

Betriebssysteme Vorlesung 4

Concurrency - Nebenläufigkeit
Wechselseitiger Ausschluss, Synchronisation,
Deadlocks und Starvation
Literatur: Stallings Kapitel 5,6

Stand der Vorlesung

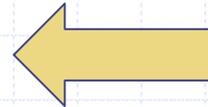
◆ Einführung: Ein Betriebssystem - Was ist das ?

◆ Prozeßmanagement

- Prozesse

- Threads

- Herausforderungen & Lösungen



HEUTE !

◆ Speicher Management

◆ Prozessor Scheduling

◆ E/A Management, Festplattenscheduling

◆ Datei Management

◆ Netzwerke

◆ Sicherheit

Übersicht: Probleme durch nebenläufige Prozesse

- ◆ Die simultane Nutzung von Ressourcen durch nebenläufige Prozesse kann eine destruktive Wirkung haben.
 - Abhilfe: Ressourcennutzung im wechselseitigen Ausschluß
Verfahren/Konzepte wechselseitigen Ausschluß zu erreichen
 - ◆ Software Lösungen: Dekker/Dijkstra, Peterson
 - ◆ Semaphore
 - ◆ Monitore
 - ◆ Nachrichtenübermittlung (Message Passing)
 - Wechselseitiger Ausschluß zeigt Nebenwirkungen
 - ◆ Verklemmung (Deadlock)
 - Bedingungen
 - Strategien: 1) ignorieren, 2) verhindern 3) vermeiden
4) erkennen & beseitigen
 - ◆ Verhungern (Starvation)
- ◆ Bei Prozessen, die durch geteilten Speicher oder Nachrichten kommunizieren, kann es ebenfalls zu Verklemmungen und Verhungern kommen.

Nutzen

- ◆ Sie lernen Konzepte kennen, mit denen sich der exklusive Zugriff auf Ressourcen / Dateien / Daten sicherstellen läßt. Anwendungsbereiche hierfür sind:
 - parallele Anwendungen mit Threads
 - parallele Anwendungen mit Prozessen
 - Datenbanken, (Transaktionskonzept und Sperren von Tabellen) und natürlich Betriebssysteme ...
- ◆ Sie lernen Phänomene kennen, die durch Synchronisation von Prozessen und den wechselseitigen Ausschluß ausgelöst werden können:
 - Verklemmungen (Deadlocks) und Verhungern (Starvation) und Gegenmaßnahmen.

Ursachen für Nebenläufigkeit

Wodurch wird Nebenläufigkeit ermöglicht ?

- Multiprogramming -> n Prozesse
- Multithreading -> n Threads in 1 Prozeß

Wer nutzt diese Möglichkeiten ?

◆ Strukturierte Anwendungen

- verteilte Berechnung mit n kommunizierenden Prozessen

◆ Betriebssystemarchitektur

- ein Betriebssystem wird mit mehreren Prozessen/Threads realisiert

Welche Probleme treten bei der gemeinsamen Nutzung von Ressourcen und bei der Kommunikation zwischen Prozessen auf ?

- Beispiel: Netzwerkdrucker
- Beispiel: Datei
- Beispiel: geteilte Variable
- Beispiel: Warten und Benachrichtigungsmuster

Anforderungen an das Prozeßmanagement

Das Ergebnis eines Prozesses darf NICHT von der Ausführungsgeschwindigkeit und -reihenfolge anderer unabhängiger Prozesse beeinflußt werden!

Entsteht das Problem nur bei mehreren CPUs ?

- Nein, Grundproblem unkontrollierter Prozeßfortschrittsgeschwindigkeit bereits bei 1 CPU, Multiprogramming, Zeitscheibenverfahren erlauben es Prozesse an beliebigen Stellen zu unterbrechen.
- Kritisch sind insbesondere Unterbrechungen zwischen Testen und Belegen einer freien Ressource

Dennoch unterschiedliche Szenarien

- Prozesse sind unabhängig, kennen sich nicht (typisch: Multiprogramming)
- Prozesse kooperieren über geteilten Speicher (z.B. Threads), kennen sich nur indirekt
- Prozesse kommunizieren über Nachrichten (Messages), kennen sich

Interaktion zwischen Prozessen und daraus resultierende Schwierig- keiten:

