Freie Universität Berlin

# Algorithmen und Programmierung IV: Nichtsequentielle Programmierung

Robert Tolksdorf

Freie Universität Berlin

# Rückblick

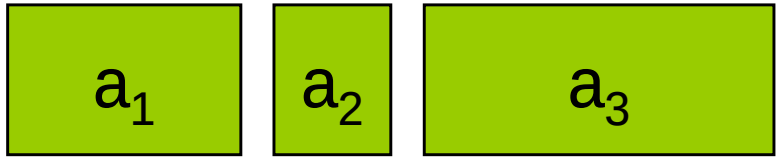
# Inhalt der Vorlesung



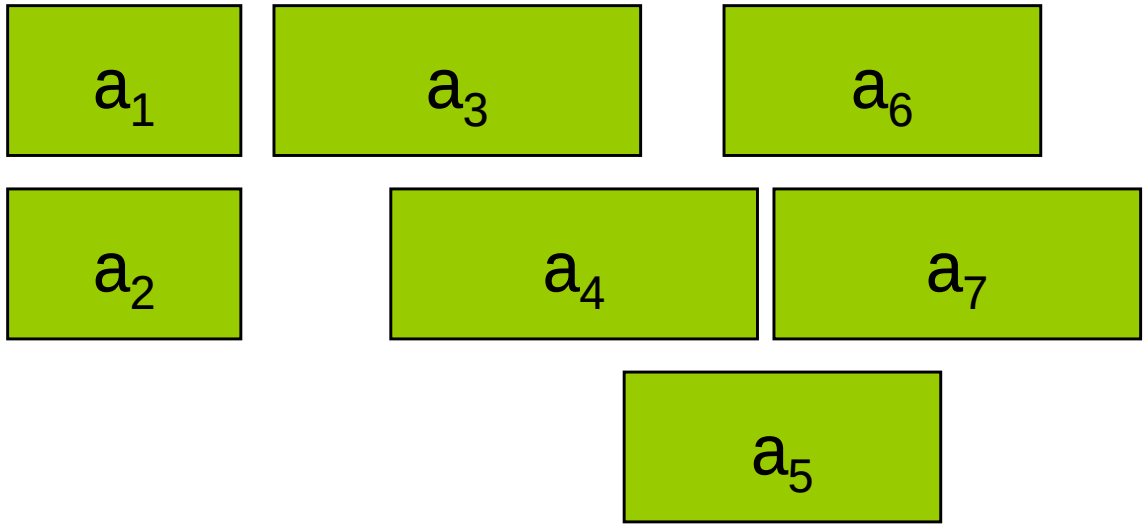
1. Organisation
2. Einführung
3. Nebenläufige Prozesse
4. Interaktion über Objekte
5. Ablaufsteuerung
6. Implementierung
7. Interaktion über Nachrichten

# Programmablauf

- Programmablauf besteht aus Aktionen und heißt
  - sequentiell (sequential),
    - wenn er aus einer Folge von zeitlich nicht überlappenden Aktionen besteht,

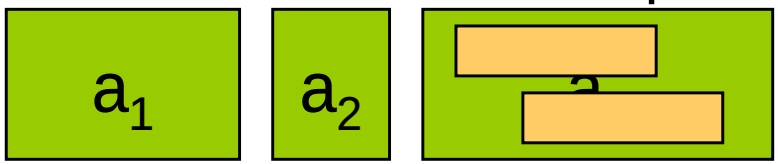



  - parallel (parallel),
    - wenn er nicht sequentiell ist



# Programmablauf

- Ein und derselbe Ablauf kann
  - auf *einer* Abstraktionsebene sequentiell,
  - auf *einer anderen* parallel sein !




- Beispiel 1:
```
x = 3; y = 7;
{ x = x+1; y = 2*y; }
z = x+y;
```
- beschreibt einen sequentiellen Ablauf von 5 Aktionen (genauer: Zuweisungen)
- aber auf Hardware-Ebene könnten manche dieser Zuweisungen parallel ausgeführt werden!

# Nebenläufigkeitskonstrukt 1: Nebenläufigkeitsanweisung

Freie Universität Berlin

- Nebenläufigkeitsanweisung (concurrent statement):

- Beispielsyntax:
  - Statement = ... | ConcurrentStatement .
  - ConcurrentStatement = CO Processes OC .
  - Processes = Process { || Process } .
  - Process = StatementSequence .

- Beispiel:

```
x = 3;  y = 7;
CO x = x+1; || y = 2*y; OC
z = x+y;
```

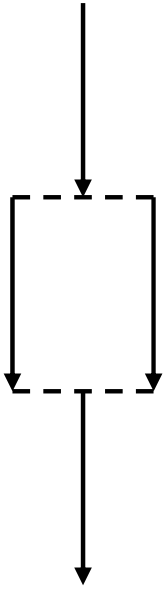
# Nebenläufigkeitsanweisung

- Semantik:
  - Die Prozesse einer Nebenläufigkeitsanweisung werden unabhängig voneinander ausgeführt.
  - Die Nebenläufigkeitsanweisung ist beendet, wenn alle Prozesse beendet sind.

```
        x = 3;
        y = 7;

CO x = x+1; || y = 2*y; OC

        z = x+y;
```



# Nebenläufigkeitskonstrukt 2: Gabelungsanweisung

Freie Universität Berlin

- Gabelungsanweisung (fork statement)

- Beispielsyntax:
  - ForkStatement = FORK Process .
  - Process = Statement .

- Beispiel

```
x = 3;
y = 7;
FORK      x = x+1;
y = 2*y;
z = x+y;
```

# Gabelungsanweisung

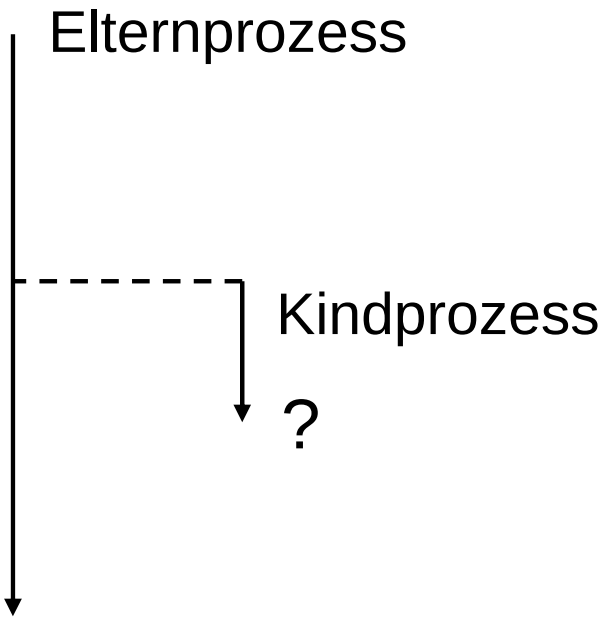


- Semantik:
  - Die Gabelungsanweisung startet den angegebenen Prozess.
  - Wann dieser beendet ist, ist *nicht* bekannt!

```
x = 3;
y = 7;

FORK      x = x+1;

y = 2*y;
z = x+y;
```



# 1.1.3 Wie wird nichtsequentiell programmiert?

Freie Universität Berlin

- Zentrale Begriffe:
  - Prozess
  - Interaktion zwischen Prozessen
  - Synchronisation von Prozessen

# Konflikt



- Def.:
  Zwei Prozesse, die nebenläufig auf eine gemeinsame Variable x zugreifen, stehen miteinander in Konflikt (conflict, interference) bezüglich x, wenn mindestens einer der Prozesse x verändert.

- Konflikte sind meistens unerwünscht (➔1.2)

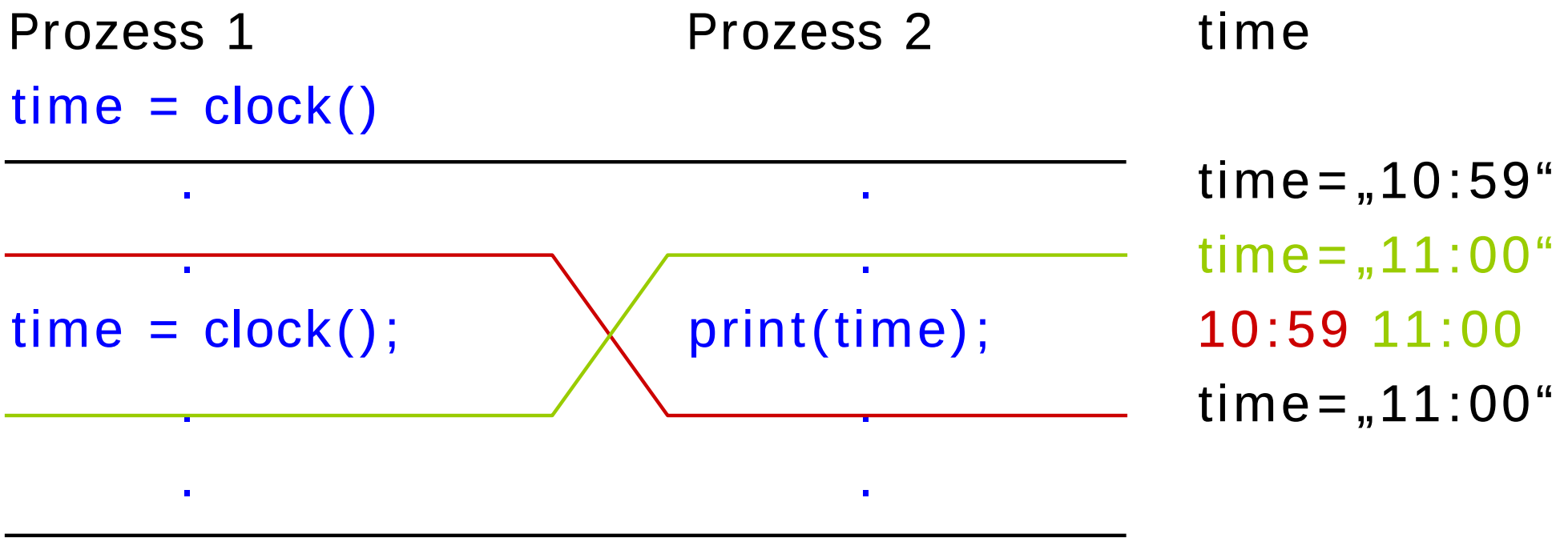
- und werden durch Synchronisationsmaßnahmen unterbunden

# 1.2 Nichtdeterminismus

- Der Ablauf eines sequentiellen Programms ist immer deterministisch
  - d.h. gleiche Eingaben führen zu gleichen Zustandsfolgen
- Das Ergebnis ist determiniert
  - d.h. gleiche Eingaben liefern gleiche Ausgaben (sofern nicht explizit mit Zufallszahlen gearbeitet wird)
- Asynchronie
  - führt in der Regel zu nichtdeterministischen Abläufen
  - möglicherweise aber trotzdem zu determinierten Ergebnissen.

# 1.2.1 Zeitschnitte

- dienen der Veranschaulichung nichtsequentieller, insbesondere nichtdeterministischer Abläufe:



# Ablaufdeterminismus/ Ergebnisdeterminiertheit

Freie Universität Berlin

| Ergebnis / Ablauf | determiniert | nicht determiniert |
|---|---|---|
| deterministisch | immer | - |
| nichtdeterministisch | möglich | meistens |

# 1.2.2 Unteilbare Anweisungen

- verhalten sich so, als würden sie „zeitlos“ ausgeführt
- Es können also keine „Zwischenzustände“ beobachtet werden
- (statt „unteilbar“ auch atomar, *indivisible, atomic*)

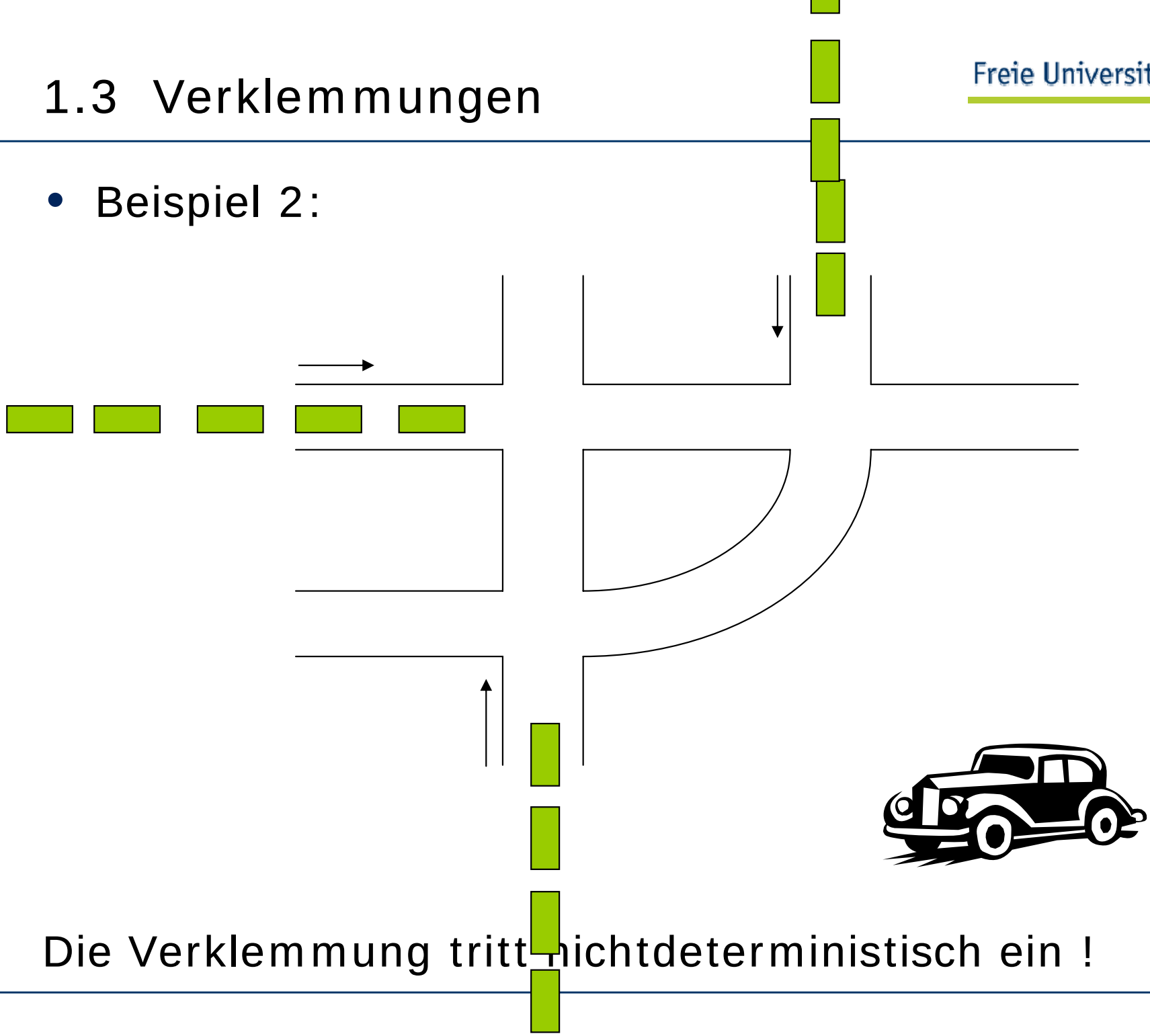
- *(Beachte:* in 1.2.1 wurde stillschweigend vorausgesetzt, dass die dortigen Anweisungen unteilbar sind.)

# 1.3 Verklemmungen

- Def.:
  Beim Ablauf eines nichtsequentiellen Programms liegt eine Verklemmung (deadlock) vor, wenn es Prozesse im *Wartezustand* gibt, die durch *keine mögliche Fortsetzung* des Programms daraus erlöst werden können.
- Bemerkung 1:
  Typischerweise entsteht eine Verklemmung dadurch, daß zwei oder mehr Prozesse *zyklisch* aufeinander warten.
- Bemerkung 2:
  Wenn *kein* zyklisches Warten vorliegt, spricht man häufig von *hangup* statt von *deadlock*.
  - Beispiel: Elternprozess bleibt bei JOIN hängen, weil Kindprozess nicht terminiert.

