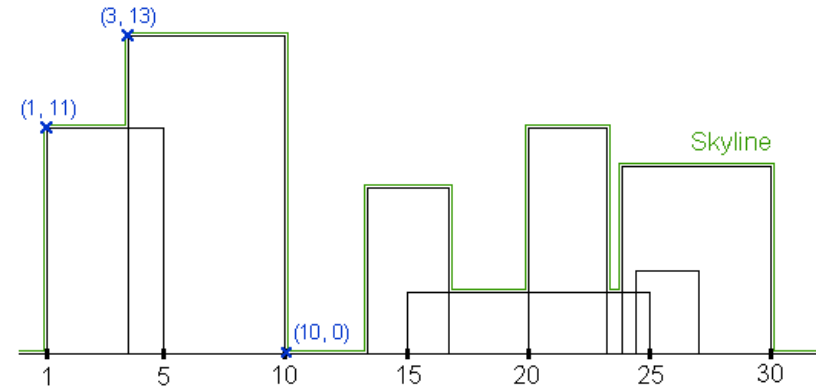


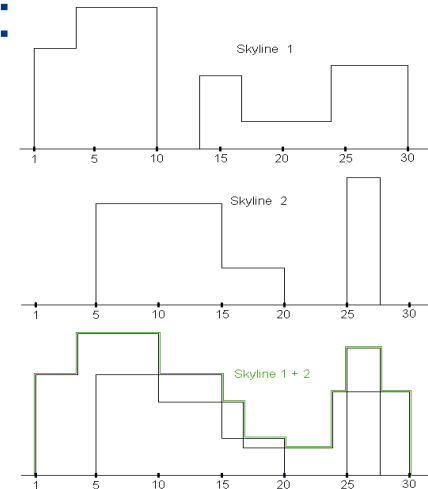
Das Skyline-Problem:

- Gegeben: n sich überlappende Häuser
- Gesucht: eine Skyline dieser Häuser
- Dieses Problem wurde in der Vorlesung mittels DuC-Algorithmus in $O(n \log n)$ Schritten gelöst



In Info 1 wurde ein *ähnliches* Problem besprochen:

- Gegeben: zwei Skylines(!)
- Gesucht: eine Skyline durch Überlappung
- Dieses Problem ist lösbar in $O(n)$ Schritten - mit n der Anzahl der Häuser in der Skyline



15.1 Maschine und Prozess

15.2 Prozesse

15.3 Ablaufplanung in Prozessmengen

15.4 Interaktion zwischen Prozessen

15.5 Verklemmungen

15.6 Parallelität



Innensicht

Algorithmenentwurf und Algorithmen (incl. Aufwände)

- Suche und Sortieren (Reihen, Folgen, Bäume)
- Graphen
- Dyn. Programmieren
- Geometr. Algorithmen

Prozesse

- Prozesse
- Prozesskommunikation
- Parallelität/Synchronisation
- Java: Thread-Programmierung

Außensicht

Skalierbarkeit und Persistenz

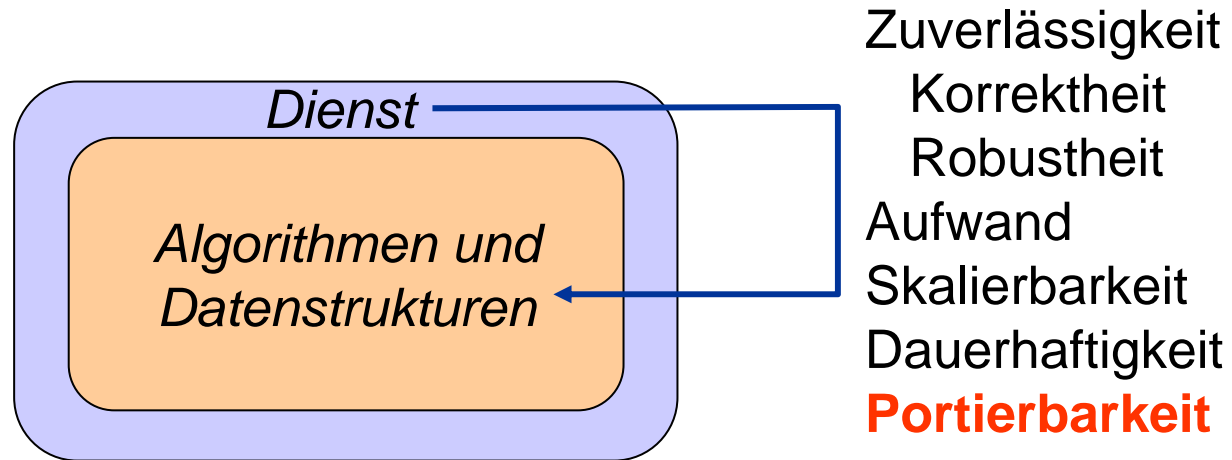
- Mengenorientierung
- Assoziativer Zugriff, Schlüsselbegriff
- Aufwände
- Polymorphie
- Schema
- Deskr. Sprachen (SQL)

Autonome Dienste

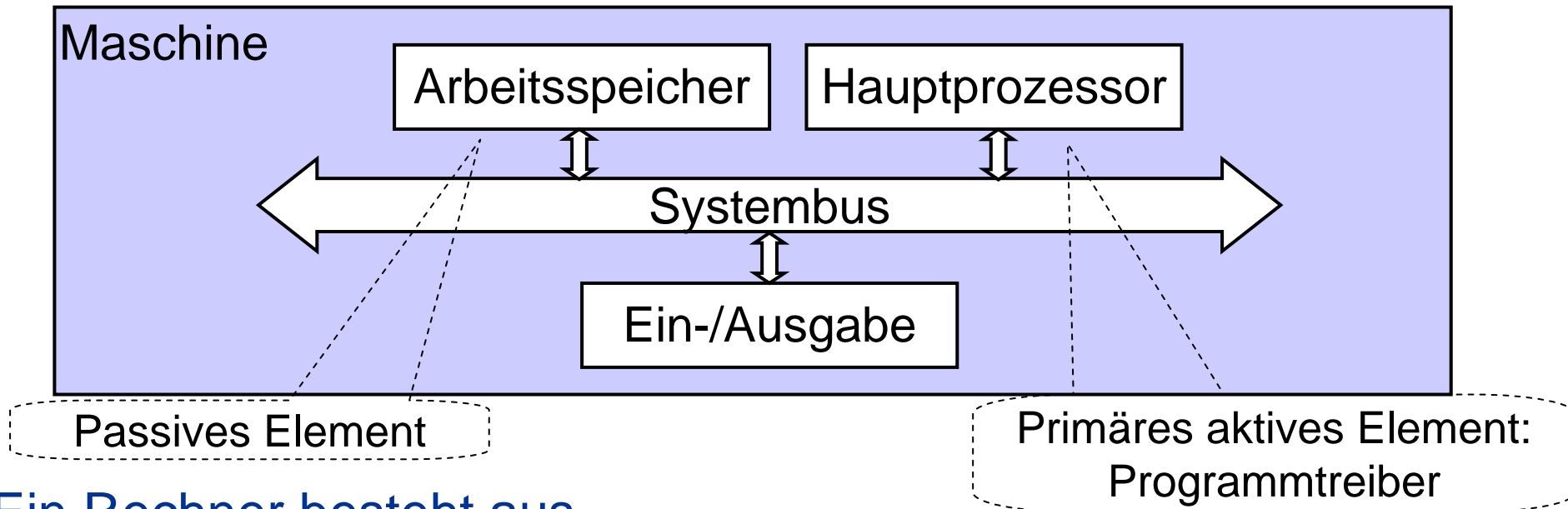
- Enge/lose Kopplung
- Protokolle / Automaten
- Verteilung



Software-/Hardware-Hierarchie



von-Neumann-Rechner als Maschine

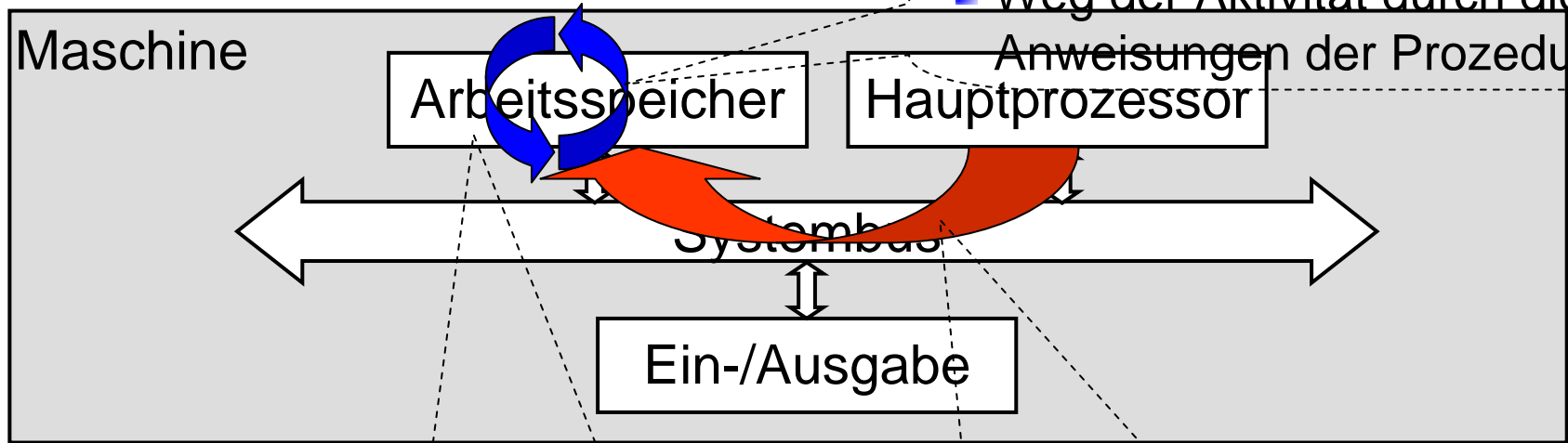


Ein Rechner besteht aus

- Arbeitsspeicher: Speicher für Daten *und* Programme
- Hauptprozessor:
 - Steuer- oder Leitwerk: steuert u.a. Ablauf über Vorgänge auf dem Systembus
 - Rechenwerk (Arithmetic Logic Unit, ALU)
- Ein-/Ausgabeprozessoren (Tastatur, Bildschirm, Magnetplatte, etc.)
- Systembus für Daten und Befehle



Abstraktion



Aktivitätsbahn:

- Weg der Aktivität durch die Anweisungen der Prozedur.

Prozedur (im weiteren Sinn):

- Funktional abgeschlossenes Programmstück inklusive Datenstruktur zur Aufnahme eines Zustands.
- Vorschrift für eine Zustandsänderung.

Aktivität:

- Zustandsänderung durch sequenzielle Ausführung der Anweisungen der Prozedur.
- Dazu muss der Prozedur ein Anfangszustand verliehen werden (Ausprägung der Prozedur).

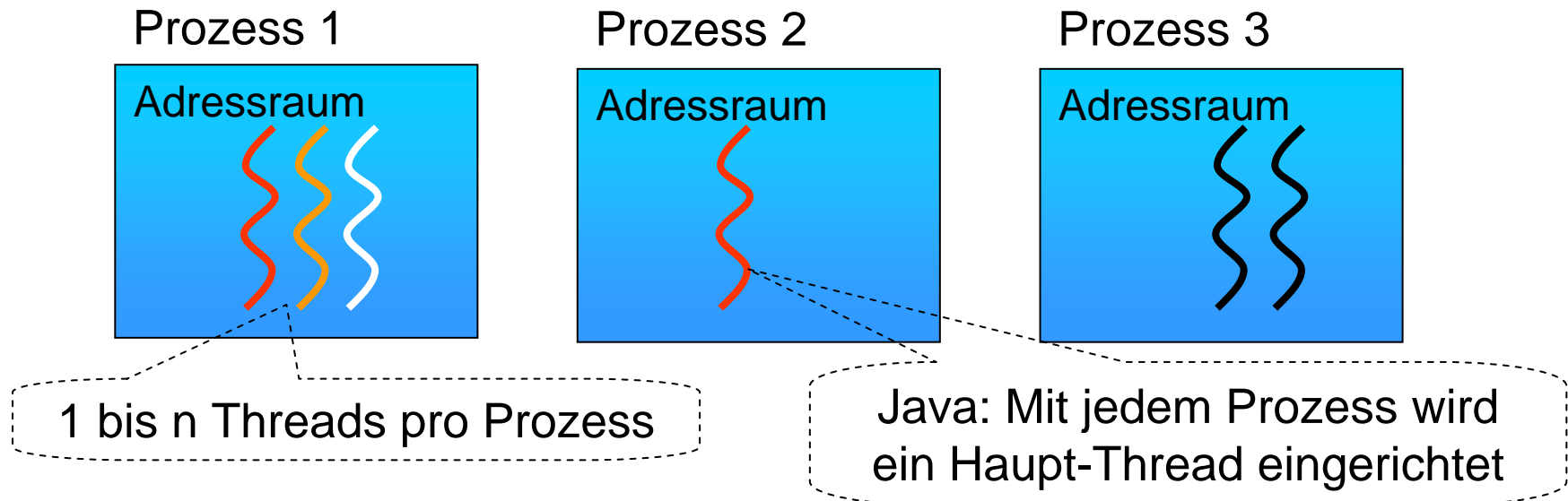


Prozess:

- Träger einer Aktivitätsbahn, die in einem eigenen Adressraum im Hauptspeicher abläuft (physische Kapselung).
 - z.B. erfolgt die Ausführung eines Anwendungsprogramms als Prozess
- Ein Prozess kann nur auf Daten in seinem Adressraum zugreifen.

Leichtgewichtiger Prozess (Prozessfaden, *thread*):

- Träger einer Aktivitätsbahn, die den Adressraum mit anderen Threads teilt.



Betriebsmittel

- Prozess benötigt Betriebsmittel (CPU, Speicher, Dateien, ...)
- Prozess ist Betriebsmittel, das vom Betriebssystem verwaltet wird (Erzeugung, Terminierung, Scheduling ...)

Prozesswechsel

- Prozessor führt in jeder Zeiteinheit maximal einen Prozess aus.
- Mehrere Prozesse auf einem Rechner erfordern Prozesswechsel. Entscheidung fällt das Betriebssystem.

Abschottung

- Prozesse sind gegeneinander abgeschottet.
- Prozesse besitzen (virtuell) eigene Betriebsmittel (z.B. Adressraum).



Prozess (process, task)

- Eine durch ein Programm spezifizierte Folge von Aktionen, deren erste begonnen, deren letzte aber noch nicht abgeschlossen ist

Prozess = Programm in Ausführung

- Instanz eines Programms
- Programm selbst ist ein passives Ding (z.B. auf Papier)

Es kann gleichzeitig mehrere Prozesse als verschiedene Instanzen des gleichen Programms geben.

- z.B. WWW-Browser mehrerer Benutzer

Eigenschaften

- Eine Folge von Maschinenbefehlen, die durch das ausgeführte Programm (program code, text section) festgelegt sind
- Der aktuelle Zustand der Bearbeitung ist durch den Programmzähler und die Registerinhalte des Prozessors beschrieben (internal state)
- Der Inhalt des Stapel-Speichers, auf dem temporäre Variablen und Parameter für Funktionsaufrufe verwaltet werden (stack)
- Der Inhalt des Speichers, in dem die globale Daten des Prozesses gehalten werden (data section)



Ein Prozess ist repräsentiert durch eine spezielle Datenstruktur, den Prozessleitblock, der alle relevante Information enthält

- Prozesscharakteristika
Prozesskennung, Name des Prozesses
- Zustandsinformation
Befehlszähler, Stapelzeiger, Registerinhalte
- Verwaltungsdaten
Priorität, Rechte, Statistikdaten

In größeren Systemen kann es Hunderte von Prozessen geben.
Daher muss effizient mit ihnen umgegangen werden

Prozesse sind Betriebsmittel, die durch das Betriebssystem verwaltet werden

- Erzeugen
- Terminieren (Freigabe sämtlicher belegter Betriebsmittel)
- Scheduling



Im einzelnen sind folgende Informationen in einem Prozessleitblock enthalten

- Prozesszustand
- Programmzähler
- Registerinhalte des Prozessors (CPU)
- Daten, die die Bearbeitungsfolge regeln (Prioritäten, Statistiken)
- Daten, die zur Verwaltung des belegten Hauptspeichers benötigt werden
- Accounting Daten zur Abrechnung von benutzten Betriebsmitteln
- Daten über Ein- / Ausgabegeräte, die dem Prozess zugeordnet sind
- Daten über geöffnete Dateien und Netzwerkverbindungen
- Parameter, die die Zugriffsrechte des Prozesses beschreiben

Prozessleitblock

Zeiger	Prozess- zustand
Prozessnummer	
Programmzähler	
Register	
Zeiger auf Speicherbereiche	
Liste geöffneter Dateien	
<div style="text-align: center;"> • • • </div>	



Prozess befindet sich immer in einem definierten Prozesszustand.

Typische Prozesszustände

- new

Neuer Prozess wird erzeugt.

- ready

Prozess ist rechenbereit und wartet auf einen Prozessor.

- running

Prozess führt Instruktionen im Prozessor aus.

- waiting

Prozess wartet auf ein Ereignis.

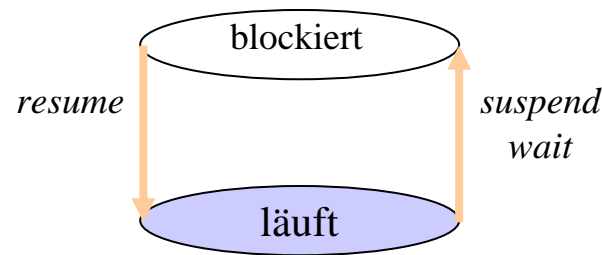
- terminated

Prozess hat seine Ausführung beendet.



