

# Systemprogrammierung

*Grundlagen von Betriebssystemen*

Teil C – X.1 Prozesssynchronisation: Nichtsequentialität

Wolfgang Schröder-Preikschat

15. November 2022



# Agenda

---

Einführung

Kausalitätsprinzip

Parallelisierbarkeit

Kausalordnung

Aktionsfolgen

Sequentialisierung

Koordinierung

Konkurrenz

Verfahrensweisen

Einordnung

Fallstudie

Lebendigkeit

Zusammenfassung



# Gliederung

---

## Einführung

Kausalitätsprinzip

Parallelisierbarkeit

Kausalordnung

Aktionsfolgen

Sequentialisierung

Koordinierung

Konkurrenz

Verfahrensweisen

Einordnung

Fallstudie

Lebendigkeit

Zusammenfassung



# Lehrstoff

---

- **Nebenläufigkeit** von Prozessen als Eigenschaft begreifen, die ein Betriebssystem fördern und schon gar nicht behindern sollte
  - um das Leistungspotential mehr- oder vielkerniger Prozessoren zu nutzen
  - **Parallelrechner** sind gang und gäbe, brauchen aber parallele Abläufe
- erkennen, dass Nebenläufigkeit jedoch nur für **kausal unabhängige Prozesse** gilt, die nicht durchgängig gegeben sind
  - problembedingte Rollenspiele von Prozessen (Konsument vs. Produzent)
  - Konkurrenzsituationen bei Zugriffen auf gemeinsame Betriebsmittel
- Prinzipien kennenlernen/vertiefen, um **kausal zusammenhängende Aktionen** nacheinander stattfinden zu lassen
  - **Sequentialisierung** von gleichzeitigen (gekoppelten) Prozessen erzwingen
  - Konkurrenz dieser Prozesse durch wechselseitigen Ausschluss koordinieren
  - verstehen, dass Aktionen auf vertikaler Ebene nicht unteilbar sein müssen
  - **Unteilbarkeit** in Bezug auf Betriebsmittel und Aktionen kennenlernen
- Verfahrensweisen zur Synchronisation erklären, mit einer **Fallstudie** Probleme und deren Lösungen aufzeigen
  - ein- und mehrseitige Synchronisation am Beispiel „*bounded buffer*“



# Gliederung

---

Einführung

Kausalitätsprinzip

Parallelisierbarkeit

Kausalordnung

Aktionsfolgen

Sequentialisierung

Koordinierung

Konkurrenz

Verfahrensweisen

Einordnung

Fallstudie

Lebendigkeit

Zusammenfassung



*Sequentielles*  $\mapsto$  *Nichtsequentielles Programm*

- **Nebenläufigkeit** (*concurrency*) bezeichnet das Verhältnis von nicht kausal abhängigen Ereignissen, die sich also nicht beeinflussen

1       foo = 4711;

2       bar = 42;

3   foofoo = foo + bar;

4   barfoo = bar + foo;

5       hal = foofoo + barfoo;

- Aktion „=“ in Zeile 1 ist nebenläufig zu der in Zeile 2

- die Aktionen „=“ und „+“ in Zeile 3 sind nebenläufig zu denen in Zeile 4

- in logischer Hinsicht sind Aktionen potentiell nebenläufig, wenn keine das Resultat der anderen benötigt
- in physischer (d.h., körperlicher) Hinsicht ist für jede dieser Aktionen ein Aktivitätsträger erforderlich, der autonom agieren kann

- **Kausalität** (lat. *causa*: Ursache) ist die Beziehung zwischen **Ursache** und **Wirkung**, d.h., die ursächliche Verbindung zweier Ereignisse

