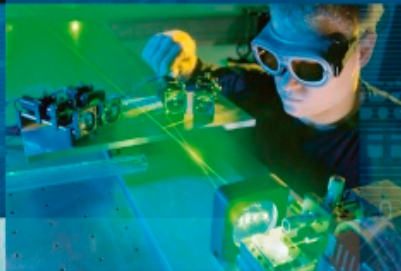
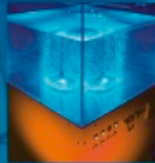


Parallele und funktionale Programmierung (1)

Prof. Dr. Michael Philippsen



■ *Teil I – Handwerkszeug des Parallelprogrammierers*

1. Einführung

Steigender Appetit aber die Zukunft ist Multi-Core; Fach-Jargon; Basiswissen über Petri-Netze

2. Erzeugung von Parallelität in Programmen

Programmiersprachliche Konzepte; Aktivitätsfäden; **Thread** und **Runnable**; Arbeitspakete; **ExecutorService**

3. Datensynchronisation

Wettlaufsituationen; gemeinsamer Zustand; kritische Abschnitte; **synchronized**; Rechnen mit Petri-Netzen

4. **Thread**-Sicherheit

Zusammenhang von Sichtbarkeit und Speichermodell; **volatile** und **Atomic**; flüchtende Objekte und typische Fehler

5. Lebendigkeitsprobleme

Verklemmung (Philosophen, Bedingungen, Gegenmaßnahmen); Verhungern

■ *Teil II – Anwendungstypen und Effizienzfragen*

6. Task-paralleles Vorgehen (1)

Klient & Dienstleister;
Chef & Arbeiter;

Granularität; Lastbalance;
Speedup; Gesetz v. Amdahl

7. Task-paralleles Vorgehen (2)

Arbeitsdiebstahl;
Fließband, Produzent & Konsument;
Paralleles teile-und-herrsche;

Parallelitätsbegrenzung;
Mindestnutzarbeit;
Umstiegspunkte

8. Datenparalleles Vorgehen (1)

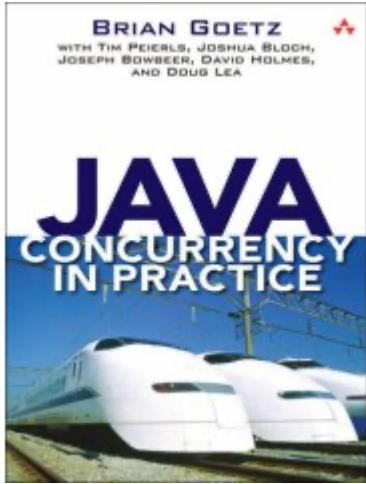
Geometrische Dekomposition;
Tournier-Ansatz zur Reduktion;

Lemma von Brent;
Schneller trotz Mehrarbeit

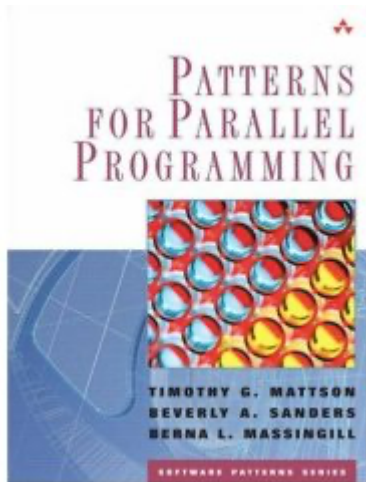
9. Datenparalleles Vorgehen (2)

Verzeigerte Datenstrukturen
Deklarativer Ansatz; MapReduce

Literatur zu den Teilen I und II

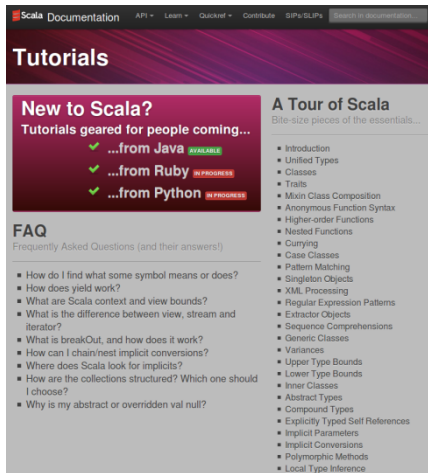


Brian Goetz et al.: *Java Concurrency in Practice*, Addison-Wesley, 2006,
ISBN 0-321-34960-1, knapp 40 €,
<http://jcip.net/>



Timothy Mattson et al.: *Patterns for Parallel Programming*, Addison-Wesley, 2004,
ISBN 0-321-22811-1, rund 40 €.

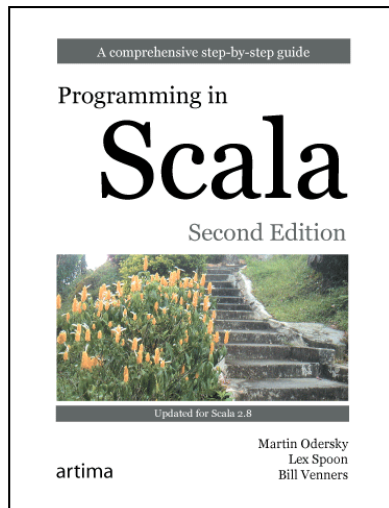
- *Teil III – Funktionale Programmierung*
 - 10. Einführung und Grundlagen
 - 11. Hauptkonzepte funktionaler Programmierung
 - 12. Parallelisierung
 - 13. Funktionale Programmierung - veranschaulicht



<http://www.scala-lang.org/>

Homepage zu Scala enthält eine Vielzahl kostenloser Informationen – insbesondere:

- „*A Brief Scala Tutorial*“ (for Java programmers)
- „*Scala By Example*“



Martin Odersky, Lex Spoon, Bill Venners:



Programming in Scala,

Artima, 2011, ISBN 978-0981531649, ca. 39 €

auch als eBook für \$29,95:

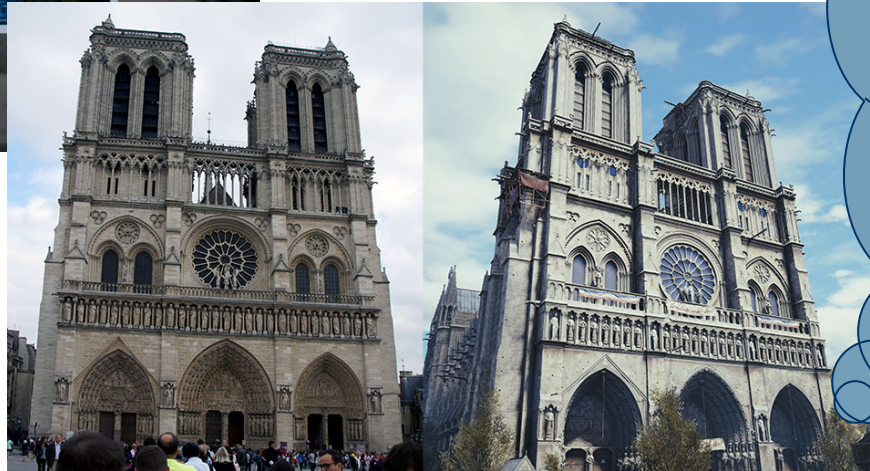
http://www.artima.com/shop/programming_in_scala_2ed

Leistungsbedarf der Anwendungen steigt

- 
- 
- Immer größere Probleme erfordern immer schnellere Systeme, z.B. Klimasimulationen, Strömungsmechanik, rechnergestützte Modellierung, animierte Filme, Computerspiele, Web-Suche, ...
 - Mit mehr Rechenleistung steigt der Appetit auf anspruchsvollere Anwendungen.
 - Leistungsanforderungen im Tera-/Peta-Flops-Bereich.
 - Auch Energieeffizienz ist wichtig!

Beispiel: Computerspiele

- Simulationen, Darstellungen werden immer realistischer, erfordern mehr Rechenkapazität.
- Beispiel „Assassin's Creed Unity“: empfohlenes System: Quad-Core CPU, 8 GB RAM, NVidia GeForce GTX 780



Spielehersteller haben heute schon „Shader“, die auf 50 mal stärkeren Grafikkarten laufen würden, um den Kartenherstellern zu zeigen, wozu die Mehrleistung nötig ist.

Beispiel: Computer-animierte Filme

- Pixar, Dreamworks, Disney, ...
- How to Train Your Dragon 2, 2014
 $90 \times 60 \times 24 = 129\,600$ Frames
398 Terabyte Animationsdaten
90 Millionen Rechenstunden
- Baymax/Big Hero 6, 2014
200 Millionen Rechenstunden,
durchschnittl. 83 h/Frame,
Cluster mit 4 600 Rechner mit
insges. 55 000 Kernen
- Figuren-Design mit Animations-
bearbeitung in Echtzeit,
verwendet bis zu 24 Cores



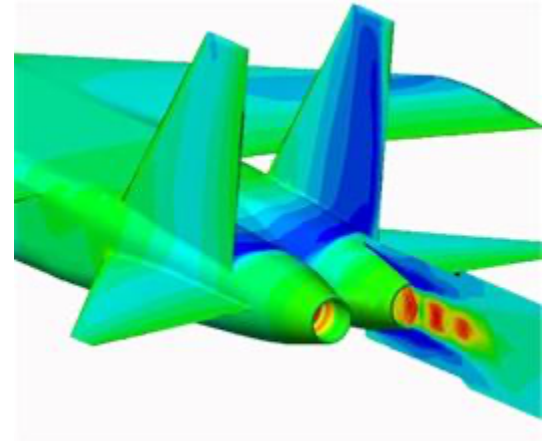
Beispiel: Simulation von 100 - 1 000 Figuren

- Hobbit, ...
 - Die Bewegungen vieler Figuren werden basierend auf Computer-Modellen berechnet.