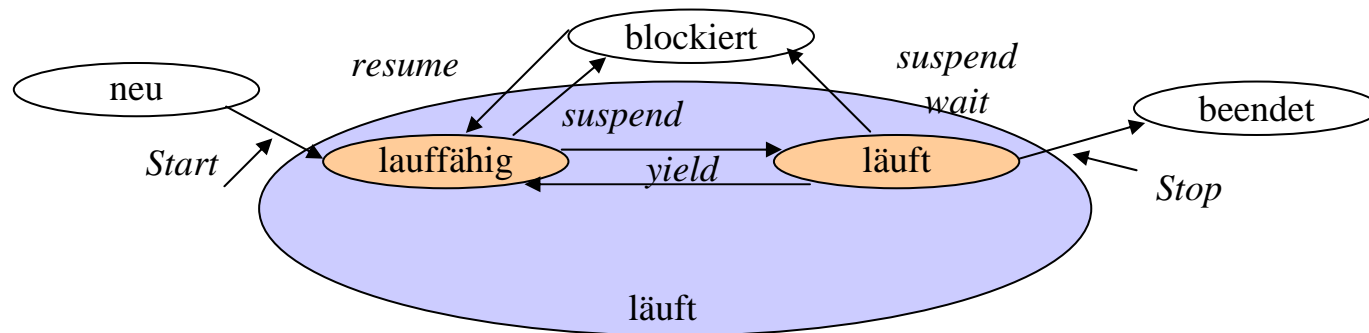
Vereinfacht betrachtet kann ein Prozess in zwei Zuständen sein

- blockiert (d.h. wartend auf Beendigung der E/A oder wartend auf ein Synchronisationssignal eines anderen Prozesses).
- läuft (d.h. wird bearbeitet)



Verfeinerung (da mehrere Prozesse quasi-gleichzeitig laufen)

- zu einem Zeitpunkt ist stets nur ein einziger Prozess aktiv, die anderen warten auf den Prozessor (lauffähig)
- unterscheide: blockiert (kann nicht weiter rechnen) und lauffähig (darf nicht weiter rechnen)
- von außen kann ein Prozess von jedem Zustand in den Zustand beendet versetzt werden
- Zustandswechsel lauffähig / läuft wird vom Betriebssystem (Scheduler) vorgenommen



Threads sind parallele Kontrollflüsse, die nicht gegeneinander abgeschottet sind

- laufen innerhalb eines Adressraums
- ggfs. sogar innerhalb eines „echten“ Prozesses
- teilen sich gemeinsame Ressourcen

Kontextwechsel zwischen Threads effizienter als zwischen Prozessen

- kein Wechsel des Adressraums
- kein Betriebssystem-Scheduling notwendig (sog. „Green Threads“)
- kein Retten und Restaurieren des Kontextes (nur Befehlszähler und Register)

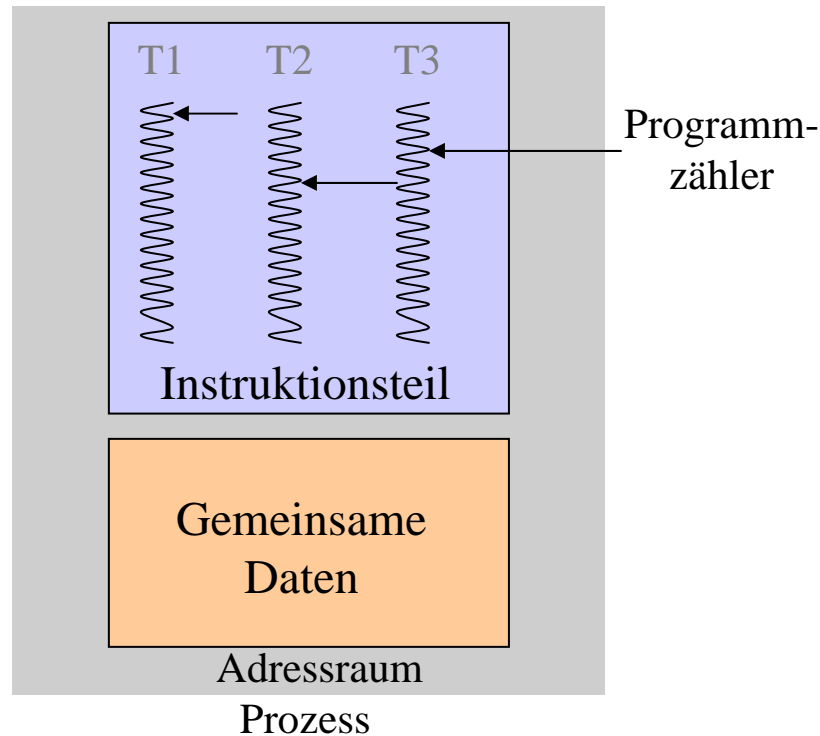
Pro Zeiteinheit sind viel mehr Threadwechsel als Prozesswechsel möglich

- wichtig für Server (z.B. WWW-Suchmaschinen), die pro Sekunde viele tausend Anfragen quasi-gleichzeitig bearbeiten müssen



Beispiel

- 3 Threads in unterschiedlichen Stadien (Programmzähler) innerhalb eines Prozesses



Man unterscheidet zwischen 2 Arten von Threads

Native Threads

- Werden vom Betriebssystem-Scheduler verwaltet
- Können auf Mehrprozessor-Maschinen effizient eingesetzt werden

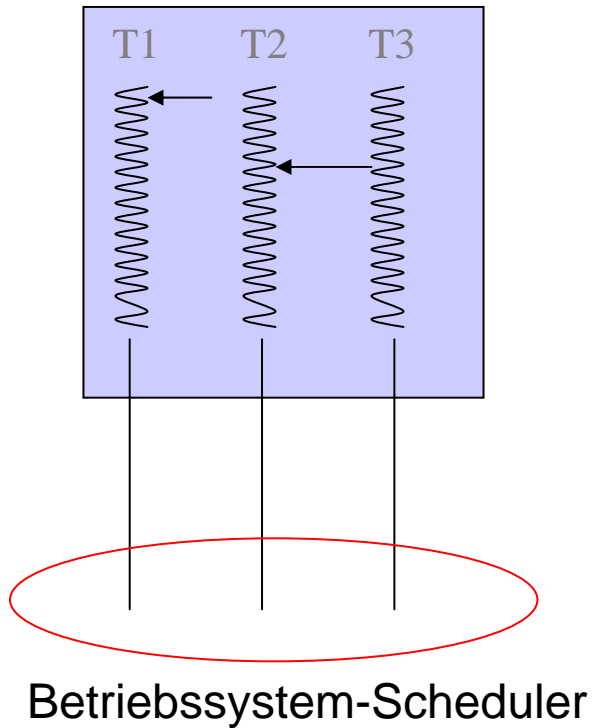
„Green“ Threads

- Sind unsichtbar dem Betriebssystem gegenüber
- Müssen ihr eigenes Scheduling implementieren
 - kann effizienter sein
 - Kontextwechsel so noch billiger
- Sind auf Mehrprozessor-Maschinen nicht parallel einsetzbar

Java kennt beide Formen von Threads

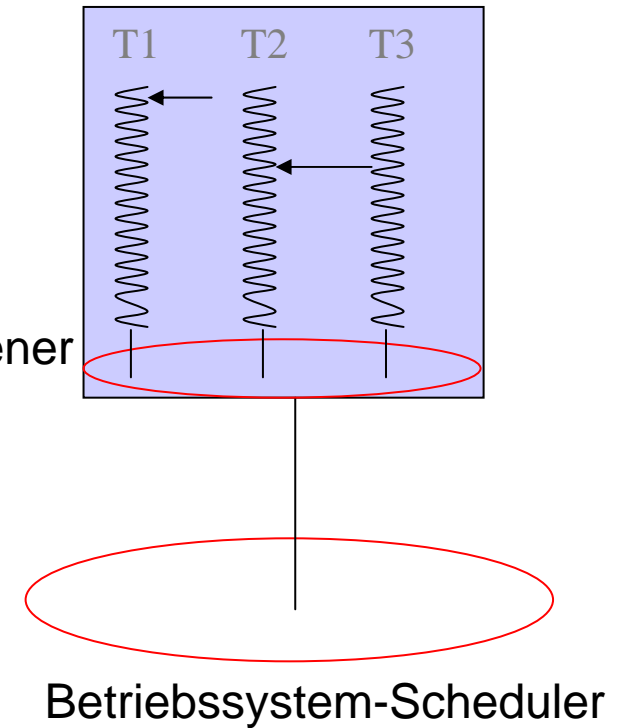


Native Threads



Green Threads

Prozess-eigener
Scheduler



Threads teilen sich gemeinsamen Adressraum

- Schutzfunktionen getrennter Adressräume ???
- → (besondere) Synchronisationsmechanismen erforderlich?



Rechner mit einem Prozessor

- Typischerweise laufen mehrere Prozesse auf einem Rechner.
- Prozesse müssen sich in der Nutzung des Prozessors abwechseln.
- Betriebssystem entscheidet, welcher Prozess wann den Prozessor benutzen darf.

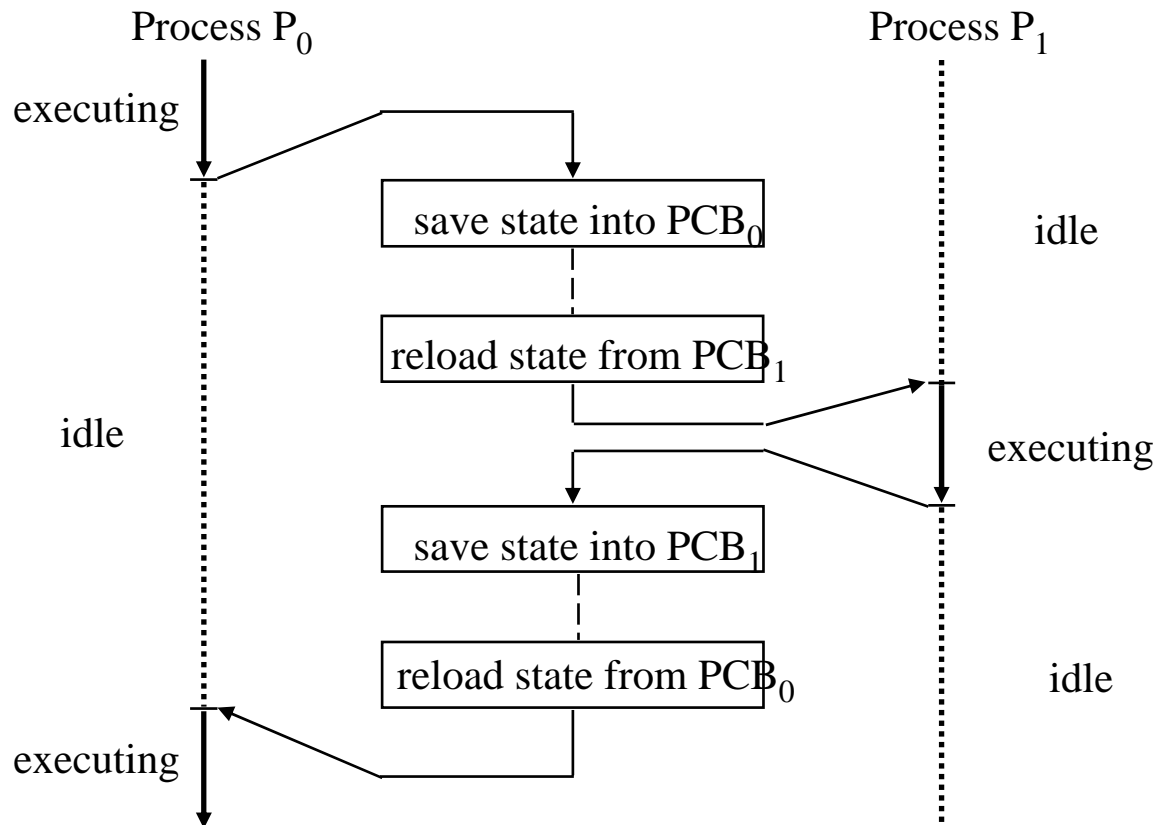
Komponenten des Scheduling

- Prozesswechselkosten
- Warteschlangenmodelle
- Scheduling-Verfahren



Wenn der laufende Prozess lauffähig oder blockiert wird, muss der aktuelle Kontext des Prozesses gesichert werden

- hierzu dienen die diversen Felder im Prozessleitblock
- Kontextwechsel sind relativ teuer (z.B. 122 μ s auf SUN SPARC 20)



Grundsätzliche Vorgehensweise bei der Ablaufplanung

- Jeder Prozess läuft in einem **virtuellen** Prozessor ab.
- Ist Zahl der Prozesse größer als die Zahl der realen Prozessoren, so muss ein **Ablaufplaner (scheduler, dispatcher)** die nebenläufige Ausführung simulieren
- Bei ihrem Start melden sich alle Prozesse beim Planer an und werden in eine **Warteschlange** einsortiert.
- Grundsatz: Suspendiere Ausführung eines Prozesses, nehme Ausführung eines neuen, wartenden (möglicherweise früher suspendierten) Prozesses auf (**verzahnte Ausführung** der Prozesse).
- Dazu erforderlich ein Signal (**Ereignis**) zum Wechsel zwischen den Prozessen.

Die Verteilung der Ressource Prozessor sollte „fair“ erfolgen.

Wir betrachten nachfolgend die Ablaufplanung für den Ein-Prozessor-Fall.



- Der Prozess-Scheduler wählt aus Prozessor-Warteschlange lauffähiger Prozesse den nächsten Prozess aus, der in den Status läuft („running“) wechseln darf

Anforderungen an einen Scheduler

- *Fairness*: Jeder Prozess soll fairen Anteil an der CPU bekommen
- *Effizienz*: CPU sollte möglichst immer belegt sein
- *Antwortzeit*: Antwortzeit für interaktive Benutzer soll minimal sein
- *Verweilzeit*: Verweildauer von (Batch-) Programmen soll möglichst gering sein
- *Durchsatz*: Anzahl bearbeiteter Prozesse pro Zeitintervall soll maximiert werden



Annahme

- Ausführungszeiten der Prozesse nicht bekannt
- Anzahl der zu rechnenden Prozesse kann sich dynamisch verändern

Ziel

- Optimierung von Leistungsmaßen wie z.B. die Durchlaufzeit durch die Bestimmung einer "geschickten" Ausführungsreihenfolge, wobei man Entscheidungen mit Strategien trifft, die mit (angenommenen) Eigenschaften von Prozessen begründet werden

Bewertung

- Analysen von allgemeinen Scheduling-Verfahren sind nur unter Verwendung von stochastischen Modellen möglich, da die Anzahl, das Ankunftsverhalten und die Rechenzeiten durch Zufallsvariablen beschrieben werden



Präemptiv (*preemptiv*)

- Laufende Prozesse können unterbrochen und später fortgesetzt werden.

Nicht-Präemptiv

- Laufender Prozess läuft, bis er blockiert oder fertig ist oder den Prozessor freiwillig freigibt (yield).



Deterministisches Scheduling

- Ausführungszeiten der Prozesse sind bekannt
- Prozesse werden zeitlich so angeordnet, dass sich ein gewünschtes Systemverhalten ergibt.
 - z.B. minimale Durchlaufzeiten

Probabilistisches Scheduling

- Lediglich die Erwartungswerte bzw. die Wahrscheinlichkeitsverteilung der Ankunfts- und Bedienzeiten sind bekannt
- Das Verhalten des jeweiligen Scheduling-Algorithmus wird unter der wahrscheinlichsten Last beschrieben
- Analyse und Bewertung probabilistischer Scheduling-Algorithmen mit Hilfe der Wahrscheinlichkeitstheorie



Sehr einfacher Scheduling-Algorithmus

- FCFS: First-Come-First-Serve
- Prozess (Task), der den Prozessor zuerst anfordert bekommt diesen
- Unterbrechung von Prozessen ist nicht gestattet (nicht-präemptiv)

Bewertung

- die mittlere Wartezeit kann sehr hoch werden
 - mittlere Wartezeit: Gesamtwartezeit aller Prozesse / Anzahl Prozesse
- Beispiel
 - $\tau := \{T_1, T_2, T_3\}$
 - Bearbeitungsdauern: $t_1 = 24, t_2 = 3, t_3 = 3$
 - Wie sieht Schedule aus, falls Ankunftsreihenfolge T_1 vor T_2 vor T_3 ?
 - Mittlere Wartelänge?

 - Wie sieht Schedule aus, falls Ankunftsreihenfolge T_3 vor T_2 vor T_1 ?
 - Mittlere Wartelänge?



Prinzip

- nicht-präemptive und präemptive Verfahren existieren
- Prozess mit kürzester Rechenzeit wird zuerst bedient, d.h., Rechenzeiten sind bekannt

Beispiel

- $\tau := \{T_1, T_2, T_3, T_4\}$
- $t(T_1) = 6, t(T_2) = 8, t(T_3) = 7, t(T_4) = 3$
- Gantt-Diagramm

t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
P																									

- Mittlere Wartezeit? ... und wie würde es bei FCFS-Scheduling aussehen?



Bewertung

- liefert minimale mittlere Wartezeit für eine gegebene Menge von Prozessen
- kurze Prozesse werden bevorzugt behandelt

Problem

- Ausführungszeiten der Prozesse in der Regel nicht im Voraus bekannt
- Schätzungen erforderlich, z.B. auf Basis der vorangegangenen Ausführungszeiten mittels „Exponential Average“
 - Sei t_n die gemessene Ausführungszeit des Prozesses P_n
 - Sei t_{n+1} die geschätzte Ausführungszeit für den nächsten Prozess P_{n+1}
 - Dann gelte:
 - $\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$
 - mit $0 \leq \alpha \leq 1$



Prinzip

- präemptiv, d.h. Prozess wird in einem Schedule i.d.R. nicht komplett ausgeführt
- Zuteilung des Prozesses jeweils für eine Zeitscheibe, danach wird er wieder in die Warteschlange eingereiht
- Abarbeitung der Warteschlange gemäß FCFS

Bewertung

- kann zu hohen mittleren Wartezeiten führen
- Leistungsfähigkeit stark von der Länge der Zeitscheiben abhängig
 - typische Längen: 10 bis 20 Millisekunden

Beispiel

- $\tau := \{T_1, T_2, T_3\}$
- $t(T_1) := 24, t(T_2) := 3, t(T_3) := 3$
- Die Länge einer Zeitscheibe sei 4
- Wie sieht das zugehörige Gantt-Diagramm aus?

t	0	4								
P										



Prinzip

- Jeder Prozess wird mit einer Priorität versehen
- Jede Prioritätsstufe besitzt eigene Warteschlange, die gemäß FCFS abgearbeitet wird
- Warteschlange mit höchster Priorität wird als erstes bearbeitet

Bewertung

- Prozesse mit niedriger Priorität können unendlich lange warten müssen („verhungern“)
- „Aging“ kann Abhilfe schaffen, d.h. die Priorität von Prozessen wird aufgrund ihres „Alters“ erhöht



Beispiel

- $t := \{T_1, T_2, T_3, T_4, T_5\}$
- Die Länge einer Zeitscheibe sei 5. Wie sieht das zugehörige Gantt-Diagramm für einen Prozessor aus?

T	1	2	3	4	5
t	10	1	2	1	5
$Prio$	3	1	3	4	2

t	
P	

- Wie hoch ist die mittlere Wartezeit?



