumente bei Timesharing-Systemen): Nutzung erzwungener Wartezeiten eines Auftrags für andere Jobs; kürzere mittlere Antwortzeiten bei Jobmix aus langen und kurzen Aufträgen
- Interne Synchronisation bei konkurrenten Aktivitäten sowie evtl. Lastbalancierung beachten
- Verschiedene denkbare Realisierungen, z.B.
  - mehrere Prozessoren bzw. Multicore-Prozessoren
  - Verbund verschiedener Server-Maschinen (Server-Farm, -Cluster)
  - feste Anzahl vorgegründeter Prozesse oder dynamische Prozesse

Konkurrente Server mit dynamischen Handler-Prozessen

- Für jeden Auftrag gründet der *Master* einen neuen *Slave*-Prozess und wartet dann auf einen neuen Auftrag
  - neu gegründeter Slave ("handler") übernimmt den Auftrag
  - Client kommuniziert dann direkt mit dem Slave (z.B. über dynamisch eingerichteten Kanal bzw. Port)
  - Slaves sind oft Leichtgewichtsprozesse ("threads")
  - Slaves terminieren i.Allg. nach Beendigung des Auftrags
  - die Anzahl gleichzeitiger Slaves sollte begrenzt werden



- Alternative: "Process preallocation": Feste Anzahl statischer Slave-Prozesse
  - u.U. effizienter (u.a. Wegfall der Erzeugungskosten)

Übungsaufgabe: Herausfinden, wie es bei Webservern typw. gemacht wird

Master/Slave

Subject: Identification of equipment sold to LA County
Date: Tue, 18 Nov 2003 14:21:16 -0800
From: "Los Angeles County"

The County of Los Angeles actively promotes and is committed to ensure a work environment that is free from any discriminatory influence be it actual or perceived. As such, it is the County's expectation that our manufacturers, suppliers and contractors make a concentrated effort to ensure that any equipment, supplies or services that are provided to County departments do not possess or portray an image that may be construed as offensive or defamatory in nature.

One such recent example included the manufacturer's labeling of equipment where the words "Master/Slave" appeared to identify the primary and secondary sources. Based on the cultural diversity and sensitivity of Los Angeles County, this is not an acceptable identification label.

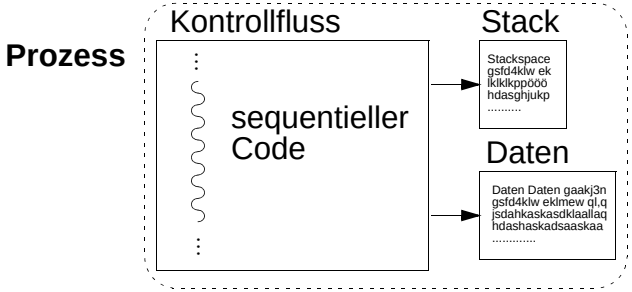
We would request that each manufacturer, supplier and contractor review, identify and remove/change any identification or labeling of equipment or components thereof that could be interpreted as discriminatory or offensive in nature before such equipment is sold or otherwise provided to any County department.

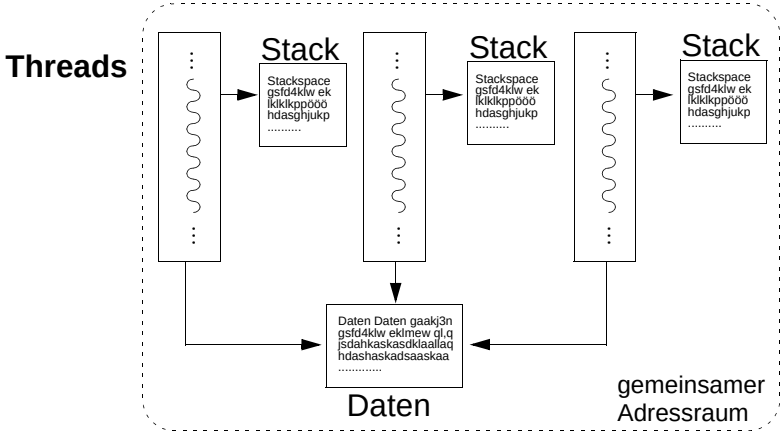
Thank you in advance for your cooperation and assistance.