- Ereignisse sind nebenläufig, wenn keines Ursache des anderen ist

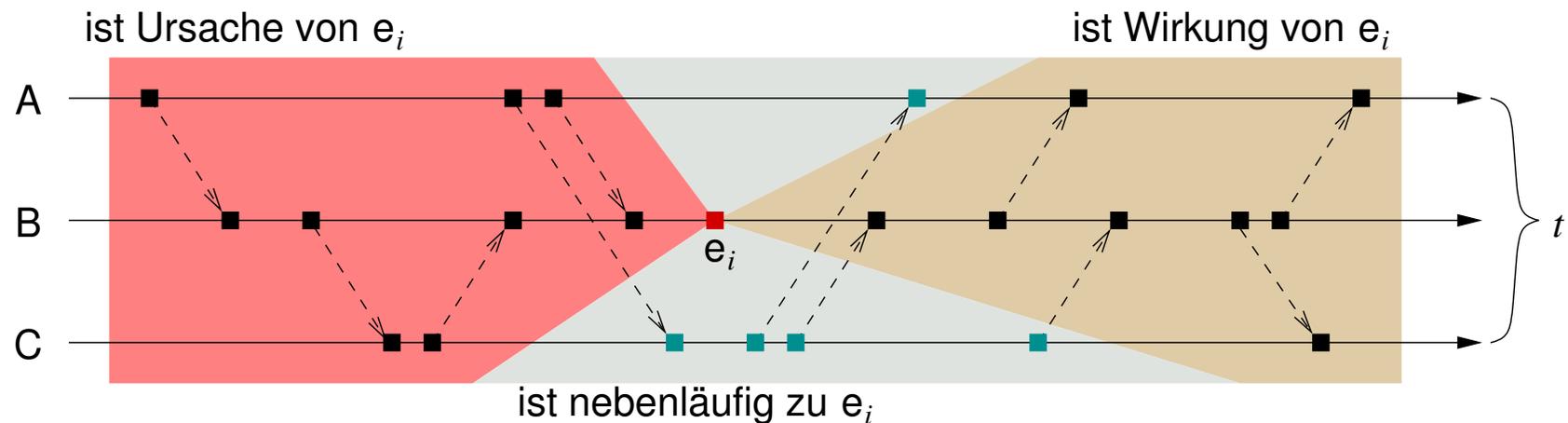
---

<sup>1</sup>Aktion: Anweisungsausführung einer (virtuellen/realen) Maschine. [8, S. 12]



# Ursache und Wirkung

*Nebenläufigkeit als relativistischer Begriff von Gleichzeitigkeit:*



- ein Ereignis **ist nebenläufig zu** einem anderen ( $e_i$ ), wenn es im **Anderswo** des anderen Ereignisses ( $e_i$ ) liegt
  - d.h., weder in der Zukunft noch in der Vergangenheit des anderen
- das Ereignis ist nicht Ursache/Wirkung des anderen Ereignisses ( $e_i$ )
  - ggf. aber Ursache/Wirkung anderer (von  $e_i$  verschiedener) Ereignisse



# Daten- und Zeit(un)abhängigkeit

- ein „im Anderswo anderer Ereignisse liegendes“ Ereignis steht für eine **nebenläufige Aktion**, sofern eben:
  - allgemein** ■ keine das Resultat der anderen benötigt (S. 6)
    - **Datenabhängigkeiten** gleichzeitiger Prozesse beachten
  - speziell** ■ keine die **Zeitbedingungen** der anderen verletzt
    - zusätzliches, zwingendes Merkmal nur für Echtzeitbetrieb
      - Zeitpunkte dürfen nicht/nur selten verpasst werden
      - Zeitintervalle dürfen nicht/nur begrenzt gedehnt werden
- je nach Art der Beziehung zwischen den Ereignissen bzw. Aktionen, ist die **Konsequenz für gleichzeitige Prozesse** verschieden

„ist Ursache von“ }  $\rightsquigarrow$  **Koordinierung** (vor/zur Laufzeit)  
„ist Wirkung von“ }  
„ist nebenläufig zu“  $\rightsquigarrow$  **Parallelität** (implizit)

- Koordinierung durch **Sequentialisierung**: Schaffen einer Ordnung für eine Menge von Aktionen entlang der Kausalordnung



## Definition (*concurrent/simultaneous processes*)

Mehrere (ggf. **nichtsequentielle**) **Prozesse**, durch die sich mehr als eine Aktionsfolge in Raum und Zeit überlappen.

- **notwendige Bedingung** dazu ist die Fähigkeit des Betriebssystems zur **Simultanverarbeitung** (*multiprocessing*) von Programmen
  - **vertikal** ausgelegt, durch Multiplexen ein und desselben Prozessors
    - Mehrbenutzer-, Teilnehmer- oder Zeitmultiplexbetrieb: *time sharing* [1]
    - pseudo Parallelität durch asynchrone Programmunterbrechungen (*interrupts*)
  - **horizontal** ausgelegt, durch Vervielfachung des Prozessors
    - symmetrischer oder asymmetrischer Multiprozessorbetrieb: *multiprocessing*
    - echte Parallelität durch mehrere physische Ausführungseinheiten
- **hinreichende Bedingung** ist die Verfügbarkeit von Programmen, durch die zugleich mehrere Ausführungsstränge möglich werden
  - ein nichtsequentielles Programm oder mehrere sequentielle Programme
  - eine beliebige Kombination derartiger Programme



## Definition (*interacting processes*)

Gleichzeitige Prozesse, die durch direkte oder indirekte Nutzung einer oder mehrerer gemeinsamer Variablen bzw. Ressourcen interagieren.

- dabei interagieren die Prozesse schon im Moment des Zugriffs, da sie dadurch **Interferenz**<sup>2</sup> in zeitlicher Hinsicht erzeugen
  - durch logisch gleichzeitige Zugriffe auf höherer Ebene, wenn diese jedoch auf tieferer Ebene nur sequentiell durchgeführt werden können/dürfen
  - z.B. Sequentialisierung durch den Bus oder einen kritischen Abschnitt
- entscheidend ist jedoch die **logische Bedeutung** der Variablen bzw. Ressource für die beteiligten gleichzeitigen Prozesse
  - Medium zur Kommunikation mit dem jeweils anderen internen Prozess
  - Instrument zur Interaktion mit einem externen Prozess (Peripherie)
- diese Bedeutung schließt **Datenabhängigkeiten** ein und bezieht sich gerade auch auf das **Rollenspiel** der Prozesse
  - Produzent/Konsument (Datum), Sender/Empfänger (Signal, Nachricht)

<sup>2</sup>Abgeleitet von (altfrz.) *s'entreferir* „sich gegenseitig schlagen“.