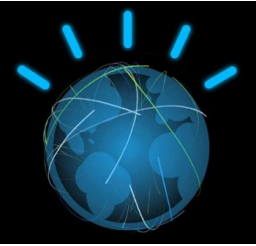


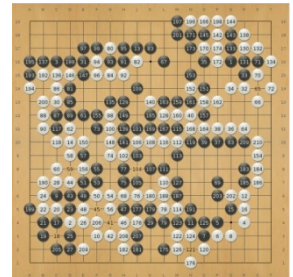
Beispiel: Strömungssimulation

- Diskretisiertes Gitter
- 5000 FLOP (Gleitkommaoperation) pro Gitterpunkt über 5000 Zeitschritte
- Flugzeugflügel: 512x64x256 Gitterpunkte
 - $2,1 * 10^{14}$ Flop erforderlich
 - 8h 20m auf Pentium 4 (Annahme: 7 GFlops)
- komplettes Flugzeug: $3,5 * 10^{17}$ Gitterpunkte
 - $8,8 * 10^{24}$ Flop erforderlich
 - Viele Jahre auf Deutschlands schnellstem Supercomputer (Platz 9 der Weltrangliste): „SuperMuc“: 26,9 PFLOP/s
www.top500.org



Weitere Beispiele

- **Google** durchsucht und indiziert einen signifikanten Anteil der Seiten im Web.
- **Folding@home** ist ein wissenschaftliches Projekt, das mithilfe von über das Internet verbundenen Computern Protein-Interaktionen zur Erforschung neuer Arzneimittel berechnet.
- „Deep Blue“ (IBM) besiegt G. Kasparov (Mai 1997).
-  „Watson“ (IBM) besiegt 2 Menschen in Jeopardy (Feb. 2011).
2880 parallele Threads, auf 90 Octocores.
- AlphaGo (Google) besiegt L. Sedol (März 2016).
1202 CPUs, 176 GPUs.

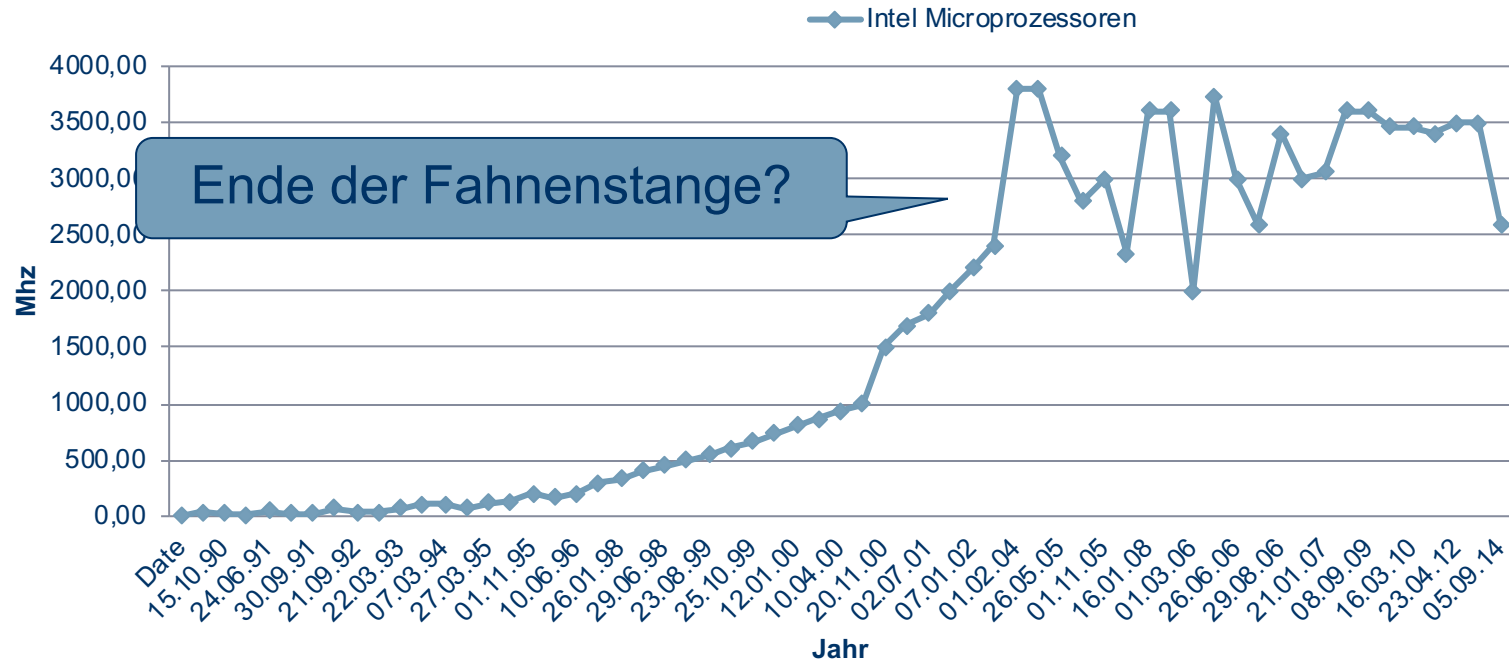


Dilemma

- Leistungssteigerungen können nicht (wie bisher üblich) durch Taktsteigerungen erzielt werden, da die klassische Chip-Technologie „an die Wand fährt“.
 - „Speicherwand“
 - „Stromverbrauchswand“
 - „Frequenzwand“

Speicherwand („memory wall“)

- Die Prozessorgeschwindigkeit verdoppelte sich ca. alle 18 Monate (Moore's Law).

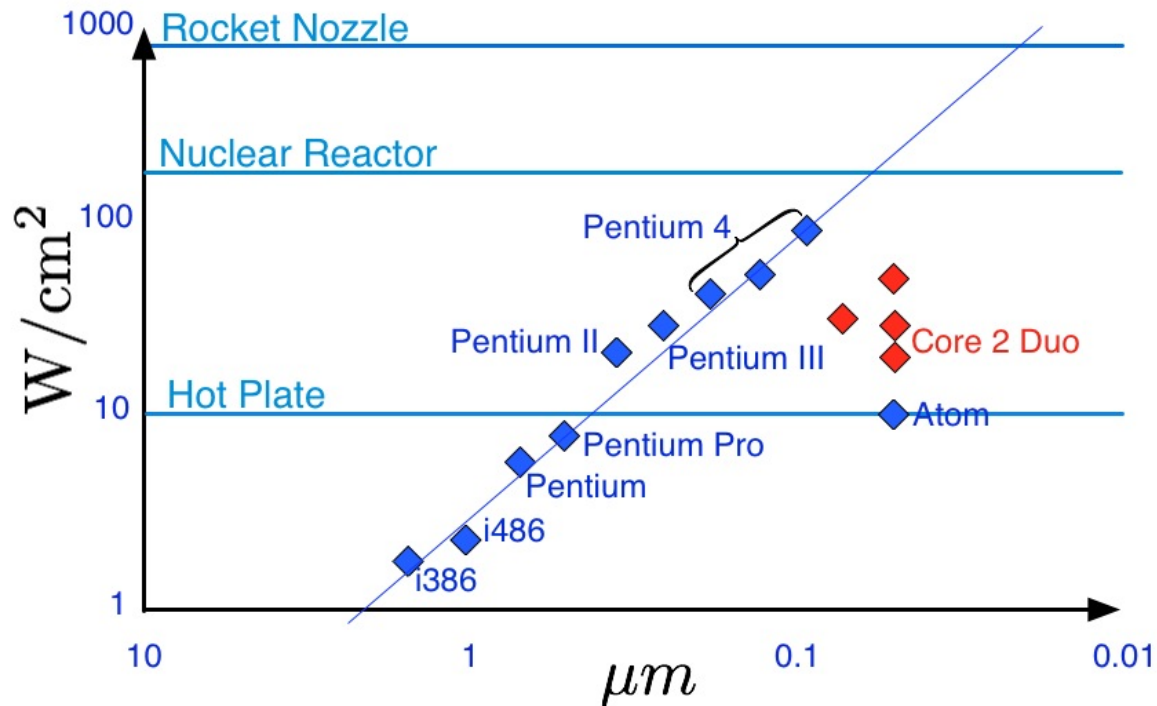
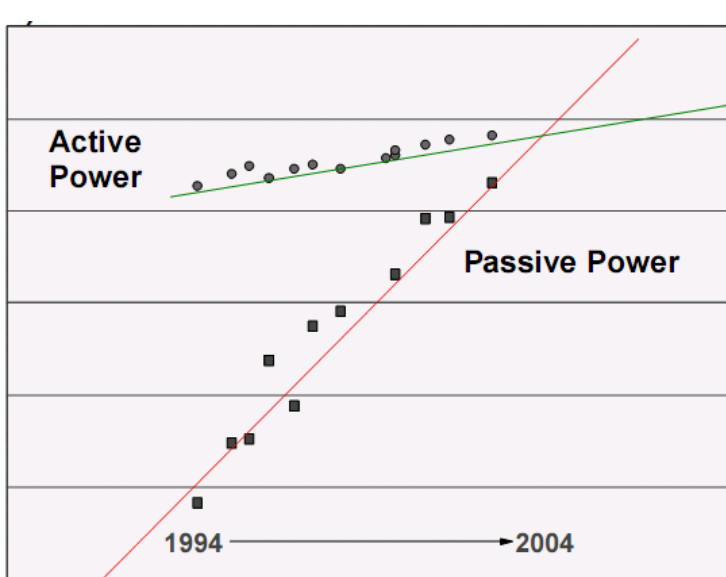


- Aber:**

- Speichergeschwindigkeit verdoppelt sich nur ca. alle 10 Jahre.
- Ein Hauptspeicherzugriff dauert heute bis zu 200 Taktzyklen.