- Mutual Exclusion
- Deadlock
- Starvation

Übersicht aus Stallings

Degree of Awareness	Relationship	Influence that one Process has on the Other	Potential Control Problems
Processes unaware of each other	Competition	<ul style="list-style-type: none"> •Results of one process independent of the action of others •Timing of process may be affected 	<ul style="list-style-type: none"> •Mutual exclusion •Deadlock (renewable resource) •Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	<ul style="list-style-type: none"> •Results of one process may depend on information obtained from others •Timing of process may be affected 	<ul style="list-style-type: none"> •Mutual exclusion •Deadlock (renewable resource) •Starvation •Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	<ul style="list-style-type: none"> •Results of one process may depend on information obtained from others •Timing of process may be affected 	<ul style="list-style-type: none"> •Deadlock (consumable resource) •Starvation

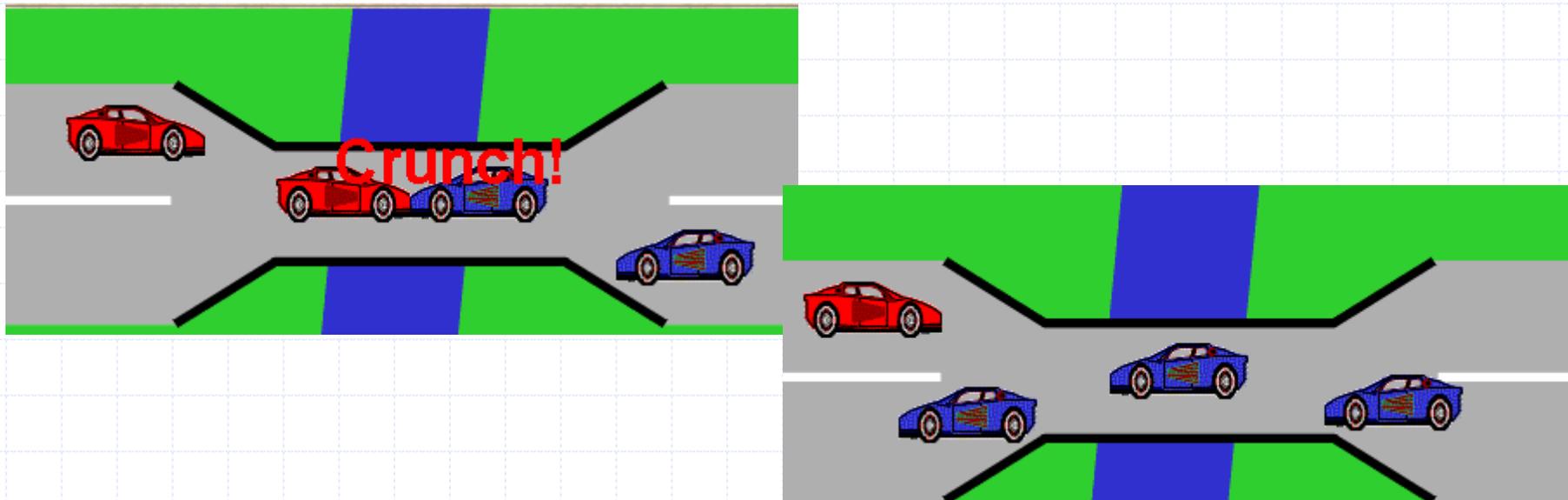
Illustration: exklusive Nutzung einer Ressource

Ressource: Brücke mit einspuriger Verkehrsführung

Prozesse: Fahrzeuge, die die Brücke aus verschiedenen Richtungen überqueren

Probleme:

- Verletzung der exklusiven Nutzung -> Unfall
- falls nachfolgende Fahrzeuge Vorrang vor entgegenkommenden haben -> Verhungern



Einfaches Beispiel im Umfeld von Programmen:

- globale, geteilte Variable

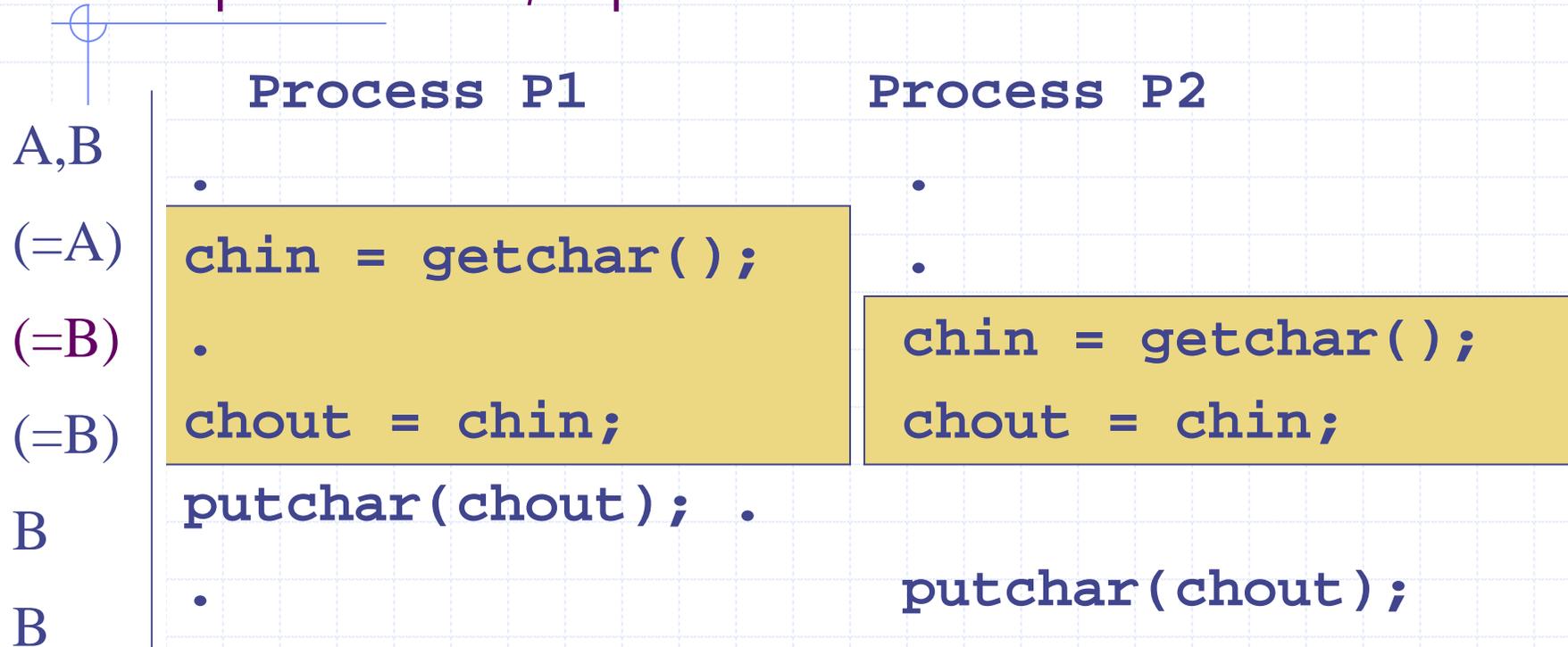
chin

lokale Variable chout

- bei 1 Prozeß ok
- bei n Prozessen, die nacheinander ablaufen, ok
- bei n Prozessen, die beliebig versetzt ablaufen (Nebenläufigkeit) ???

```
void echo()  
{  
    chin = getchar();  
    chout = chin;  
    putchar(chout);  
}
```

Beispiel mit n=2, Input: A B



Wesentlich für das Problem:

chin ist geteilte Variable (abstrakt: Ressource)

Lsg: nur jeweils 1 Prozeß im kritischen Abschnitt 

kritischer Abschnitt (critical section):

zusammenhängender Teil eines Programms, in dem auf geteilte Variable lesend oder schreibend zugegriffen wird.

Wechselseitiger Ausschluß, Mutex = Mutual Exclusion

Wechselseitiger Ausschluß:

- ◆ seien n Prozesse gegeben, die jeweils kritische Abschnitte beinhalten, in denen sie auf eine geteilte Variable (geteilte Ressource) zugreifen
- ◆ zu jedem Zeitpunkt darf nur maximal 1 Prozeß sich in seinem kritischen Abschnitt befinden

Grundidee aller Lösungen:

- ◆ jeder Prozeß testet, ob die Ressource R frei ist
- ◆ falls R frei, wird R belegt, der Abschnitt durchlaufen und R anschließend freigegeben
- ◆ falls R belegt ist, wartet der Prozeß darauf, dass R frei wird.