# Gliederung

---

Einführung

Kausalitätsprinzip

Parallelisierbarkeit

Kausalordnung

Aktionsfolgen

**Sequentialisierung**

**Koordinierung**

**Konkurrenz**

Verfahrensweisen

Einordnung

Fallstudie

Lebendigkeit

Zusammenfassung



# Koordination von Konkurrenz

- **gleichzeitige Aktionen** überlappen einander in Raum und Zeit
  - i der **Moment** ihres Zusammentreffens ist i. A. nicht vorherbestimmt
  - ii Aktionen können komplex sein (d.h., **mehrere Einzelschritte** umfassen)
  - iii ihre besondere Eigenschaft ist die **Teilbarkeit in zeitlicher Hinsicht**
- kausal zusammenhängende Aktionen müssen nacheinander stattfinden
  - off-line*
    - statische Einplanung, Daten- und Kontrollflussabhängigkeiten
    - der Ablaufplan sorgt für die **implizite Synchronisation**
    - *analytischer Ansatz*, der Vorabwissen erfordert (s. aber i, oben)
  - on-line*
    - dynamische Einplanung, ausgelöst durch externe Ereignisse
    - **explizite Synchronisation** durch Programmanweisungen
    - *konstruktiver Ansatz*, der ohne Vorabwissen auskommen muss
- explizite Prozesssynchronisation kann **Wettstreit** hervorbringen
  - bei Mitbenutzung (*sharing*) desselben wiederverwendbaren Betriebsmittels
  - bei Übergabe (*handover*) eines konsumierbaren Betriebsmittels

## Hinweis

Die gewählte Methode sollte *minimal invasiv* auf die Prozesse wirken, bei expliziter Synchronisation ist **Interferenz** unvermeidbar...



# Atomare Aktionen

## Definition (atomare Aktion)

Eine Aktion, **deren Einzelschritte** nach außen sichtbar im Verbund **scheinbar gleichzeitig stattfinden**.

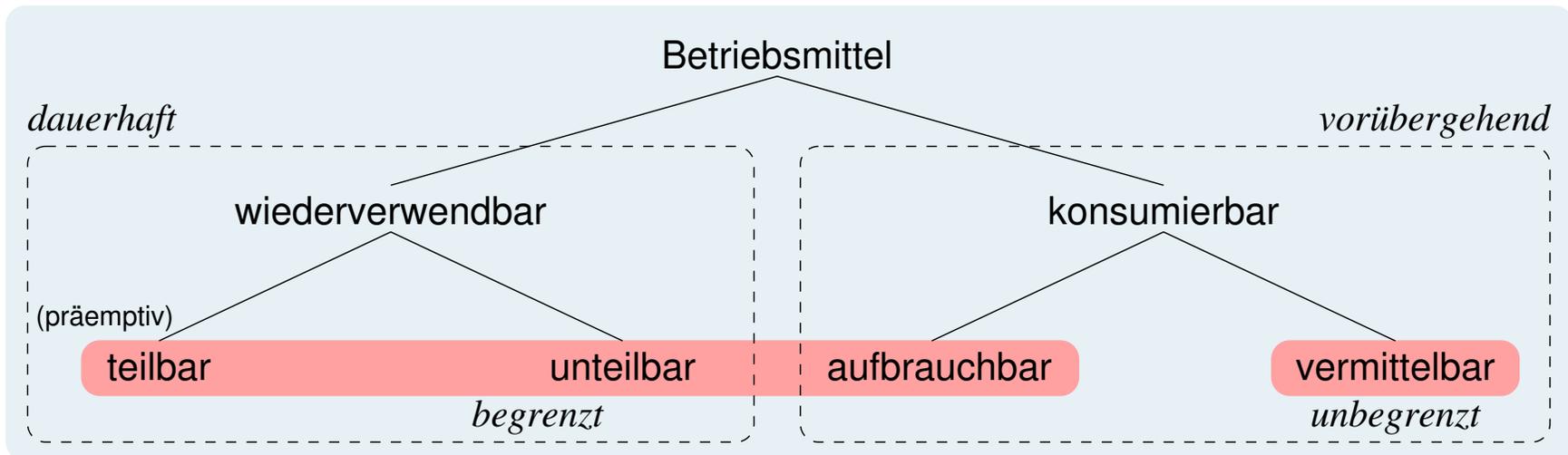
- dabei wird das Herstellen von Gleichzeitigkeit (Simultanität) durch **Synchronisation**<sup>3</sup> der gekoppelten Aktionen/Prozesse erreicht
  - **Koordination** der Kooperation und Konkurrenz zwischen Prozessen [6]
  - Sequentialisierung von Ereignissen entlang einer Kausalordnung
  - Aktionen gleichzeitig/in einer bestimmten Reihenfolge stattfinden lassen
- zentrales Konzept, um **gleichzeitige Aktionen** zu **koordinieren**, ist der **wechselseitige Ausschluss** (*mutual exclusion*)
  - i ein **kritischer Abschnitt** [2, S. 11] der Maschinenprogrammzebene
  - ii eine **Elementaroperation** (*read-modify-write*) der Befehlssatzebene
- dabei ist die Auswirkung auf die beteiligten Prozesse je nach Ebene der Abstraktion bzw. Paradigma sehr unterschiedlich
  - d.h., die Synchronisation wirkt blockierend (i) oder nichtblockierend (ii)