# 1.3 Verklemmungen

- Beispiel 2:



Die Verklemmung tritt nichtdeterministisch ein !

- Achtung:
  Es gibt Programme, die nicht terminieren sollen – aber natürlich auch verklemmungsfrei sein sollen

- Daher: Erweiterte Definition nötig

- Def.: Korrektheit = Sicherheit + Lebendigkeit

- Sicherheit (*safety*) eines Programms:
  - das Programm produziert nichts Falsches
    (⇔ partielle Korrektheit, Determiniertheit)
- Lebendigkeit (*liveness*) eines Programms:
  - das Programm bleibt nicht hängen
    (⇔ Termination, Verklemmungsfreiheit)

- Variante 1 – prozedurbezogene Gabelungsanweisung:

- FORK ProcedureInvocation
  FORK p(x)
  - Aufrufer veranlasst asynchrone Ausführung
  - nach vollzogener Parameterübergabe!

- process ProcedureInvocation
  - Burroughs Extended Algol (1971)

- Jede Prozedur kann asynchron ausgeführt werden:
  *Asynchronie an Aufruf gebunden und vom Aufrufer bestimmt*

# 2.1.2 Prozeduren als Prozesse

Freie Universität Berlin

- Variante 2 – asynchrone Prozedur:
- process ProcedureDeclaration (1.1.3.1 ←)
  - d.h. *Asynchronie ist an die Prozedur gebunden*
  - ebenfalls nach vollzogener Parameterübergabe
- Jac - Java with Annotated Concurrency:
  (http://page.mi.fu-berlin.de/~haustein/jac/doku.html)
  - Annotation asynchroner Methoden
  - async MethodDeclaration oder auto MethodDeclaration
  - public class Storage {
    /**
    * save data to disk
    * @jac.async
    * @jac.require d != null */
    public void writeToDisk(Data d) {...}
  - Aufruf der Operation writeToDisk kehrt unmittelbar nach der Überprüfung der Voraussetzung zurück, während die Abarbeitung in einem separaten Kontrollfluss stattfindet

- Java sieht kein Schlüsselwort für Prozesse vor, sondern bestimmte Klassen/Schnittstellen.

- Mit anderen Worten: der Prozessbegriff wird mit Mitteln der Objektorientierung eingeführt.

- Bewertung (Peter Löhr):
- hübsche Übung in Objektorientierung, aber nicht angenehm für den Benutzer, weil eher implementierungsorientiert als problemorientiert

# 2.2.1 Thread und Runnable



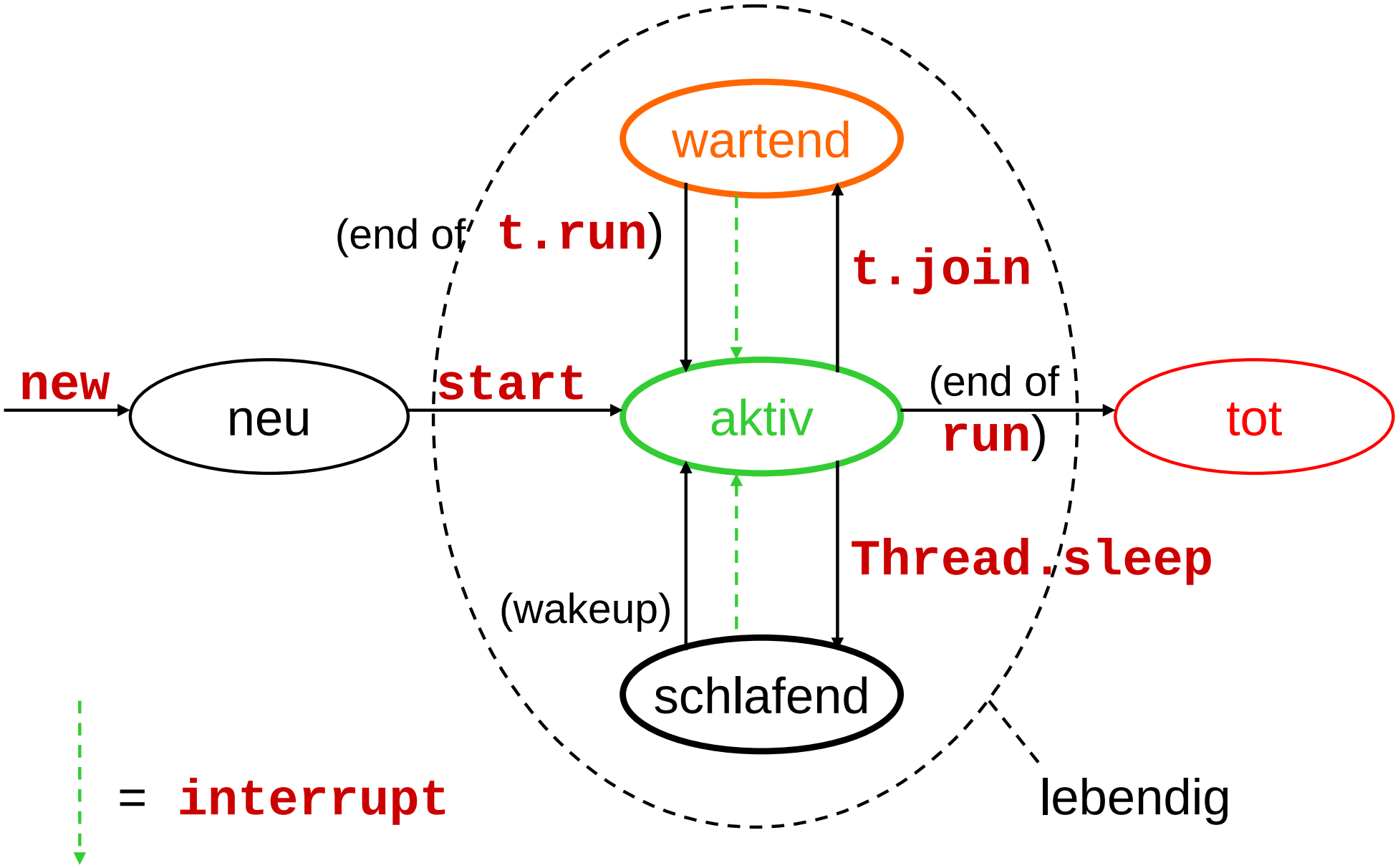
- Im Paket java.lang befinden sich

```
interface Runnable {
  public void run();
}

public class Thread implements Runnable {...
public void run()   {}    // thread activity
public void start() {...} // start thread
public void join()  {...} // wait for thread
}
```

- Ein Thread-Objekt ist ein Prozess, der durch die Operation start gestartet wird und dann selbsttätig run ausführt ...

# Zustandsübergänge im Bild



# 2.2.3 Speichermodell



- *gibt weniger Garantien* als es der naiven Vorstellung von der Ausführung eines Programms entspricht,
- erlaubt damit dem Übersetzer *beträchtliche Freiheiten* auf raffinierten Rechnerarchitekturen
- *gibt immerhin gewisse Garantien*, an die man sich beim Programmieren halten kann/muss

Freie Universität Berlin

- dient der Vermeidung unkontrollierter nebenläufiger Zugriffe auf gemeinsame Datenobjekte und der damit potentiell verbundenen Schmutzeffekte (1.2←)

- Zur Erinnerung:
  wenn nur gelesen wird, droht keine Gefahr (Beispiel: Zeichenketten in Java sind *immutable objects*)

- Gefahr droht, wenn mindestens einer der beteiligten Prozesse, das Datenobjekt *modifiziert*.

- Nachteile von ⟨ ⟩ (1.2.2🡐):
  - i.a. nicht effizient implementierbar,
  - meist unnötig restriktiv. Beispiel:

```
co ... ⟨ a++; ⟩ ...
 || ... ⟨ a--; ⟩ ...
 || ... ⟨ b++; ⟩ ...
 || ... ⟨ b--; ⟩ ...
 oc
```

verhindert überlappende Manipulation von a und b – ohne Not!

# 3.1.1 Kritische Abschnitte

- Syntax in Java
  Java Statement = ..... | SynchronizedStatement
  SynchronizedStatement =
  synchronized ( Expression ) Block

- Der angegebene Ausdruck muß ein Objekt bezeichnen.
- Der angegebene Block heißt auch kritischer Abschnitt (critical section).

# 3.1.1 Kritische Abschnitte

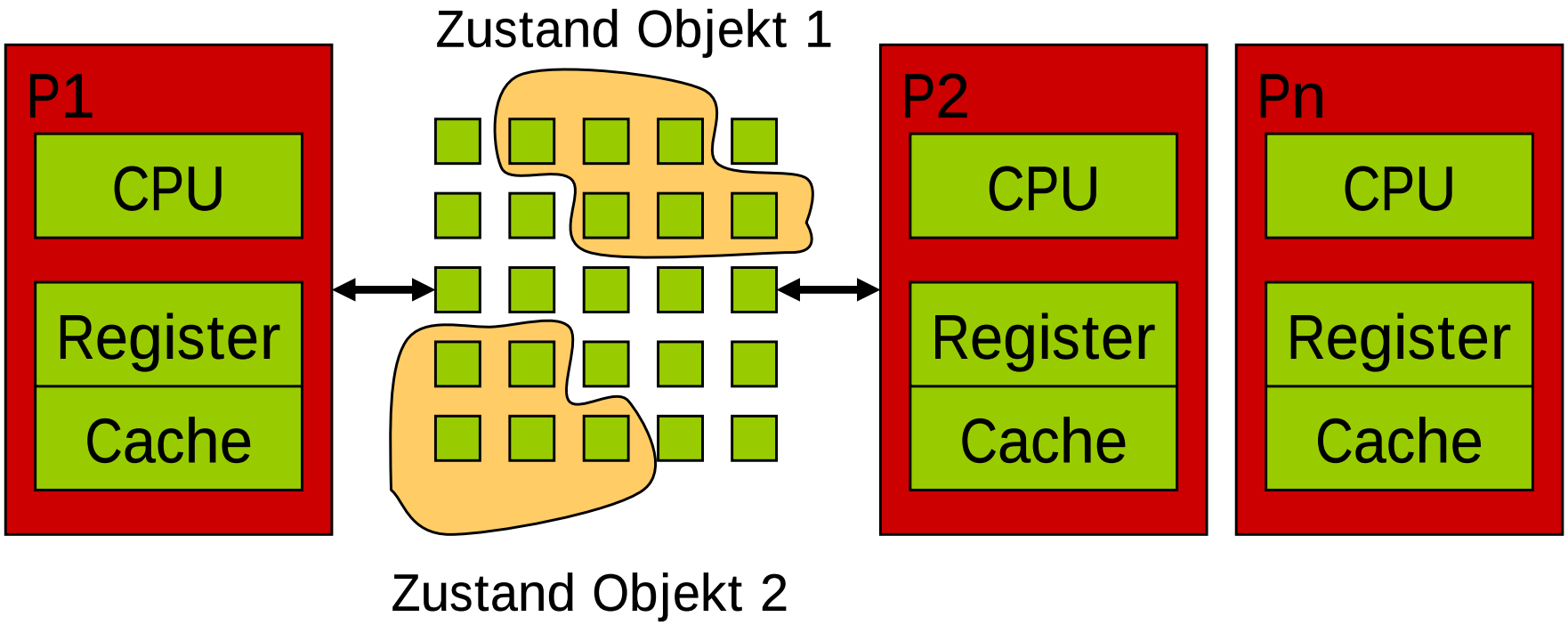
- Alternative Terminologie:

  „Jedes Objekt hat ein Schloss (*lock*), das anfangs offen ist. Ist es offen, ist der Eintritt in den kritischen Abschnitt möglich. Beim Eintritt wird das Schloss geschlossen (gesperrt, *locked*), beim Austritt wieder geöffnet (*unlocked*).“

- Achtung!
  „Das Objekt wird gesperrt“ ist falsche, irreführende Terminologie! Es gibt keinen notwendigen Zusammenhang zwischen dem Objekt und den im kritischen Abschnitt manipulierten Daten.

# Speichermodell in Java

- Speichermodell: Idealer Mehrprozessorbetrieb
  - Jeder Prozess arbeitet auf einem Prozessor
  - Alle Prozesse teilen sich Speicher
  - Jeder Prozessor hat lokale Kopien des gemeinsamen Speichers (Register, Cache)



# Concurrency Utilities



- Klassen für atomaren Zugriff auf Variablen und Referenzen
  - java.util.concurrent.atomic
- Erweitertes Collections Paket optimierten nebenläufig nutzbaren Varianten
  - java.util.concurrent
- Erweiterte Sperren
  - java.util.concurrent.locks
- Erweiterte Synchronisationsmittel
  - java.util.concurrent
- ...
- Siehe http://java.sun.com/j2se/1.5.0/docs/api/overview-summary.html

# 3.1.2 Monitore
(Dijkstra, Brinch Hansen, Hoare, 1972-1974)

Freie Universität Berlin

- Def.:
  Ein Monitor ist ein Objekt mit
  - vollständiger Datenabstraktion
  - vollständigem wechselseitigem Ausschluss zwischen allen seinen Operationen

- Ideale Syntax (nicht Java!) :
  - Ersetzung von class durch MONITOR oder SYNCHRONIZED CLASS
  - Vorteil: Sicherheit – Invariante bleibt garantiert erhalten!
  - Nachteil: evtl. unnötig restriktiv

# 3.1.3 Nachrüsten von Ausschluss…

- … bei Klassen, die für *sequentielle* Benutzung gebaut wurden,
- für Objekte, die *nichtsequentiell* benutzt werden,
- durch Bereitstellung eines Monitors, der als synchronisierte („*thread-safe*“) Version der Klasse eingesetzt werden kann,
- typischerweise mit *gleicher Schnittstelle*.

# 2 Alternativen:

1. *Vererbung:* Monitor ist *Unterklasse* der Originalklasse
2. *Delegation:* Monitor ist *Adapter* für die Originalklasse

- Beispiel:

```
interface SortedSet { ...
  boolean add     (Object x);
  boolean contains(Object x);
}
class TreeSet implements SortedSet { ...
  public boolean add     (Object x) {.....}
  public boolean contains(Object x) {.....}
}
```

voll synchronisiert, da die Implementierung des Originals i.a. unbekannt ist !

# Leser/Schreiber-Ausschluss



- Wünschenswertes Sprachkonstrukt anstelle von synchronized :
  - Statement = ..... | **READING**(Expression) Block
    | **WRITING**(Expression) Block

- *Semantik:*
  Wie bei synchronized
  - wechselseitiger Ausschluss von Abschnitten, deren Expression sich auf das gleiche Objekt bezieht –
  - außer wenn beide READING sind
- Statische oder dynamische Schachtelung möglich, z.B. auch *upgrading:*
  **READING**(x){... **WRITING**(x){...} ... }

- Datenbank *(database)*
  enthält große Menge von langzeitgespeicherten Daten

- Transaktion *(transaction)*
  besteht aus mehreren lesenden/schreibenden Einzelzugriffen auf Daten der Datenbank:

```
BEGIN  ...
    ...          → ABORT   Abbruch möglich
    ...
COMMIT
```

# Serialisierbarkeit



- Def.: Ein Ablauf nebenläufiger Operationen heißt serialisierbar, wenn er den gleichen Effekt und die gleichen Ergebnisse hat, als würden die Operationen *in irgendeiner Reihenfolge streng nacheinander* ausgeführt.
- Wünschenswert ist, daß alle Abläufe serialisierbar sind !

- Def.: Korrektheitskriterium für nebenläufig benutzte Objekte:
  - Ein Objekt heißt serialisierbar, wenn jeder mögliche Ablauf serialisierbar ist.
  - (→3.1.7)

# Zwei-Phasen-Sperren



- Diese Idee ist Grundlage des Zwei-Phasen-Sperrens *(two-phase locking, 2PL)* bei Datenbanken
- 4 Varianten:
  1. *konservativ und strikt:*
     alle benötigten Sperren am Anfang anfordern und am Schluss freigeben
  2. *konservativ:*
     alle benötigten Sperren am Anfang anfordern, aber *baldmöglichst freigeben*
  3. *strikt:*
     jede Sperre *spätmöglichst anfordern* und alle am Schluss freigeben
  4. *(sonst:)*
     jede Sperre *spätmöglichst anfordern* und *baldmöglichst freigeben*
- *„2 Phasen“: erst Sperren setzen, dann Sperren lösen.*

- *Motivation:*

- Wenn für eine Operation auf einem Objekt die *Voraussetzung nicht erfüllt* ist, ist es häufig angebracht, auf deren Erfüllung zu warten, anstatt false zu liefern oder eine *Ausnahme* zu melden.