Joe Sandoval, Division Manager
Purchasing and Contract Services
Internal Services Department
County of Los Angeles

Threads

- Threads = leichtgewichtige Prozesse mit gemeinsamem Adressraum

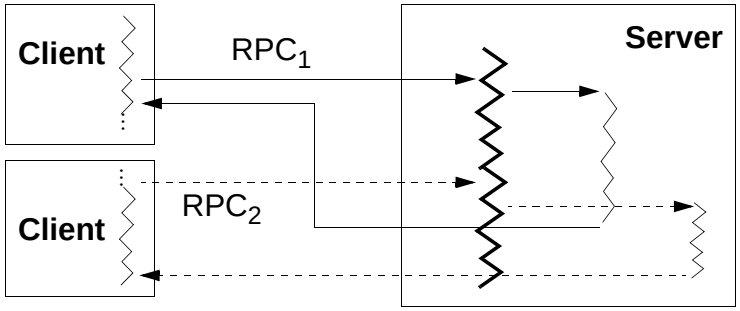




- Einfache Kommunikation zwischen Kontrollflüssen
  - aber: kein gegenseitiger Schutz
- Thread hat weniger Zustandsinformation als ein Prozess
- Kontextwechsel daher i.Allg. wesentlich schneller
  - kein Umschalten des Adressraumkontexts
  - Cache und Translation Look Aside Buffer (TLB) bleiben erhalten
  - kein aufwändiger Wechsel in / über privilegierten Modus

Wozu Multithreading beim Client-Server-Konzept?

- *Server*: quasiparallele Bearbeitung von Aufträgen
  - Server bleibt ständig empfangsbereit



- *Client*: Möglichkeit zum „asynchronen RPC“
  - Hauptkontrollfluss delegiert RPCs an nebenläufige Threads
  - keine Blockade durch Aufrufe im Hauptfluss
  - echte Parallelität von Client (Hauptkontrollfluss) und Server

Problematik von Threads

- Fehlender gegenseitiger Adressraumschutz
  → schwierig zu findende Fehler
- Stackgrösse muss bei Gründung i.Allg. statisch festgelegt werden
  → unkalkulierbares Verhalten bei Überschreitung
- Von asynchronen Meldungen (“Signale”, “Interrupts”) soll i.Allg. nur der “richtige” Thread betroffen werden
- Schwierige Synchronisation → Deadlockgefahr

Aufrufe des Betriebssystem-Kerns können problematisch sein, wenn diese nicht dafür geeignet (“thread safe”) sind:

a) *nicht ablaufinvariante* (“non-reentrant”) Systemroutinen

- interne Statusinformation, die ausserhalb des Stacks der Routine gehalten wird, kann bei paralleler Verwendung überschrieben werden
- z.B. *printf*: ruft intern Speichergenerierungsroutine auf; diese benutzt prozesslokale Freispeicherliste, deren “gleichzeitige” nicht-atomare Manipulation zu Fehlverhalten führt
- “Lösung”: Verwendung von “Wrapper-Routinen”, die gefährdete Routinen kapseln und Aufrufe wechselseitig ausschliessen

b) *blockierende* (“synchrone”) Systemroutinen

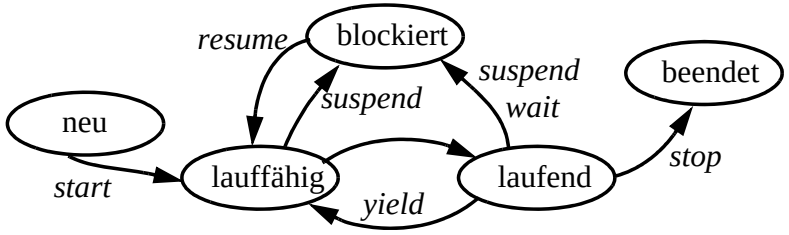
- z.B. synchrone E/A, die *alle* Threads eines Prozesses blockieren würde (statt nur den einen aufrufenden Thread)