<sup>3</sup>(gr). *sýn*: zusammen, *chrónos*: Zeit



# Betriebsmittel und Aktionen

- je nach **Betriebsmittelart** (vgl. [8, S. 9–10]) ist die Nutzung durch gleichzeitige Prozesse eingeschränkt:



- bereits Aktionen zum **Zugriff** auf ein unteilbares wiederverwendbares Betriebsmittel unterliegen dem wechselseitigen Ausschluss
  - **mehrseitige/multilaterale Synchronisation** gekoppelter Prozesse
- wohingegen die Aktion der **Entgegennahme** eines konsumierbaren Betriebsmittels nur auf einen Prozess verzögernd wirkt
  - **einseitige/unilaterale Synchronisation** gekoppelter Prozesse



# Unteilbarkeit

## Definition (in Anlehnung an den Duden)

(Betriebssystem) Umstand, der die Verteilung der Betriebsmittel auf mehrere Prozessoren *oder* Prozesse verhindert.

- unteilbar ist ein Betriebsmittel, wenn es zu einem Zeitpunkt von nur genau einem Prozessor/Prozess genutzt werden darf
  - Zugriffsoperationen darauf können/dürfen nicht zeitlich zerteilt werden
  - sie müssen **atomar**, d.h., als **Elementaroperation** ausgeführt werden
  - ↪ Aktion/Aktionsfolge mehrerer kausal abhängender Einzelschritte
- teilbar ist ein Betriebsmittel, wenn mehrere Prozessoren/Prozesse es gleichzeitig benutzen dürfen
  - es dem einem entzogen und einem anderen gegeben werden darf
  - Zugriffe auf das Betriebsmittel können/dürfen zeitlich zerteilt werden
  - ↪ Aktion/Aktionsfolge mehrerer kausal unabhängiger Einzelschritte
- **beachte**: ein Betriebsmittel besonderer Art ist der Prozessor im Falle eines kritischen Abschnitts in einem nichtsequentiellen Programm
  - das *Unteilbarsein* auf Maschinenprogramm- oder Befehlssatzebene (S. 13)



# Wettstreit

## Definition (Duden)

Bemühen, einander in etwas zu übertreffen, einander den Vorrang streitig zu machen.

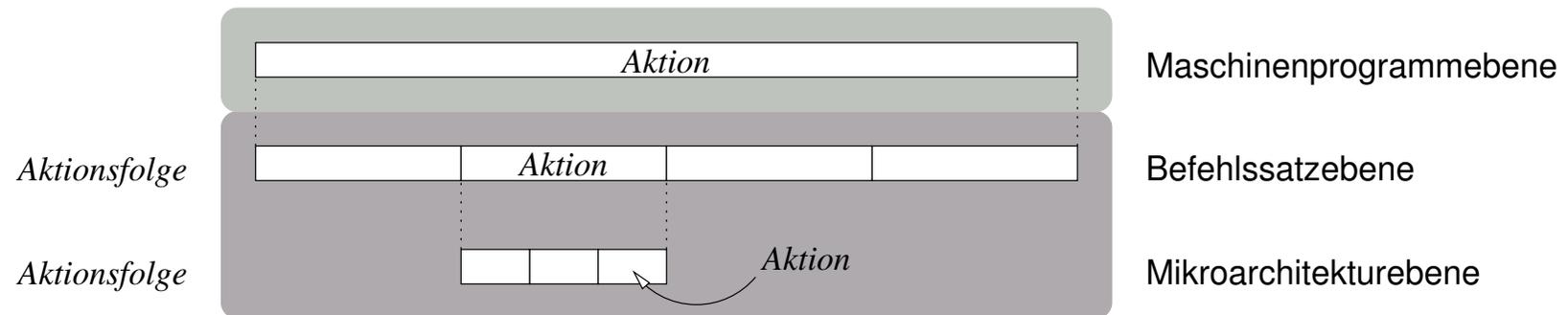
- ein unter gleichzeitigen Prozessen auftretender **Konflikt**, der diese implizit koppelt und damit zur **Interaktion** zwingt, wenn:
  1. Zugriffe auf wenigstens ein **gemeinsames Betriebsmittel** erfolgen,
  2. nur eine **begrenzte Anzahl** dieses Betriebsmittels vorrätig ist und
  3. die betreffenden Betriebsmittel **unteilbar** und von derselben Art sind
- es entsteht eine **Konkurrenzsituation** (*contention*), wenn einer dieser Prozesse ein Betriebsmittel anfordert, das ein anderer bereits besitzt
  - der anfordernde Prozess blockiert und wartet auf die Freigabe des Betriebsmittels durch den Prozess, der das Betriebsmittel belegt
  - der das Betriebsmittel belegende Prozess löst den auf die Freigabe des Betriebsmittels wartenden Prozess aus, deblockiert ihn wieder