- *Beispiele:*
  Virtueller Drucker (3.1.1←), Puffer(3.1.2 ←)

# 3.2.1 Verzögerte Monitore

- Verzögerter Monitor *(delayed monitor)* =

- Monitor, in dem auf die *Erfüllung einer Bedingung* gewartet wird
- Beim Blockieren wird der *Ausschluss aufgegeben*, so dass ein anderer Prozess in den Monitor eintreten kann – dessen Aktionen vielleicht dazu führen, dass die Bedingung erfüllt ist
- Ein im Monitor blockierter Prozess kann *frühestens* dann fortfahren, wenn ein anderer Prozess den Monitor verlässt – oder selbst blockiert.
- Blockierte Prozesse haben *Vorrang* vor Neuankömmlingen.

# 3.2.1.1 Warteanweisungen



- Syntax (Nicht Java!)
  - Statement = . . . | AwaitStatement
  - AwaitStatement = AWAIT Condition ;
- Diese Warteanweisung ist nur im statischen Kontext eines Monitors erlaubt, und die gemeinsamen Variablen in der *Condition* müssen Monitorvariable sein.
- *Semantik:*
  - Wenn nach der Warteanweisung fortgefahren wird, ist die angegebene Bedingung garantiert erfüllt.
  - Muss gewartet werden, wird der Monitor freigegeben.
  - *Fairness:* blockierte Prozesse haben Vorrang (s.o.)

# 3.2.1.2 Wachen

- *Deklarative Variante* der Warteanweisung:
  - Wartebedingung als **Wache** *(guard) vor* einem Operationsrumpf eines Monitors (*gesperrt* wird allerdings *vor* Auswertung der Wache)
- *Vorteil:*
  Voraussetzung *(precondition) bewirkt* Synchronisation
- Beispiel (**nicht Java!**):

```
public void send(M m) WHEN count<size {
        cell[rear] = m;
        count++;
        rear = (rear+1)%size;
}
```

# Implementierung?



- **Effiziente Übersetzung** von **AWAIT / WHEN** ist schwierig !

1. Nach jeder Zuweisung bei allen wartenden Prozessen die Wartebedingungen zu überprüfen ist undenkbar.
2. Daher:
   - gemeinsame Variablen in Bedingungen dürfen nur Monitorvariable sein
   - dadurch ist Überprüfung auf den Monitor, bei dem zugewiesen wird, beschränkt
   - sofern es sich nicht um Verweisvariable handelt!
3. Es genügt, die Wartebedingungen dann zu überprüfen, wenn der Monitor freigegeben wird, also beim Verlassen des Monitors oder bei einem Warten bei **AWAIT**.
   Das kann immer noch ineffizient sein.

# 3.2.2 Ereignissynchronisation



- Prozesse warten auf Ereignisse *(events)*,
  die von anderen Prozessen ausgelöst werden.

- *Mehrere* Prozesse können auf *ein* Ereignis warten;
  - beim Eintreten des Ereignisses werden entweder
    - *alle* Prozesse oder
    - nur *ein* Prozess aufgeweckt;
  - das Ereignis „geht verloren“, wenn *keiner* wartet.

# 3.2.2.2 Signal-Varianten

1. *signal-and-wait:*
   - aus WAIT aufgeweckter Prozess übernimmt Monitor, und aufweckender Prozess *blockiert in* SIGNAL (!).
   - *Begründung:* Monitorübergabe ohne Zustandsänderung.
2. *signal-and-continue:*
   - aus WAIT aufgeweckter Prozess übernimmt Monitor erst dann, wenn aufweckender Prozess ihn freigibt.
   - *Begründung:* Effizienz.
3. *signal-and-return:*
   - SIGNAL ist mit Verlassen des Monitors verbunden.
   - *Begründung:* Vorteile von 1) und 2), begrenzter Nachteil.

# 3.2.2.3 Ereignissynchronisation in Java

Freie Universität Berlin

- mittels Operationen der Wurzelklasse Object –
- *sofern* der ausführende Thread das Objekt gesperrt hat (*andernfalls* IllegalMonitorStateException):

- void wait() **throws** InterruptedException
  - blockiert und gibt die Objektsperre frei
- void notify()
  - weckt einen blockierten Thread auf (sofern vorhanden), gibt aber noch nicht die Sperre frei (signal-and-continue)
  - aufgeweckter Thread wartet, bis er die Sperre wiederbekommt
- void notifyAll()
  - entsprechend für *alle* blockierten Threads

# Ereignissynchronisation in Java

- Mit anderen Worten:

- Für jedes Objekt (Monitor) gibt es nur ein „Standard-Ereignis“ – das notify-Ereignis

- *Fairness:*
  - *keinerlei* Garantien!
  - Insbesondere konkurrieren nach notifyAll *alle* aufgeweckten Threads *und* die von außen hinzukommenden Threads um die Sperre.
- *Warten mit Timeout:*
- void wait(long msecs)
  - wie wait, mit dem Zusatz, daß der blockierte Thread nach der angegebenen Zeit geweckt wird.

# Condition Objekte in java.util.concurrent

Freie Universität Berlin

- An Objektsperren gibt es nur eine einzige Bedingung auf die mit wait() gewartet werden kann
- Condition Objekte machen diese Bedingung explizit
- Ein Sperrobjekt kann mehrere Bedingungsobjekte haben
- In der Lock Schnittstelle weiterhin:
  Condition newCondition()
  Erzeugt und bindet ein Condition-Objekt an Sperre
- Methoden
  - void await()
  - boolean await(long time, TimeUnit unit)
  - boolean awaitUntil(Date deadline)
    Auf Signal warten
  - void signal()
    Entspricht notify()
  - void signalAll()
    Entspricht notifyAll()
- Können fair sein

# synchronized



- synchronized legt lock/unlock Klammern um Block:
  - synchronized void doit() {*Block*} mit o.doit:
    monitorenter o;
    *Block*;
    monitorexit o;
  - synchronized(x) {*Block*}
    monitorenter x;
    *Block*;
    monitorexit x;
- Dynamische Klammerung über Zähler (siehe Def monitorenter und monitorexit)

# Ablauf von wait/notify/notifyAll



- Objekt O hat
  - Sperre (ca. Thread x Zähler)
  - Waitset (Menge von Threads)

- T macht wait() auf O
  - Voraussetzung: T hat Sperre (Zähler ist n)
  - T wird in das Waitset eingefügt
  - T wird blockiert (ist nicht zur Ausführung bereit)
  - n unlock-Operationen werden durchgeführt

# Ablauf von wait/notify/notifyAll

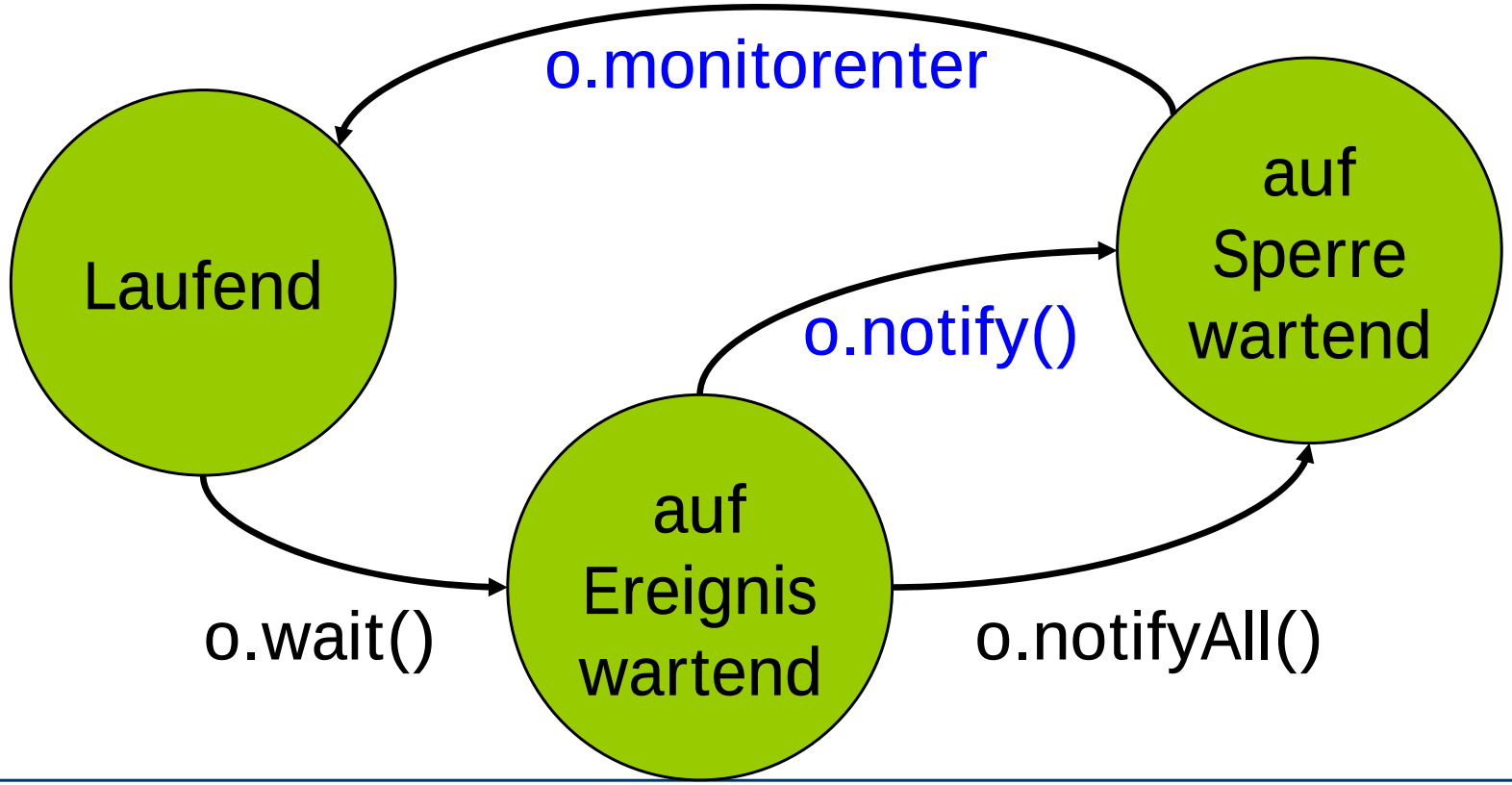


- Drei Ereignisse können T beeinflussen
  - ein notify wird auf O gemacht und T wird zur Benachrichtigung ausgewählt
  - ein notifyAll wird auf O gemacht
  - wait war mit einem Timeout t versehen und dieser ist abgelaufen
- Dann:
  - T wird aus waitset enfernt
  - T wird als bereit zur Ausführung markiert
  - T führt lock-Operation aus (in Konkurrenz mit anderen)
  - n-1 zusätzliche lock-Operationen werden ausgeführt
  - Ein „Resume T“ wird ausgeführt, wait() kehrt zurück, Zustand ist unverändert

# Ablauf von wait/notify/notifyAll

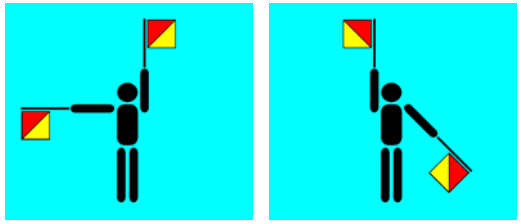


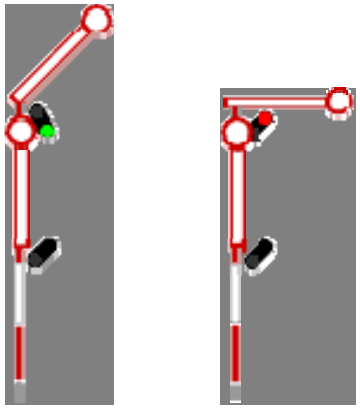
- Thread S, der notify() und notifyAll auf O macht, hat Sperre auf O
- T kann erst dann versuchen, Sperre zu erhalten wenn S Sperre aufgibt
- T steht dann in Konkurrenz mit anderen



# 3.3 Semaphore (Dijkstra 1968)

- sind einfache Synchronisationsobjekte, ähnlich wie Ereignisse, aber unabhängig von Monitoren:
  - (das) Semaphor: altes Flügelsignal – wie bei der Bahn
  - (engl.) semaphore: auch Flaggensignal bei der Marine





- Operationen:
  - P() steht für (holl.) passeren
  - V() steht für (holl.) verhogen oder vrijgeven
- Synonyme für P/V:
  - up/down ,
  - wait/signal

# 3.3.1 Abstrakte Definition



- Sei für einen Semaphor s
  - p die Anzahl der abgeschlossenen P-Operationen auf s
  - v die Anzahl der abgeschlossenen V-Operationen auf s
  - n ein Initialwert, der n „Passiersignale“ in der Semaphore einrichtet.
- Dann gilt die Invariante:

  $$p <= v + n$$

- Mit anderen Worten:
  - V kann stets ausgeführt werden,
  - P aber nur dann, wenn p < v+n

- Semaphor ist eigentlich ein Zähler für Passiersignale (als abstrakter Datentyp)

# Semaphore



- Beachte: P/V entsprechen einfachen Operationen request/release auf einem Ressourcen-Pool

- Achtung!
  Semaphore nicht mit Ereignissen verwechseln!
- Weitere Terminologie – entsprechend dem aktuellen Verhalten:
  - Boolesches Semaphor:
    - hat Invariante signals <= 1 ,
    - verhält sich wie Sperrvariable (3.1.5←)
  - privates Semaphor:
    - nur ein Prozess führt P-Operationen darauf aus
  - (Konsequenz: Frage nach Fairness ist irrelevant)

# 3.3.2 Sperren mit Semaphoren

- Ein Semaphor kann als Sperrvariable verwendet werden:

- der Effekt von
  `x.lock(); ..... ; x.unlock();`

- wird erzielt durch
  `mutex.P(); ..... ; mutex.V();`

- mit `Semaphore mutex = new Semaphore(1);`
- ( = Boolesches Semaphor)

# Semaphore in java.util.concurrent

Freie Universität Berlin

- Konstruktoren
  - Semaphore(int permits)
  - Semaphore(int permits, boolean fair)
- P Operation:
  - void acquire()
    Ein Freisignal nehmen (blockierend)
  - void acquire(int permits)
    permits Freisignale nehmen (blockierend)
  - int drainPermits()
    Alle gerade freien Freisignal nehmen (nicht blockierend)
  - boolean tryAcquire()
    Ein Freisignal nehmen (nicht blockierend)
  - boolean tryAcquire(int permits)
    permits Freisignale nehmen (nicht blockierend)
  - boolean tryAcquire(long timeout, TimeUnit unit)
    Ein Freisignal nehmen (blockierend mit Timeout)
  - boolean tryAcquire(int permits, long timeout, TimeUnit unit)
    permits Freisignale nehmen (blockierend mit Timeout)
- V Operation:
  - void release()
    Ein Freisignal freigeben
  - void release(int permits)
    permits Freisignale freigeben

# 3.3.4 Äquivalenz von Semaphoren und Monitoren

Freie Universität Berlin

- Semaphore können mit Monitoren implementiert werden (3.3.1←)
- Die Umkehrung gilt ebenfalls – siehe unten !
- Somit sind Semaphore und Monitore hinsichtlich ihrer Synchronisationsfähigkeiten gleich mächtig.

# 3.4.1 Vererbungsanomalien

Freie Universität Berlin

- *(inheritance anomalies)*
- Eine Vererbungsanomalie liegt vor, wenn

  - in einer Unterklasse einer synchronisierten Klasse *allein aus Synchronisationsgründen* das Umdefinieren *(overriding)* einer ererbten Operation erforderlich wird.