(*) Grundproblem: testen & belegen ist nicht 1 atomare Aktion

Lösungen variieren in der Behandlung von (*) und in der Gestaltung des Wartens:

- a) aktives Warten (Busy Waiting, Polling): Prozeß fragt selbst nach
- b) passives Warten: Prozeß wartet auf Signal, wird geweckt

Was muß ein Konzept für Mutual Exclusion gewährleisten?

1. Zu 1 Zeitpunkt darf nur 1 Prozeß im kritischen Abschnitt bzgl einer Ressource sein.
2. Das Anhalten eines Prozesse außerhalb eines kritischen Abschnittes darf andere Prozesse nicht behindern.
3. Keine Verklemmungen, kein Verhungern
4. Wenn der kritische Abschnitt frei ist, dann soll der nächste Prozeß ohne Verzögerung eintreten können.
5. Keine Annahmen über Fortschrittsgeschwindigkeit eines Prozesses oder über die Anzahl Prozesse
6. Ein Prozeß verweilt nur eine endliche Zeit im kritischen Abschnitt. (-> eher eine Annahme, Voraussetzung)

Lösungsvariante: Reine Software Lösungen

Algorithmen arbeiten stets mit Locks (Sperrern), die gesetzt und aufgehoben werden. Locks sind einfache, geteilte Variable. An dieser Stelle wiederholt sich das Problem Mutual Exclusion für Lock Variable sicherzustellen ...

Schema: "while (kritischerAbschnittBelegt) do nothing ;
setze kritischerAbschnittBelegt ;

- Grundannahme: read, write Operationen atomar
- 1. Problem: Variable testen und setzen nicht atomar
- 2. Problem: Busy Waiting verschwendet CPU Zyklen

Zur Lösung des 1. Problems werden Algorithmen sehr trickreich und damit auch für einen allgemeinen Einsatz zu fehlerträchtig.

Funktionierende Lösungen: Dekker-Dijkstra, Peterson

In Stallings, Kap 5.2 werden einige Fehlversuche diskutiert, lohnenswert zu lesen !

Das 2. Problem läßt Software Lösungen vergleichsweise ineffizient werden. Andere Lösungen sind attraktiver.

Petersons Algorithmus für 2 Prozesse

```
boolean flag[2] ; int turn ;  
void main() { flag[0]=false ; flag[1]=false ; parbegin(P0,P1) }
```

```
void P0()  
{  
while (true) {  
    flag[0] = true;  
    turn    = 1;  
    while(flag[1] && turn==1)  
        /* do nothing */ ;  
    /* critical section */ ;  
    flag[0] = false;  
    /* rest */  
}  
}
```

```
void P1()  
{  
while (true) {  
    flag[1] = true;  
    turn    = 0;  
    while(flag[0] && turn==0)  
        /* do nothing */ ;  
    /* critical section */ ;  
    flag[1] = false;  
    /* rest */  
}  
}
```

Lösungsvariante: SW Lösung mit etwas HW Support

- ◆ Atomare test&set oder exchange Operationen als Spracherweiterung für Programmiersprachen bieten Lösung für Problem 1
- ◆ Pro: funktioniert auch bei Multiprozessorsystemen, einfach & daher leicht zu verifizieren, unterstützt mehrere kritische Abschnitte
- ◆ Contra:
 - Problem 2 bleibt: Busy waiting verschwendet CPU Zyklen
 - Starvation: die Auswahl aus wartenden Prozessen ist unklar
 - Deadlock:
 - ein Prozeß A im kritischen Abschnitt wird durch hochpriorigen Prozeß B unterbrochen,
 - der Prozeß B wartet dann mit Busy Waiting auf Freigabe des Abschnittes
 - der Abschnitt kann aber nicht von A freigegeben werden, weil A wegen B nicht bearbeitet wird.