- Protokoll zur **Sequentialisierung** gleichzeitiger Aktionen bei Zugriff auf ein gemeinsames wiederverwendbares/unteilbares Betriebsmittel:
  - Vergabe**  $\mapsto$  vor der Aktion das Betriebsmittel sperren
    - im Moment der Anforderung eines gesperrten Betriebsmittels wird die betreffende Aktion blockiert
    - die blockierte Aktion erwartet (mit/ohne Prozessorabgabe) das Ereignis zur Freigabe des gesperrten Betriebsmittels
  - Freigabe**  $\mapsto$  nach der Aktion das Betriebsmittel entsperren
    - sollten Aktionen die Freigabe dieses Betriebsmittels erwarten, wird es zur **Wiedervergabe** bereitgestellt; das bedeutet:
      - alle Aktionen deblockieren, erneut das Vergabeprotokoll durchlaufen oder
      - eine Aktion deblockieren, für sie das Betriebsmittel (weiterhin) sperren
    - normalerweise nur durch den das Betriebsmittel „besitzenden“ Prozess
- dabei beziehen sich die Aktionen auf ein und dieselbe **Phase** in einem Soft- oder Hardwareprozess, je nach Betrachtungsebene
  - d.h., der Maschinenprogramm- oder Befehlssatzebene (S. 13)



# Teilbarkeit in vertikaler Hinsicht



Beachte: Aktion  $\sim$  Programmablauf (vgl. [8, S. 11–12])

Ein und derselbe Programmablauf kann auf einer Abstraktionsebene sequentiell, auf einer anderen parallel sein. [9]

- wechselseitiger Ausschluss ist eine Methode der Maschinenprogramm- oder Befehlssatzebene, um **atomare Aktionen** zu schaffen
  - kritischer Abschnitt auf höherer Ebene, Elementaroperation auf tieferer
  - letztere entspricht einem kritischen Abschnitt in der Hardware. . .
- je nach **Bezugssystem** wirken sich diese Methoden blockierend oder nichtblockierend auf zu synchronisierende gleichzeitige Prozesse aus