Problem

- Zeitschranken müssen eingehalten werden

Strikte Echtzeitsysteme

- Verletzung von Zeitvorgaben bedeutet Katastrophe
- Beispiel: Steuerung von Fabriken

Schwache Echtzeitsysteme

- Verletzung von Zeitvorgaben hat störenden Charakter, führt aber nicht zu Katastrophen
- Beispiel: Übertragung von Audio- und Videodaten



Bereitzeit

- Frühestmöglicher Ausführungsbeginn einer Aktivität

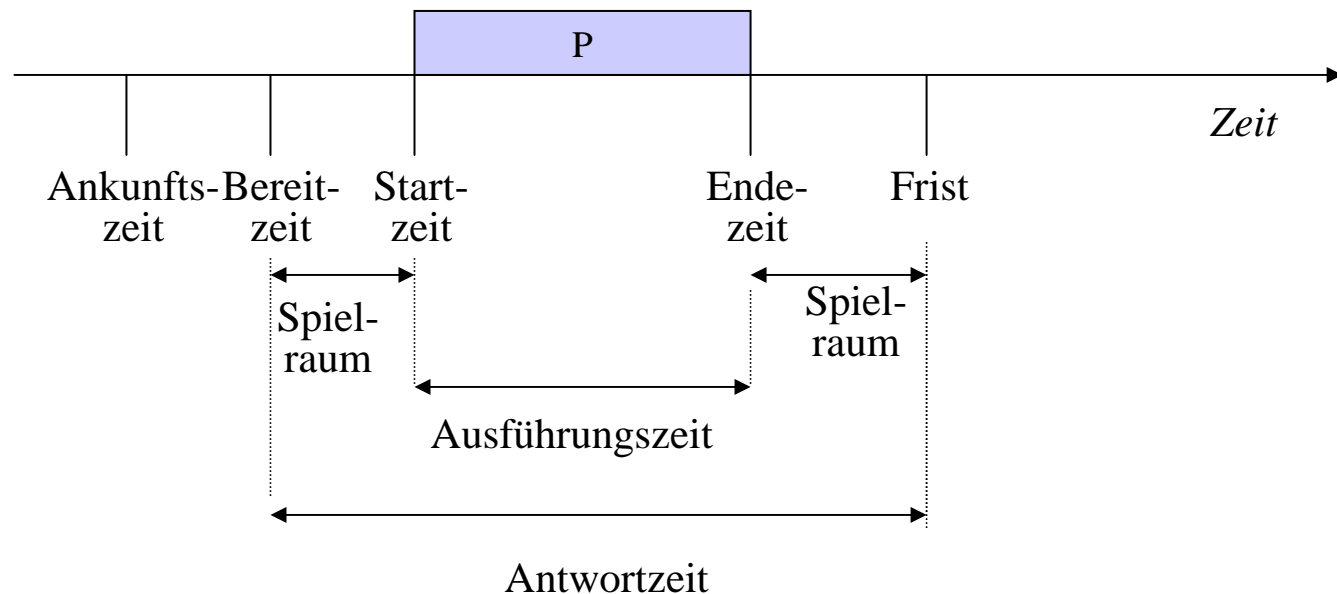
Frist

- Spätester Zeitpunkt für die Beendigung einer Aktivität

Ausführungszeit

- Worst-Case-Abschätzung für das zur vollständigen Ausführung der Aktivität notwendige Zeitintervall

Schema



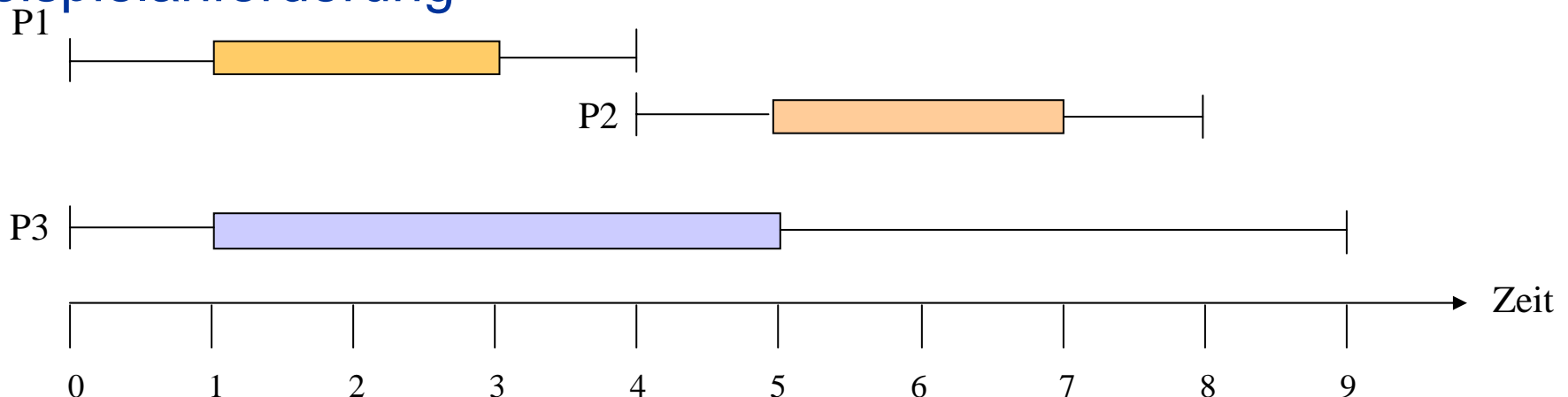
Prinzip

- Prozessor wird immer dem Prozess mit der nächsten Frist zugeordnet
- Ist kein Prozess bereit, verbleibt der Prozessor im Leerlauf

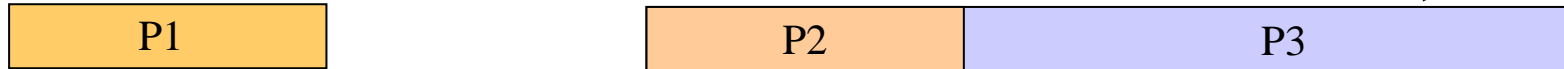
Varianten

- präemptiv und nicht präemptiv
- nicht präemptive Variante findet nicht immer eine Abarbeitungsreihenfolge
- präemptive Variante findet immer eine Abarbeitungsreihenfolge, falls eine solche existiert

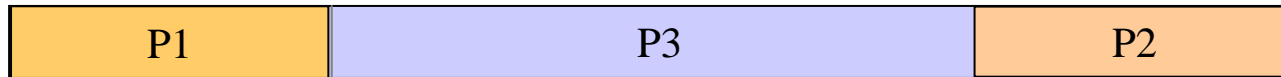
Beispielanforderung



Nicht präemptive Schedulingreihenfolgen



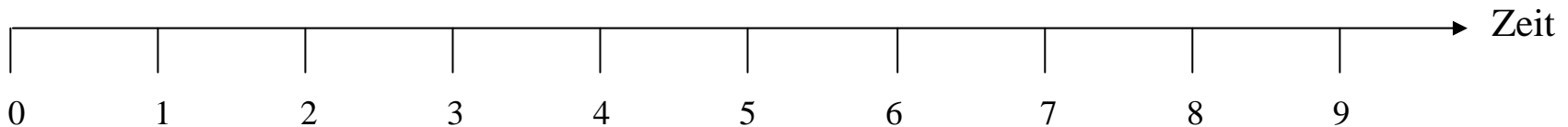
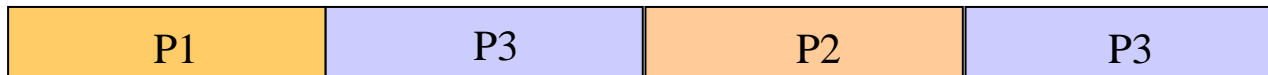
(a) EDF-basierte Reihenfolge mit Fristverletzung



(b) Mögliche Reihenfolge ohne Fristverletzung



Präemptive Schedulingreihenfolge



Beachte: Auch Prozessorzeit ist eine Ressource

- Prozessorzeit ist eine Ressource, die sich alle Threads im System teilen (dazu später mehr).
- Auch hier kann es zum „Verhungern“ einzelner Threads kommen

Betrachte z.B. Round-Robin-Scheduling mit Prioritäten!



Prozessinteraktion

Wechselwirkungen zwischen Prozessen

Voraussetzung

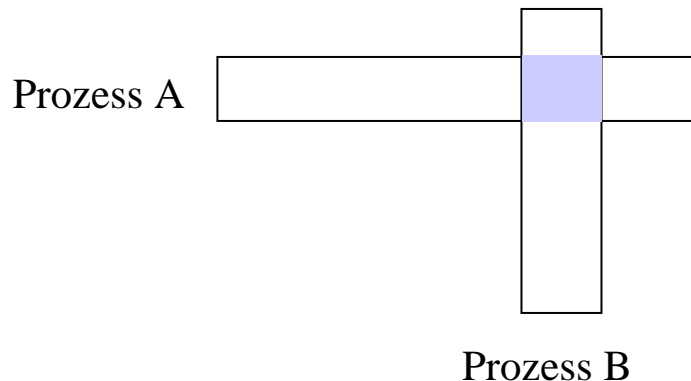
Kopplung unter den Prozessen

Disjunkte Prozesse

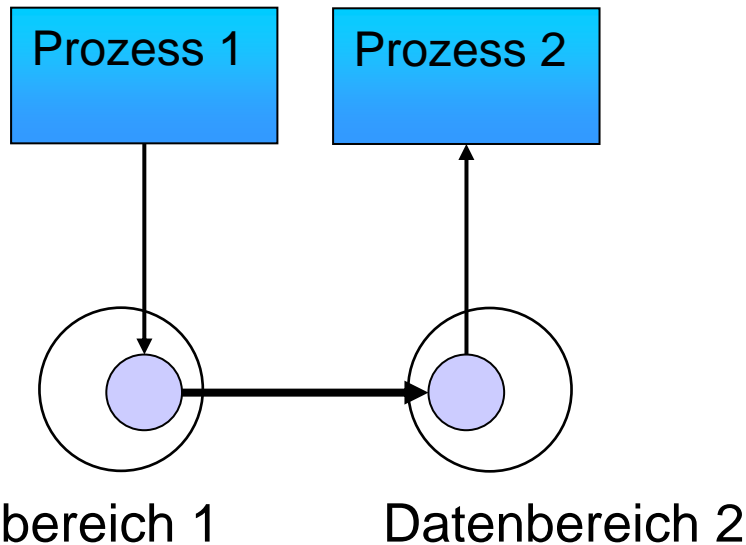
Prozessinteraktion ist ausgeschlossen

Gekoppelte Prozesse

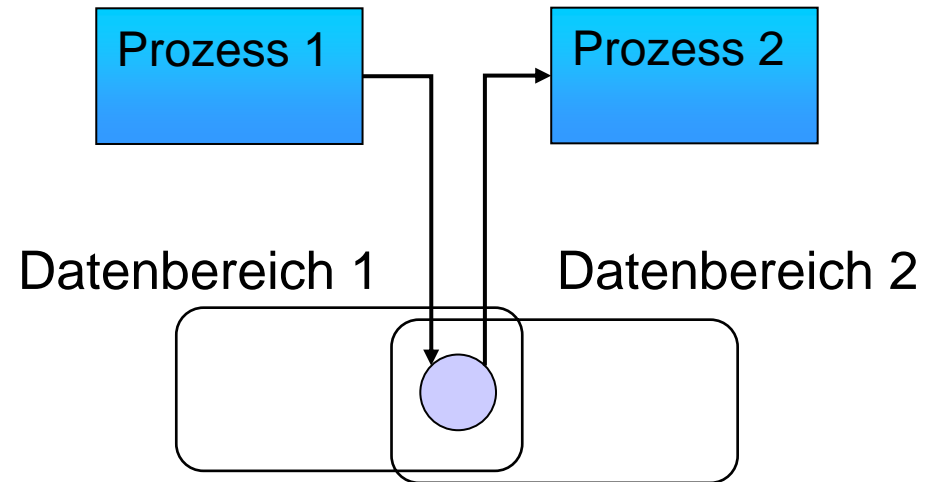
hängen über gemeinsamen Speicher oder Kanäle zusammen



Grundformen Datenaustausch



Kommunikation: Transport der Daten von einem Datenbereich in den anderen



Kooperation: Ablegen von Daten in einem gemeinsamen Datenbereich

Problemstellung

- Zwei Prozesse besitzen gemeinsamen Puffer mit begrenzter Länge (bounded buffer).
- Ein Prozess (Erzeuger, producer) schreibt Information in den Puffer.
- Ein Prozess (Verbraucher, consumer) liest Information aus dem Puffer.
- Erzeuger darf nicht in den vollen Puffer einfügen.
- Verbraucher darf nicht aus leerem Puffer lesen.

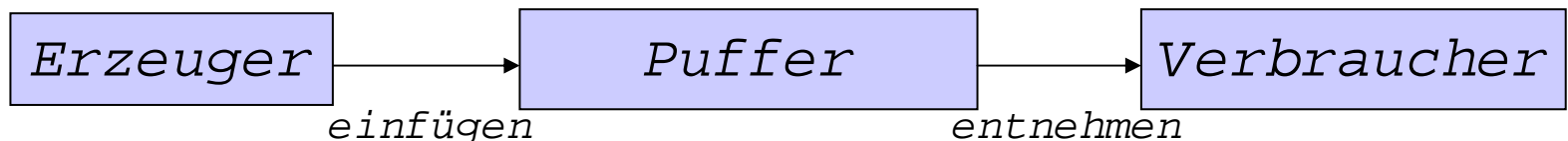
Fehlerhafte Lösung

```

While (true) {
    produce(item);
    while(count==N) sleep(1);
    buffer[in]:= item;
    in := (in+1) % N;
    count := count + 1;
}
    
```

```

While (true) {
    while(count==0) sleep(1);
    item = buffer[out];
    out := (out+1) % N;
    count := count - 1;
    consume(item);
}
    
```



Die Anweisungen `count := count+1` und `count := count-1` werden typischerweise zu den folgenden Maschinenbefehlen

P1: `register1 := count`

P2: `register1 := register1+1`

P3: `count := register1`

C1: `register2 := count`

C2: `register2 := register2-1`

C3: `count := register2`

Annahme

- Count habe den Wert 5
- Ausführung der Befehle in folgender Reihenfolge
 - P1, P2, C1, C2, P3, C3
- ... und in folgender Reihenfolge
 - P1, P2, C1, C2, C3, P3

Problem

- Ergebnisse hängen von der Bearbeitungsreihenfolge der Prozesse ab.



Definition

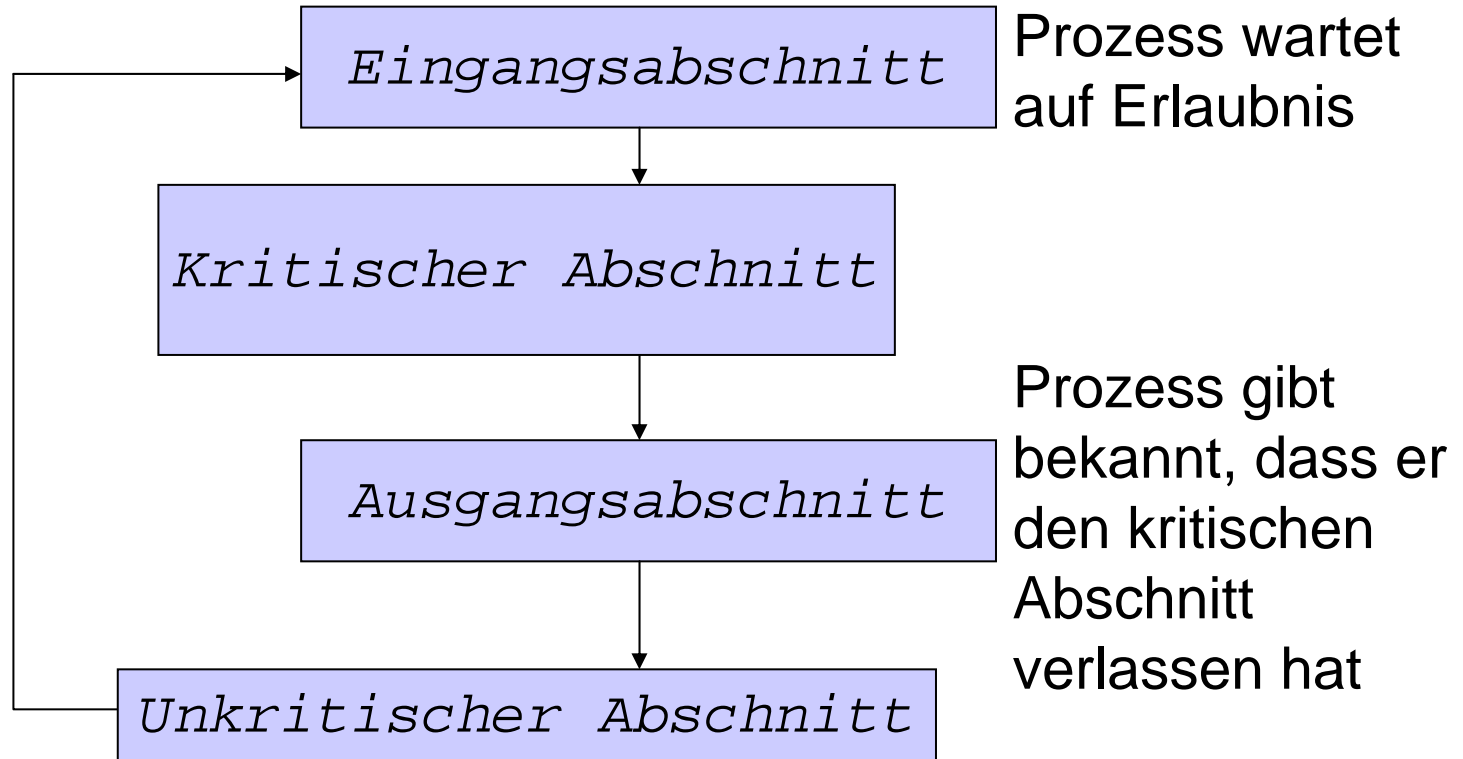
- Ein kritischer Abschnitt (critical section) eines Prozesses ist eine Menge von Instruktionen, in der das Ergebnis der Ausführung auf unvorhergesehene Weise variieren kann, wenn Variablen, die auch für andere parallel ablaufende Prozesse verfügbar sind, während der Ausführung verändert werden.