# 3.4.2.1 Autonome Operationen

Freie Universität Berlin

- Wünschenswert:
  Klasse kann **autonome Operationen** enthalten
- Syntax: (nicht Java!)
  - AutonomousMethodDecl = **AUTO** Identifier Block
- Eine solche Operation wird
  - nach Initialisierung *ohne expliziten Aufruf* automatisch gestartet
  - nach Beendigung automatisch *erneut* gestartet
- Es gibt hier
  - *weder* Modifizierer
  - *noch* Argumente
  - *noch* Ergebnis!

# Asynchrone Operation

Freie Universität Berlin

- Alternative:
  Asynchrone Operation **(nicht Java!),** d.h. Asynchronie ist Eigenschaft der Operation (und Gabelungsanweisung entfällt)
- Syntax:
  - AsynchronousMethodDecl = **ASYNC** MethodDecl

- ASYNC ist ein Modifizierer *(modifier).*
- Vorteile:
  - Semantik!
    Es ist i.a. *nicht gleichgültig*, ob eine Operation synchron oder asynchron abläuft.
  - Effizienz:
    Übersetzer kann Threading optimieren.

- Asynchrone Operationen *mit Ergebnis* **(nicht Java!):**

```
class Printer {
    ...
  public ASYNC Result print(File f) {
      ..... // do print f
  }
}
```

Verweistyp, nicht primitiver Typ!

- Aufruf in

```
Result result = printer.print(File f);
```

liefert einen Ergebnisvertreter *(„future“)* result

# Verzögerte Synchronisation



- Implizite Synchronisation mit der Operations-Beendigung bei erstem Zugriff auf den Ergebnisvertreter –
  verzögerte Synchronisation
  *(lazy synchronization, wait-by-necessity)*

```
Result result = printer.print(f);
...
... // do other business
...
Status s = result.getStatus();
    // await termination of print,
    //       then access print result
```

# Asynchrone Aufrufe in java.util.concurrent

Freie Universität Berlin

- Etwas asynchron aufrufbares ist ein Callable

```
interface java.util.concurrent.Callable<V> {
  V call()
}
```

  - Liefern Ergebnis zurück
  - Können Ausnahmen werfen

- Ein Callable kann einem ExecutorService zur Ausführung gegeben werden
  - `<T> Future<T> submit(Callable<T> task)`
  - `Future<?>  submit(Runnable task)`
  - `<T> Future<T> submit(Runnable task, T result)`

- Objekt vom Typ Future<T> wird sofort zurückgegeben

# Asynchrone Aufrufe in java.util.concurrent

Freie Universität Berlin

- Future ist das Resultat einer asynchronen Berechnung
- Lesezugriff darauf blockiert falls Ergebnis noch nicht vorliegt
- Schnittstelle Future<V>
  - V get()
    Wert auslesen (blockierend)
  - V get(long timeout, TimeUnit unit)
    Wert auslesen mit Timeout
  - boolean cancel(boolean mayInterruptIfRunning)
    Berechnung abbrechen
  - boolean isCancelled()
    Wurde abgebrochen?
  - boolean isDone()
    Liegt Ergebnis vor?

# Ablaufsteuerung



- Ablaufsteuerung *(scheduling)* =

- Zuteilung von *Betriebsmitteln (resources)*
  - wieder verwendbaren oder
  - verbrauchbaren

  an Prozesse, z.B.

  - Prozessor (wieder verwendbar),
  - Speicher (wieder verwendbar),
  - Nachrichten (verbrauchbar),
  - Ereignisse (verbrauchbar)

# 4.1 Prozeßprioritäten

- Def.:
  Priorität eines Prozesses =

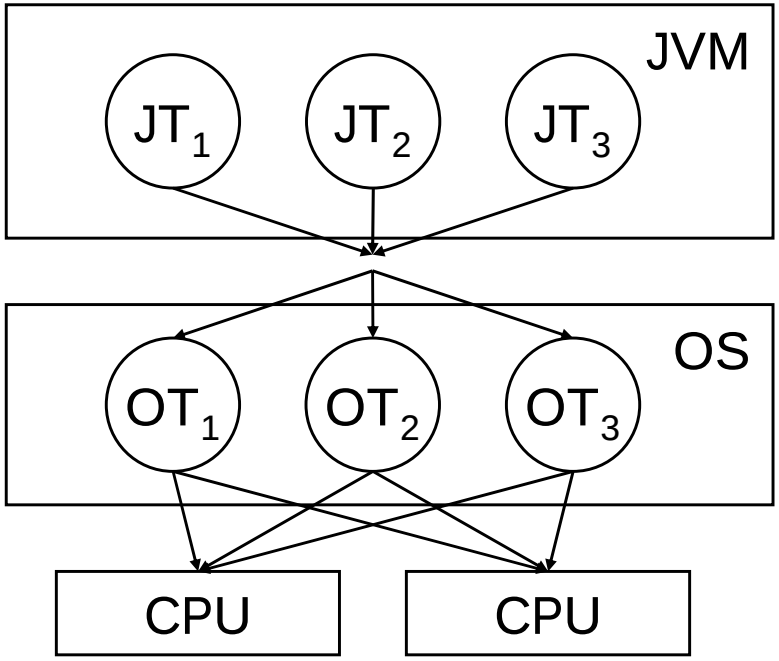
  einem Prozess *(task, thread, ...)* zugeordnetes Attribut, üblicherweise eine natürliche Zahl, das zur Entscheidung über die Zuteilung von Betriebsmitteln herangezogen wird.

- Priorität wird einem Prozess bei der Erzeugung (explizit oder implizit) zugeteilt und kann eventuell programmgesteuert verändert werden.

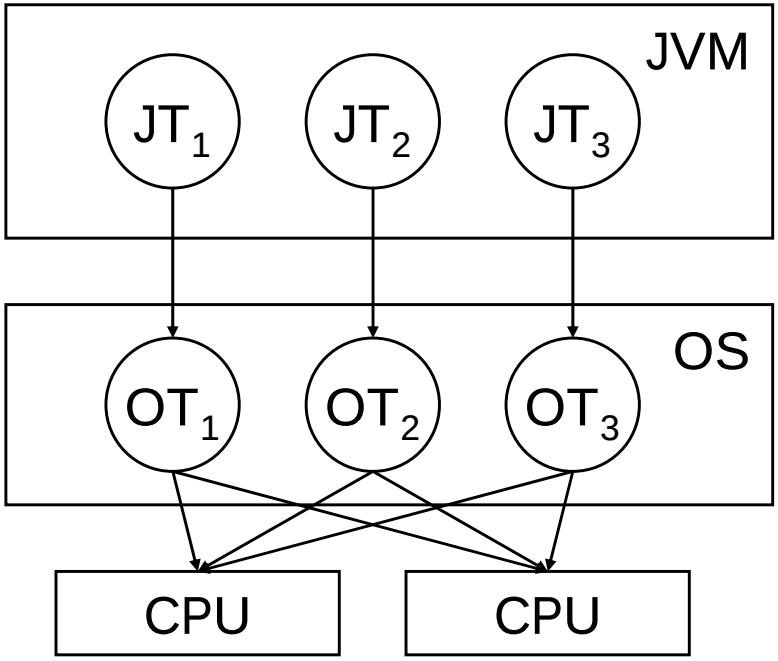
# Betriebssystem Threads in Java

- Moderne Java Implementierungen nutzen Threads die vom Betriebssystem angeboten werden

- Solaris – Pools




Win32 - Bindung



# Prioritätenvergabe vs. Synchronisation

- **Achtung 1:**
  Prioritätenvergabe ist kein Ersatz für Synchronisation!
- **Achtung 2:**
  Gefahr der **Prioritätsumkehr** *(priority inversion)*, z.B. bei Einprozessorsystem:
  - 3 Prozesse A, B, C mit abfallenden Prioritäten a,b,c
  - C betritt kritischen Abschnitt k, weckt B und wird von B verdrängt, d.h. belegt weiterhin k;
  - B weckt A und wird von A verdrängt,
  - A blockiert beim Eintrittsversuch in k;
  - solange B nicht blockiert, hat A keine Chance!
- *Lösungstechnik:* spezielle Sperroperationen, die für temporäre Prioritätserhöhung sorgen.

- *Problem:*
  Wenn *mehrere* Prozesse auf Betriebsmittelzuteilung bzw. auf ein bestimmtes Ereignis warten (mit **when**, wait, **P**, join, ...)

  **und**

  beim Eintreten des Ereignisses *nicht alle* aufgeweckt werden können/sollen, welche ?

- → Auswahlstrategie *(scheduling policy)*

# Fairness



- Def.: Eine Auswahlstrategie heißt fair, wenn jedem wartenden Prozess garantiert ist, dass ihm nicht permanent und systematisch andere Prozesse vorgezogen werden.
  (Auch: Von einer endlichen Anzahl nebenläufiger Prozesse ist in einem unendlichen Betriebsablauf jeder einzelne unendlich oft aktiv)
- Eine Auswahlstrategie heißt streng fair (strong fairness, compassion): Wenn ein Prozess blockiert, kann eine *obere Schranke* für die Anzahl der Prozesse angegeben werden, die ihm vorgezogen werden;
  - (Auch: Wenn ein Prozess unendlich oft ausführbereit ist, wird er unendlich oft aktiviert)
- Eine Auswahlstrategie heißt schwach fair (weak fairness, justice): sonst
  - (Auch: Wenn ein Prozess ausführbereit bleibt, wird er unendlich oft aktiviert)

# Prioritätsgesteuerte Semaphore

```
  public void V() { // this is the modified V
      mutex.P();
      count++;
      if(count<=0) {
        Semaphore ready = prios.rem();
        ready.V(); }
      mutex.V();
  }
}
```

- *Beachte:*
  1. Die Semaphore ready sind private Semaphore, d.h. bei ihnen blockiert jeweils höchstens ein Thread.
  2. Die kritischen Abschnitte (mutex) sind kurz genug, dass sie keinen Engpass darstellen.

  ➔ Daher ist es *irrelevant*, welche Auswahlstrategie von Semaphore praktiziert wird.

# Anforderungsbezogene Auswahl

Freie Universität Berlin

- Beispiele für effizienzorientierte Auswahlstrategien bei request(n)/ release(n):

- *SRF (smallest request first):*
  die *kleinsten* Anforderungen sollen vorrangig bedient werden
- *LRF (largest request first):*
  die *größten* Anforderungen sollen vorrangig bedient werden
- (jedenfalls nicht die, welche am längsten warten; weitere Alternativen sind möglich.)

- **Beispiel:**
  Alterungsmechanismus für SRF:
  - Auswahl nach *Rang*ordnung 1,2,3,...
    (hoher Rang = kleine Rangzahl)
  - *Rang* = Anforderung + *Malus*
  - *Malus* = Zeitpunkt des request, gemessen in Anzahl der begonnenen requests
  - Einem release wird genau dann stattgegeben, wenn die Anforderung erfüllt werden kann **und** die kleinste Rangzahl hat. (Bei Gleichrangigkeit zweier erfüllbarer Anforderungen...)

- ... drohen überall, wo synchronisiert wird

- Def. 1 (1.3🡐):

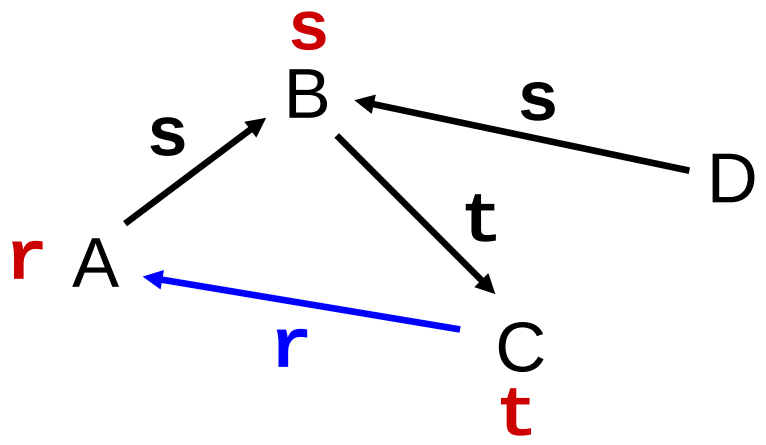
- Beim Ablauf eines nichtsequentiellen Programms liegt eine Verklemmung *(deadlock)* vor, wenn es Prozesse im *Wartezustand* gibt, die durch *keine mögliche Fortsetzung* des Programms daraus erlöst werden können.

- Def. 2:
  Eine Verklemmung heißt *deterministisch*, wenn sie bei jedem möglichen Ablauf – mit gleichen Eingabedaten – eintritt (andernfalls *nichtdeterminstisch*).
- Def. 3:
  Eine Verklemmung heißt *total* oder *partiell,* je nachdem, ob alle oder nicht alle Prozesse verklemmt sind.
- 3 Alternativen für den Umgang mit Verklemmungen:
  1. **Erkennung** *(detection)* + Auflösung
  2. **Umgehung / Vermeidung** *(avoidance)*
  3. **Ausschluß** *(prevention)*

# Bsp. mit 4 Prozessen A,B,C,D und 3 Ressourcen r,s,t

Freie Universität Berlin

| Zeit | A | B | C | D |
|---|---|---|---|---|
| 0 | lock(r) | lock(s) | lock(t) | |
| 1 | lock(s) | | | |
| 2 | | | | lock(s) |
| 3 | | lock(t) | | |
| 4 | | | lock(r) | |



# Verklemmungserkennung



- Eine Verklemmung liegt genau dann vor, wenn der Anforderungsgraph einen Zyklus enthält.

- **Verklemmungs-Erkennung:**
  Ständige Überwachung des Anforderungsgraphen:
  bei jedem Hinzufügen einer Kante P→Q prüfen, ob damit ein Zyklus geschlossen wird, d.h. ob es einen *Weg von Q nach P* gibt.

- Ohne Backtracking, weil jede Ecke höchstens eine divergierende Kante hat !

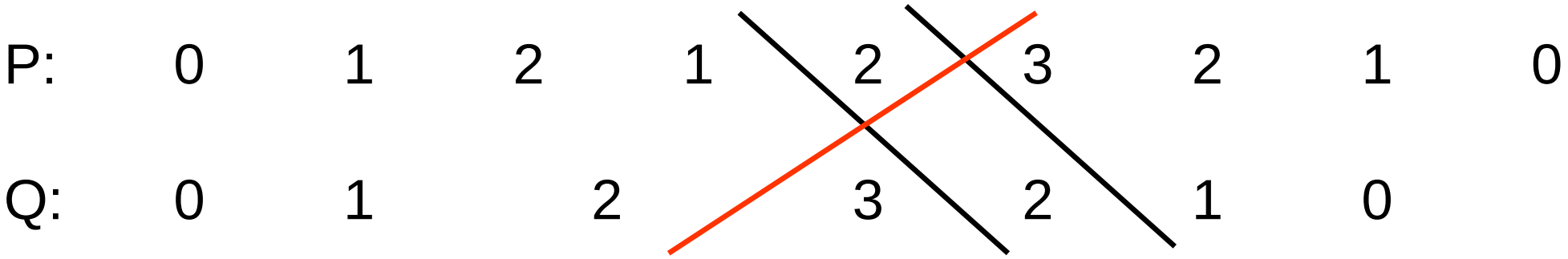
# Verklemmungsauflösung



- **Verklemmungs-Auflösung** durch Wahl eines Opfers, dem gewaltsam die Ressource entzogen wird – mit Konsequenzen:
  - Bei Datenbank-Transaktion:
    Transaktion abbrechen und neu starten
  - Falls Sicherungspunkt vorhanden (Checkpointing):
    Prozess zurücksetzen zum letzten Checkpoint
  - sonst:
    Prozess abbrechen

# 4.3.2 Vermeidung am Beispiel 1/N

- Beispiel:
  - 2 Prozesse P,Q mit $n = 4$ Exemplaren eines Ressourcen-Typs
  - Vor.:
    Maximalbedarf jedes Prozesses ist bekannt und ist $\leq n$.