Stromverbrauchswand („power wall“)

- Die Leistungsdichte der Chips steigt extrem an.
- Das Verhältnis von aktiver Energie (tatsächlich zum Rechnen eingesetzt) zu passiver Energie (Betriebsbereitschaft ohne Funktion, Leitungswiderstand) wird immer ungünstiger.



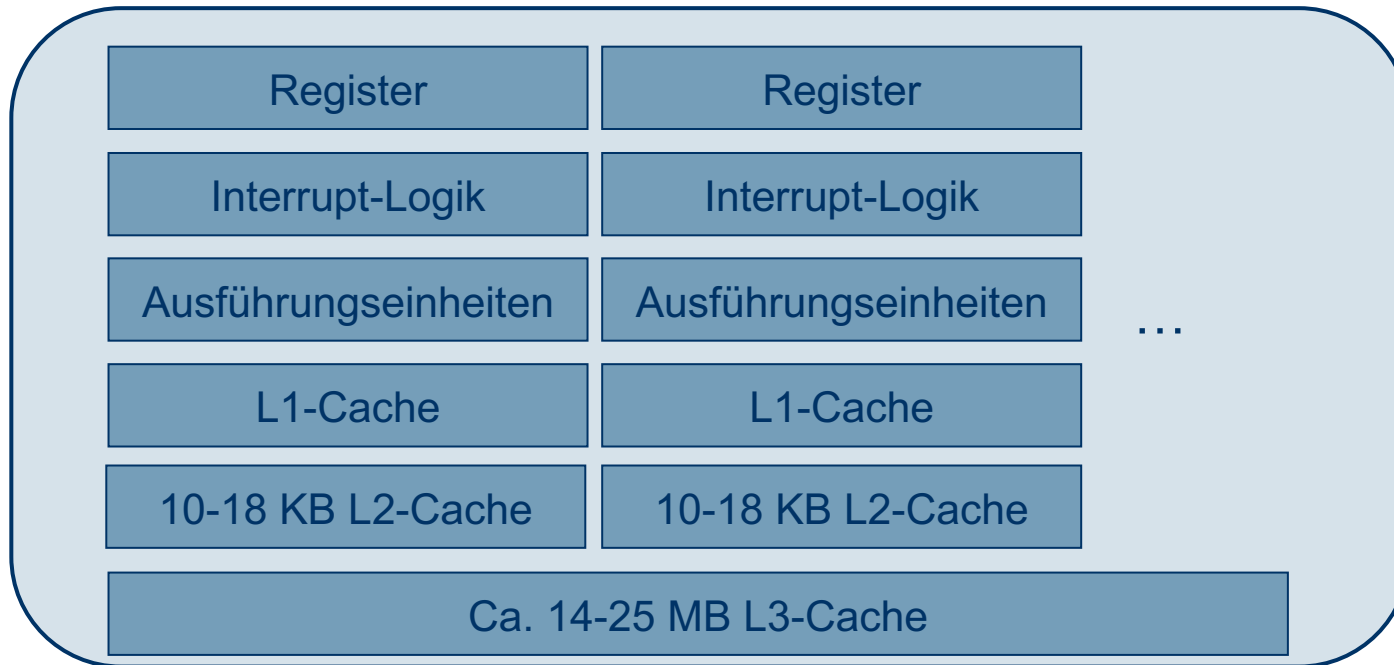
Quelle: Mandy Pant, "Microprocessor Power Impacts", May 2010

Frequenzwand („frequency wall“)

- Erhöhung der Taktfrequenzen und längere Pipelines erzielen keine höhere Rechenleistung mehr,
- vor allem im Verhältnis zur Energieaufnahme.
- „Gegenmaßnahmen“:
 - Höhere Registerzahlen
 - Längere Pipelines
 - falsch vorhergesagte Sprünge kosten deutlich mehr

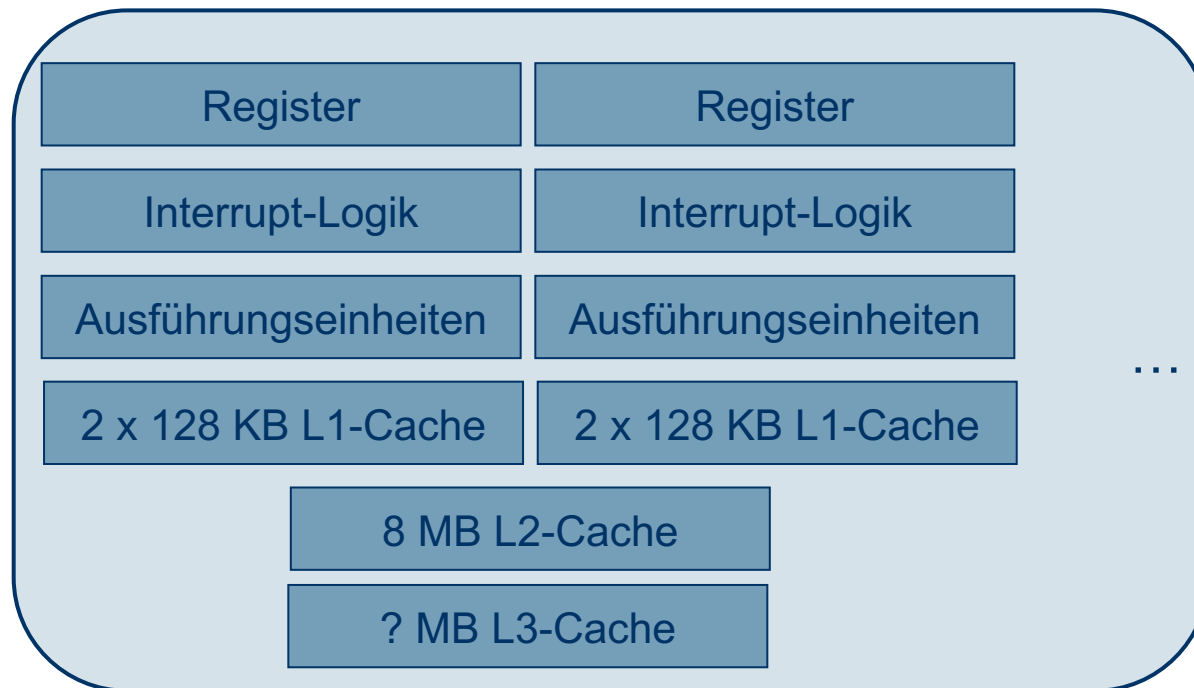
Beispiele für Multi-Core-Architekturen (1)

- Intel Core i9
 - 10-18 Kerne
 - 2,6 - 3,6 GHz
 - 95 - 140 W Verlustleistung
 - 2019 Laptop-Variante mit 8 Kernen