Anforderungen an eine Lösung

- Zwei Prozesse dürfen nicht gleichzeitig in ihrem kritischen Abschnitt sein
- Es dürfen keine Annahmen über die Bearbeitungsgeschwindigkeit von Prozessen gemacht werden
- Kein Prozess, der außerhalb eines kritischen Bereichs ist, darf andere Prozesse beim Eintritt in den kritischen Abschnitt behindern
- Kein Prozess darf ewig auf den Eintritt in den kritischen Abschnitt warten



Prinzipieller „Lebenszyklus“ eines Prozesses oder Threads



Prozesse interagieren miteinander über Speicherzellen

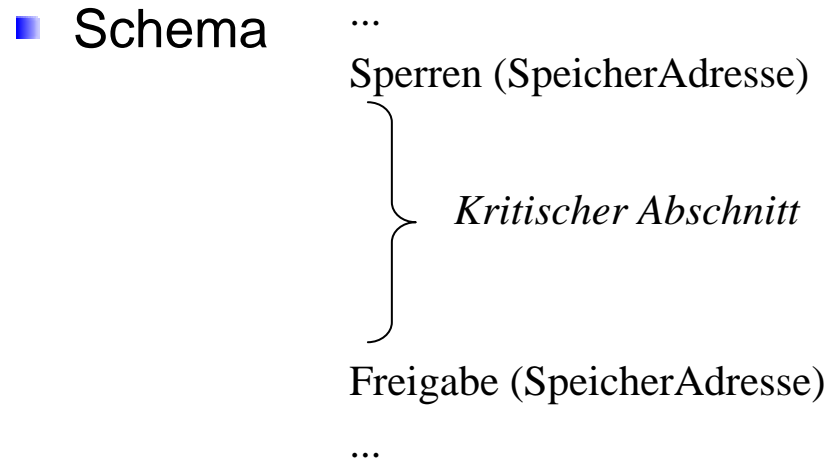
Dabei Problem

- Konkurrierende Prozesse greifen auf die gleiche Speicherzelle zu

Exklusives Sperren

- Sperren (SpeicherAdresse)
- Freigeben (SpeicherAdresse)

→ Benutzung in einem *kritischen Abschnitt*

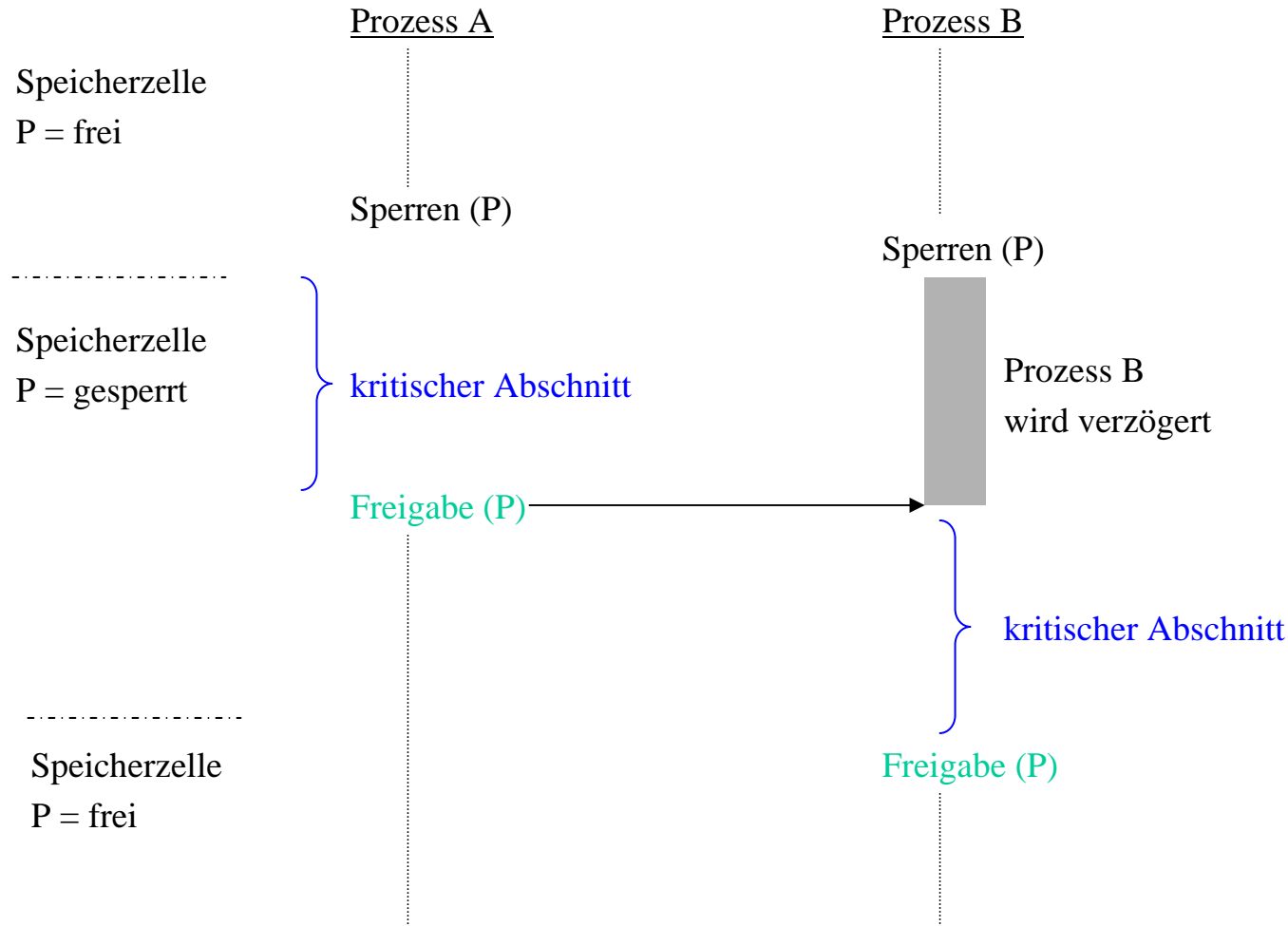


- Zwei Prozesse dürfen sich nicht gleichzeitig in ihrem kritischen Abschnitt sein

→ *Sperrflag* ist die zur Synchronisation benutzte Speicherzelle



Synchronisation mittels kritischer Abschnitte



Sitzplätze eines Flugzeugs

- (Sitz)Platz [Anzahl] mit
 - Platz.Status : (frei, belegt);
Platz.Kunde : String;
- int Freiplätze = Anzahl;
- k Prozesse greifen gleichzeitig auf die Variablen zu
 - PROCESS {
 int I;
 LOOP {
[1] Warte auf Signal von Terminal;
[2] if (Freiplätze>0) {
[3] I = SuchePlatz();
[4] Platz[I].Status = belegt;
[5] Platz[I].Kunde = ReadName();
[6] Freiplätze --;
[7] Print(I);
[8] } else Print („Flugzeug belegt“);
 }}
}



Zwei zeitlich überlappende Buchungsprozesse

- Prozess A möchte Kunden K. Müller einbuchen
- Prozess B möchte Kunden M. Zink einbuchen

Zeitliche Verzahnungen der Anweisungsfolge

- a) A1 ... A8, B1 ... B8
- b) B1 ... B8, A1 ... A8
- c) A1, B1, A2, B2, A3, B3, A4, B4, A5, B5, A6, B6, A7, B7, A8, B8
→ resultiert in einer Doppelbelegung
 - Problem: nahezu gleichzeitiger Zugriff auf Zustandsinformation des Platzes 7

Konsistenzprobleme sind zu erwarten, wenn Zugriff auf gemeinsame Daten unkoordiniert verläuft



Sitzplatzbelegungen bei verschiedenen Buchungsvorgängen

	1		6	7	8	9	n
a)	belegt (F.Hill)	alle Sitze belegt	belegt (K.Horn)	frei	frei	frei	frei

$$\text{Freiplätze} = n - 6$$

							n
b)	belegt (F.Hill)	alle Sitze belegt	belegt (K.Horn)	belegt (K.Müller)	belegt (M.Zink)	frei	frei

$$\text{Freiplätze} = n - 8$$

						n	
c)	belegt (F.Hill)	alle Sitze belegt	belegt (K.Horn)	belegt (M.Zink)	belegt (K.Müller)	frei	frei

$$\text{Freiplätze} = n - 8$$

Freiplatze = n - 8

d)	belegt (F.Hill)	alle Sitze belegt	belegt (K.Horn)	belegt (M.Zink)	frei	frei	frei
----	--------------------	-------------------	--------------------	--------------------	------	------	------

$$\text{Freiplätze} = n - 8$$



Problem

- „Protokoll“ erforderlich, an das sich alle Prozesse oder Threads halten und das die Semantik des kritischen Abschnitts erhält.

Anforderungen

- Zwei Prozesse dürfen nicht gleichzeitig in ihrem kritischen Abschnitt sein (safety)
- Es dürfen keine Annahmen über die Bearbeitungsgeschwindigkeit von Prozessen gemacht werden
- Kein Prozess der außerhalb eines kritischen Bereichs ist, darf andere Prozesse beim Eintritt in den kritischen Abschnitt behindern
- Kein Prozess darf ewig auf den Eintritt in den kritischen Abschnitt warten müssen (fairness)
- Möglichst passives statt aktives Warten, da aktives Warten einerseits Rechenzeit verschwendet und andererseits Blockierungen auftreten können, wenn Prozesse/Threads mit niedriger Priorität gewartet werden muss.



Es wurden eine Reihe von Lösungen für die Synchronisation von Prozessen entwickelt, von denen die wichtigsten sind

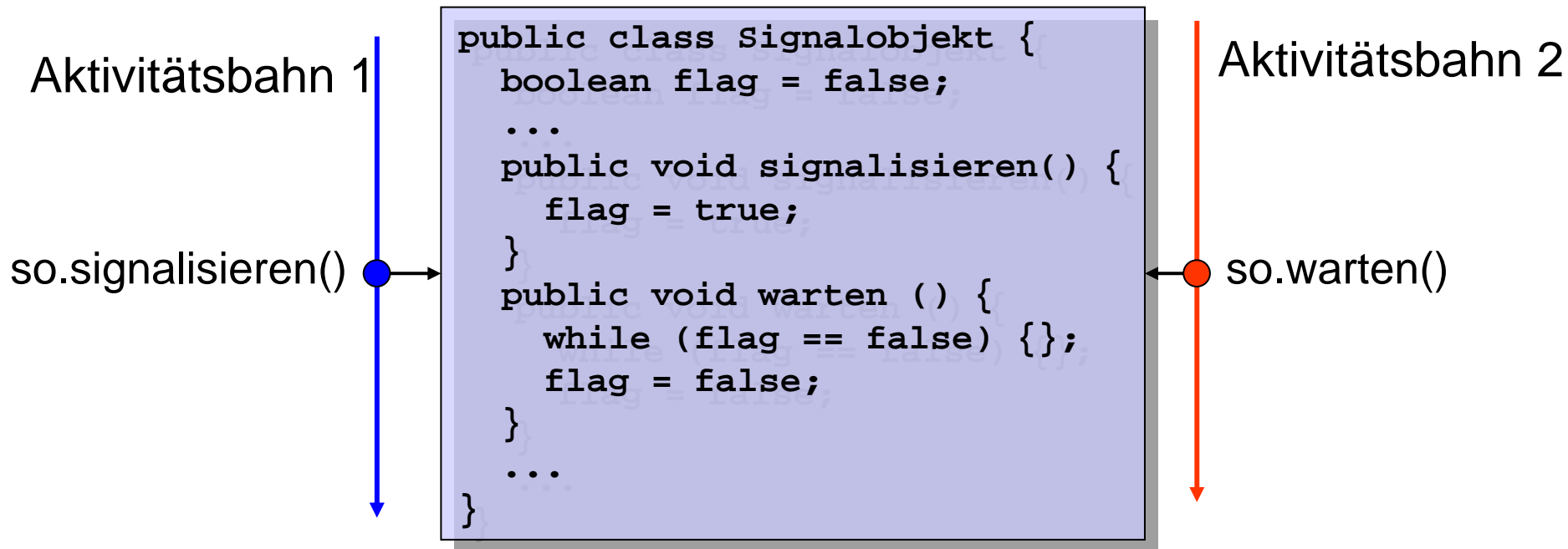
- Semaphore
- Mutexe
- Monitore

Sie stellen Lösungen für die Synchronisation bei Nutzung gemeinsamer Variable dar.

- Bei Nachrichtenkommunikation wird diese Form der Synchronisation nicht benötigt.



- Koordination über Signalobjekt
- Die Koordination kann über ein **Signalobjekt (*semaphore*)** erfolgen, auf das beide Zugriff haben.



Definition

- Ein Semaphor ist eine geschützte Variable, auf die nur die unteilbaren (atomaren) Operationen `up` (signal, V) und `down` (wait, P) ausgeführt werden können
 - `down(s)`

```
{  
    s := s - 1;  
    if (s < 0) queue_this_process_and_block();  
}
```
 - `up(s)`

```
{  
    s := s + 1;  
    if (s <= 0) wakeup_process_from_queue();  
}
```

Bemerkungen

- Die angegebene Definition beschreibt ein zählendes Semaphor („counting semaphore“)
- Semaphore, die nur die Werte 0 und 1 annehmen können, heißen binäre Semaphore („binary semaphores“)
- Zählende Semaphore lassen sich mit binären Semaphoren implementieren



Implementierung

- Durch spezielle Systemaufrufe, die die geforderten Operationen up und down realisieren.

Programmiersprachen

- Semaphore können in beliebigen Programmiersprachen benutzt werden, da sie letztlich einem Systemaufruf entsprechen.

Warten

- Passives Warten bis zum Eintritt in den kritischen Bereich



Fähigkeit zu zählen

- Oft bei Semaphoren nicht benötigt.
- Binäre Aussage ob der kritische Abschnitt frei ist oder nicht genügt häufig.

Einfachere Variante

- Mutex (mutual exclusion)



- Durch das Konkurrieren von Threads um Ressourcen kann es zu „invertierter Priorität“ kommen.
- Ein nieder-priorer Thread hält die Ressource, die ein hoch-priorer Thread benötigt – dieser blockiert
- Ein Thread mit mittlerer Priorität verdrängt den mit niedriger (und den mit hoher) und kann laufen.



- Tasks on the Pathfinder spacecraft were executed as threads with priorities that were assigned in the usual manner reflecting the relative urgency of these tasks.
- Pathfinder contained an "information bus", which you can think of as a shared memory area used for passing information between different components of the spacecraft. A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus.
- Access to the bus was synchronized with mutual exclusion locks (mutexes). The meteorological data gathering task ran as an infrequent, low priority thread, and used the information bus to publish its data. When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex.
- If an interrupt caused the information bus thread to be scheduled while this mutex was held, and if the information bus thread then attempted to acquire this same mutex in order to retrieve published data, this would cause it to block on the mutex, waiting until the meteorological thread released the mutex before it could continue.
- The spacecraft also contained a communications task that ran with medium priority.
- Very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread. In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running, consequently preventing the blocked information bus task from running.

Resultat: kompletter Software-Reset des „Pathfinder“-Systems



Das Erzeuger-Verbraucher Problem lässt sich elegant mit drei Semaphoren lösen

1. Ein Semaphor zum Betreten der kritischen Abschnitte (mutex)
2. Ein Semaphor, das die freien Plätze im Puffer herunter zählt und den Prozess blockiert, der in einen vollen Puffer schreiben will (empty)
3. Ein Semaphor, das die belegten Plätze im Puffer herauf zählt und den Prozess blockiert, der von einem leeren Puffer lesen will (full)

Beispiel

semaphore mutex = 1, empty = N, full = 0;

```
while (true) {  
    produce(item);  
    down(empty);  
    down(mutex);  
    add(item);  
    up(mutex);  
    up(full);  
}
```

Erzeuger

```
while (true) {  
    down(full);  
    down(mutex);  
    remove(item);  
    up(mutex);  
    up(empty);  
    consume(item);  
}
```

Verbraucher



- Semaphore lassen sich als Systemaufrufe implementieren, wobei kurzzeitig sämtliche Unterbrechungen unterbunden werden. (Da zur Implementation nur wenige Maschinenbefehle benötigt werden, ist diese Möglichkeit akzeptabel.)
- Eine Voraussetzung zur Implementation von Semaphoren auf parallelen Maschinen ist die Existenz eines gemeinsamen Hauptspeichers
- Auf Mehrprozessor-Systemen muss ein Semaphor mit einer unteilbaren Operation implementiert werden, die das Semaphor vor gleichzeitigen Änderungen in anderen Prozessoren schützt

Beispiel: Test-And-Set

```
boolean test_and_set(boolean target) {  
    boolean v = target;  
    target = true;  
    return v;  
}  
  
while (true) {  
    while (test_and_set(lock)) do no-op;  
    critical_section();  
    lock = false;  
}
```

Beachte Call-by-Value
Aufrufe!



Kleine Programmierfehler bei der Benutzung eines Semaphors können zu Verklemmungen oder inkorrekten Ergebnissen führen. Typische Fehler

Dazu später mehr

- Sprünge aus kritischen Bereichen, ohne mutex Semaphor freizugeben
- Sprünge in kritische Bereiche, ohne mutex Semaphor zu setzen
- Vertauschungen von Semaphoren zum Schutz von kritischen Abschnitten und Semaphoren, die vorhandene Betriebsmittel zählen

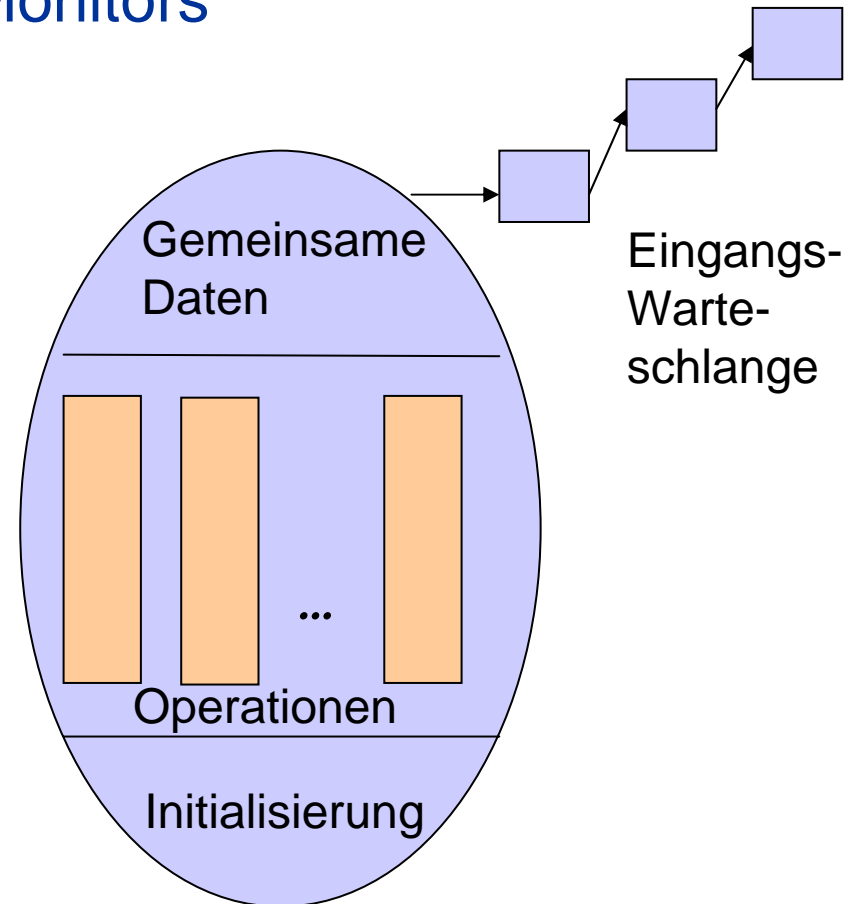
Fazit

- Semaphore sind eine „low-level“ Lösung, die erhebliche Disziplin vom Programmierer verlangt.
- Eine komfortablere Lösung bieten Monitore.