Exkurs: Threads in Java

- Hier nur ein *Überblick*; zu weiteren Aspekten vgl. die Dokumentation (online bzw. in Büchern)

- *Konstruktor*:
  - **public** Thread()

- *Methoden*:
  - **void** start()
  - **void** suspend()
  - **void** stop()
  - **void** resume()
  - **void** wait()
  - **static void** yield()



- **static void** sleep(**long** millis) // blockiert für eine gewisse Zeit
- **void** join() // Synchronisation zweier Threads
- **void** setPriority(**int** prio)
- **int** getPriority()
- **void** setDaemon (**boolean** on)

"Hintergrundprozess": terminiert nicht mit dem Erzeuger

- Jeder Thread (genauer: jede von Thread abgeleitete Klasse) muss eine void-Methode *run()* enthalten
  - diese macht die eigentlichen Anweisungen des Threads aus!
  - "run" ist in Thread nur abstrakt definiert

Erzeugen von Threads

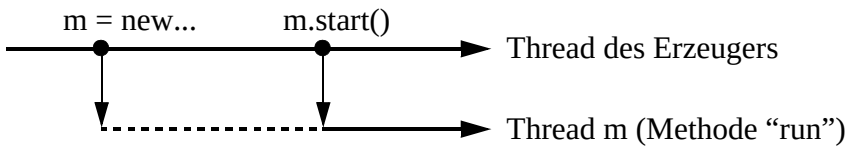
- Typisches Gerüst für einen Thread:

```
class Mythread extends Thread
{ int mynumber;
  public Mythread(int number)
  { mynumber = number; }

  public void run()
  { // hier die Anweisungen des Threads
    ...
  }

  // hier andere Methoden
}
```

Konstruktor



- Erzeugen aus einem anderen Thread heraus:

```
Mythread m = new Mythread(5);
m.start();
```

mit dieser Nummer identifizieren wir einen Thread individuell

damit kann man den Thread kontrollieren (z.B. m.suspend();)

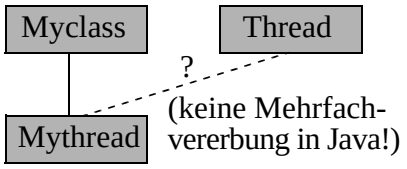
- Alternative: "Anonyme" Erzeugung

```
new Mythread(5).start();
```

Mythread "kann" start, da dies von Thread ererbt ist

- Zusammenfassen der beiden Anweisungen
- dann aber keine Kontrolle möglich, da kein Zugriff auf den Thread

Abgeleitete Klassen als Threads



- Myclass sei eine Klasse, die nicht von Thread abgeleitet ist
- Mythread soll Unterklasse von Myclass sein; gleichzeitig aber auch einen Thread darstellen

- Lösung über die Runnable-Schnittstelle:

```
class Mythread extends Myclass implements Runnable
{ ...
  public void run()
  { ... }
  ...
}
```

Definition ("Implementierung") von run muss hier erfolgen

- Erzeugung aus einem anderen Thread heraus:

```
Mythread m = new Mythread(...);

Thread t = new Thread (m);
t.start();
```

m "kann" kein start, da dies nicht in runnable enthalten; erst t als Thread kann start

zweite Form des Konstruktors!

- Es geht auch so:

```
Mythread m = new Mythread(...);
```

In der Klasse Mythread dann an geeigneter Stelle:

```
t = new Thread(this);
t.start();
```

Assoziation des Threads zur Methode run herstellen

Ein Thread-Beispiel

```
class T extends Thread
{ String wer; int delay = 0;

  T (String s, int zeit)
  { wer = s; delay = zeit;
    System.out.println(wer + " : " + delay);
  }

  public void run()
  { try
    { while (true)
      { System.out.println(wer);
        sleep(delay);
      }
    }
    catch (InterruptedException e) {return;}
  }

  public static void main(String args[])
         throws InterruptedException
  { int i = Integer.parseInt(args[0]);
    int j = Integer.parseInt(args[1]);
    new T("She loves me", i).start();
    sleep(i/2);
    new T("  not", j).start();
    System.out.println("Jetzt sind beide
                        Threads gestartet");
  }
}
```

Konstruktor

Exception, falls während des sleep ein Interrupt ausgelöst wird

Methode run und damit den Thread beenden

statt try / catch

Gründen zweier Threads als Instanzen der eigenen Klasse

Die Threads in Aktion

> java T 500 500

She loves me: 500
She loves me
 not : 500
Jetzt sind beide Threads gestartet
 not
She loves me
 not
She loves me
 not
She loves me
 not
She loves me
 not
She loves me
 not
She loves me
....