# Gliederung

---

Einführung

Kausalitätsprinzip

Parallelisierbarkeit

Kausalordnung

Aktionsfolgen

Sequentialisierung

Koordinierung

Konkurrenz

Verfahrensweisen

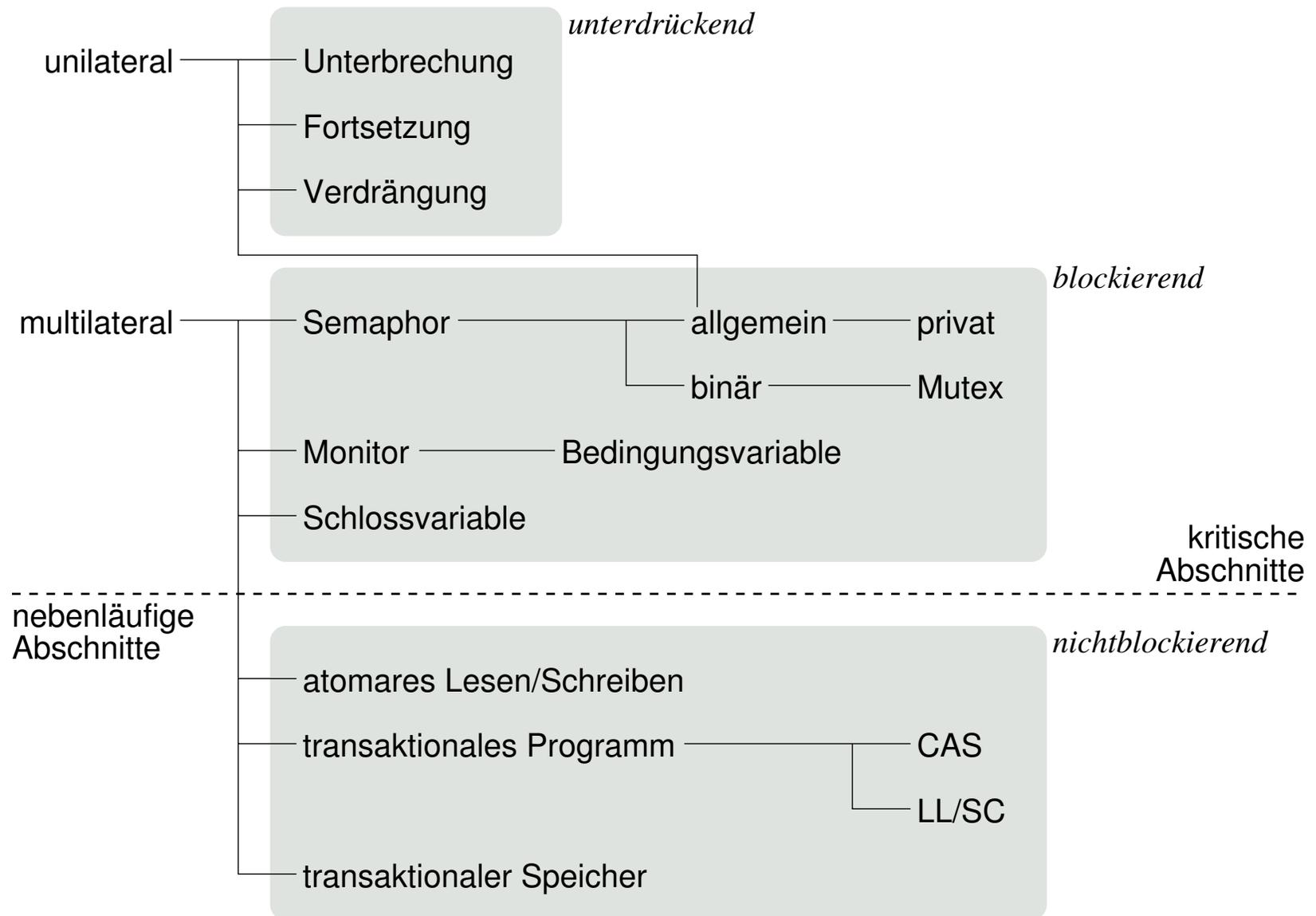
Einordnung

Fallstudie

Lebendigkeit

Zusammenfassung





# Wirkung

- je nach Art und Technik ist der Effekt von Synchronisation auf die gleichzeitigen Prozesse sehr unterschiedlich:
  - unterdrückend** ■ verhindert die **Prozessauslösung** anderer Prozesse
    - unabhängig des eventuellen gleichzeitigen Geschehens
    - betrifft konsumierbare Betriebsmittel
  - blockierend** ■ sperrt die **Betriebsmittelvergabe** an Prozesse
    - ist nur bei gleichzeitigem Geschehen wirksam
    - betrifft wiederverwendbare/konsumierbare Betriebsmittel
  - nichtblockierend** ■ unterbindet **Zustandsverstetigung** durch Prozesse
    - ist nur bei gleichzeitigem Geschehen wirksam
    - betrifft wiederverwendbare Betriebsmittel: **Speicher**
  - der aktuelle Prozess wird möglicherweise zurückgerollt
- es gibt keine einzige Methode, die nur Vorteile hat, allen Ansprüchen genügt und jeder Anforderung gerecht wird. . .



- die Verfahren wirken lediglich auf einen der beteiligten Prozesse
    - die anderen beteiligten Prozesse schreiten ungehindert fort<sup>4</sup>
  - dabei gibt ein **logischer Programmablauf** die **Bedingungen** vor
- ## Bedingungssynchronisation
- der Fortschritt des einen Prozesses ist abhängig von einer Bedingung, die in einem nichtsequentiellen Programm formuliert ist
  - der andere Prozess, der diese Bedingung aufhebt, erfährt dabei keine Verzögerung in seinem Ablauf

## logische Synchronisation

- die Maßnahme resultiert aus der logischen Abfolge der Aktivitäten
  - vorgegeben durch das „Rollenspiel“ der beteiligten Prozesse
- beachte: andere Prozesse sind jedoch nicht gänzlich unbeteiligt
    - die Aufhebung der Bedingung, die zum Warten eines Prozesses führte, ist von einem anderen Prozess zu leisten
    - **gekoppelte Prozesse** müssen ihrer jeweiligen Rolle gerecht werden. . .

---

<sup>4</sup>Ungeachtet der Gemeinkosten (*overhead*) der Verfahren.



- die Verfahren wirken auf alle in dem Moment beteiligten Prozesse
  - welcher dieser Prozesse ungehindert fortschreitet, ist unbestimmt
- den **Fortgang** der beteiligten Prozesse explizit kontrollieren

**blockierend** ~ pessimistisch: wahrscheinliche, häufige Konkurrenz

- der **wechselseitige Ausschluss** gleichzeitiger Prozesse
  - Warten mit (passiv) oder ohne (aktiv) Prozessorabgabe
- die Verfahren profitieren von der **Maschinenprogrammebene**
  - Systemfunktionen zur Einplanung und Einlastung von Prozessen
- im Regelfall zeitlich begrenzte, **exklusive Betriebsmittelvergabe**

**nichtblockierend** ~ optimistisch: unwahrscheinliche, seltene Konkurrenz

- auf Basis einer **Transaktion** zwischen gleichzeitigen Prozessen
  - eine Folge von Aktionen, die nur komplett oder gar nicht stattfinden
- den Verfahren genügen Merkmale der **Befehlssatzebene**
  - **Spezialbefehle** mit atomaren Aktionen der Mikroarchitekturebene
- ungeeignet für wiederverwendbare unteilbare Betriebsmittel