# Sicherheit

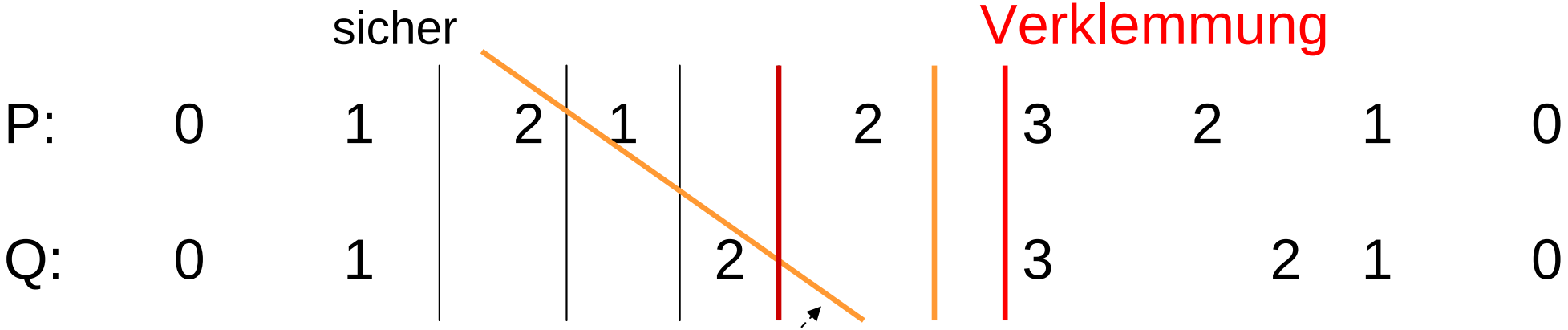
- **Def.:**
  Ein Zustand in einem Ablauf eines 2-Prozess-Systems heißt **sicher** *(safe)*, wenn in ihm für mindestens einen nichtblockierten Prozess der Maximalbedarf befriedigt werden kann.

- **Feststellung 1:**
  Ein direkter Übergang von einem sicheren Zustand in einen Verklemmungszustand ist nicht möglich.
  - weil damit nach dem ersten Prozess auch der zweite Prozess blockieren müsste – was der Sicherheit widerspricht

- **Feststellung 2:**
  Verklemmungen werden vermieden, wenn das System stets in einem sicheren Zustand gehalten wird.

# Sicherheit

- Maximalforderung
  - von P: 3
  - von Q: 3




**unsicher,**
d.h. Anforderung P: 1 → 2
wird nicht stattgegeben

- Bereits dieser Zustand würde nicht zugelassen werden, d.h. der Anforderung von Q (1 → 2) würde nicht stattgegeben werden
  – obwohl nicht notwendig Verklemmung eintritt.

# Bankiers-Algorithmus (banker‘s algorithm) [Dijkstra 65]

Freie Universität Berlin

- beantwortet die Frage nach der Sicherheit für *beliebig viele* Prozesse:
- Ein Zustand heißt **sicher**,
  - wenn in ihm mindestens ein Prozess unter Gewährung seines Maximalbedarfs zum Abschluss gebracht werden kann
  - **und** mit den damit freiwerdenden Ressourcen ein weiterer Prozess zum Abschluss gebracht werden kann
  - **und** mit den damit freiwerdenden .....
  - **usw.**

- *Präzisierung:*

- Zur Erinnerung: *Anforderungsgraph* (4.3.1←)

- **Behauptung:**
  - Die Betriebsmittel seien linear geordnet, und die Prozesse tätigen ihre Anforderungen nur in der Reihenfolge dieser Ordnung
    (Prozess im Besitz von r fordert s nur dann, wenn s > r).
  - *Dann bleibt der Anforderungsgraph zyklenfrei.*

# Beispiel

```
volatile boolean lock1 = false;
volatile boolean lock2 = false;
```

Prozess 1

```
do
        lock1 = true;
        lock1 = !lock2;
while (!lock1);
```

kritischer Abschnitt

```
lock1 = false;
```

Prozess 2

```
do
        lock2 = true;
        lock2 = !lock1;
while (!lock2);
```

kritischer Abschnitt

```
lock2 = false;
```

- *„busy waiting“* (normalerweise verpönt)

# Strukturtreue Implementierung

Prozess ⟷ ? ⟷ Prozessor

Daten ⟷ ? ⟷ Speicher

- Def.:
  Strukturtreue Implementierung:

1. *Mehrprozessorsystem (multiprocessor)*
   mit privaten und gemeinsamen Speichern für private und gemeinsame Variable
2. *Parallele*,
   d.h. echt simultane, Ausführung aller Prozesse

# Beispiele für *nicht* strukturtreue Implementierung

Freie Universität Berlin

- Mehrprozessorsystem *ohne private* Speicher

➔ alle Daten im zentralen Arbeitsspeicher

- Weniger Prozessoren als Prozesse, z.B. Einprozessorsystem

➔ quasi-simultane, verzahnte Ausführung im Mehrprozessbetrieb *(multitasking, multiprogramming)*

- Mehrrechnersystem *ohne gemeinsamen* Speicher *(multicomputer)*

➔ nachrichtenbasierte Prozessinteraktion

- Fragestellung:
  Wie kann die Sperroperation lock (3.1.5) implementiert werden?

```
MONITOR Lock {// setzt bereits Sperrmechanismus voraus !
  private boolean lock = false;
  public void lock() when !lock { lock = true; }
  public void unlock() { lock = false; }
}
```

# Sperrsynchronisation



- *Idee:*
  - Wiederholt lock prüfen – solange lock gesetzt ist;
  - sobald lock nicht gesetzt ist, selbst setzen.
- *Aber:*
  naive Umsetzung dieser Idee führt nicht zum Erfolg.

- Hardware bietet Unterstützung: unteilbare Instruktionen

- *Beispiel:*
  Instruktion TAS *(test-and-set)* für unteilbares Lesen/Schreiben im Arbeitsspeicher

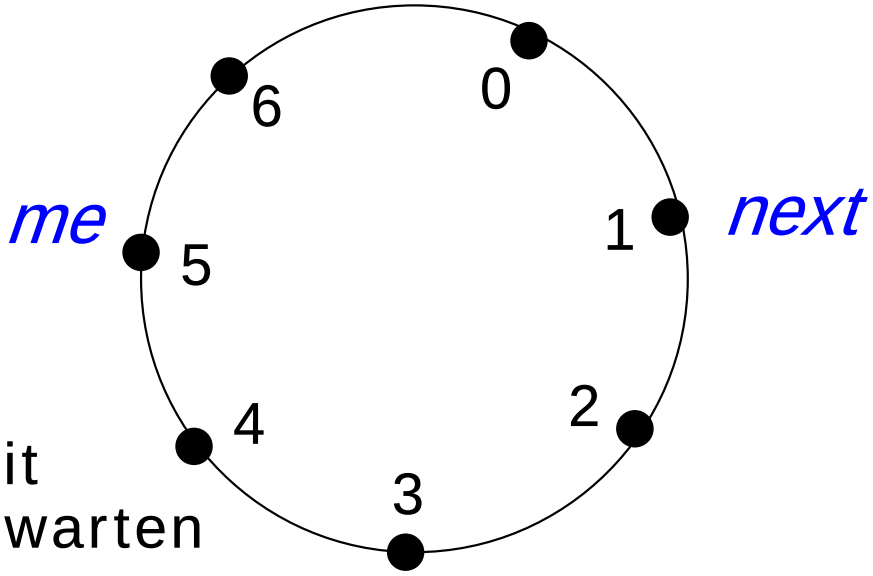
```
boolean TAS(VAR boolean lock) {
<       boolean result = lock;
        lock = true;            >
        return result;
}
```

# 5.1.2 Sperren ohne spezielle Instruktionen



- Nur mit atomarem Lesen bzw. Schreiben natürlicher Zahlen (Dekker, Dijkstra, Habermann, Peterson, .....)

- n Prozesse
- Gemeinsame Variable:
  - *next* = 0,1,...,n-1, anfangs 0
  - next ist der Prozess, der als nächster den Lock hat
  - Solang ein Prozess nicht der mit der Nummer next ist, muss er warten
- Private Variable:
  - *me* = Prozessnummer
  - Zustand = *idle, eager, critical,* anfangs *idle*
  - Zustandsübergänge immer idle > eager > critical > idle



# Sperren ohne spezielle Instruktionen

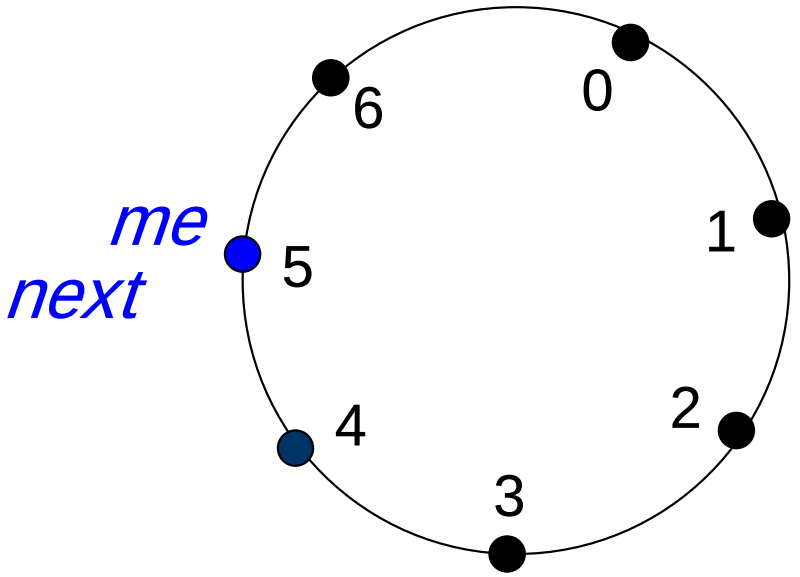
unlock:

set *next* to *me* ⊕ 1,

set *idle*

- 4 gibt Sperre auf
- next wird zu 5
- 5 hat Sperre

- Zusammengefasst
  - Prozesse sind geordnet
  - versuchen Sperre zu bekommen, wenn das keiner „vor“ ihnen tut
  - Falls das nebenläufig passiert entsteht Konflikt
  - Dieser wird aufgehoben durch Ordnung beim nochmaligen Versuch



- *Voraussetzung:*
  (zunächst) nur 1 Prozessor

- *Idee:*
  Prozessor ist *abwechselnd* mit der Ausführung der beteiligten Prozesse befasst
- („quasi-simultane“ Ausführung, *processor multiplexing*)

# Koroutine (coroutine)



- Interaktion mit *exchange jump* RESUME statt *call/return:*
- A: RESUME B bewirkt
- Sprung von A-Inkarnation nach B-Inkarnation an denjenigen Punkt, an dem die B-Inkarnation das letzte Mal (mit RESUME) verlassen wurde
- Jede Inkarnation „kennt“ ihren eigenen letzten RESUME-Punkt.

# Koroutinen in Modula-2

Freie Universität Berlin

- Modula-2
  - Imperative Programmiersprache
  - Pascal plus Module (Schnittstelle und Implementierung)
- Modula-2 hat Standardbibliotheken, die Module
- Im Modul SYSTEM sind Koroutinen vorhanden
- Basierend auf den Koroutinen können in Modula-2 Ablaufsteuerung und Synchronisationsmechanismen implementiert werden
- siehe auch www.modula2.org
  - http://www.modula2.org/reference/isomodules/isomodule.php?file=PROCESSE.DEF
  - http://www.modula2.org/reference/isomodules/isomodule.php?file=SEMAPHOR.DEF

# 5.3 Koroutinen in Java

- Java kennt keine Koroutinen
  - Man kann sie aber implementieren

- Idee:
  - Es gibt immer einen aktiven Thread – die Koroutine
  - Resume() auf einer Koroutine „schaltet um“
  - „alte“ Koroutine macht wait()
  - „neue“ Koroutine bekommt Signal per notify()

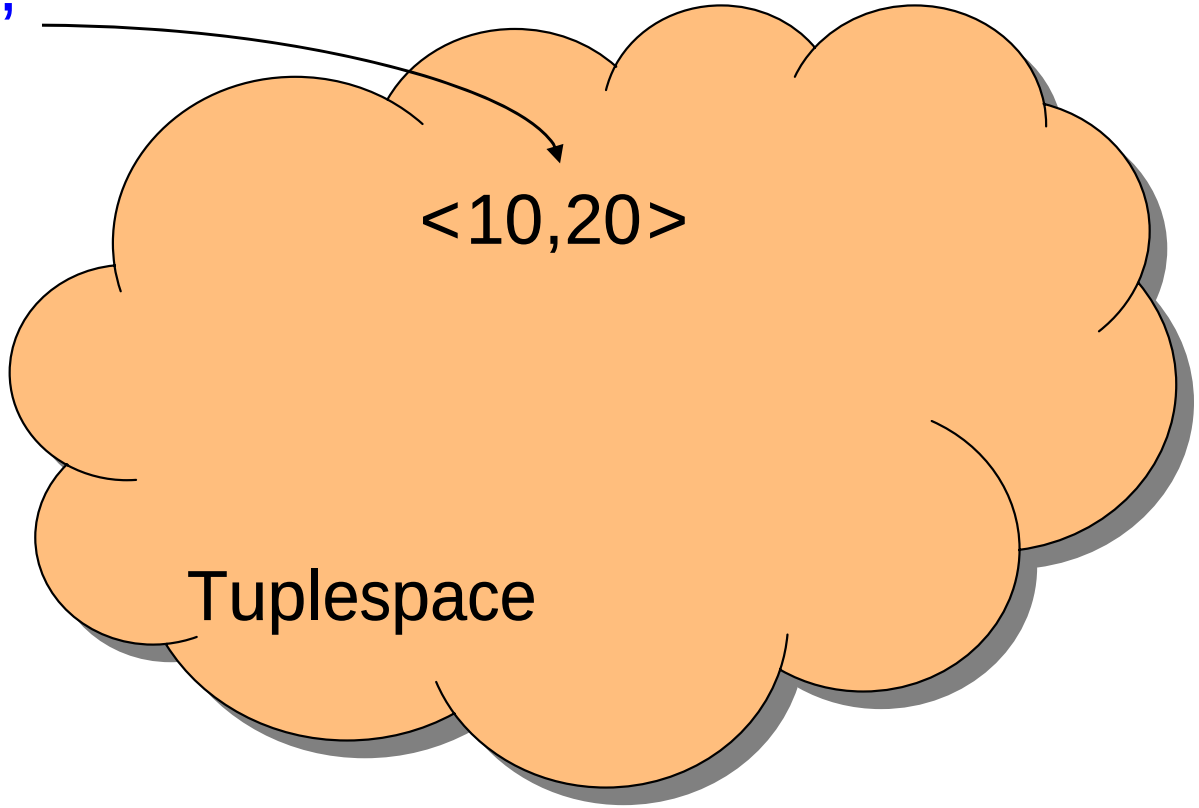
- Basiert auf [Ole Saustrup Kristensen http://dspace.ruc.dk/handle/1800/764?mode=full]

- Interaktion von Prozessen über *zentralen Umschlagplatz* für alle zwischen Prozessen ausgetauschten Daten:
  **Tupelraum** *(tuple space)*
  (Carriero/Gelernter, Yale 1985)