Vielleicht hätte man für ein Happy-End doch die Parameter anders wählen sollen?

> java T 1000 300

She loves me: 1000
She loves me
 not : 300
Jetzt sind beide Threads gestartet
 not
 not
She loves me
 not
 not
 not
She loves me
 not
 not
 not
 not
She loves me
 not
 not
 not
She loves me
 not
 not
 not
She loves me
....

Ein weiteres Thread-Beispiel

- Ein Thread “Hin” schreibt Sterne; ein Thread “Her” löscht Sterne - beide arbeiten (quasi)parallel
  - nach einer Idee von Ralf Kühnel (“Die Java-Fibel”, Addison-Wesley)

```
public class HinHer
{ public static void main (String args[])
  { System.out.print("********************");
    System.out.flush();
    new Hin().start();
    new Her().start();
  }
}

class Hin extends Thread
{ public void run()
  { try
    { while(true)
      { sleep((int)(Math.random() * 1000));
        repaint();
        System.out.flush();
      }
    }
    catch (InterruptedException e) {return;}
  }

public void repaint()  // Sterne hinzufuegen
  { System.out.print("*****"); }
}

class Her extends Hin
{ public void repaint()  // Sterne loeschen
  { System.out.print("\b\b\b\b\b    \b\b\b\b\b"); }
}
```

Übung: Ausprobieren, wer gewinnt!

Thread-Kontrolle

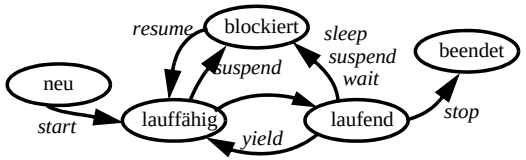
laufend oder lauffähig

- Ein Thread läuft so lange, bis

  von aussen oder durch sich selbst

  - seine run-Methode zuende ist
  - er mit stop() abgebrochen wird

- Ein Thread kann *sich selbst*



  - die cpu entziehen: *yield()*
    (Übergang in Zustand “lauffähig”; wird automatisch wieder “laufend”, wenn keine wichtigeren Threads mehr laufen möchten)
  - schlafen legen: *sleep()*
    (automatisches resume nach gegebener Zeit)
  - anhalten: *suspend()* bzw. *wait()*
  - beenden: *stop()*
  - in der Priorität verändern: *setPriority()*

    Prioritäten: normal 5, minimal 1, maximal 10 (anfangs: Priorität des erzeugenden Prozesses)

- Ein Thread kann einen *anderen* Thread t

  hierzu ist eine Referenz auf den Thread notwendig

  - *starten*: t.start()
  - *anhalten*: t.suspend()
  - *fortsetztbar machen*: t.resume();
  - *beenden*: t.stop()
  - in der *Priorität verändern*: t.setPriority()

Beachte: *stop*, *suspend* und *resume* führen, unbedacht angewendet, zu unsicheren Programmen und sollte daher eigentlich nicht mehr verwendet werden

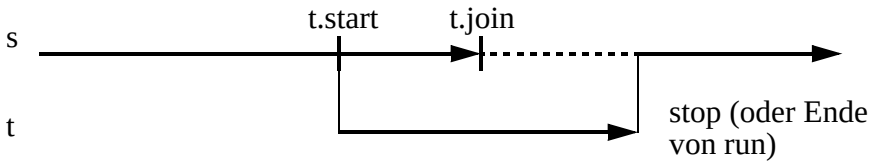
Warten auf Threads

- Methode *join* verwenden, wenn auf die Beendigung eines anderen Threads gewartet werden soll
  - z.B. weil auf die von ihm berechneten Daten zugegriffen werden soll