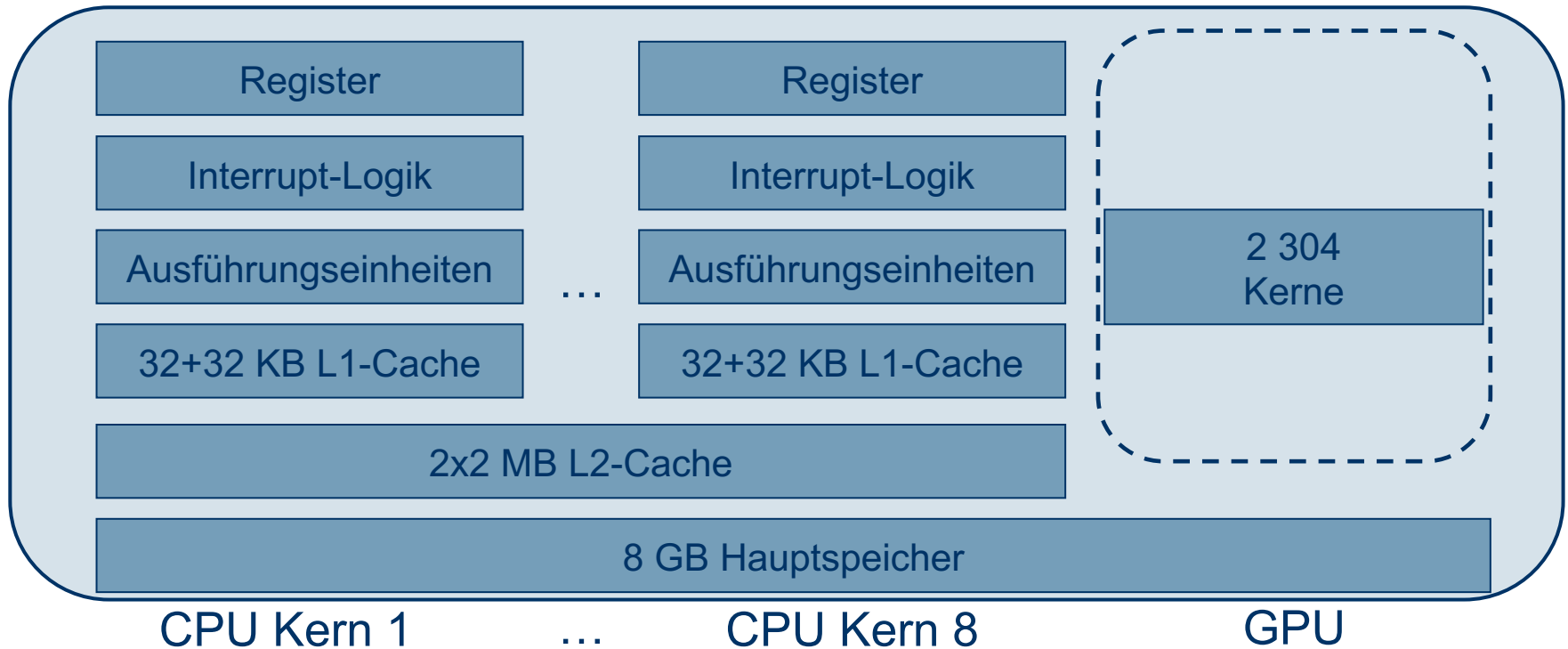
Beispiele für Multi-Core-Architekturen (2)

- Apple iPad pro 12,9“ (2018)
 - A12X, basierend auf ARMv8-A
 - 8 Kerne
 - 2,9 GHz
 - Apple Bionic GPU mit 7 mal ? SIMD



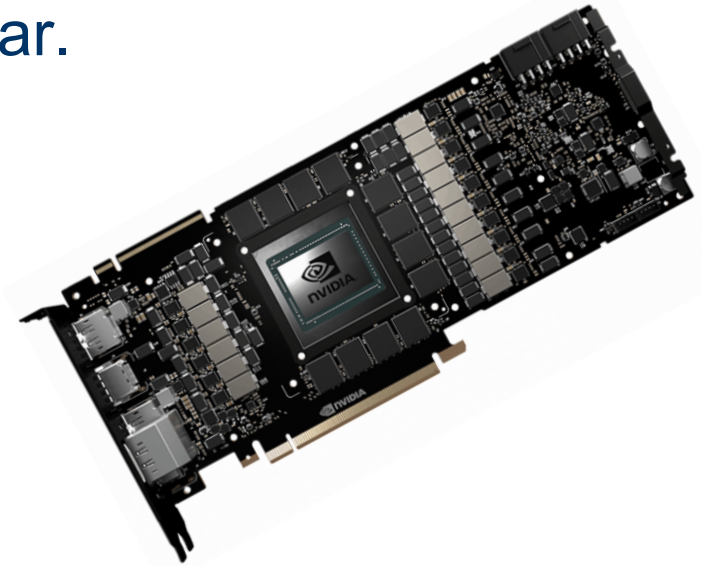
Beispiele für Multi-Core-Architekturen (3)

- AMD Accelerated Processing Unit
 - Z.B. in Playstation 4 pro
 - Heterogene Architektur mit
 - Multi-Core CPU (8 Core AMD)
 - GPU (AMD Radeon)
 - Gemeinsamer Hauptspeicher



Beispiele für Multi-Core-Architekturen (4)

- Graphikkarten für allgemeine *parallele* Rechenaufgaben verwendbar.
- NVIDIA-Graphikkarten
 - Varianten für Endbenutzer zum „Gaming“ oder für Berechnungsaufgaben.
 - Programmierung mittels CUDA oder OpenCL.



	Pascal	Volta	Turing
Kernanzahl	3 840	5 120	4 608
Taktfrequenz	1404 Mhz	1200 MHz	1590 Mhz
Verlustleistung	250 W	250 W	250 W

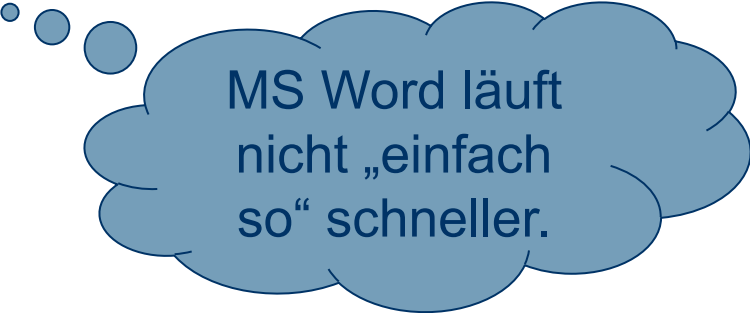
Multi-Core liegt im Trend

- Multi-Core-Architekturen existieren bereits
 - sowohl im High-Performance-Computing
 - als auch im Heimanwenderbereich.
- Sie stellen eine mögliche Lösung für die Probleme der Hardware-Entwicklung dar.
- Mehr Leistung bei meist geringerem Energieverbrauch.
- Teilweise heterogene, teilweise homogene Architekturen.
- Heute i9 „Coffee Lake“ 8 Kerne,
in spätestens 5 Jahren könnte typischer „CIP-Pool“-Rechner
vermutlich 32 Kerne haben.

„*The free lunch is over.*“

Herb Sutter

- Früher: schnellere CPUs \Rightarrow kürzere Laufzeiten
- Heute: mehr CPUs \Rightarrow *kürzere Laufzeiten?*
 - Nicht automatisch!
 - Parallele Programmierung
 - lohnend ... notwendig
 - aufwändig ... nicht-trivial



MS Word läuft
nicht „einfach
so“ schneller.

Gründe für diese Vorlesung (1)

- Warum will man Parallelität in den Programmen?
 - Reduktion der Bearbeitungszeit für ein Problem.
 - Berechnung größerer Probleme bei gleicher Rechenzeit.
 - Berechnung in Echtzeit.
 - Höherer Durchsatz der Anwendungsprogramme.
Wartezeiten (z.B. E/A-Geräte) mit anderen Aktivitäten nutzen.
 - Höhere Reaktionsgeschwindigkeit.
Aktivitäten mit unterschiedlichen Prioritäten versehen.
 - Klarere Programmstrukturen, insbesondere für Aktivitäten, die externe Geräte steuern.
- Parallele Programmierung als Schlüssel zur Leistung moderner Architekturen.

Gründe für diese Vorlesung (2)

- Warum will man Funktionale Programmiersprachen?
 - Gemeinsam genutzte Datenstrukturen erschweren die fehlerfreie Programmierung, sowohl von sequentiell als auch von parallelem Code.
 - Mit funktionalen Sprachen kann man diese Komplexität für den Programmierer reduzieren.

- Im ersten Semester haben Sie wie folgt programmiert:

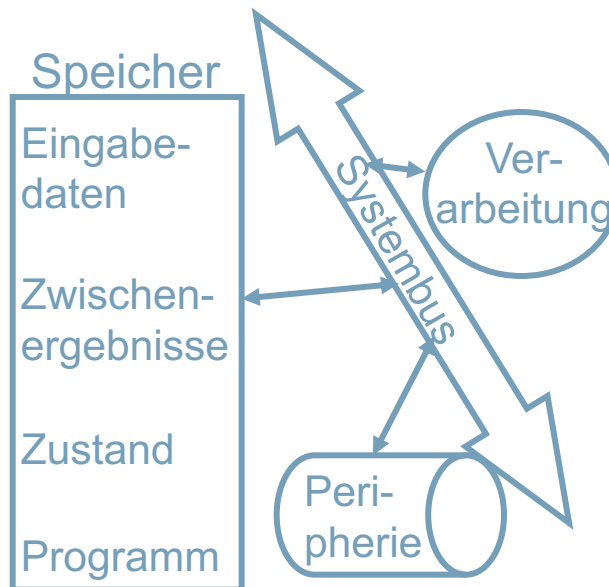
- *ein Programm* arbeitet alleine mit
- *seinen* Datenstrukturen, die im Hauptspeicher liegen.

} ein Instruktionsstrom
„Single Instruction“

} ein Datensatz
„Single Data“

SISD

Die Ausführung erfolgt konzeptuell auf einem Prozessor nach Von-Neumann.