Lösungsvariante: Spezielle Datenstrukturen, Semaphore

Idee:

Zählvariable, die inkrement / dekrement Operationen zur Verfügung stellt. (Inkrement: $x = x+1$ Dekrement: $x = x-1$)

Bei Dekrement Operationen kann die Operation abhängig vom Wertestand blockieren und erst dann weiterlaufen, wenn eine ausreichende Anzahl von Inkrement Operationen durchgeführt wurde.

Eine Semaphore ist eine spezielle Variable, ein Objekt s

- mit einer ganzzahligen Variable $s.count$, $initial \geq 0$, d.h.
 - $s.count \geq 0$: gibt die Anzahl freier Zutritte an,
 - $s.count < 0$: gibt die Anzahl wartender Prozesse an
- Operation **wait**(s): $s.count = s.count - 1$
falls $s.count < 0$, wird der **aufrufende Prozeß suspendiert**
- Operation **signal**(s): $s.count = s.count + 1$,
falls $s.count \leq 0$, wird ein **wartender Prozeß aktiviert**

◆ **wait, signal** sind atomar (kein Interrupt)

◆ Suspendierte/wartende Prozesse werden in einer Warteschlange verwaltet.

Semaphore

Grundidee für Mutex:

Semaphore sichert Zutritt zu kritischem Abschnitt

Dijkstra '65: binäre Semaphore mit Werten $\{0,1\}$ und Operationen P, V

Schwache Semaphore

- Auswahl des nächsten wartenden Prozesses aus der Warteschlange ist beliebig, Verhungern ist daher möglich

Starke Semaphore

- FIFO-Auswahl aus der Warteschlange, d.h. der Prozeß, der am längsten gewartet hat, wird als erster fortgeführt
- Verhungern nicht möglich

Beurteilung:

- ◆ Pro: kein aktives Warten, signal() Operation weckt ggfs wartenden Prozeß
- ◆ Contra: bei mehreren Semaphoren ist die Verteilung der wait-signal Operationen im Code sehr fehlerträchtig
- ◆ Implementierung: führt zu Mutex Problem auf nächster Ebene für wait, signal
 - Hardware, Firmware Lösung, Dekker, Peterson, test&set Operation

Lösungsvariante: Monitor

Ein Monitor ist ein Software Modul

- ◆ Lokale Variable nur für Monitor sichtbar (Kapselung)
 - ◆ Prozeß nutzt (betritt) Monitor über Monitorfunktionen
 - ◆ Nur 1 Prozeß kann zu 1 Zeitpunkt Monitor aktiv nutzen, andere Prozesse werden durch den Monitor suspendiert
 - ◆ Innerhalb des Monitors können Prozesse mit **wait** zurücktreten und später von anderen Prozessen im Monitor mit **notify** geweckt werden
- } Typisch für Objekte

Wir betrachten Ansatz von Lampson/Redell:

- ◆ cnotify legt die Verantwortung für die Bedingung auf wartende Prozesse
- ◆ cwait(c): Prozeß blockiert ggfs wg Bedingung **c** und gibt dabei den Monitor für andere Prozesse frei
- ◆ cnotify(c): Prozeß gibt Signal zum Wecken für 1 suspendierten Prozeß, kann dann aber selbst fortfahren, ganz analog cnotifyAll(c)
- ◆ Suspendierte Prozesse, die durch ein notify gelegentlich geweckt werden, müssen Bedingung erneut prüfen daher stets:
while (not c) cwait(c) ;
- ◆ Vorteile: einfachere Programmierung, modular, robuster

Monitore in Java

In Java gehört zu jedem Objekt ein eigener Monitor.

Überwachte Methoden sind als **synchronized** gekennzeichnet.

Monitor und Objekt kapseln somit Daten + Zugriffsmethoden, so daß bzgl der Methoden wechselseitiger Ausschluß gilt. Zusätzlich existiert die Möglichkeit zur freiwilligen, vorübergehenden Suspendierung des aktiven Threads, wobei der Monitor für andere Prozesse/Threads frei wird (Mutex !!!)

```
while (not c) wait() ;
```

Die Änderung einer Bedingung kann die jeweils aktive Funktion (der aktive Prozeß/Thread) den suspendierten Threads mittels **notify()** bzw **notifyAll()** mitteilen. In **Java**: der aktive Thread bleibt aktiv. Geweckte Threads prüfen zu späterem Zeitpunkt die Gültigkeit der Bedingung.