- = Multimenge von Tupeln beliebiger Typen

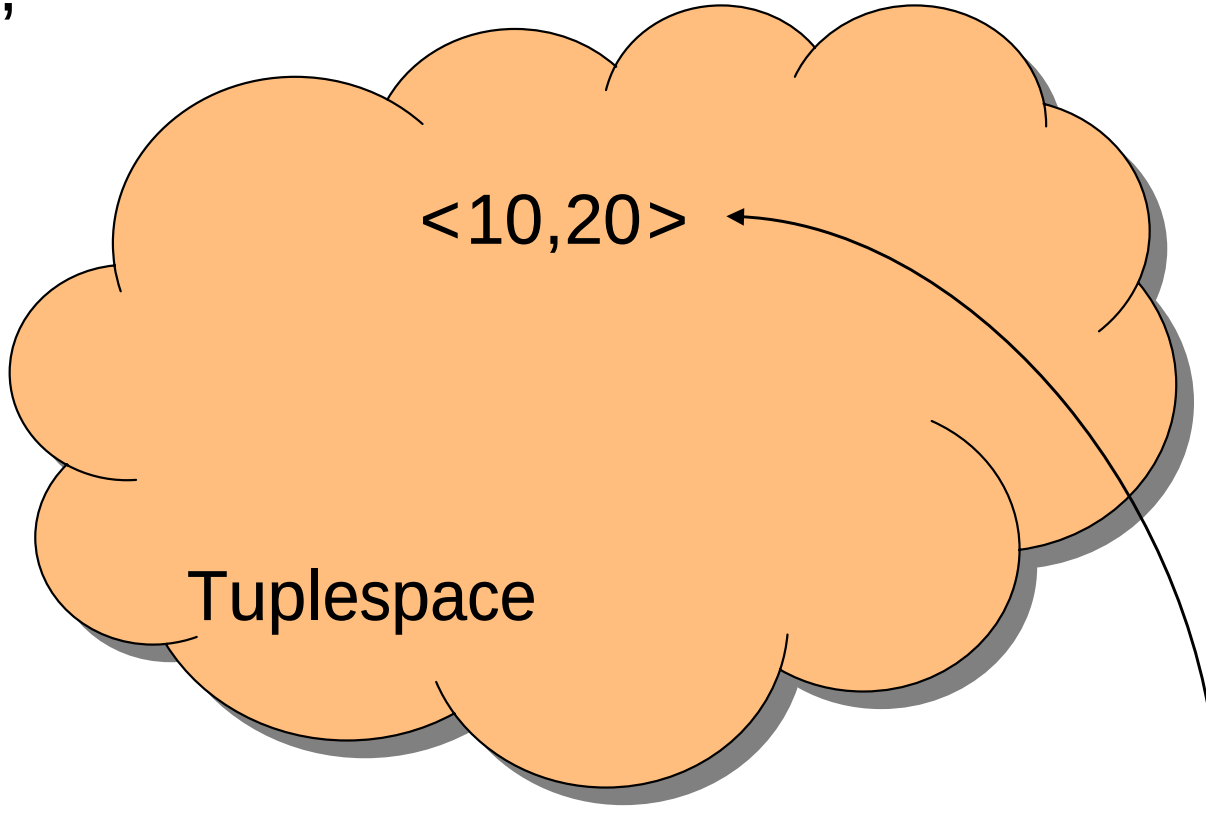
# Indirekte Interaktion

out(10,20);
in(?result);