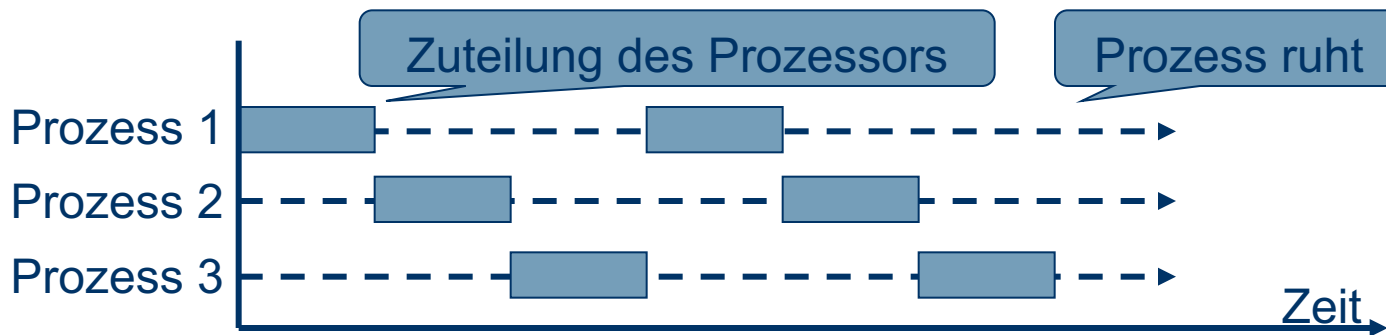


In der Klassifikation von Flynn

- *Prozess* = ein im Speicher(zugriff) befindliches Programm mit seinen dafür nötigen Datenstrukturen:
 - Anwendungsdaten
 - Verwaltungsdaten, wie Methodenstapel, Programmzähler, ...
 - Systemzustand (geöffnete Dateien, ...)

Rolle des Betriebssystems (1)

- Moderne Betriebssysteme können mehrere Prozesse nebeneinander ausführen.
 - Tatsächlich parallele Ausführung bei mehreren Prozessoren.
 - *Quasi-parallele* Ausführung, wenn nur ein Prozessor vorhanden ist.
- } *nebenläufig, asynchron*



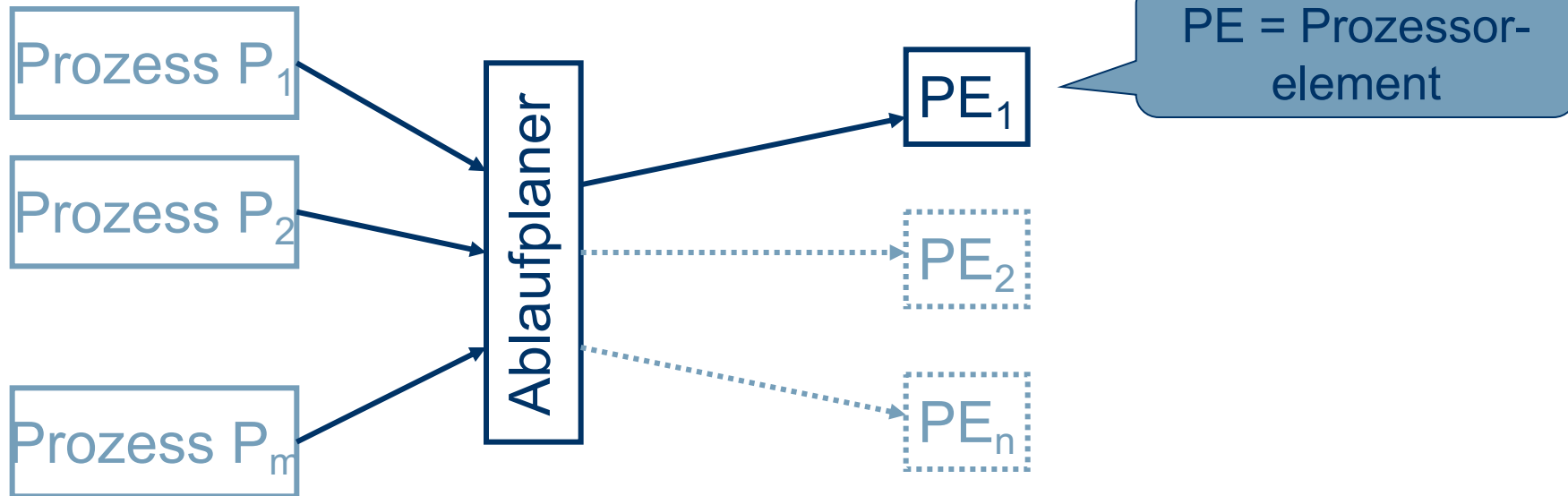
Alle Prozesse scheinen gleichzeitig voranzukommen (wenn auch langsam).

Rolle des Betriebssystems (2)

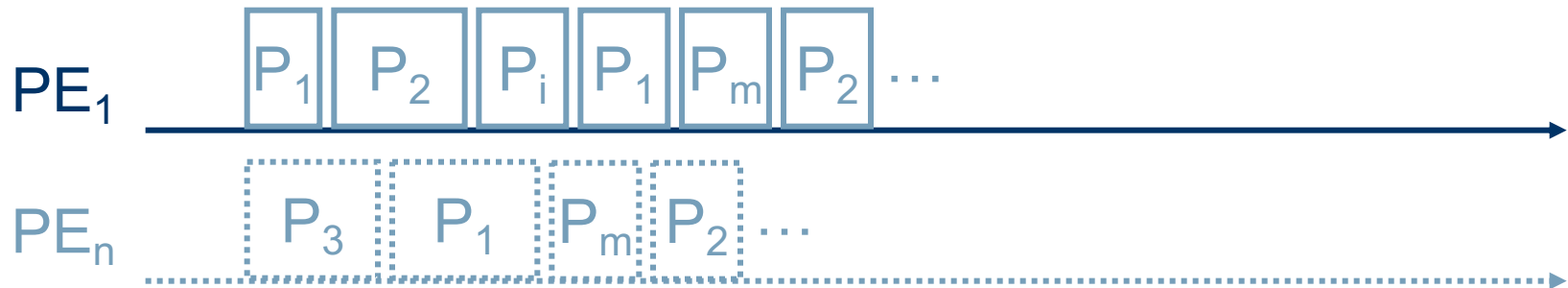
- Der *Ablaufplaner („scheduler“)* teilt die Prozesse dem Prozessor bzw. den Prozessoren zu, damit diese (abwechselnd) vorankommen.
 - Er nutzt dabei Wartezeiten (Festplatte, ...) aus.
- Die Schutzfunktionen des Betriebssystems sorgen dafür, dass diese Prozesse *getrennte Adressräume* (und sonstige Ressourcen) haben und ein Prozess nicht auf die Datenstrukturen anderer Prozesse zugreifen kann.
- Ideal, wenn Prozesse *völlig unabhängig* voneinander sind.

Zuteilung (1)

- Zuteilung räumlich:



- Zuteilung zeitlich:



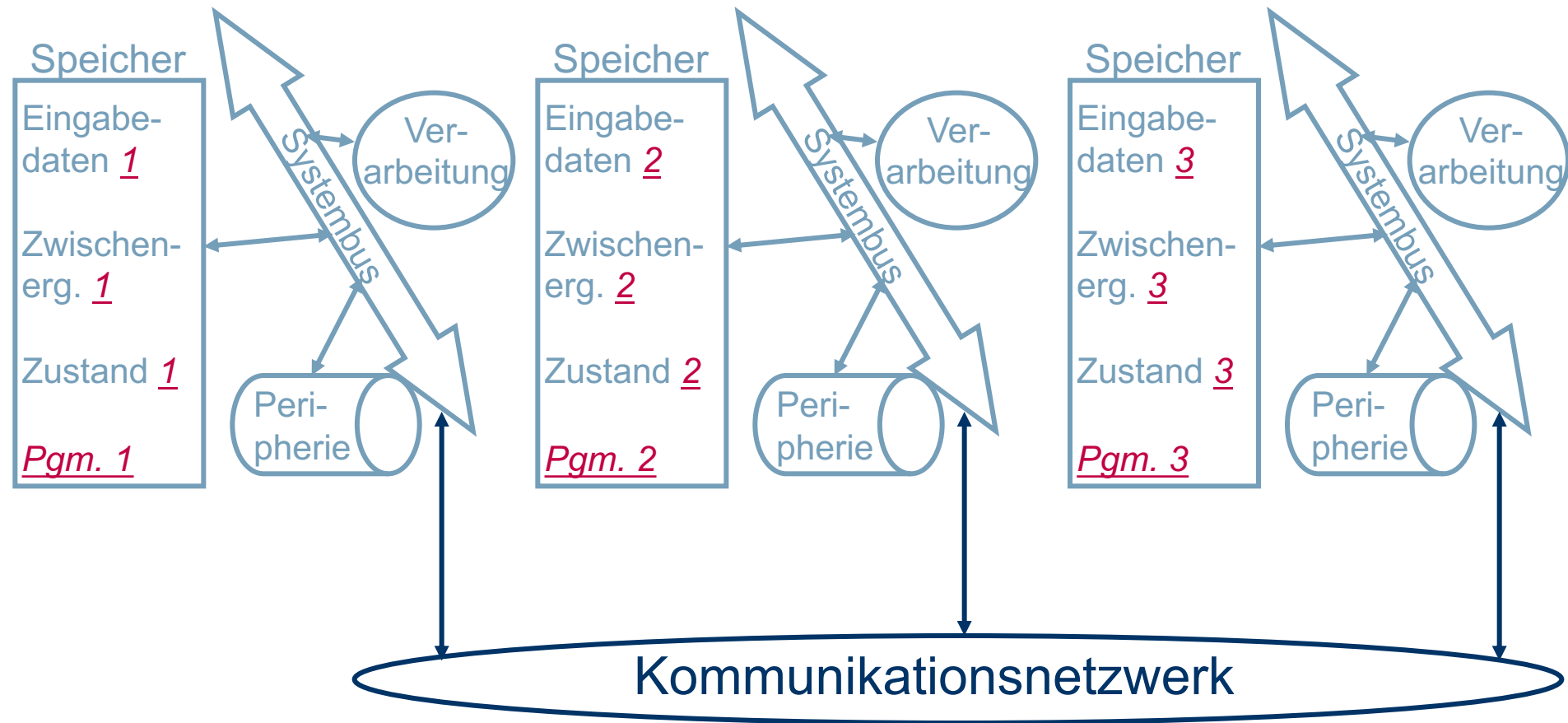
MIMD-Software (1)