- angenommen, die folgenden Unterprogramme (put und get) werden in beliebiger Reihenfolge und gleichzeitig ausgeführt:

```
1 char buffer[64];
2 unsigned in = 0, out = 0;
3
4 void put(char item) {
5     buffer[in++ % 64] = item;
6 }
7
8 char get() {
9     return buffer[out++ % 64];
10 }
```

↪ mit buffer als **wiederverwendbares** und item als **konsumierbares** Betriebsmittel

- put und get unterliegen der uni- und multilateralen Synchronisation
  - eine uneingeschränkt gleichzeitige Ausführung darf nicht geschehen

- logische Probleme:

- gepufferte Daten werden ggf. überschrieben: **Überlauf**
- ein leerer Puffer gibt ggf. Daten zurück: **Unterlauf**

- andere Probleme:

- **überlappendes Schreiben** an dieselbe Speicherstelle
- **überlappendes Lesen** von derselben Speicherstelle
- **überlappendes Addieren** gibt ggf. falsche Zählerwerte



# Nichtsequentieller Programmablauf

... Lösung mit „Untiefe“

```
1 char buffer[64];
2 unsigned in = 0, out = 0;
3
4 void put(char item) {
5     if (((in + 1) % 64) == out) await(get);
6
7     buffer[FAA(&in, 1) % 64] = item;
8     cause(put);
9 }
10
11 char get() {
12     if (out == in) await(put);
13
14     char item = buffer[FAA(&out, 1) % 64];
15     cause(get);
16
17     return item;
18 }
```

## „Verlorenes Aufwachen“

Überlappt die Aktion zur Ereignisanzeige mit der Überprüfung der Wartebedingung, kann das Ereignis unbemerkt bleiben.

## await(*e*)

- erwarte Ereignis *e*

## cause(*e*)

- zeige Ereignis *e* an

## FAA(*c*, *n*)

- verändere Zähler *c* um Wert *n*
- liefere vorherigen Zählerstand
- tue dies unteilbar



# Vorbeugung des Ereignisverlusts

- `if (condition) await(event):` **wettlaufkritische Anweisung**

## *lost wake-up*

Zwischen Feststellung der Wartebedingung eines Prozesses und seiner daraufhin logisch korrekten **Blockierung**, wird diese Bedingung durch einen gleichzeitigen Prozess aufgehoben.

- die Aktionsfolge 1. **Prüfen und** ggf. 2. **Warten** findet unwiderruflich statt
- sie eröffnet eine **Konkurrenzsituation** zwischen gleichzeitigen Prozessen
- die Anweisung ist als **bedingter kritischer Abschnitt** auszuführen
  - dabei definiert die Wartebedingung ein **Prädikat** über die im kritischen Abschnitt von den Prozessen gemeinsam verwendeten Daten
  - Auswertung und Folgerung erfolgen im kritischen Abschnitt, der während der Wartezeit des Prozesses für andere Prozesse aber frei sein muss [7]
- alternative und für das Pufferbeispiel besser geeignete Lösung:
  - **allgemeiner Semaphor**, der die Anzahl freier/belegter Einträge mitzählt



- Aussagen zur **Lebendigkeit** (*liveliness*) nichtsequentieller Programme **behinderungsfrei** (*obstruction-free*)
  - ein einzelner, in Isolation stattfindender Prozess wird seine Aktion in begrenzter Anzahl von Schritten beenden
  - der Prozess findet isoliert statt, sofern alle anderen Prozesse, die ihn behindern könnten, zurückgestellt sind
- sperrfrei** (*lock-free*), umfasst Behinderungsfreiheit
  - jeder Schritt eines Prozesses trägt dazu bei, dass die Ausführung des nichtsequentiellen Programms insgesamt voranschreitet
  - systemweiter Fortschritt ist garantiert, jedoch können einzelne Prozesse der **Aushungerung** (*starvation*) unterliegen
- wartefrei** (*wait-free*), umfasst Sperrfreiheit
  - die Anzahl der zur Beendigung einer Aktion auszuführenden Schritte ist konstant oder zumindest nach oben begrenzt
  - garantiert systemweiten Fortschritt und ist frei von Aushungerung
- Merkmale von Verfahren für die **nichtblockierende Synchronisation**
  - Eigenschaften der Algorithmen, unabhängig von Umgebungswissen