- Monitor ist eine Sammlung von Prozeduren, Variablen und Datenstrukturen, die in einem Modul gekapselt sind
- Prozesse können Prozeduren des Monitors aufrufen, aber keine internen Daten ändern
- Monitore besitzen Eigenschaft, dass immer nur genau ein Prozess im Monitor aktiv sein kann
- Monitore sind Konstrukte einer Programmiersprache und erfordern daher spezielle Compiler
- Es ist die Aufgabe des Compilers, Maschinenbefehle zu generieren, die den wechselseitigen Ausschluss im Monitor garantieren

Schematischer Aufbau eines Monitors



Bedingungsvariablen (condition variables) eines Monitors mit zugehörigen Operationen `wait()` und `signal()` erlauben es, im Monitor auf andere Prozesse zu warten

- `wait(c)` Der aufrufende Prozess blockiert bis ein `signal()` auf der Bedingungsvariablen `c` ausgeführt wird. Ein anderer Prozess darf den Monitor betreten
- `signal(c)` Ein auf die Bedingungsvariable `c` wartender Prozess wird aufgeweckt. Der aktuelle Prozess muss den Monitor sofort verlassen
- Der durch `signal()` aufgeweckte Prozess wird gemäß Hoare (1974) zum aktiven Prozess im Monitor, während der Prozess, der `signal()` ausgeführt hat, blockiert.
- Bedingungsvariablen sind keine Zähler. Ein `signal(c)` auf einer Variablen `c` ohne ein `wait(c)` geht einfach verloren.



```
monitor ErzeugerVerbraucher
    condition full, empty; integer count;

    procedure enter
        if count = N then wait(full);
        enter_item();
        count := count + 1;
        if count = 1 then signal(empty);
    end;

    procedure remove
        if count = 0 then wait(empty);
        remove_item();
        count := count - 1;
        if count = N-1 then signal(full);
    end;

    count = 0;
end monitor;
```

```
procedure Erzeuger
    while true do begin
        produce_item;
        ErzeugerVerbraucher.enter;
    end
end;

procedure Verbraucher
    while true do begin
        ErzeugerVerbraucher.remove;
        consume_item;
    end
end;
```



Eigenschaften

- Erlauben Synchronisation von Prozessen oder Threads, die auf verschiedenen Rechnern ohne gemeinsamen Hauptspeicher laufen.
- Lassen sich auf Benutzerebene implementieren und können somit ohne spezielle Übersetzer überall benutzt werden
- Sind normalerweise langsamer als Synchronisationsverfahren, die auf gemeinsamem Speicher beruhen.



Kommunikationsmuster

- Meldung
 - Einweg-Nachricht vom Sender zum Empfänger
- Auftrag
 - Zweiweg-Nachricht
 - Versenden des Auftrags, Übergabe einer Erfolgsbestätigung (Quittung)

Synchronität

- definiert Kopplungsgrad zwischen Prozessen
- asynchron
 - Sender und Empfänger sind zeitlich voneinander entkoppelt
- synchron
 - Sender und Empfänger sind zeitlich gekoppelt
 - Sender blockiert, bis zur Beendigung einer Nachrichtentransaktion



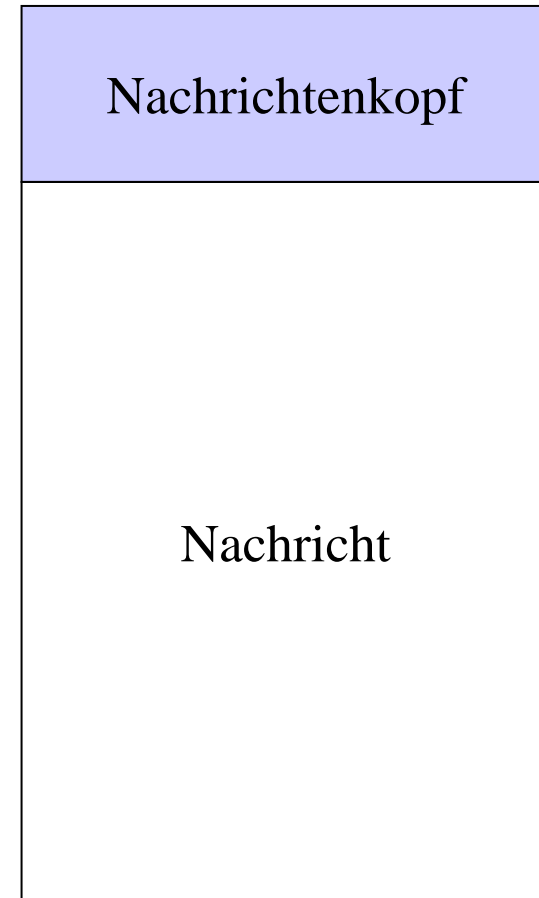
Klassifikationsschema

		Synchronität	
		asynchron	synchron
Kommunikations- muster	Meldung	asynchrone Meldungen	synchrone Meldungen
	Auftrag	asynchrone Aufträge	synchrone Aufträge



Nachrichtenkopf

- Nachrichtentyp
- Länge der Nachricht
- Absender
- Empfänger
- Sequenznummer
- Priorität
- Hinweis auf Folgenachrichten

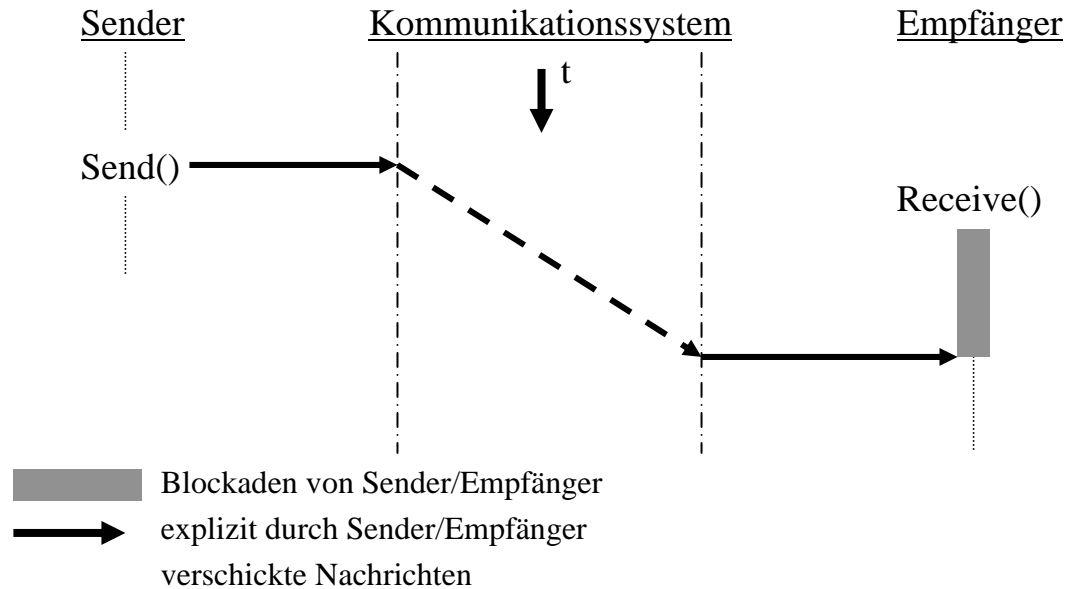


Kontrollnachrichten

- dienen der Realisierung von Protokollen
- sind ausserhalb des Kommunikationssystems unsichtbar

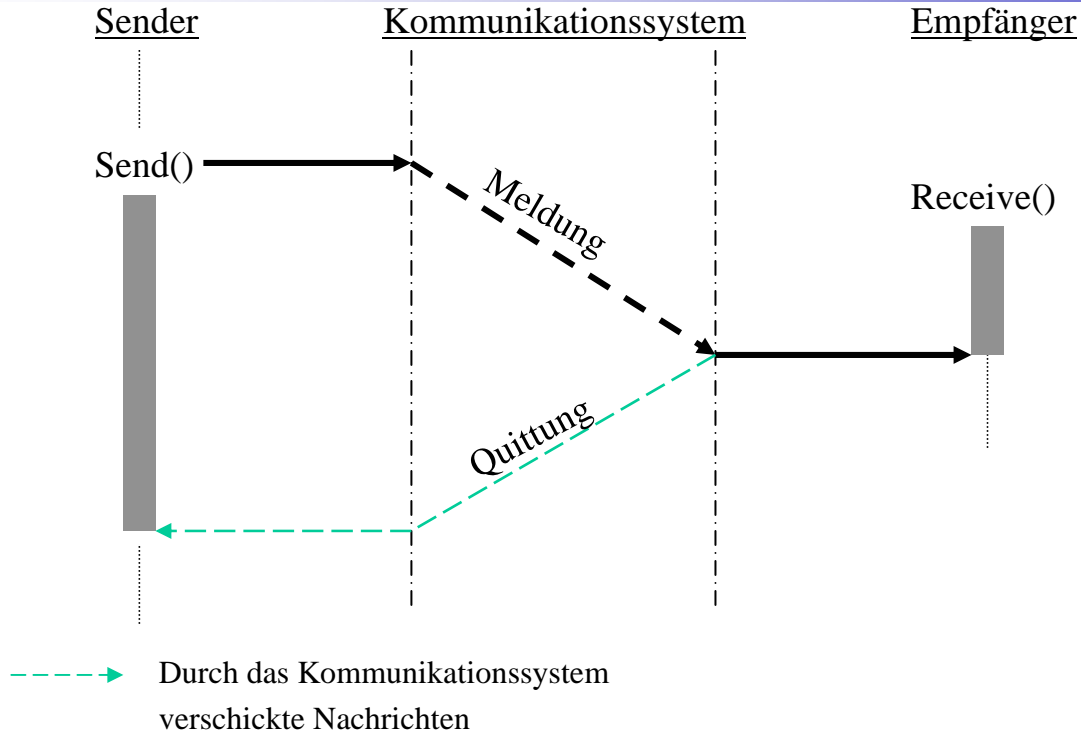


Schema



- Sender wird nicht blockiert
- Empfänger blockiert bis zum Empfang der Nachricht
- Parallelarbeit zwischen Sender und Empfänger möglich
- Kommunikationssystem muss Nachrichten puffern können

Schema



Sender ist bis zur Ablieferung der Nachricht blockiert

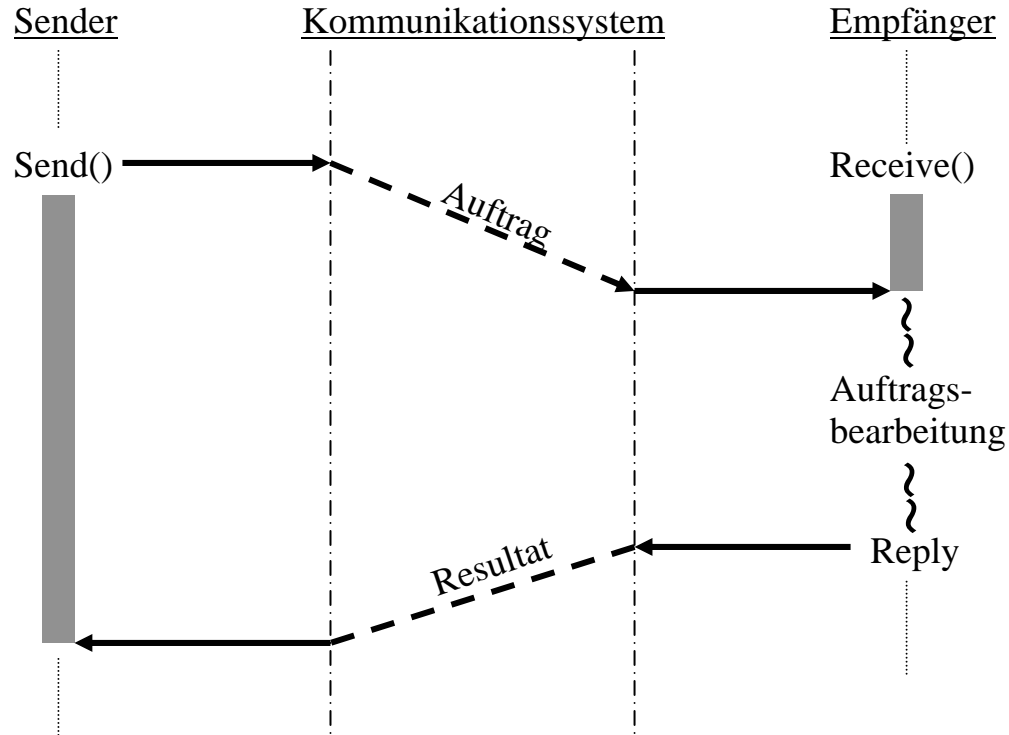
- kann Nachrichten nicht schneller erzeugen als sie konsumiert werden

Variante

- Sender und Empfänger stellen Sende- bzw. Empfangsbereitschaft vor dem Austausch der Nachrichten her
 - Rendezvous-Technik



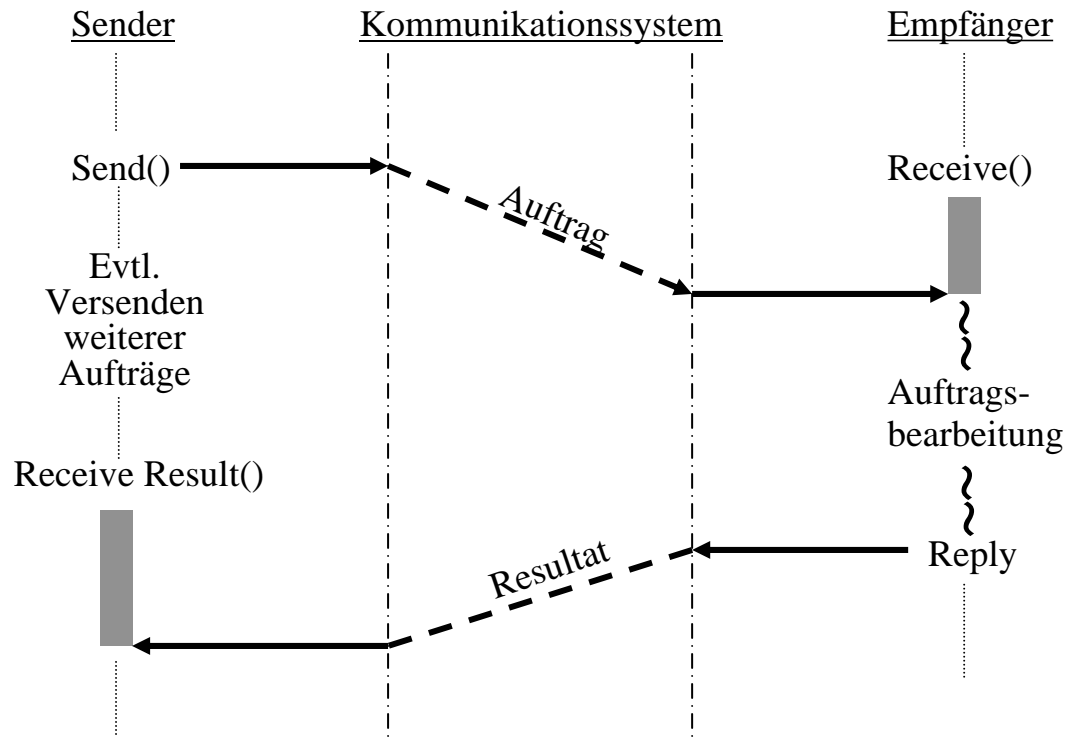
Schema



- Auftragsbearbeitung beim Empfänger ist Teil der Nachrichtentransaktion
- Funktion `reply()` wird benötigt
- Schränken potentielle Parallelarbeit stark ein
- Remote Procedure Call (RPC) unterstützt diese Variante



Schema



Auftrag und Resultat sind unabhängige Meldungen

- werden durch Auftragskennung eindeutig zugeordnet

Vorgehensweise

- Gemeinsamer Speicher wird durch Nachrichten und Mailbox ersetzt
- Verbraucher-Prozess erzeugt zunächst N leere Nachrichten, die er dem Erzeuger-Prozess sendet
- Erzeuger-Prozess wartet auf eintreffende leere Nachrichten und schickt diese mit Daten gefüllt an den Verbraucher-Prozess zurück.

```
While (true) {  
    produce(item);  
    receive(consumer, m);  
    build_message(m, item)  
    send(consumer, m)  
}
```

```
For (i=0, i<N, i++)  
    send(producer, m)  
While (true) {  
    receive(producer, m);  
    extract_message(m, item);  
    send(producer, m);  
    consume(item);  
}
```



Problem

- In manchen Situationen ist sofortige Reaktion des Empfängers bei Eintreffen einer Nachricht erforderlich
- Elementare Kommunikationsoperationen basieren auf Blockade des Empfängers, falls beim Absetzen der Receive-Operation keine Nachricht vorliegt

Ansatz

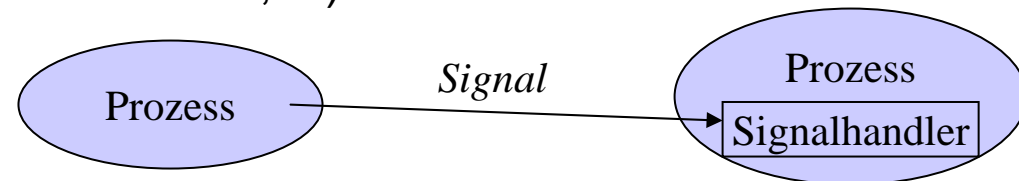
- Signale basieren auf nicht blockierenden Receive-Operationen

Signale sind wichtige Meldungen

- kurz (nur wenige Bytes)
- lösen bei der Ankunft beim Empfänger die sofortige Verarbeitung aus

Anwendungen

- Intra-Prozesssignalisierung
 - Ausnahmen (Overflow, Division durch 0, ...)
 - I/O-Signale
- Inter-Prozesssignalisierung



Operationen

- Receive(Signalisierungsursache, HandlerAdresse)
 - übergibt dem Kommunikationssystem die Adresse einer Routine, die beim Eintreffen des Signals zu aktivieren ist
- Signal(Empfänger, Signalursache, Parameter)
 - schickt ein Signal an einen Empfänger

Umsetzung

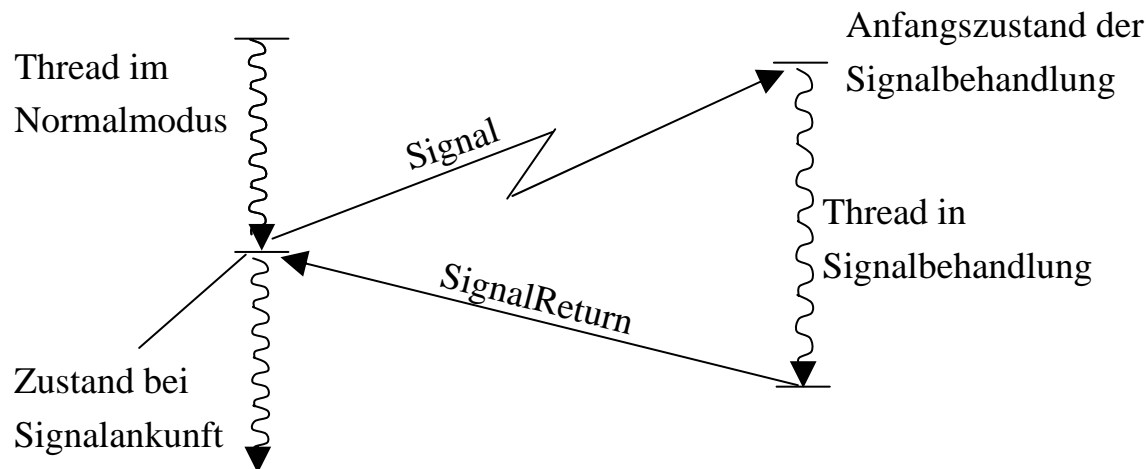
- Neuer Thread für jedes eintreffende Signal
- Verknüpfung des Threads mit der Handler-Routine
- Übergabe der Parameter als Stack
- Beendigung der Signalbehandlung mit Selbstterminierung des Threads
- Erfordert gleichzeitige Unterstützung mehrerer Kernbasierter Threads in einem Adressraum



Alternative Umsetzung als Software-Interrupt

- Bei Eintreffen eines Signals wird Prozessleitblock gerettet
- Thread mit Signal-Handler-Routine
- Thread befindet sich in besonderem Ausführungsmodus
 - Signalmodus
 - Beenden nur über SignalReturn()
- Es wird immer nur eine Signalbehandlung ausgeführt, die anderen werden gepuffert

Ablauf



Definition

- Eine Menge von Prozessen befindet sich in einer Verklemmung (deadlock), wenn jeder Prozess der Menge auf ein Ereignis wartet, dass nur ein anderer Prozess aus der Menge auslösen kann.