- Es gibt Probleme, die in wenige/mehrere Prozesse *grobgranular* zerlegt werden können, die für dieses Ausführungsmodell des Betriebssystems geeignet sind.
 - Solche Prozesse können sich explizit Botschaften zusenden.
 - Eine *explizite Sendeoperation* in einem Prozess muss mit einer *expliziten Empfangsoperation* gepaart werden.
 - Es gibt Bibliotheken mit höheren Kommunikationsprotokollen, z.B. Rundruf-Funktion, ...
 - Der Programmierer schreibt also *mehrere Programme*, die auf mehreren Prozessoren laufen; jedes hat seine *eigenen Daten im eigenen Adressraum*.
- *MIMD-Software, „Multiple Instruction Multiple Data“*

In der Klassifikation von Flynn.

MIMD-Software (2)

- MIMD-Software auf n Von-Neumann-Rechnern:



MIMD-Rechner, DMP „distributed memory parallel“, lose gekoppelt

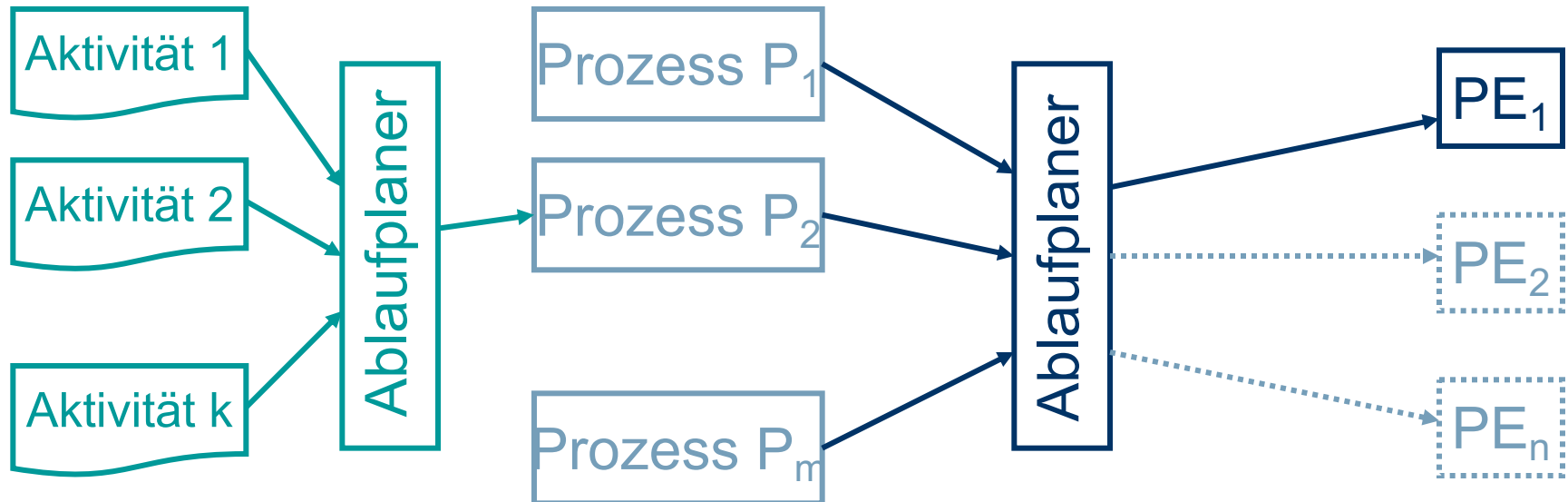
MIMD-Software (3)

- Wenn es schon schwierig ist, ein Programm korrekt zustande zu bringen, wie schwer ist dann erst MIMD?
 - Einfachere Subklasse: *SPMD*, „Single Program Multiple Data“
 - Dasselbe Programm läuft auf allen Prozessoren.
 - Die Prozesse haben getrennte Adressräume.
 - Im Allg. ist nur die Prozessnummer verschieden.
- In zukünftigen Modulen Ihres Studiums:
- Rechnerbündel/Cluster Computing
 - Verteilte Systeme
 - MPI-Bibliothek („message passing interface“).
 - ...

- Diese Vorlesung: *SMP*, „Shared Memory Parallel“
 - *Ein* Programm, bei dem die Parallelität
 - *in einem Adressraum* stattfindet.
 - Für eng gekoppelte Rechner (ein Arbeitsspeicher).

Zuteilung (2)

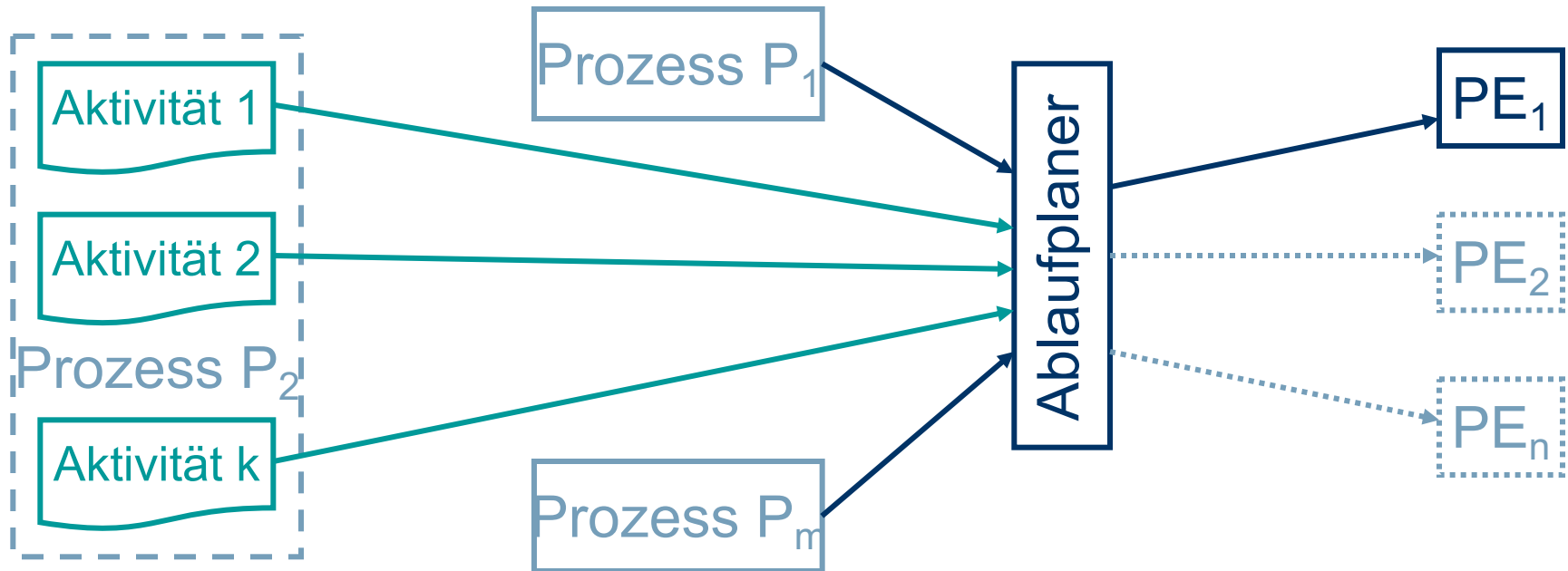
- Separater Ablaufplaner der Anwendung:



- Wenn der Prozess P_2 einem Prozessor zugeteilt ist, dann wählt der Ablaufplaner der Anwendung aus, welche Aktivität weitergeführt wird.
- Java: die JVM hat *manchmal* einen separaten Ablaufplaner.

Zuteilung (3)

- Aktivitätsgewahrer Ablaufplaner des Betriebssystems:



- E/A-Unterbrechung einer Aktivität verursacht nicht automatisch einen Prozesswechsel (viel aufwändiger).



Nach Carl-Adam Petri (deutscher Mathematiker, 1926-2010) benannte bipartite Graphen, die gut zur Modellierung des Verhaltens nebenläufiger Systeme geeignet sind.

Petri-Netze finden sich heute z.B.

- in Form von Aktivitätsdiagrammen in der UML,
- in der Workflow-Modellierung,
- ...