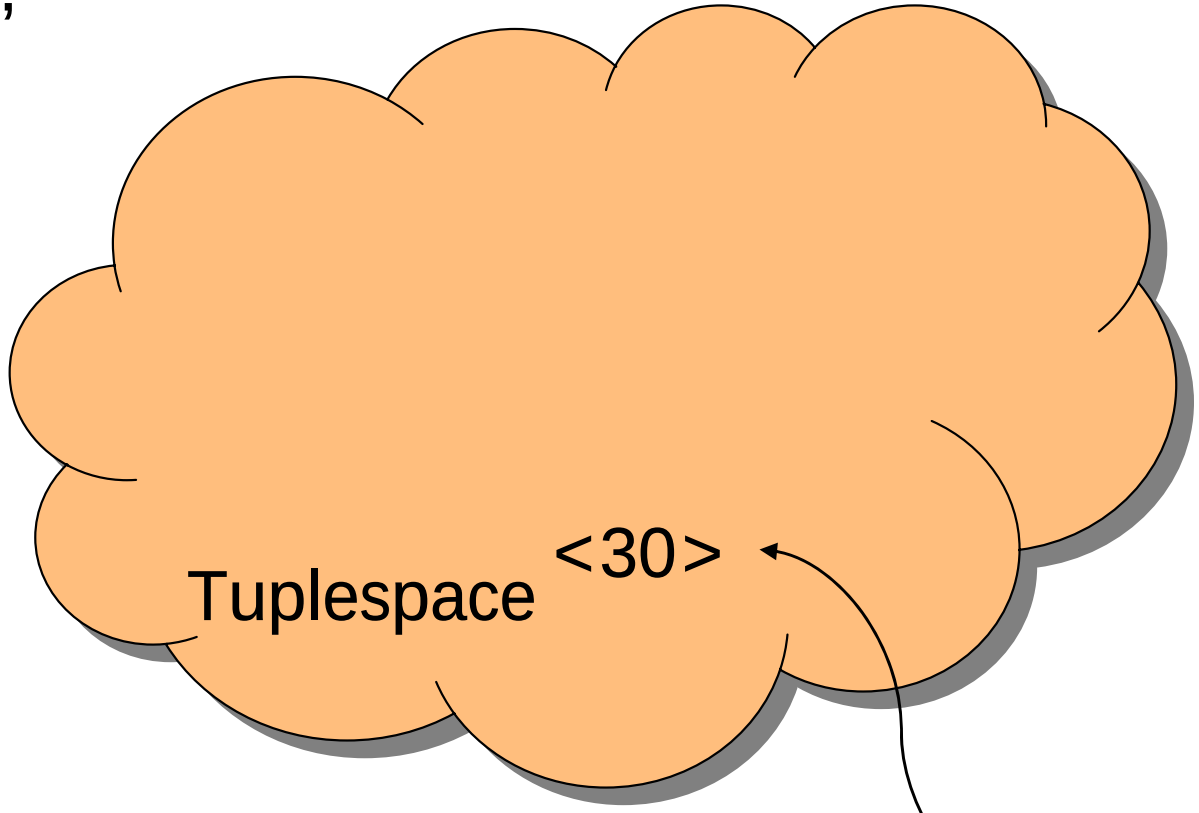

in(?a,?b);
out(a+b);

```
out(10,20);
in(?result);
```

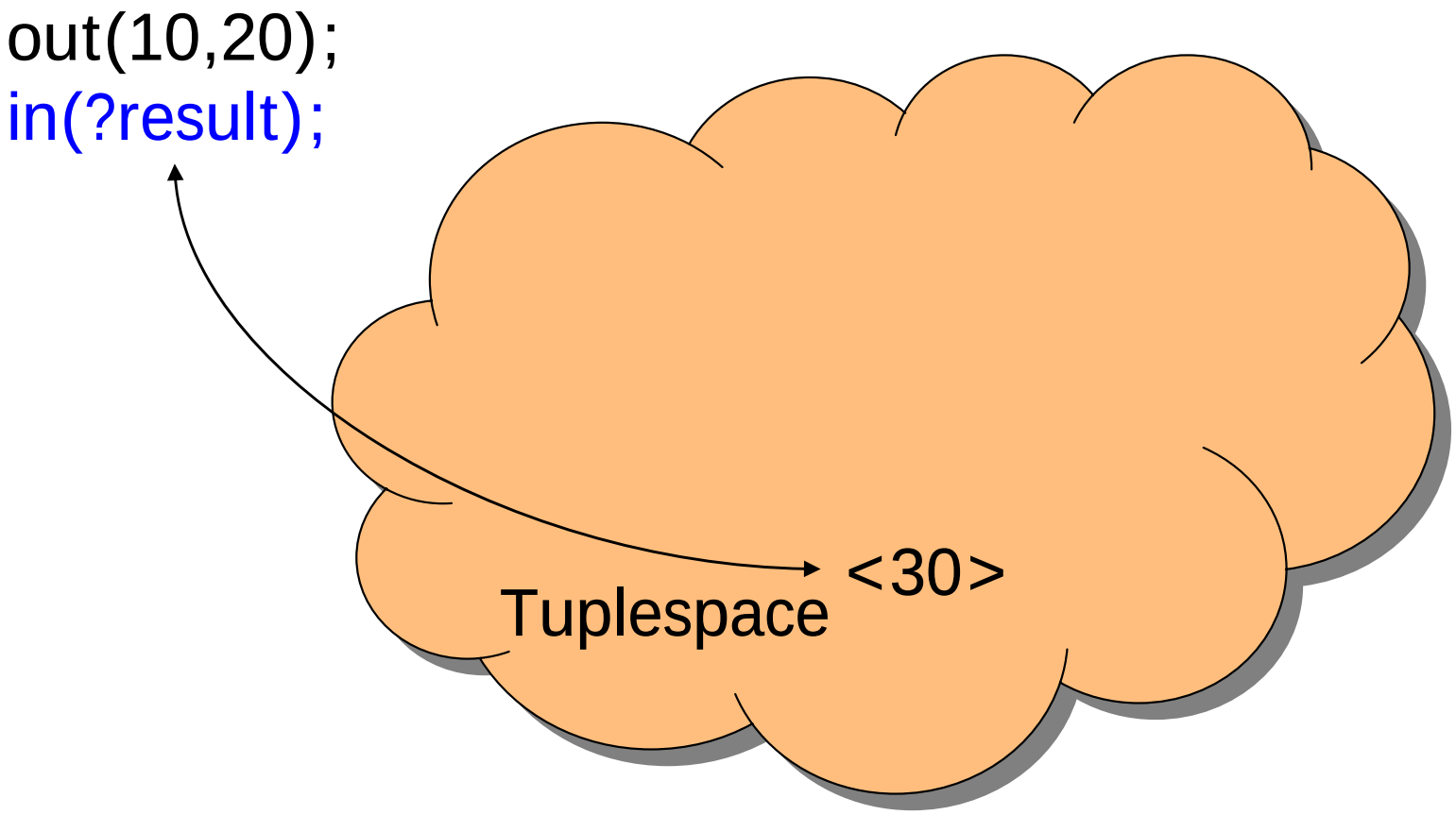



```
in(?a,?b);
out(a+b);
```

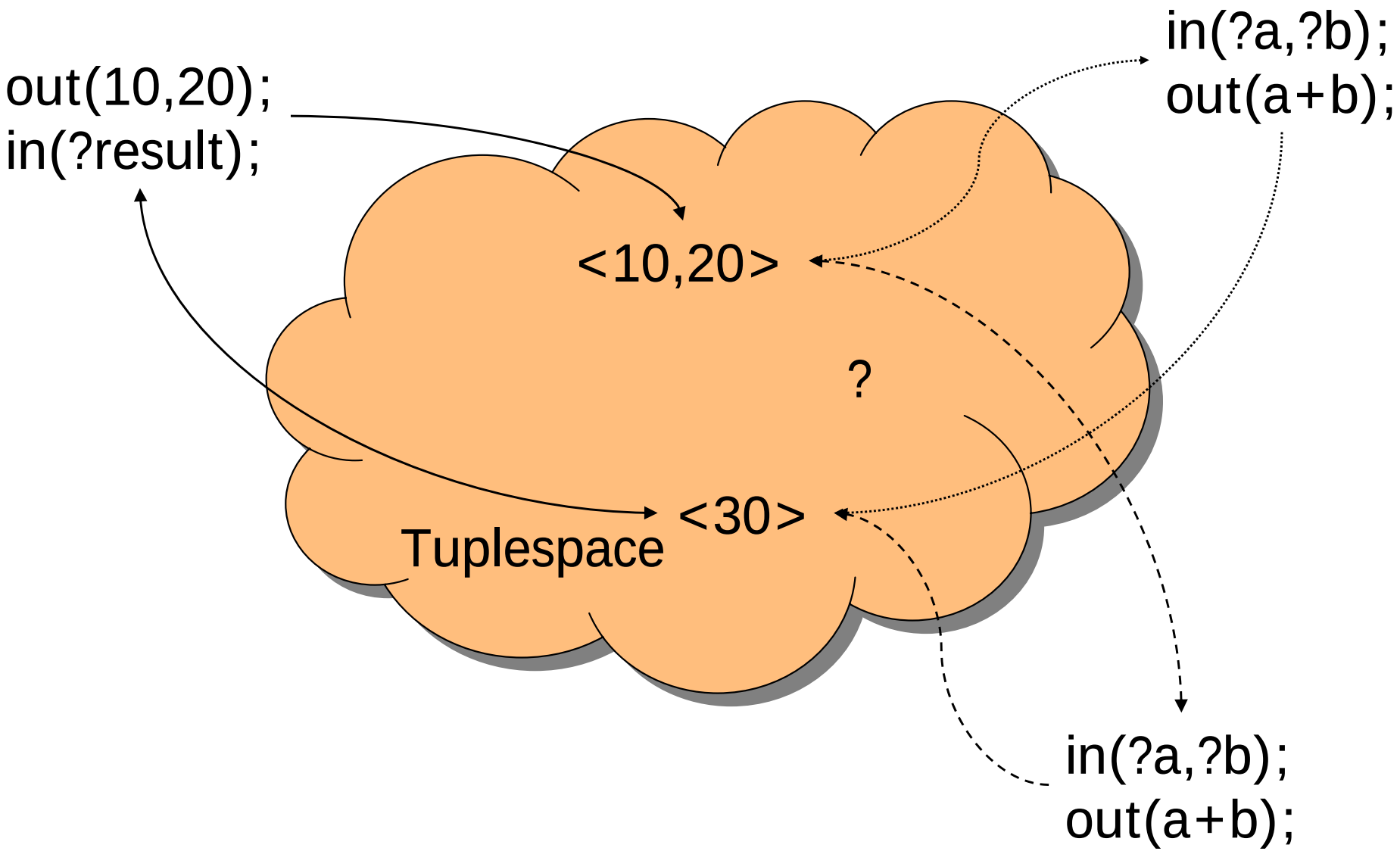
```
out(10,20);
in(?result);
```



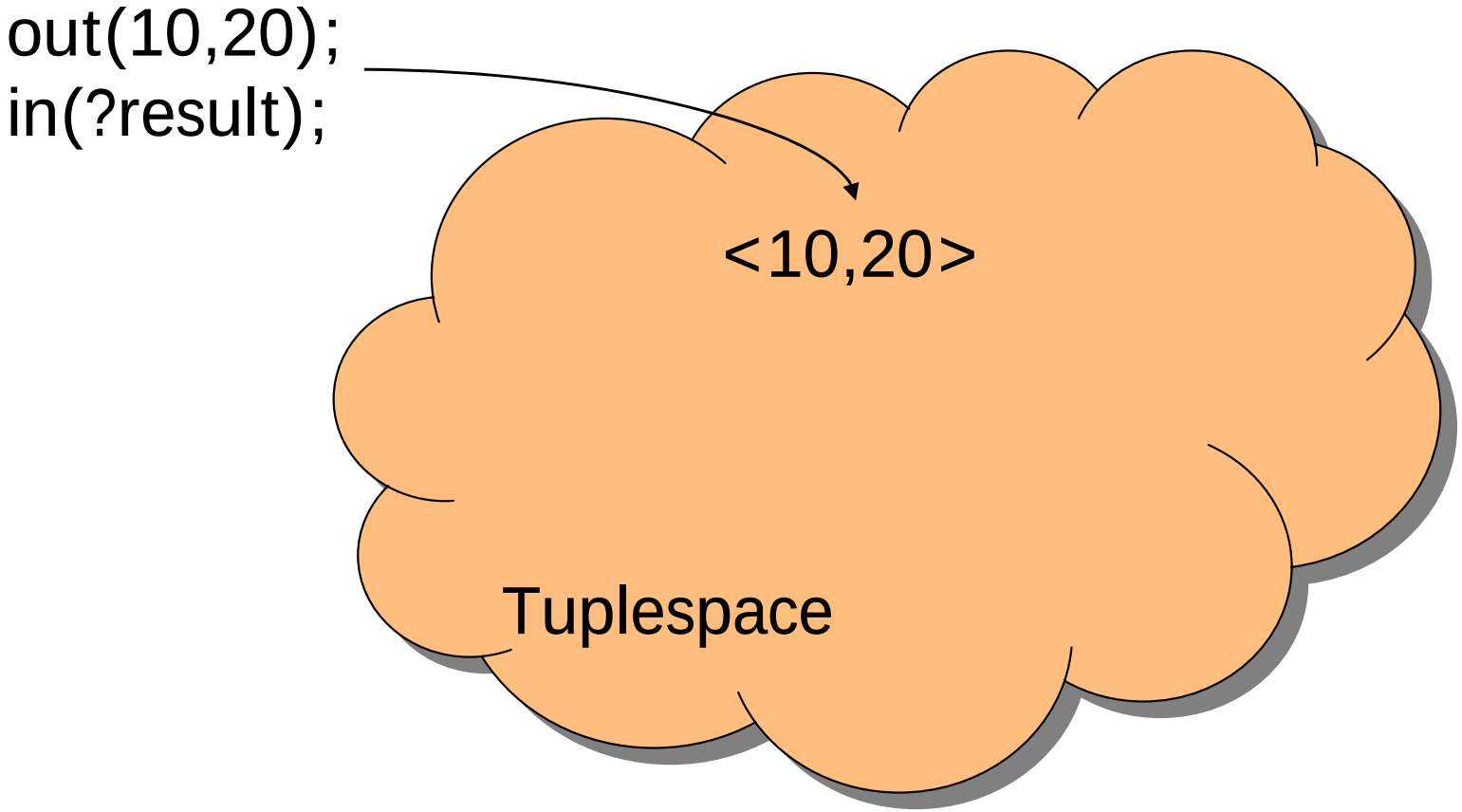
# Indirekte Interaktion



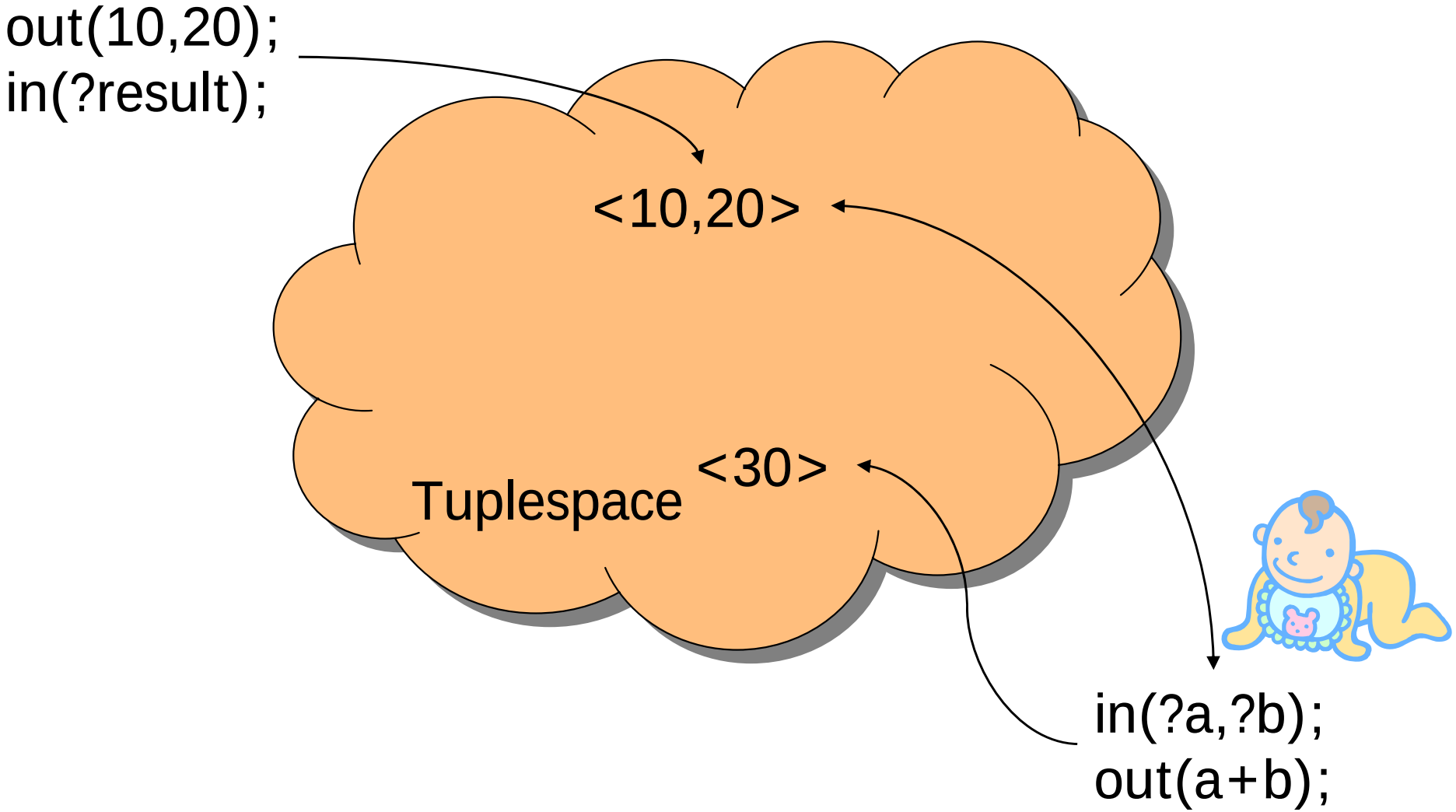
# Anonyme Interaktion



# Unterschiedliche Lebensdauer



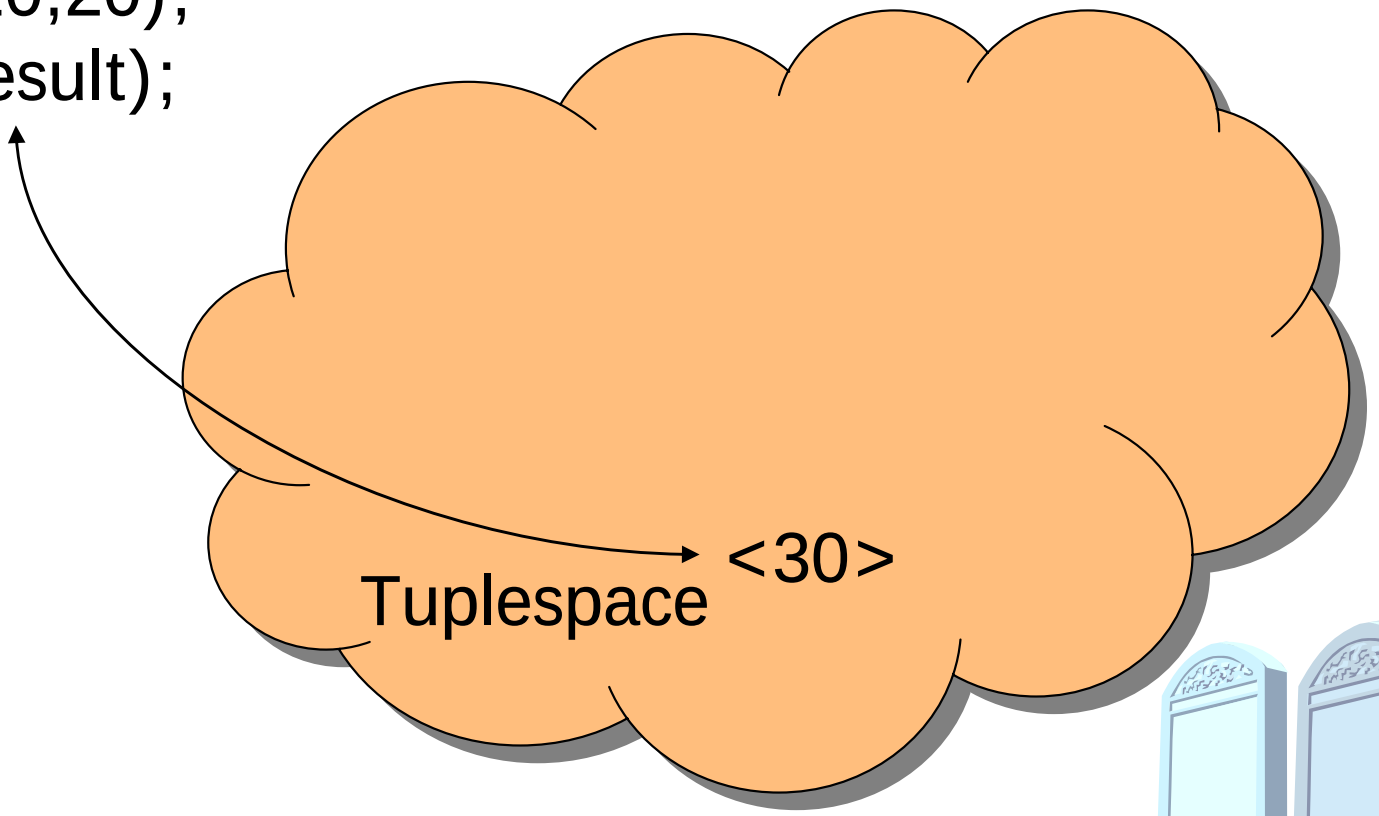
# Unterschiedliche Lebensdauer

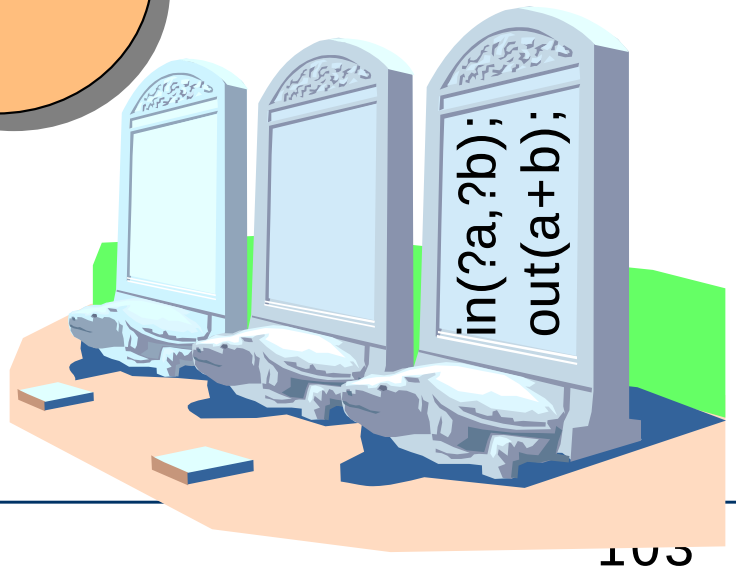


# Unterschiedliche Lebensdauer

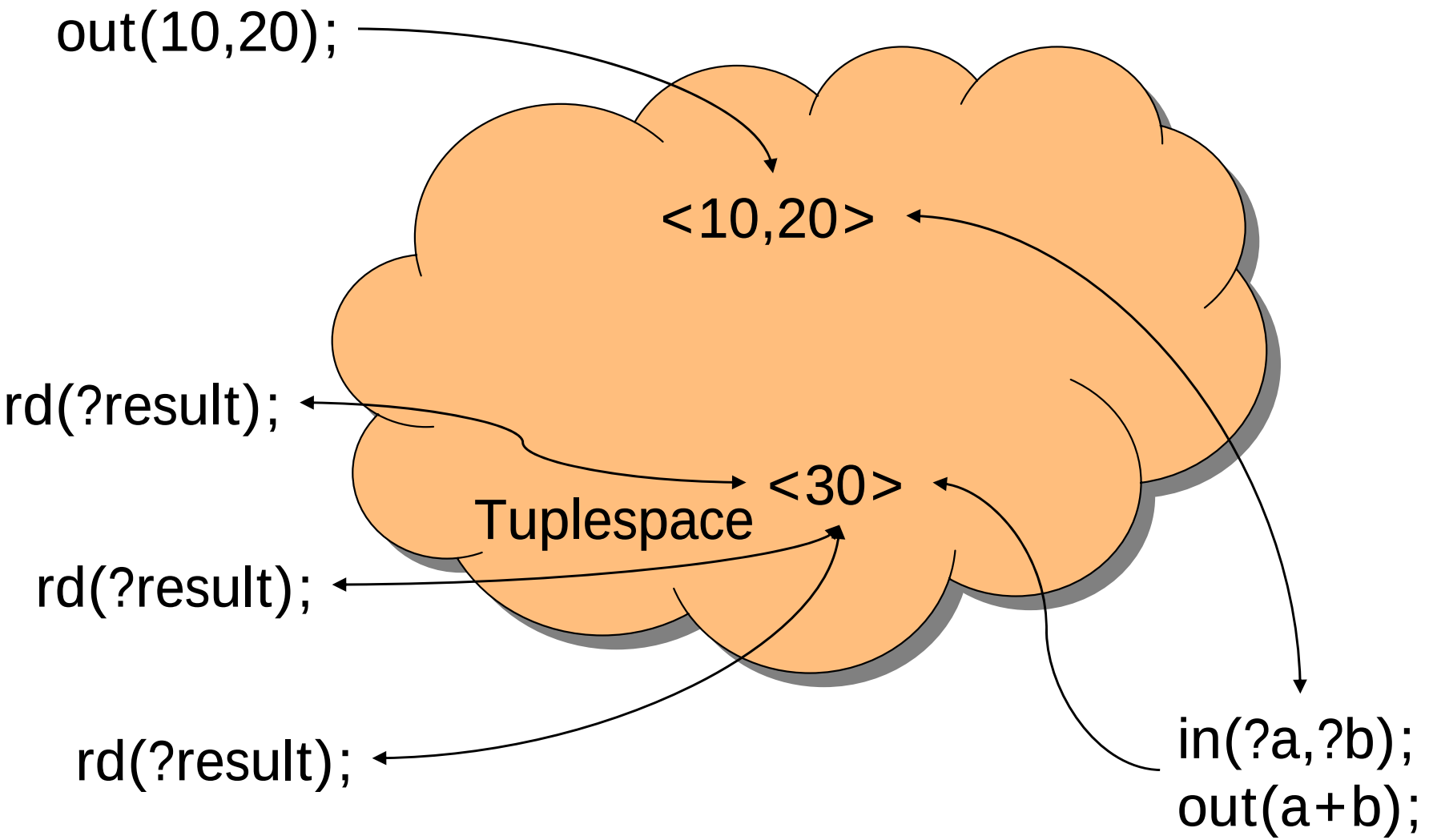


out(10,20);
in(?result);



# Mehrparteieninteraktion



# Programmiermuster: Semaphore

Freie Universität Berlin

- Semaphore
  - V-Operation: out("sem")
  - P-Operation: in ("sem")
  - Initialisierung: out("sem") n Mal wiederholen
  - P(2)-Operation: in("sem"); in("sem");
    - Verklemmungsgefahr!
  - P("sem1","sem2") nicht direkt in Tupel realisierbar
- Kann auch verteilt sein, wenn ein verteilter Tuplespace verwendet wird

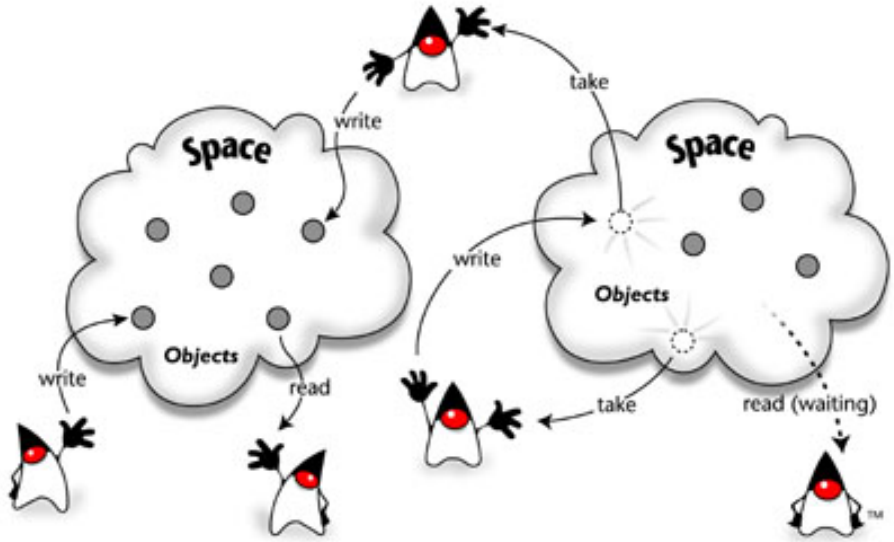
# LighTS



- LighTS (http://lights.sourceforge.net/) ist eine leichtgewichtige Linda Implementierung in Java
  - Nicht verteilt
  - Mit einigen Erweiterungen
  - Basis für eigene Experimente

- = Java-Objekte mit Tupelraum-Schnittstelle und Tupel-Objekten, die als *(tiefe) Kopien* in den / aus dem Tupelraum gelangen

- Änderungen/Erweiterungen gegenüber Linda:
  - Objektorientierung
  - mehrere Tupelräume
  - Transaktionen
  - Ereignisse




- http://java.sun.com/products/javaspaces
- http://wwws.sun.com/software/jini/specs/jini1.2html/js-title.html

# Interaktion über Nachrichten



- Bisherige Synchronisationsmittel nehmen gemeinsamen Speicher und gemeinsame Objekte an
- Aber: Es gibt auch Rechnerarchitekturen bei denen Prozesse *keinen gemeinsamen Speicher* haben, in dem sie gemeinsame Objekte unterbringen könnten
- Z.B.
  - Parallelrechner mit verteiltem Speicher *(distributed-memory parallel computer)* (auch „Mehrrechnersystem“ – *multi-computer*)
  - Rechnernetz *(computer network)* mit enger Kopplung *(cluster)* oder LAN oder WAN

# Verteiltes System

- **Def.:**
  Verteiltes System *(distributed system)* =
  System von Prozessoren, Prozessen, Threads, ...,
  die mangels gemeinsamen Speichers *nicht über Datenobjekte, sondern über Nachrichten* interagieren
  (Gegensatz: zentralisiertes System)
- **Beachte:**
  Die Klassifikation eines Systems als *verteilt* oder *zentralisiert* ist u.U. abhängig von der Abstraktionsebene, auf der das System betrachtet wird
  (Beispiel: System lokal kommunizierender Prozesse)

# Kommunikationsoperationen



- Prozesse kommunizieren unter Verwendung von Kommunikationsoperationen *(communication primitives)*
  - send
    Nachricht versenden (auch write, ...); die Nachrichtenquelle heißt Sender *(sender)* oder Produzent *(producer)*
  - recv
    Nachricht entgegennehmen (auch read, ...); die Nachrichtensenke heißt Empfänger *(receiver)*, Verbraucher *(consumer)*
  - in vielen möglichen Varianten (s.u.)