Beispiel

- Zwei Prozesse möchten auf einem Bandgerät gespeicherte Daten auf einem Drucker ausgeben. Der erste Prozess fordert zunächst den Drucker an und bekommt ihn vom Betriebssystem zugewiesen. Der zweite Prozess fordert zuerst das Bandgerät an und erhält ebenfalls exklusiven Zugriff. Nun können beide Prozesse das fehlende Betriebsmittel nicht mehr belegen.

Modell

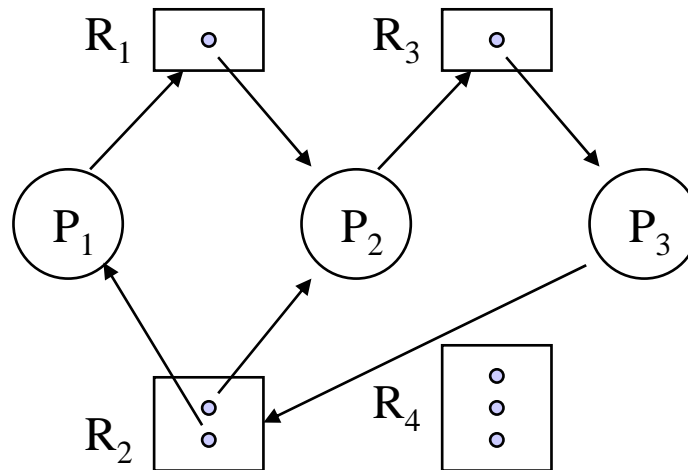
- Im folgenden werden allgemein Prozesse und von ihnen benötigte bzw. belegte Betriebsmittel betrachtet. Ein Betriebsmittel kann aber auch eine einfache Datei oder eine gemeinsame Variable im Hauptspeicher sein.



Definition

- Graph besteht aus einer Menge von Prozessen $P=\{P_1, P_2, \dots, P_n\}$, einer Menge von Betriebsmitteln $R=\{R_1, R_2, \dots, R_m\}$ und gerichteten Kanten
- Gerichtete Kante von Prozess P_i zu Betriebsmittel R_j beschreibt, dass Prozess P_i ein Exemplar des Betriebsmittels R_j angefordert hat
- Gerichtete Kante von Betriebsmittel R_j zu Prozess P_i beschreibt, dass ein Exemplar des Betriebsmittels R_j P_i zugewiesen ist

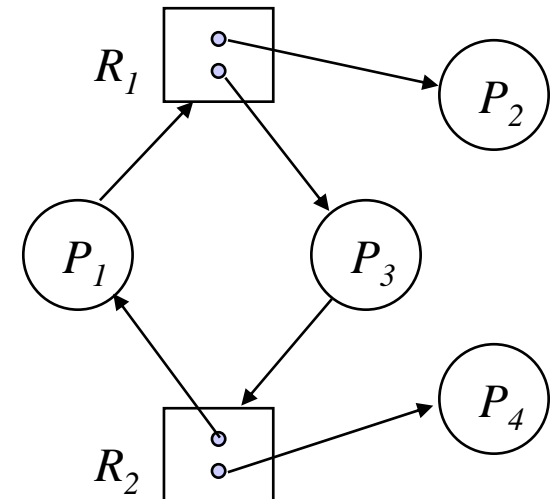
Schema



- Enthält ein Betriebsmittel-Zuweisungsgraph keine Zyklen, dann existiert auch kein Deadlock.
- Besitzt ein Betriebsmittel-Zuweisungsgraph einen Zyklus und existiert von jedem beteiligten Betriebsmittel nur genau ein Exemplar, dann existiert ein Deadlock.
- Besitzt ein Betriebsmittel-Zuweisungsgraph einen Zyklus und von den beteiligten Betriebsmitteln existieren mehrere Exemplare, so ist ein Deadlock möglich, aber nicht unbedingt auch eingetreten.

Existieren im vorigen Beispiel Deadlocks?

...und wie sieht's in diesem Beispiel aus?



Wechselseitiger Ausschluss („mutual exclusion“)

- Betriebsmittel ist entweder genau einem Prozess zugeordnet oder es ist verfügbar

Wartebedingung („hold and wait“)

- Es gibt einen Prozess, der ein Betriebsmittel belegt und auf ein anderes Betriebsmittel wartet, das von einem anderen Prozess belegt wird.

Keine Verdrängung („no preemption“)

- Einem Prozess kann ein Betriebsmittel nicht entzogen werden.

Zirkuläres Warten („circular wait“)

- Es gibt eine Menge $\{P_1, P_2, \dots, P_n\}$ von Prozessen, so dass P_1 auf ein Betriebsmittel wartet, das P_2 belegt, P_2 wartet auf ein Betriebsmittel das P_3 belegt, ..., und P_n wartet auf ein Betriebsmittel das P_1 belegt.



Sobald *eine* der notwendigen Bedingungen für eine Deadlock nicht existiert, kann ein Deadlock vermieden werden

Verhinderung von wechselseitigem Ausschluss

- ist wichtig für Betriebsmittel, die nicht gleichzeitig benutzt werden können. Bei gemeinsam nutzbaren Betriebsmitteln (z.B. read-only Datei) unproblematisch.
- Beispiel Drucker
 - Keine exklusive Zuordnung von Drucker zu Prozessen
 - Ablegen von Druckausgaben in Dateien und einfügen in Warteschlange
 - Spezieller Prozess (printer daemon) erhält exklusiven Zugriff auf den Drucker und arbeitet die Warteschlange ab. Dieser Prozess fordert selbst keine weiteren Betriebsmittel an.
- Problem
 - einige Betriebsmittel können nicht gleichzeitig von mehreren Prozessen benutzt werden.



Verhinderung der Wartebedingung

- Prozess darf nur dann Betriebsmittel anfordern, wenn er selbst keine anderen Betriebsmittel belegt
- Anforderung sämtlicher Betriebsmittel vor Abarbeitung
 - Beispiel: Prozess, der Daten vom Bandlaufwerk in eine Datei kopiert und dann druckt
 - Bandlaufwerk, Datei und Drucker müssen zu Beginn allokiert werden
 - Drucker wird erst am Ende benötigt
 - benötigte Betriebsmittel müssen vor der Abarbeitung bekannt sein



Probleme dabei

- geringe Ressourcenauslastung
- „Aushungern“ („starvation“) möglich

Prozess muss alle Betriebsmittel abgeben, bevor er neue anfordern kann

- Beispiel wie oben
- zunächst werden Bandlaufwerk und Datei angefordert, anschließend Datei und Drucker
- Problem
 - es können nicht immer alle Betriebsmittel freigegeben werden
 - „Aushungern“ möglich



Entzug von zugewiesenen Betriebsmitteln

- Bereits belegte Betriebsmittel können einem Prozess wieder entzogen werden.

Situation

- Prozess werden Betriebsmittel entzogen, um Anforderungen anderer Prozesse befriedigen zu können
- Entzogene Betriebsmittel werden in die Liste der Betriebsmittel hinzugefügt, auf die der Prozess wartet

Probleme

- Zustand der Betriebsmittel muss beim Entzug gesichert und später restauriert werden
- Nicht jedes Betriebsmittel (z.B. Drucker) erlaubt Sicherung und Restaurierung des Zustandes
- häufige Anwendung bei Betriebsmitteln, deren Zustand leicht gesichert werden kann, wie z.B. CPU-Registerinhalte oder Hauptspeicherbereiche



Verhindern von zirkulärem Warten

- Definieren einer totalen Ordnung auf allen Betriebsmitteln.
Betriebsmittel müssen in steigender Reihenfolge angefordert werden.
 - Werden mehrere Instanzen eines Betriebsmittels benötigt, so müssen sie in einem Aufruf angefordert werden.
- Es wird eine Funktion $F: R \rightarrow \mathbb{N}$ definiert, d.h. eine Abbildung von Betriebsmitteln (Ressourcen) auf natürliche Zahlen
 - Beispiel
 - $F(\text{Bandlaufwerk}) = 1$,
 - $F(\text{Plattenlaufwerk}) = 5$,
 - $F(\text{Drucker}) = 12$
- Problem
 - Finden einer totalen Ordnung, die allen Anforderungen gerecht wird ist schwierig.
- Korrektheit
 - Wie kann diese nachgewiesen werden?



Ansatz

- Es wird zusätzliche Information über Betriebsmittelanforderungen gefordert, um die Betriebsmittel besser zu nutzen und den Systemdurchsatz zu erhöhen.
- Ständiges Überwachen des Belegungszustandes, um zirkuläres Warten zu vermeiden.

Beispiel

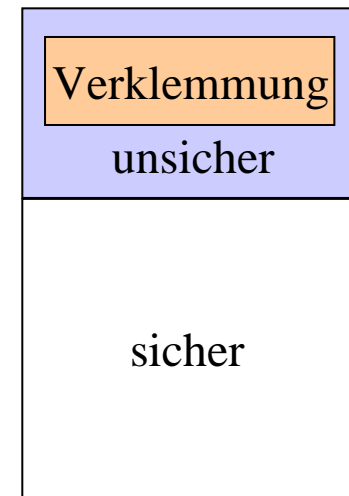
- Prozess P fordert zunächst das Bandlaufwerk und dann den Drucker. Prozess Q fordert erst Drucker, dann Bandlaufwerk
- Annahme
 - Die maximale Anforderung von Betriebsmitteln ist im Voraus bekannt.

Sichere Zustände

- Ein System ist in einem sicheren Zustand, wenn es zu jedem Prozess die maximal geforderten Betriebsmittel zuordnen kann, ohne dass das System in eine Verklemmung gerät.

Unsichere Zustände

- Es existiert keine Sequenz wie im sicheren Zustand.
- Unsichere Zustände können zu Verklemmungen führen.
- Verklemmungen sind unsichere Zustände.



Ziel

- Gewährleistung, dass System in einem sicheren Zustand bleibt

Ausgangssituation

- Jeder neue Prozess gibt zu Beginn maximal erforderlichen Ressourcen an
- n sei die Anzahl der vorhandenen Prozesse
- m sei die Anzahl unterschiedlicher Ressourcen

Datenstrukturen

- Available
 - Vektor der Länge m .
 - $\text{Available}[j] = k$ bedeutet, dass von der Ressource R_j k Exemplare vorhanden sind.
- Max
 - $n \times m$ Matrix, die die maximalen Anforderungen der Prozesse beschreibt.
 - $\text{Max}[i, j] = k$ bedeutet, dass Prozess P_i höchstens k Elemente der Ressource R_j anfordert.



■ Allocation

- $n \times m$ Matrix, die die momentan durch die einzelnen Prozesse belegten Ressourcen beschreibt.
- $\text{Allocation}[i, j] = k$ bedeutet, dass Prozess P_i derzeit k Elemente der Ressource R_j belegt.

■ Need

- $n \times m$ Matrix, die noch ausstehende max. Ressourcenanforderungen der Prozesse angibt.
- $\text{Need}[i, j] = k$ signalisiert, dass P_i maximal noch k Elemente von Ressource R_j benötigt.



Algorithmus

- Hilfsvariablen: Vektor Work (Länge m) und Vektor Finish (Länge n)
- Ablauf
 1. Setze $\text{Work}[i] := \text{Available}[i]$ und $\text{Finish}[i] := \text{false}$ für alle i .
 2. Bestimme ein i mit
 $\text{Finish}[i] = \text{false}$ und
 $\text{Need}[i] \leq \text{Work}[i]$.
Falls kein solches i existiert, gehe zu Schritt 4.
 3. $\text{Work}[i] := \text{Work}[i] + \text{Allocation}_i$ und $\text{Finish}[i] := \text{true}$.
Gehe zu Schritt 2.
 4. Wenn $\text{Finish}[i] = \text{true}$ für alle i gilt, so befindet sich das System in einem sicheren Zustand.

Anmerkung

- Allocation und Need werden als Vektoren behandelt
- Allocation_i und Need_i beziehen sich auf Prozess P_i



Beispiel: Sichere/unsichere Zustände

Prozesse A, B und C und eine Ressource mit 10 Elementen

All. Max

A	3	9
B	2	4
C	2	7

Available: 3

All. Max

A		
B		
C		

Available:

All. Max

A		
B		
C		

Available:

All. Max

A		
B		
C		

Available:

All. Max

A		
B		
C		

Available:

Der Zustand im linken Teilbild ist sicher/unsicher?

All. Max

A	3	
B	2	
C	2	

Available: 3

All. Max

A		
B		
C		

Available: 2

All. Max

A		
B		
C		

Available:

All. Max

A		
B		
C		

Available:

Der Zustand im zweiten Teilbild von links ist sicher/unsicher?



Algorithmus

- Request_i sei Request-Vektor für P_i
- Ablauf
 1. Wenn Request_i ≤ Need_i, gehe zu Schritt 2. Ansonsten Fehler.
 2. Wenn Request_i ≤ Available[i], gehe zu Schritt 3. Ansonsten muss P_i warten, da die angeforderten Betriebsmittel nicht verfügbar sind.
 3. Setze Available[i] := Available[i] - Request_i und
Allocation_i = Allocation_i + Request_i und
Need_i = Need_i - Request_i
Wenn der sich daraus ergebende Zustand sicher ist, dann wird die Betriebsmittelanforderung gewährt. Anderenfalls muss P_i warten und die Zuweisungen werden zurückgenommen.



Szenario

- 5 Prozesse $P_0 \dots P_4$ und 3 Ressourcen A, B, C.
- Ressource A hat 10 Elemente, B hat 5 Elemente und C 7 Elemente.

<i>Allocation</i>				<i>Max</i>				<i>Available ?</i>	<i>Need ?</i>
	A	B	C		A	B	C		
P_0	0	1	0	P_0	7	5	3		
P_1	2	0	0	P_1	3	2	2		
P_2	3	0	2	P_2	9	0	2		
P_3	2	1	1	P_3	2	2	2		
P_4	0	0	2	P_4	4	3	3		

- Ist der Zustand sicher? Wie sieht es nach folgender Anforderung aus?
 - $\text{Request}_1 = (1,0,2)$?



Problem

- Wird kein Algorithmus zur Vermeidung von Deadlocks benutzt, so können Deadlocks entstehen.
- Erkennen und Behebung von Deadlocks ist erforderlich

Erkennungsverfahren

- Das Betriebssystem überprüft periodisch, ob ein Deadlock vorliegt

Behebungsverfahren

- Im Fall eines Deadlocks ergreift das Betriebssystem Maßnahmen zur Auflösung

Notwendige Rahmenbedingungen

- Information über belegte und angeforderte Betriebsmittel muss einfach zugänglich sein.
- Aufwand zur Entdeckung von Verklemmungen muss vertretbar sein.
 - während der Ausführung des Erkennungsverfahrens dürfen keine Betriebsmittel angefordert oder freigegeben werden
- Kosten zur Behebung von Verklemmungen müssen vertretbar sein

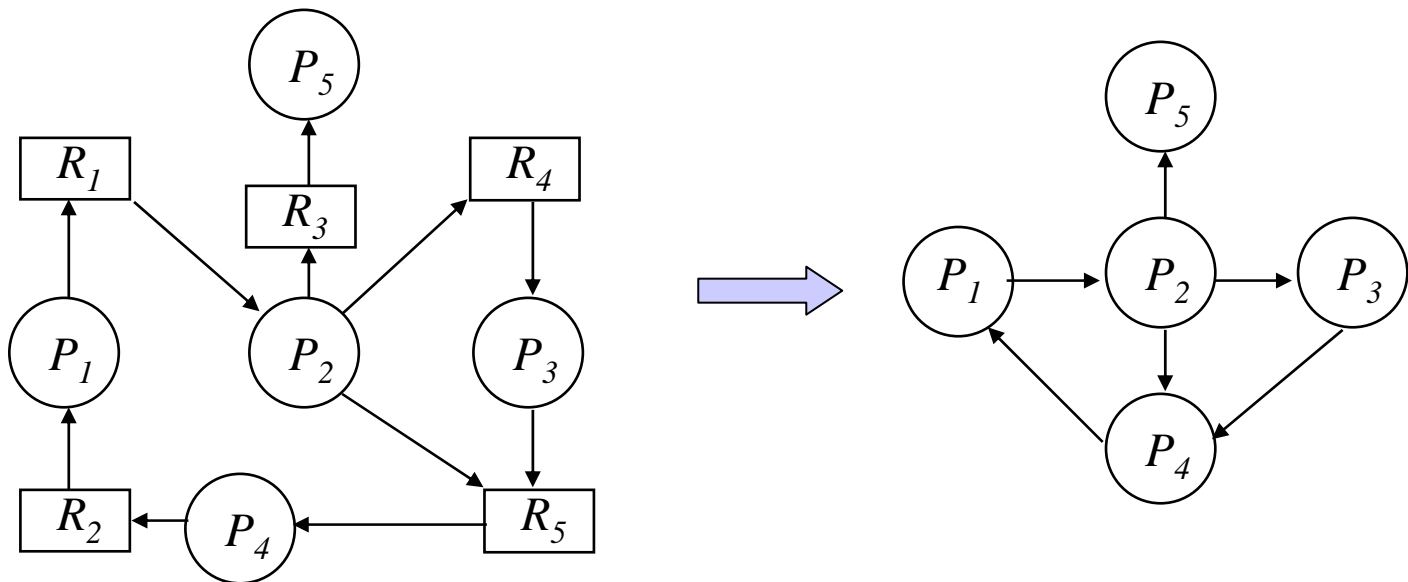


Vorgehensweise

- Aus dem Betriebsmittel-Zuweisungsgraphen wird ein *Wartegraph* („wait-for graph“) konstruiert, wobei alle Knoten, die Betriebsmittel repräsentieren, entfernt werden.
- Eine Kante von Prozess P_i zu Prozess P_j existiert im Wartegraph genau dann, wenn der Betriebsmittel-Zuweisungsgraph zwei Kanten $P_i \ltimes R_q$ und $R_q \ltimes P_j$ besitzt.