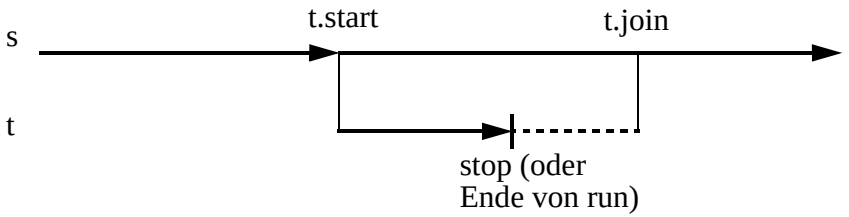
- Das Objekt eines beendeten Threads existiert weiter
  - auf dessen Zustand kann also noch zugegriffen werden

- Auf einen beendeten Thread kann *start* aufgerufen werden
  - run-Methode wird dann erneut ausgeführt

- Beispiel: Thread s wartet tatsächlich auf t:



- Thread t ist eher fertig:



- Nach t.join ist jedenfalls garantiert, dass t beendet ist

Thread-Scheduling

- Scheduling: Planvolle Zuordnung der cpu an die einzelnen Threads (jeweils für eine gewisse Zeit)

- Genaue Scheduling-Strategie ist *nicht* Bestandteil des Java-Sprachstandards
  - kann jede Implementierung für sich entscheiden (und damit Eigenheiten des zugrundeliegenden Betriebssystems effizient nutzen)
  - man darf sich daher nicht auf "Erfahrungen" verlassen
  - genauer: nicht auf die Wirkung von Zeitscheiben oder Prioritäten etc.

  sonst nicht deterministisch und nicht portabel!

- Konsequenzen:
  - Test und Debugging ist sehr schwierig
  - alle denkbaren verzahnten Abläufe ("interleavings") berücksichtigen
  - Menge der verzahnten Abläufe durch geeignete *Synchronisation* einschränken (nur "korrekte" Abläufe zulassen)
  - ggf. mit "yield" Scheduling teilweise selbst realisieren (z.B. um andere Prozesse am Verhungern zu hindern)

- Vorgabe: Ein Thread-Scheduler *soll* Threads mit höherer Priorität bevorzugen
  - Priorität eines Threads entspricht zunächst der des Erzeugers
  - Priorität kann verändert werden (*setPriority*)
  - Thread mit der höchsten Priorität *sollte* immer laufen (ohne Garantie!)
  - wenn ein Thread mit höherer Priorität als der gegenwärtig ausgeführte lauffähig wird, wird der gegenwärtige i.a. unterbrochen

  auch bei einem Rechner mit zwei cpus?

Thread-Scheduling (2)

- Wie lange läuft ein Thread?

  im Sinne von "laufend"

  (sofortiger) Threadwechsel ist aber nicht garantiert!
  - bis ein Thread mit höherer Priorität lauffähig wird
  - bis er sich beendet (mit "stop" oder dem Ende von "run")
  - bis er in den "blockiert"-Zustand übergeht (explizit mit "suspend", "wait", "sleep" etc. ; implizit durch E/A etc. - es ist aber umgekehrt nicht garantiert, dass ein auf E/A-wartender Thread die cpu freigibt!)
  - bis er mit "yield" die Kontrolle dem Scheduler übergibt

- Scheduling mit Zeitscheiben *kann* vom System realisiert sein, *muss* aber *nicht*
  - Thread läuft längstens bis zum Ablauf der Zeitscheibe (dann i.a. Round-Robin-Scheduling unter Threads gleicher maximaler Priorität)

- Konsequenzen:
  - Thread mit Endlosschleife kann gegebenenfalls das ganze System blockieren (andere Threads "verhungern")
  - Threads gleicher Priorität verhalten sich besonders willkürlich
  - "yield" ist insbesondere bei Systemen ohne Zeitscheiben wichtig
  - Prioritäten sollten besser nicht als Synchronisationsmittel (zum Erzwingen einer bestimmten Reihenfolge etc.) eingesetzt werden
  - Portabilität ist bei diletantischer Nutzung von Threads eingeschränkt

---

Exkurs-Ende (Threads in Java)