# Gliederung

---

Einführung

Kausalitätsprinzip

Parallelisierbarkeit

Kausalordnung

Aktionsfolgen

Sequentialisierung

Koordinierung

Konkurrenz

Verfahrensweisen

Einordnung

Fallstudie

Lebendigkeit

Zusammenfassung



- **Nebenläufigkeit** setzt voneinander unabhängige Prozesse voraus
  - bezeichnet das Verhältnis von nicht kausal abhängigen Ereignissen
  - schränkt sich ein aus Gründen von Daten- oder Zeitabhängigkeit
- gleichzeitige abhängige Prozesse implizieren **Koordinierung**
  - nämlich der Kooperation und Konkurrenz zwischen Prozessen
  - durch analytische (implizite) oder konstruktive (explizite) Techniken
- **Synchronisation** zeigt einen großen Facettenreichtum
  - klassifiziert nach der jeweiligen Auswirkung auf beteiligte Prozesse:
    - einseitig oder mehrseitig
    - unterdrückend, blockierend oder nichtblockierend
    - behinderungs-, sperr- oder wartefrei
  - klassifiziert nach der Ebene im Rechensystem  $\rightsquigarrow$  nächsten Vorlesungen:
    - Hochsprachenebene** Bedingungsvariable, Monitor
    - Maschinenprogrammebene** Verdrängungssteuerung, Semaphor
    - Befehlssatzebene** Schlossvariable, Spezialbefehle (CPU)
- Aussagen zur „Lebendigkeit“ nichtsequentieller Programme leiten sich aus den **Fortschrittsgarantien** der Synchronisationsverfahren ab



# Literaturverzeichnis I

---

- [1] CORBATÓ, F. J. ; MERWIN-DAGGETT, M. ; DALEX, R. C.:  
An Experimental Time-Sharing System.  
In: *Proceedings of the AIEE-IRE '62 Spring Joint Computer Conference*, ACM, 1962,  
S. 335–344
- [2] DIJKSTRA, E. W.:  
Cooperating Sequential Processes / Technische Universiteit Eindhoven.  
Eindhoven, The Netherlands, 1965 (EWD-123). –  
Forschungsbericht. –  
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New  
York, NY, 1996)
- [3] HANSEN, P. B.:  
Concurrent Processes.  
In: *Operating System Principles*.  
Englewood Cliffs, N.J., USA : Prentice-Hall, Inc., 1973. –  
ISBN 0–13–637843–9, Kapitel 3, S. 55–131
- [4] HERLIHY, M. :  
Wait-Free Synchronization.  
In: *ACM Transactions on Programming Languages and Systems* 11 (1991), Jan., Nr.  
1, S. 124–149



# Literaturverzeichnis II

---

- [5] HERLIHY, M. ; LUCHANGCO, V. ; MOIR, M. :  
Obstruction-Free Synchronization: Double-Ended Queues as an Example.  
In: *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS 2003), May 19–22, 2003, Providence, Rhode Island, USA, IEEE Computer Society, 2003, S. 522–529*
- [6] HERRTWICH, R. G. ; HOMMEL, G. :  
*Kooperation und Konkurrenz — Nebenläufige, verteilte und echtzeitabhängige Programmsysteme.*  
Springer-Verlag, 1989. –  
ISBN 3–540–51701–4
- [7] HOARE, C. A. R.:  
Towards a Theory of Parallel Programming.  
In: HOARE, C. A. R. (Hrsg.) ; PERROT, R. H. (Hrsg.): *Operating System Techniques.*  
New York, NY : Academic Press, Inc., Aug. – Sept. 1971 (Proceedings of a Seminar at Queen’s University, Belfast, Northern Ireland), S. 61–71
- [8] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. :  
Prozesse.  
In: LEHRSTUHL INFORMATIK 4 (Hrsg.): *Systemprogrammierung.*  
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien), Kapitel 6.1



# Literaturverzeichnis III

---

- [9] LÖHR, K.-P. :  
Nichtsequentielle Programmierung.  
In: INSTITUT FÜR INFORMATIK (Hrsg.): *Algorithmen und Programmierung IV*.  
Freie Universität Berlin, 2006 (Vorlesungsfolien)



# Logische Synchronisation zählender Semaphor (*counting semaphore*)

```
1 semaphore free = 64, data = 0;
2
3 char buffer[64];
4 unsigned in = 0, out = 0;
5
6 void put(char item) {
7     P(&free);    /* block iff buffer is full: free = 0 */
8     buffer[FAA(&in, 1) % 64] = item;
9     V(&data);    /* signal data availability */
10 }
11
12 char get() {
13     P(&data);    /* block iff buffer is empty: data = 0 */
14     char item = buffer[FAA(&out, 1) % 64];
15     V(&free);    /* signal buffer-place availability */
16
17     return item;
18 }
```

- Prinzip „**begrenzter Puffer**“ (*bounded buffer*), siehe auch [8, S. 30–33]



## ■ binärer Semaphor, Lösung auf Maschinenprogrammzebene: ☹️

```
1 int FAA(int_t *ref, int val) {
2     P(&ref->mutex);
3     int aux = ref->value;
4     ref->value += val;
5     V(&ref->mutex);
6
7     return aux;
8 }
```

```
9 typedef struct {
10     int value;
11     semaphore_t mutex;
12 } int_t;
13 int_t in = {0,1}, out = {0,1};
```

## ■ atomarer Spezialbefehl, Lösung auf Befelssatzebene: 😊

```
14 inline int FAA(int *ref, int val) {
15     int aux = val;
16
17     asm volatile ("xaddl %0, %1"
18         : "=g" (aux), "=m" (*ref) : "0" (aux), "m" (*ref)
19         : "memory", "cc");
20
21     return aux;
22 }
```