Deadlock genau dann, wenn ein Zyklus im Wartegraph existiert.

Beispiel



Vorgehensweise

- Deadlocks lassen sich erkennen, indem man versucht, eine Abarbeitungsfolge der Prozesse P_1, \dots, P_n zu finden, so dass alle bekannten Betriebsmittelanforderungen erfüllt werden können.
- Sei Available der Vektor der noch verfügbaren Betriebsmittel und Allocation eine Matrix, die die zur Zeit belegten Betriebsmittel den Prozessen zuordnet.
- Sei Request die Matrix, die die noch ausstehenden Betriebsmittelanforderungen beschreibt.
 1. Setze $Work := Available[i]$ und $Finish[i] := false$, falls P_i noch offene Anforderungen hat.
 2. Bestimme ein i mit $Finish[i] = false$ und $Request[i] \leq Work$. Falls kein solches i existiert, gehe zu Schritt 4.
 3. $Work := Work + Allocation[i]$ und $Finish[i] := true$. Gehe zu Schritt 2.
 4. Existiert ein i mit $Finish[i] = false$, so existiert ein Deadlock. Alle Prozesse i mit $Finish[i] = false$ sind an dem Deadlock beteiligt.

Welche Annahmen werden in Schritt 3 gemacht?



Erscheinungsformen der Parallelität:

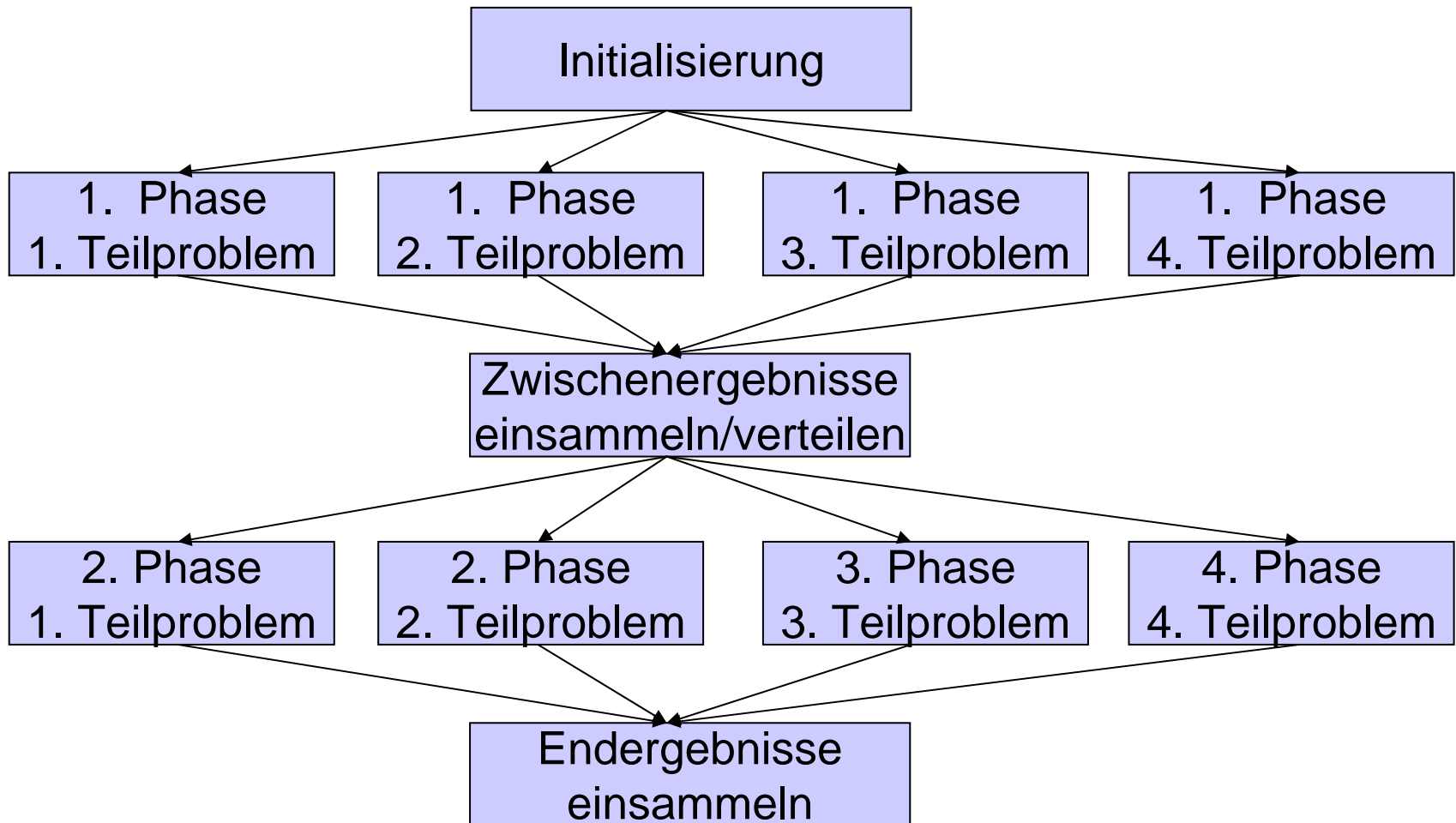
- Echter Parallelrechner mit mehreren Prozessoren.
- Rechnernetz mit mehreren Rechnern.
- Rechner mit einem Prozessor, der so tut, als hätte er mehrere (Mehrbenutzer-Betriebssysteme).
- Hintergrundjobs (z.B. Druckerspooler).

Grundgedanke paralleler Algorithmen:

- Ein Problem lässt sich so in Teilprobleme zerlegen, dass deren Berechnungsreihenfolge beliebig ist.
- Dann kann jedes Teilproblem von einer eigenen Recheneinheit gelöst werden.
- Ziel der Parallelisierung ist eine Beschleunigung der Ausführung.



Probleme können stückweise parallelisierbar sein



Kommunikation zwischen den Prozessoren erforderlich

- Verteilen der Teilprobleme
- Einsammeln der Teilergebnisse

Prozesse müssen synchronisiert werden

Schlechtes Kommunikationsdesign

- Hoher Kommunikationsaufwand (zusätzlich zum Rechenaufwand)

Schlechte Synchronisation

- Prozesse stehen still und warten.
- Gleichzeitige Kommunikationsversuche erzeugen „race conditions“



Beispiel zu Race Conditions

Überweisen (vonKonto, nachKonto, Betrag)

```
{
    vonKontoNeu = vonKonto - Betrag;
    nachKontoNeu = nachKonto + Betrag;

    vonKonto = vonKontoNeu;
    nachKonto = nachKontoNeu;
}
```

Zwei parallele Prozesse

- Prozess 1: Überweisen (k1, k2, 50)
- Prozess 2: Überweisen (k1, k2, 30)

Prozess 1:

K1Neu=50
K2Neu=150
K1=50

Prozess 2:

K1Neu=20
K2Neu=130
K1=20
K2=130

Prozess 1:

K2=150

K1: 20 €
K2: 150 €

K1: 100 €
K2: 100 €

K1: 50 €
K2: 100 €

K1: 20 €
K2: 130 €



Beschleunigung (*speedup*):

- Sei $T(n,1)$ der Aufwand eines Algorithmus mit einem Prozessor und $T(n,p)$ der Aufwand mit p Prozessoren.
- Der „Speedup“ ist definiert als

$$S(p) = \frac{T(n,1)}{T(n,p)}$$

- Er ist **optimal** bzw. **perfekt**, wenn $S(p)=p$.

Effizienz

- Gibt die durchschnittliche Auslastung (Ausnutzung) der Prozessoren an:

$$E(n,p) = \frac{S(p)}{p} = \frac{T(n,1)}{pT(n,p)}$$

Beachte: Es gibt Probleme, die leicht zu parallelisieren sind und Probleme, die gar nicht parallelisierbar sind.



Modell eines Parallelrechners

- p Prozessoren P_1, \dots, P_p
- können alle auf einen gemeinsamen Speicher zugreifen
- jeder Prozessor verfügt außerdem über einen privaten Arbeitsspeicher
- Die p Prozessoren sind synchronisiert, d.h. sie führen Rechenschritte gleichzeitig aus
 - Rechenschritt besteht aus drei Phasen: lesen (gemeinsamer Speicher), rechnen (privater Arbeitsspeicher), schreiben (gemeinsamer Speicher)
 - Kommunikation erfolgt über gemeinsamen Speicher
 - Datenaustausch in zwei Rechenschritten möglich

Weitere Unterscheidung: Speicherzugriff

- EREW, CREW, CRCW, ERCW

Exclusive

Exclusive

Read

Write

Concurrent

Concurrent



Exclusive Read/Write

- Nur ein Prozessor kann in einem Zeittakt eine Speicherzelle lesen/schreiben

Concurrent Read/Write

- Beliebig viele Prozessoren können in einem Zeittakt dieselbe Speicherzelle lesen/schreiben
- Wenn mehrere Prozessoren gleichzeitig schreiben dürfen, ist eine Regelung erforderlich
 - Alle gleiches Datum → ok, sonst Abbruch
 - Datum des Prozessors mit kleinster/größter Nummer zählt
 - Datum eines zufälligen Prozessors zählt



PRAMs

- Zugriff unbeschränkter Anzahl von Prozessoren auf die gleiche Speicherzelle unrealistisch

Kommunikation mehrerer Prozessoren über Verbindungsnetzwerk

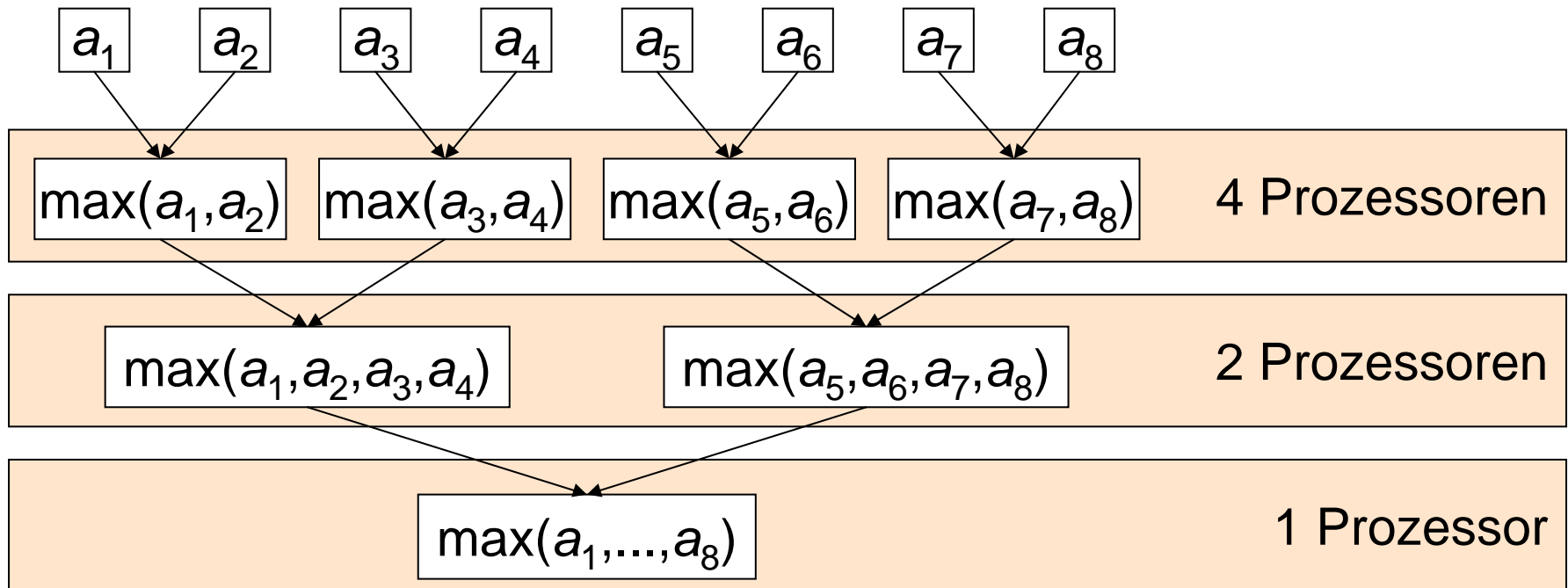
- Identische Prozessoren sind an den Knoten eines Graphen platziert
- Prozessoren kommunizieren untereinander entlang der Kanten
- Struktur des Verbindungsnetzwerkes bestimmt, für welche Aufgaben der Parallelrechner gut geeignet ist



Gegeben: Reihung mit den Elementen a_1, \dots, a_n

Gesucht: Maximales Element a_i

Naheliegender Algorithmus (z.B. $n=8$):



Aufwand

- $O(\log n)$ mit $n/2$ Prozessoren



Die Effizienz des Algorithmus ist ziemlich schlecht:

- Nach dem ersten Schritt liegen die Hälfte, nach dem zweiten drei Viertel aller Prozessoren brach.

Effizienz:

- $n-1$ Vergleiche für sequentiellen Algorithmus:

$$T(n, 1) = n-1$$

- $n/2$ Prozessoren, Aufwand $O(\log_2 n)$:

$$T(n, n/2) = \log_2 n$$

$$E(n, n/2) = \frac{T(n, 1)}{(n/2)T(n, n/2)} = \frac{n-1}{(n/2)\log_2 n} \approx 2/(\log_2 n)$$

Frage: geht es auch mit weniger als $n/2$ Prozessoren?



Statischer Paralleler Algorithmus

- Zuteilung der Rechenschritte zu Prozessoren liegt im Voraus fest.

Problem

- Wann ist es möglich, mit Hilfe einer besseren Lastverteilung die Effizienz zu steigern?

Satz von Brent

- Wenn es einen EREW-Algorithmus gibt mit Aufwand $T(n,p)=O(t)$, der insgesamt s Rechenschritte macht, dann gibt es auch einen mit Aufwand $T(n,s/t)=O(t)$



Rechenschritte des vorgestellten Algorithmus

- $n/2 + n/4 + \dots = n-1$ Schritte

Berechneter Aufwand

- $T(n, n/2) = O(\log n)$

Laut Satz von Brent sollte ein Algorithmus existieren mit

- $T(n, \frac{n-1}{\log n}) = O(\log n)$
- Also ein Algorithmus der bei gleichem Aufwand nur noch $O(n/\log n)$ Prozessoren benötigt



... jetzt mit $n/\log n$ Prozessoren

Vorgehensweise

- Aufteilen von a_1, \dots, a_n in $n/\log n$ Gruppen mit jeweils $\log n$ Elementen
- Erste Phase
 - Jeder Prozessor berechnet (sequentiell) das Maximum in einer Gruppe (in $O(\log n)$ Schritten)
- Zweite Phase
 - Maximum der $n/\log n$ Gruppenmaxima wird mit $n/\log n$ Prozessoren in $\log(n/\log n) \approx \log n$ Schritten berechnet



PRAM-Architektur ist geeignet für wenige Prozessoren

- Z.B. Workstations mit mehreren CPUs

Bei sehr vielen Prozessoren ($10^3 - 10^5$) wird Anschluss an gemeinsamen Speicher hardwaretechnisch sehr aufwändig.

→ Bei sehr vielen Prozessoren besser

- Jeder Prozessor hat seinen eigenen Speicher
- Prozessoren kommunizieren über vorgesehene Verbindungen



Verbindungsnetzwerke

- Können als Graphen betrachtet werden
- Prozessoren sind Knoten
- Kommunikationsleitungen sind Kanten

Verschiedene Topologien möglich



Verschiedene Topologien für verschiedene Algorithmen

Zugriff auf Daten

- Nur im prozessoreigenen Speicher direkt
- Alles andere über Kommunikation

Vollständige Vermaschung

- Meist technisch zu aufwändig
- Übertragung von Daten von Prozessor A zu Prozessor B läuft häufig über mehrere Prozessoren

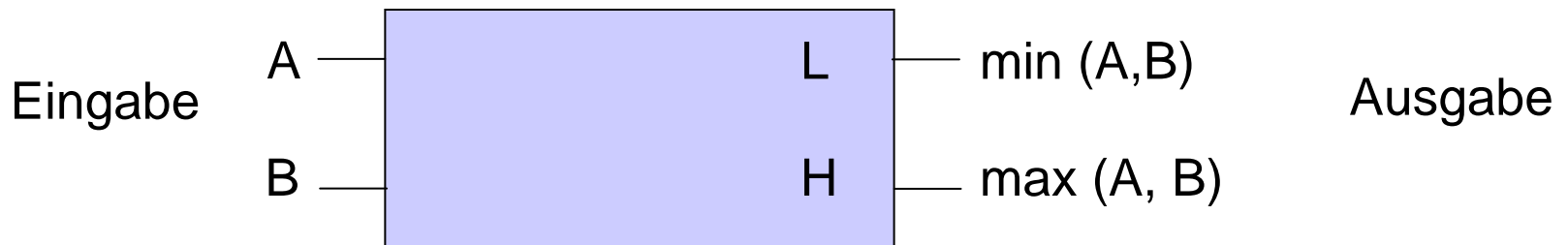


Untersuchung der seriellen Sortiervverfahren auf ihre Parallelisierbarkeit

- Typischer Schritt in serielltem Sortiervverfahren
 - Vergleich zweier Schlüssel
 - Jedem Prozessor Paar von Schlüsseln zuordnen

Modell

- Parallelrechner habe eine große Zahl an „Compare-Exchange“-Moduln
 - Liest, vergleicht Werte und gibt sie geordnet aus



- Module sollen nicht über gemeinsamen Speicher kommunizieren sondern über ein festes Verbindungsnetz

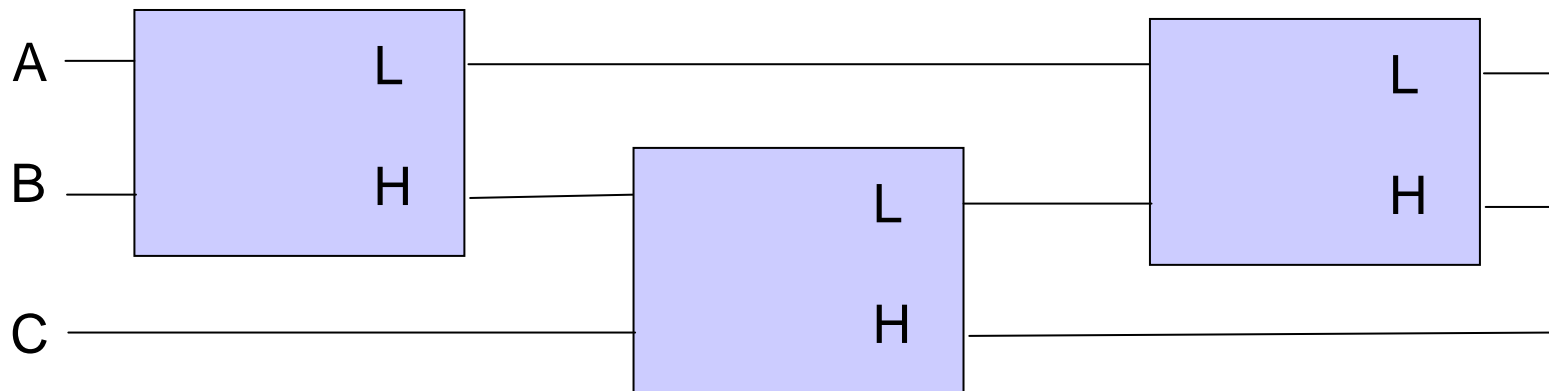
Problem

- Verbindungsnetz kann seine Struktur nicht dynamisch an Problemstellung bzw. Eingabedaten anpassen

Serielles Programm

- If $A > B$ then vertausche (A,B);
 if $B > C$ then begin
 vertausche (B, C);
 if $A > B$ then vertausche (A, B);
 end

Netz aus drei „Compare-Exchange“-Moduln



Bei seriellen Sortiervverfahren

- Strategien zum Mischen spielen eine wichtige Rolle

→ Für paralleles Sortieren sind geeignete Mischverfahren erforderlich.