# Klassifizierung

- Softwaretechnische Klassifizierung:
  1. Datenfluss-Architektur *(dataflow)* (➔6.1):
     Filter *(filter)* wandelt Eingabedaten in Ausgabedaten, *weiß nichts* von seiner Umgebung
  2. Dienst-Architektur *(client/server)*:
     Klienten *(clients)* beauftragen Dienstanbieter *(servers)*, *erwarten bestimmte* Funktionalität und Antwort
  3. Ereignisbasierte Systeme *(event-based systems)* (➔6.3):
     Abonnenten *(subscribers)* sind an bestimmten Ereignissen interessiert, über deren Eintreten sie von den auslösenden Ereignisquellen *(publishers)* benachrichtigt werden
  4. Verteilte Algorithmen *(distributed algorithms)*:
     gleichberechtigte Partner *(peers)* kooperieren, *setzen* vereinbartes Verhalten *voraus*

- Prozess arbeitet als Filter *(filter)*, d.h.
  - *empfängt* Daten, evtl. über verschiedene Kanäle,
  - *berechnet* daraus neue Daten,
  - *versendet* diese, evtl. über verschiedene Kanäle.

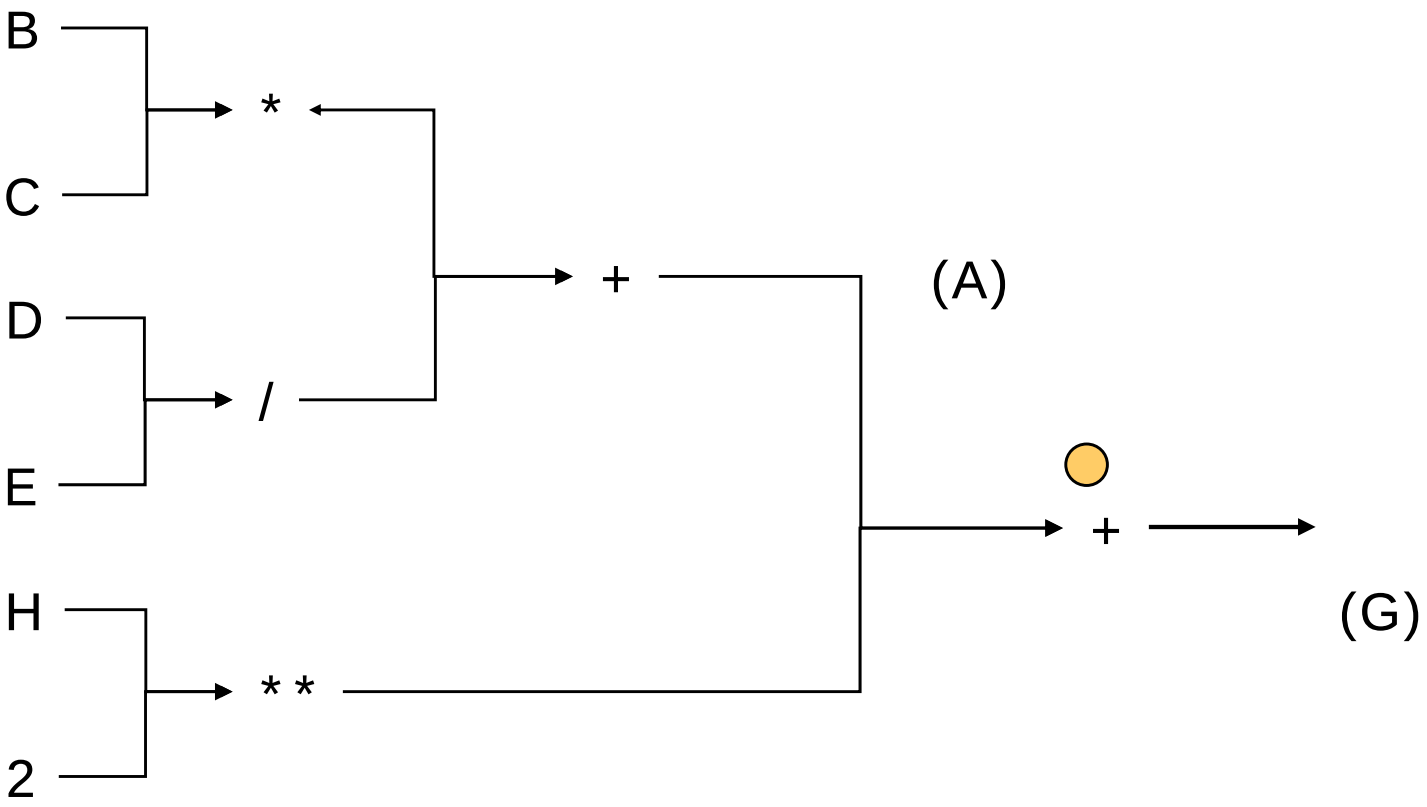
- Nicht notwendig, aber *typisch* ist:
  - Kommunikationspartner bzw. -kanäle sind *formale Parameter* des Prozess-Codes: Ports *(ports)*
  - Parameter ist *entweder* Eingabe- *oder* Ausgabekanal *(input port, output port)*

# Berechnung nach Datenabhängigkeiten

- Berechnungsausschnitt:

  A := B*C + D/F;

  G := H**2 + A;



# Programmieren mit Datenfluss-Graph



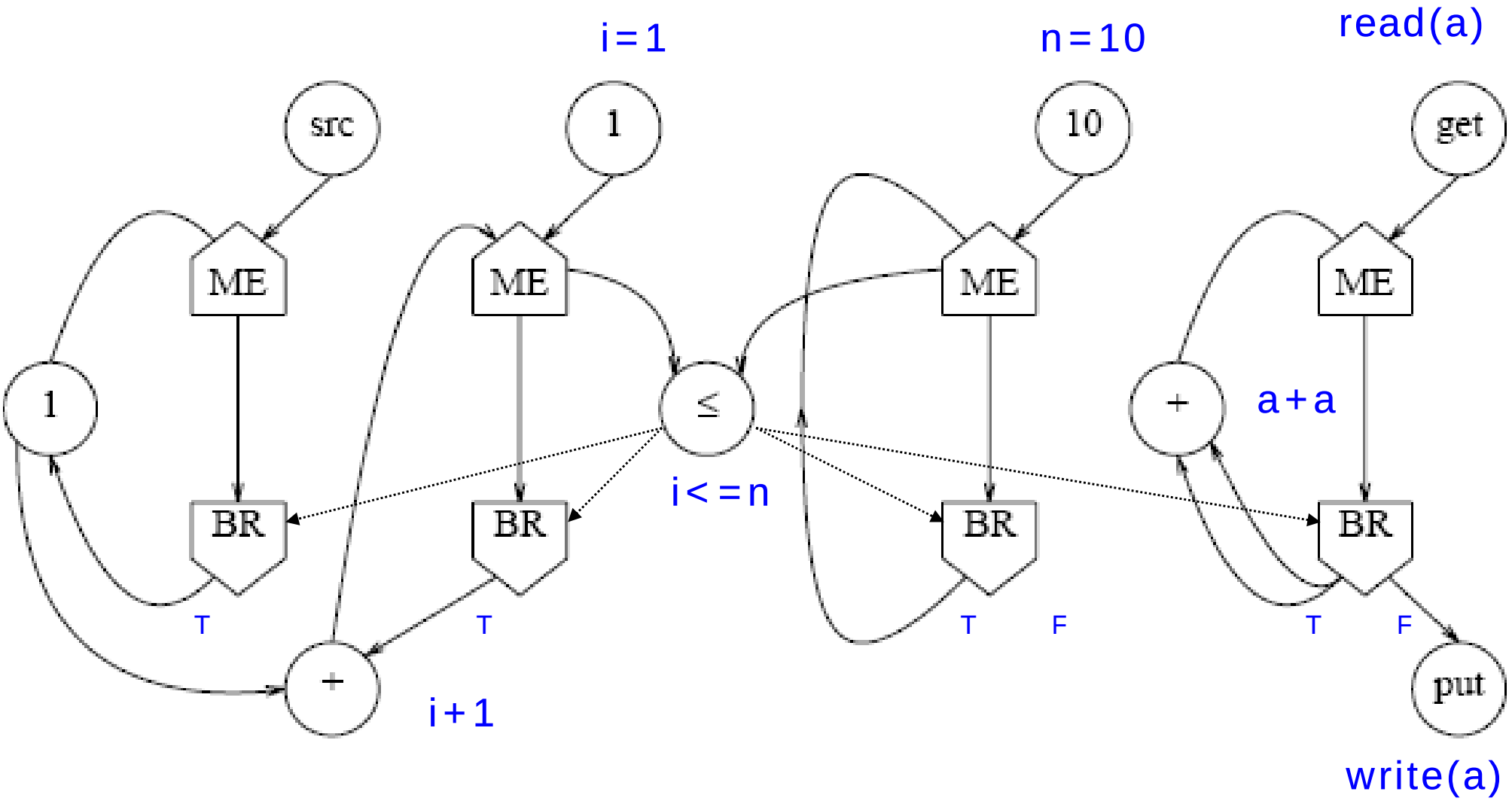
- *Einfache Zählschleife*

```
read (a);
n = 10;
for i:=1 to n
  a := a + a
end
write (a);
```

- Transformiert:

```
read (a);
n := 10;
i := 1;
while (i <= n)
  a := a + a;
  i := i + 1;
end
write (a);
```

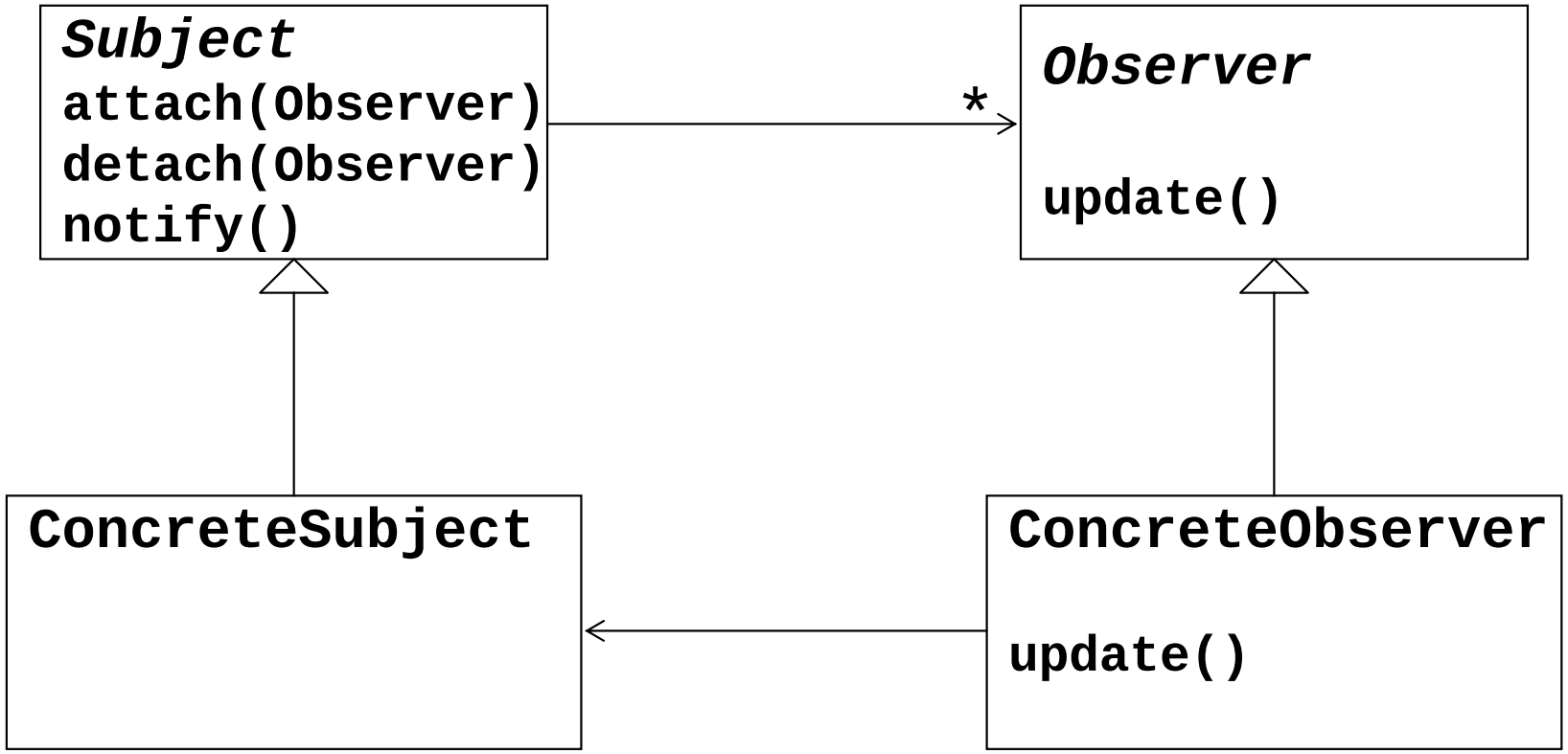
# Programmieren mit Datenfluss-Graph



- Ein interaktives Programm mit einfacher, textbasierter Benutzer-Interaktion
  - liest von **Standard-Eingabe** *(standard input)* – hinter der sich die Tastatur verbirgt
    (in Java durch Benutzung von System.in),
  - schreibt nach **Standard-Ausgabe** *(standard output)* – hinter der sich der Bildschirm verbirgt
    (in Java durch Benutzung von System.out)

- Ereignis *(event)*:
  - eine **Ereignis-Quelle** *(event source, publisher)* generiert Benachrichtigung *(event notification)*,
  - an der i. a. *mehrere* **Ereignis-Senken** *(event sinks, subscribers)* interessiert sind.
- Z.B.
  - „Telekom-Aktie fällt unter 5!“
  - „Druck steigt über 4,5!“
- Entkopplung:
  - Für die **Quelle** ist es *irrelevant*, *wer* auf ein Ereignis *wie* reagiert.
  - Die **Senken** sind an ganz *bestimmten* Ereignissen interessiert.

- *(observer pattern)* ist ein objektorientiertes Entwurfsmuster *(design pattern)*, das als *sequentielle* Variante der ereignisbasierten Interaktion betrachtet werden kann:



# 6.3.3 Das Ereignis-Modell von Java

- (für GUI-Ereignisse: Paket java.awt.event u.a.) orientiert sich am Beobachter-Muster:
  - *event listener* = Beobachter (= Ereignissenke)
  - *event source* = Beobachteter (= Ereignisquelle)
  - *event type* + *event source* (= Ereigniskanal/typ) (*event type* in AWT per Namenskonvention)
  - *event object* = Benachrichtigung