Petri-Netze (2)

- **Petri-Netze** bestehen aus: Für Zustände, Bedingungen
 - **Stellen** (Plätze) werden als Kreise dargestellt.
 - Stellen können ganzzahlig bewertet werden.
Man spricht von der **Belegung der Stelle**.
Wenn nichts *in der Stelle (im Kreis)* angegeben ist, wird 0 angenommen.
 - Statt mit Zahlen kann die Belegung der Stelle auch durch **Token** (Punkte, Marken) in der Stelle angegeben werden.
 - Stellen können eine Kapazität haben, die angibt, wie viele Marken maximal aufgenommen werden können.
Wenn nichts außen an der Stelle (am Kreis) angegeben ist, wird ∞ angenommen, also eine unbegrenzte Kapazität.
 - **Transitionen** (Schalter) werden als Striche (manchmal auch als Rechtecke) dargestellt.
 - ...



Für Aktionen, Ereignisse

- *Petri-Netze* bestehen aus:

- ...

- *Pfeilen*,

- die je eine Stelle mit einer Transition verbinden oder umgekehrt,
 - aber nicht 2 Stellen oder 2 Transitionen miteinander verbinden.
 - Pfeile können ganzzahlig bewertet werden (wenn nichts annotiert ist, dann wird 1 angenommen).

- Die Anzahl der Token in allen Stellen des Netzes heißt *Markierung* (oder *Belegung*) des Netzes.

- Anfangsmarkierung

*Eingangs-
stelle* von t



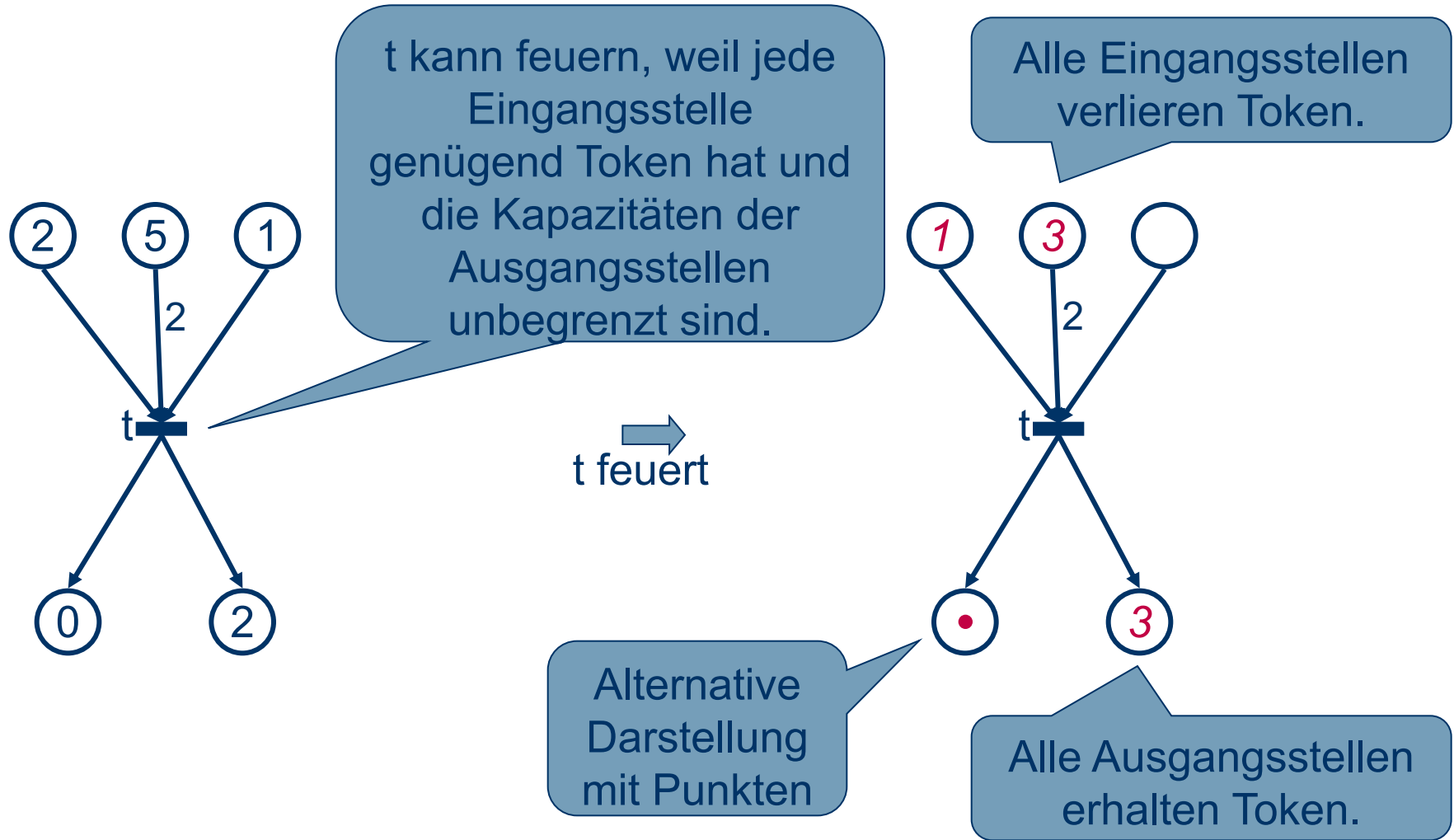
*Ausgangs-
stelle* von t

Kurznotation:

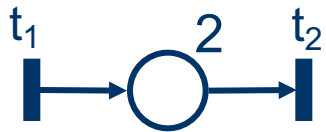


- Schaltregeln:
 - Eine Transition t *kann* feuern (schalten),
 - wenn jede Eingangsstelle e von t mit mindestens so vielen Token belegt ist, wie das Gewicht des Pfeils $e \rightarrow t$ angibt *und*
 - wenn in jeder Ausgangsstelle a die Zahl der vorhandenen Marken *nach dem Schalten* von t die Kapazität von a nicht übersteigt.
 - Wenn die Transition t *schaltet*, werden in jeder ihrer Eingangsstellen entsprechend dem Pfeilgewicht Token entfernt. Ebenso werden in jeder Ausgangsstelle Token (entsprechend dem Pfeilgewicht) hinzugefügt.
 - Das Schalten erfolgt atomar und braucht keine Zeit. Diese Modellierung *ignoriert die Zeit*.
- Ein Petri-Netz feuert, wenn eine der Transitionen feuert.

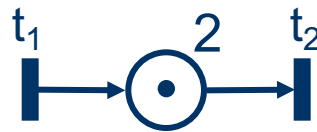
Schaltregeln am Beispiel (1)



Schaltregeln am Beispiel (2)

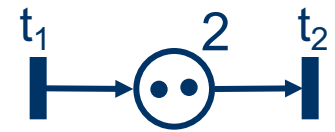


t_1 feuert



t_2 feuert

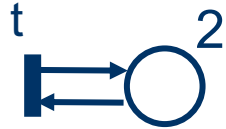
t_1 feuert



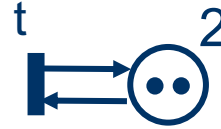
t_2 feuert

Hier können
spontan neue
Marken
entstehen.

Schaltregeln am Beispiel (3)