- Zwei sortierte Schlüsselfolgen müssen mit der immer gleichen Operationsfolge zu einer sortierten Folge verschmolzen werden.

Beispiele

- Odd-even-Mergesort
- Bitonic Mergesort



Gegeben: Zwei jeweils aufsteigend sortierte zu mischende Folgen a_1, \dots, a_n und b_1, \dots, b_n

Gesucht: Aufsteigend sortierte Folge x_1, \dots, x_{2n}
(Mehrfachvorkommen möglich)

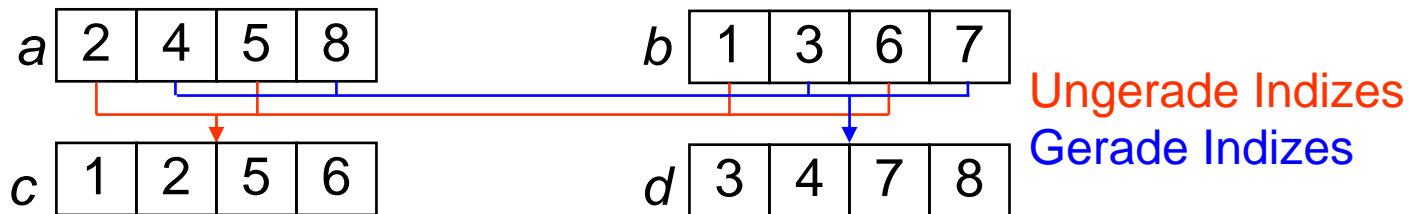
Rekursive Ansatz

- Vereinfachende Annahme: $n = 2^k$ für ein $k \geq 0$

Grundgedanke:

- Mischen der Elemente an den ungeraden Positionen ergibt c_1, \dots, c_n
- Mischen der Elemente an den geraden Positionen ergibt d_1, \dots, d_n

Beispiel



Es gilt :

- Für jedes i , $1 \leq i < n$
das Element c_{i+1} muss unmittelbar vor oder unmittelbar nach dem Element d_i der Größe nach eingeordnet werden.

Aus den Folgen c_1, \dots, c_n und d_1, \dots, d_n kann eine aufsteigend sortierte Folge e_1, \dots, e_{2n} hergestellt werden.

- Beobachtung
 - $c_i \geq c_1, \dots, c_i$ und $c_i \geq d_1, \dots, d_i$ also $c_i \geq e_{2i}$
 - $d_{i+1} \geq d_1, \dots, d_{i+1}$ und $d_{i+1} \geq c_1, \dots, c_{i+1}$ also $d_{i+1} \geq e_{2i+1}$
- Vorgehensweise: Durch Vergleich ist zu bestimmen, ob c_i oder d_{i+1} an die Position e_{2i} gelangt. Das andere Element nimmt Position e_{2i+1} ein.
 - $e_1 = c_1$
 - $e_{2i} = \min(c_{i+1}, d_i)$, für $1 \leq i < n$
 - $e_{2i+1} = \max(c_{i+1}, d_i)$, für $1 \leq i < n$
 - $e_{2n} = d_n$



Sortierte
Listen

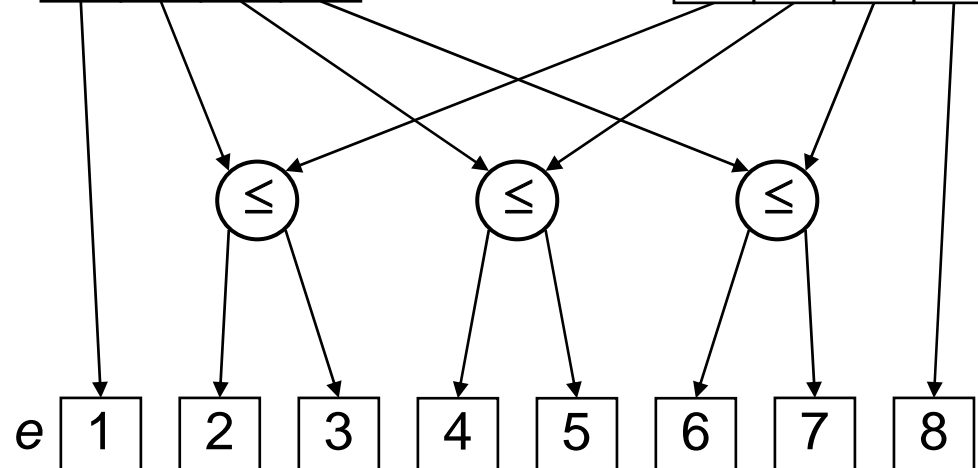


Paralleles
Mischen

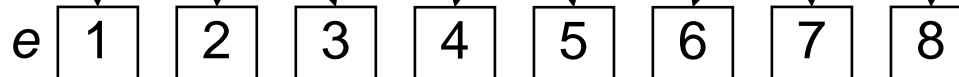


Ungerade Indizes
Gerade Indizes

Vergleichen und
Tauschen



Sortierte
Gesamtliste



Gegeben seien die aufsteigend sortierten Folgen a und b

- a: (2, 15, 19, 43)
- b: (4, 8, 17, 47)

Verschmelzen der Teilfolgen mit gradzahligem bzw. ungradzahligem Index

- c: (2, 4, 17, 19)
- d: (8, 15, 43, 47)

Vergleichen und gegebenenfalls Vertauschen der Paare (c_{i+1}, d_i) ergibt die aufsteigend sortierte Folge

- e: (2, 4, 8, 15, 17, 19, 43, 47)



Für $n=1$

- Genau ein Vergleichsmodul

Für $n > 1$ (es gelte $n = 2^k$ für $k > 0$)

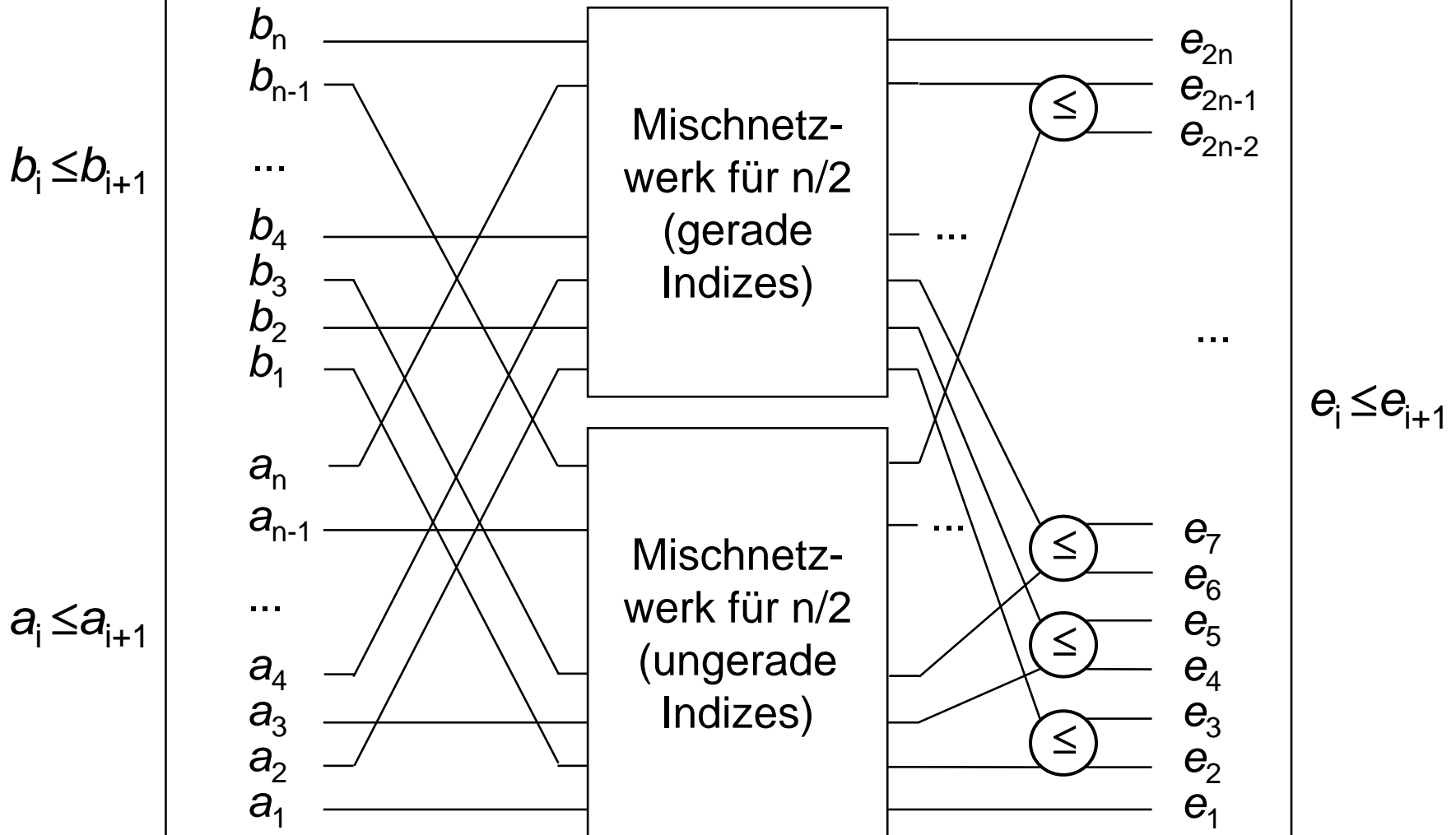
- Netzwerk hat $2n$ Eingabeleitungen, die linear angeordnet sind für die gradzahligen bzw. ungradzahligen Indexe
 - $a_1, b_1, a_3, b_3, \dots, a_{n-1}, b_{n-1}$ und $a_2, b_2, a_4, b_4, \dots, a_n, b_n$
- $2n$ Ausgabeleitungen existieren e_1, e_2, \dots, e_{2n}

Annahme

- Netzwerk zum Mischen zweier Folgen der Länge $n/2$ existieren
- Zusammensetzen des Netzwerks aus vorhandenen Moduln
 - Linker Teil sich kreuzender Linien gehört nicht zum Netzwerk. Sorgt lediglich für die richtige Reihenfolge der Eingaben.



Mischnetzwerk für n

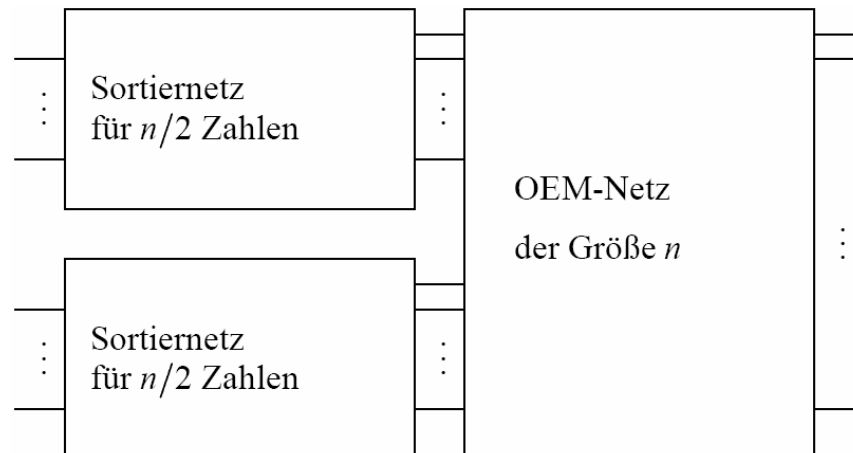


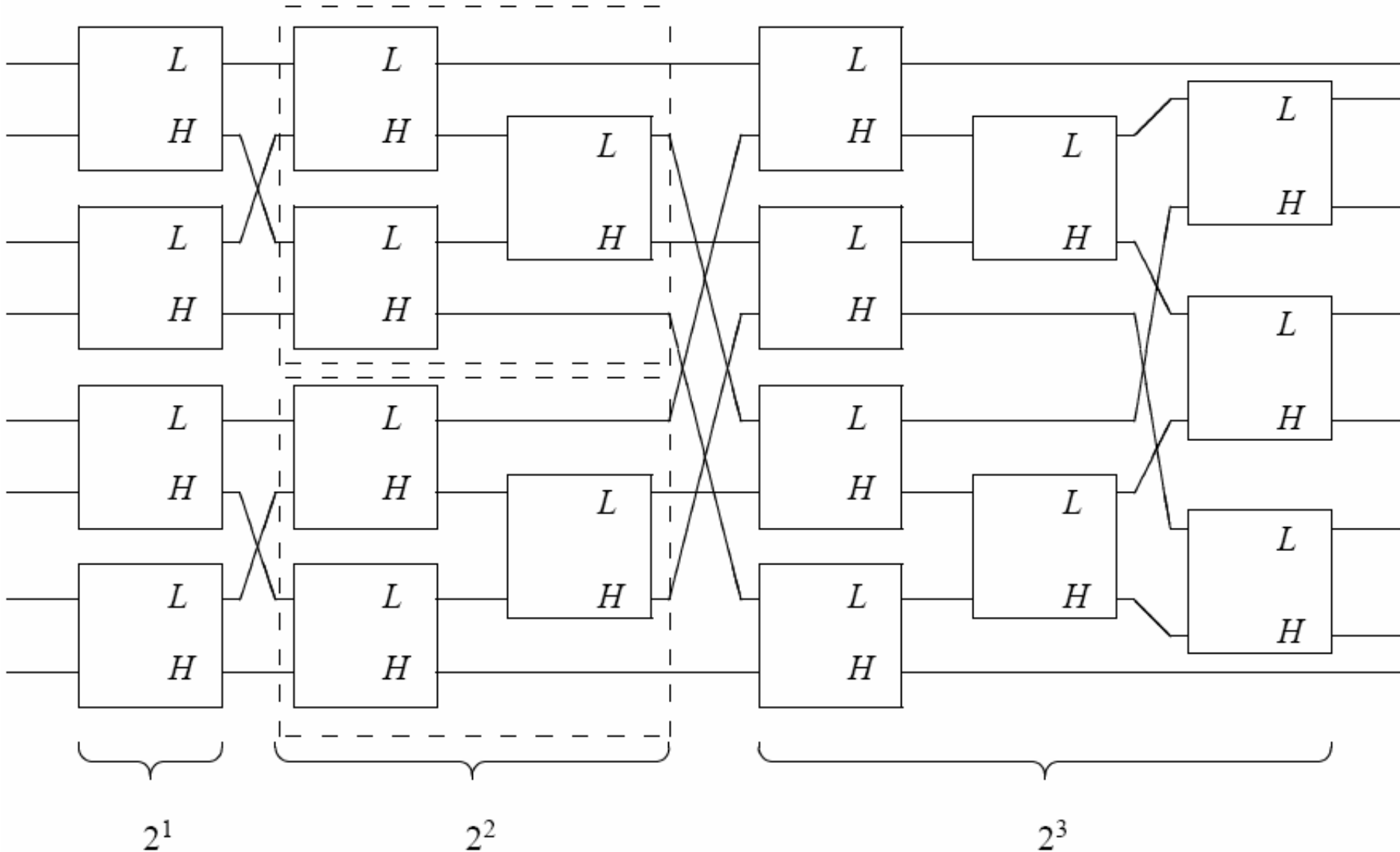
Sortieren von $n = 2^k$ Zahlen

- Start mit n Folgen der Länge 1
- Gleichzeitiges Mischen mit 2^{k-1} Odd-Even Mischnetzen der Größe 2^1 zu $n/2$ Folgen der Länge 2
- Dann $n/2$ Folgen der Länge 2 mischen mit 2^{k-2} Odd-Even Mischnetzen der Größe 2^2

Konstruktionsprinzip

- ... Sortiernetz für zwei Zahlen ist ein Vergleichsmodul
- Sortiernetz für $n > 2$ Zahlen zusammensetzen aus zwei Sortiernetzen für $n/2$ Zahlen und einem Mischnetz der Größe n





- Ein Vergleich von d_i und c_{i+1} erfordert $O(1)$.
- $n+n$ Mischen benötigt $n/2+n/2$ Mischnetzwerke.
- Insgesamt also $O(\log n)$ Ebenen.

Rechenschritte insgesamt:

$$T(n) = 2 T(n/2) + n/2 - 1 \quad \text{und} \quad T(2) = 1$$

$$\Rightarrow T(n) = O(n \log n)$$

Prozessoren: $\log n$ Ebenen zu je $n \Rightarrow n \log n$ Prozessoren.

Effizienz:

$$E(n, n \log n) = \frac{T(n, 1)}{n \log n T(n, n \log n)} = \frac{O(n)}{n \log n (O(\log n))} \approx \frac{c}{\log^2 n}$$



Anzahl Vergleichsmodule pro Schlüssel

- Höchstens $1 + 2 + \dots + k = k(k+1)/2$

Anzahl der Vergleichsmodule im Netz

- Odd-Even Sortiernetz der Größe n enthält höchstens $(1 + 2 + \dots + k) n/2$ Vergleichsmodule
 - Für große n sogar deutlich weniger

Wegen $k = \log n$ folgt

- N Zahlen können in Zeit $O(\log^2 n)$ mit Hilfe eines aus $O(n \log^2 n)$ Vergleichsmoduln bestehenden Netz sortiert werden.