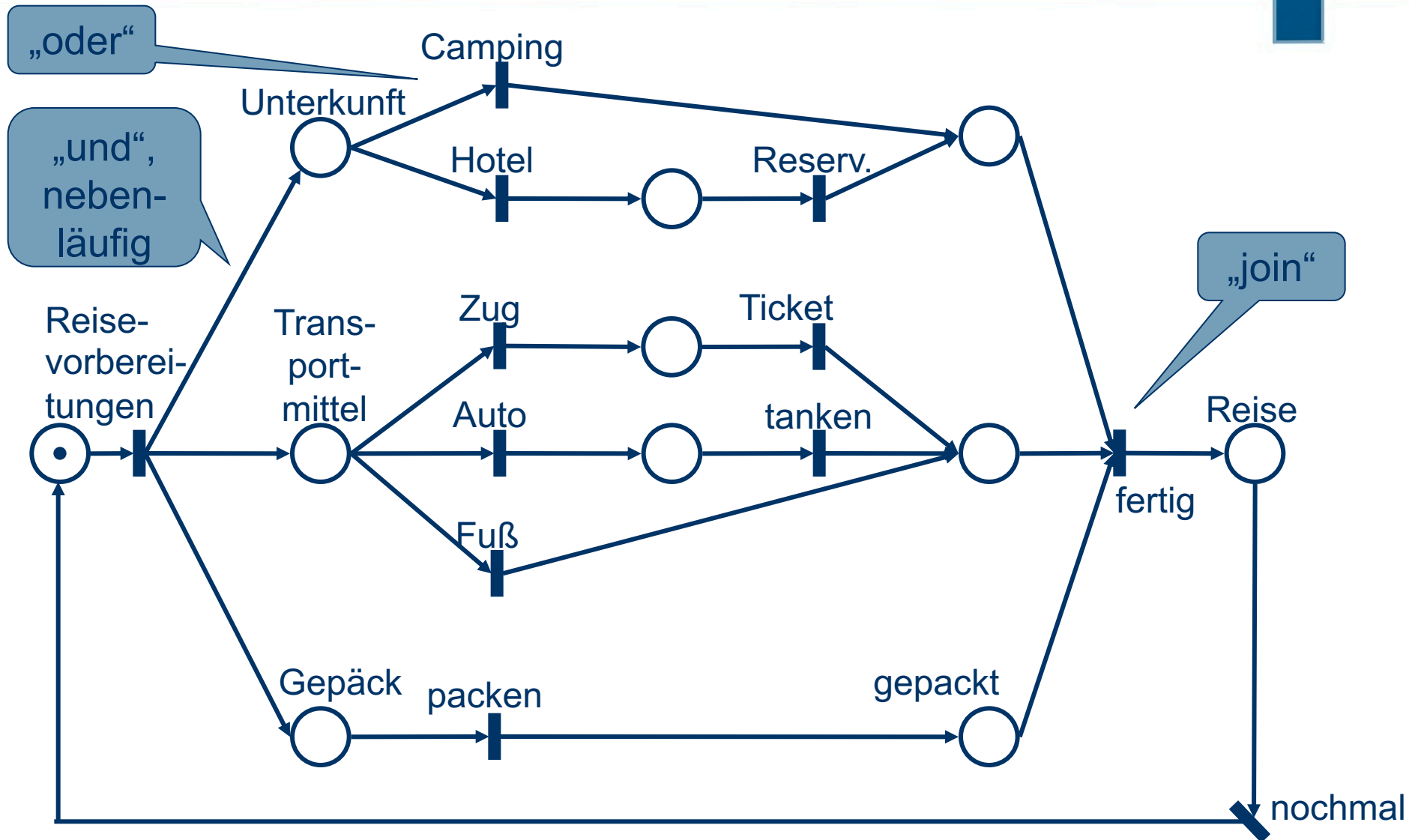


Kann **nicht** feuern, da an der Eingangskante keine Marke anliegt (Bedingung 1).



Kann feuern, da
1.) an der Eingangskante eine Marke anliegt und
2.) beim Schalten eine Marke entfernt und eine Marke erzeugt wird.
Danach hat die Stelle noch immer höchstens 2 Marken.

Modellierung mit Petri-Netz am Beispiel

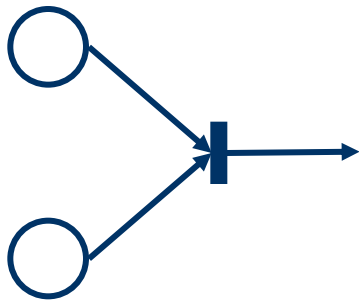


Modellierungselemente (1)

- Vorher-Nachher-Beziehung

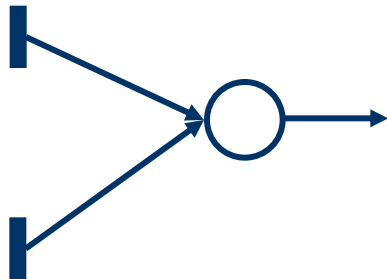


- Notwendige Bedingung



Erst wenn beide Eingangsstellen ein Token haben, geht es weiter.

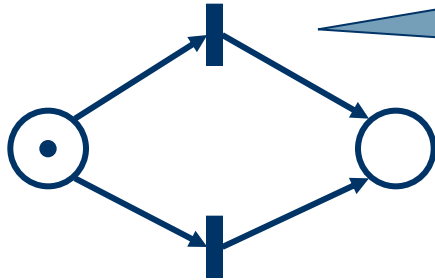
- Hinreichende Bedingung



Es reicht, wenn eine der Transitionen feuert, damit es weiter geht.

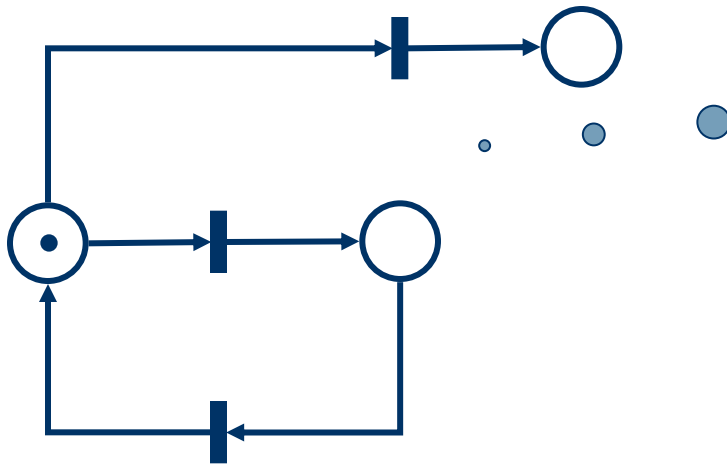
Modellierungselemente (2)

■ Alternative



Konflikt: zwei Transitionen brauchen dieselbe Marke zum Schalten.

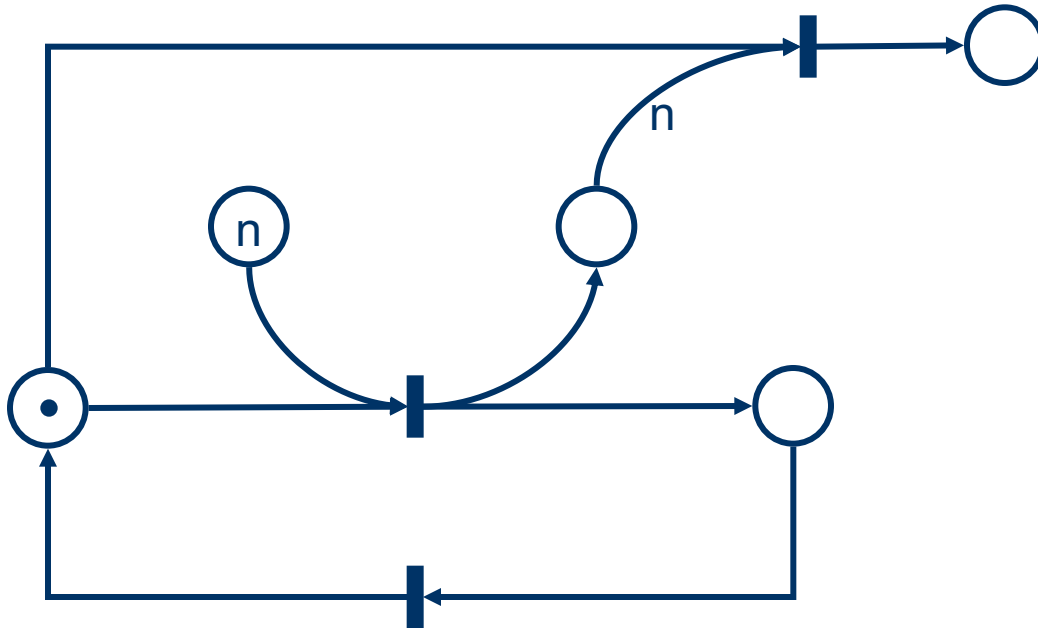
■ Wiederholung



Schleifenabbruch
ist hier zufällig.

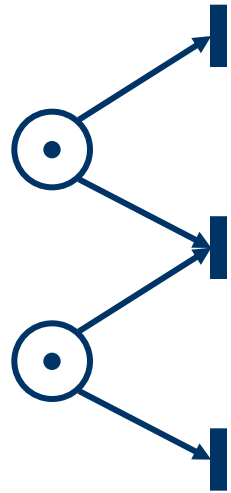
Modellierungselemente (3)

- for-Schleife mit n Durchläufen

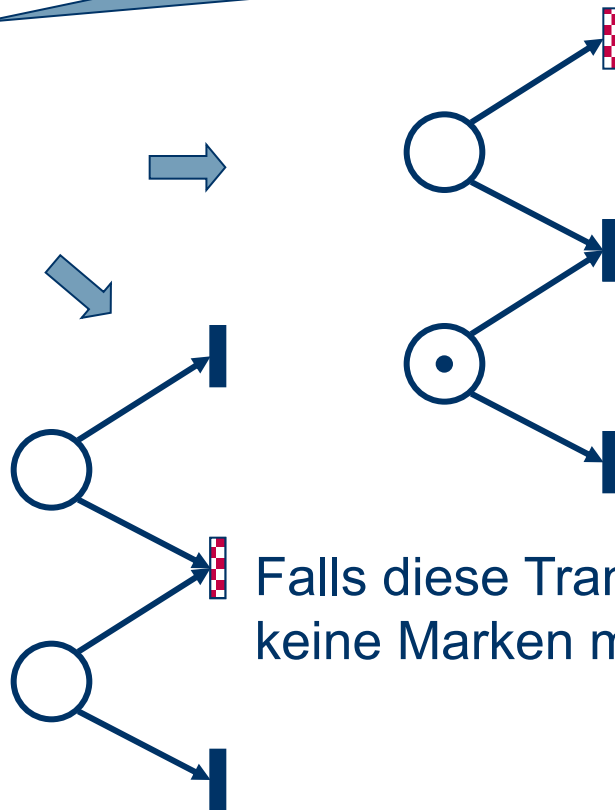


Modellierungselemente (4)

■ Konfusion



Transition liegt im Konflikt mit 2 Transitionen.
Mit dem ersten Schalten einer Transition wird die Konfusion aufgelöst. Danach ist kein Konflikt mehr vorhanden.



Falls diese Transition zuerst schaltet, dann kann die mittlere Transition nicht mehr feuern.
→ Kein Konflikt mehr.

Falls diese Transition zuerst schaltet, dann sind keine Marken mehr vorhanden → Kein Konflikt mehr.

Vielen Dank für Ihre Aufmerksamkeit!