Das Monitorkonzept hat entscheidende Vorteile:

- implizite Handhabung zur Belegung / Freigabe von Locks
- Kapselung von Belegung und Freigabe
- kein aktives Warten

Lösungsvariante: Message Passing

Message Passing = Kommunikation zwischen Prozessen durch Nachrichtenübermittlung

- ◆ erlaubt Synchronisation von Prozessen (wichtig für Mutual Exclusion)
- ◆ erlaubt Informationsaustausch
- ◆ Zusatznutzen: auch für verteilte Systeme ok
- ◆ typische Operationen:
 - send (Ziel, Nachricht)**
 - receive (Quelle, Nachricht)**
- ◆ Implizite, aber realistische Annahmen bei 1 Mailbox für mehrere (empfangende) Prozesse
 - falls Nachricht in Mailbox, dann erhält sie nur 1 Prozeß, andere Prozesse blockieren beim Aufruf von receive()
 - falls Mailbox leer, dann blockieren alle Prozesse bei receive().
 - Bei Eintreffen einer Nachricht wird nur 1 Prozeß aktiviert und nur 1 Prozeß erhält die Nachricht.

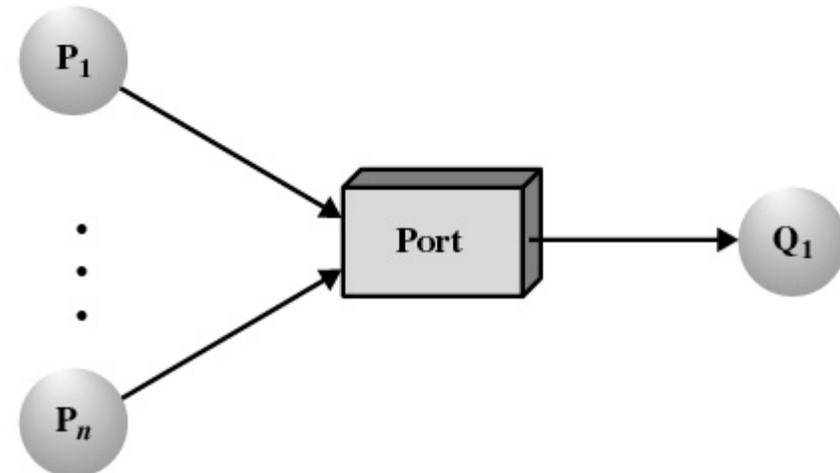
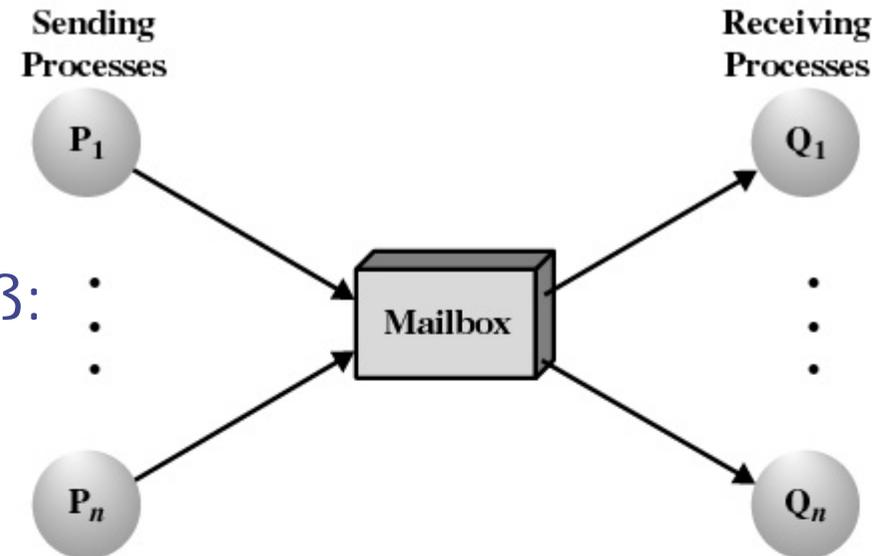
Message Passing mit nicht-blockierendem Senden, blockierendem Empfangen

◆ löst wechselseitigen Ausschluß:

- Mailbox **mutex** als Semaphore
- Eintrittsbedingung:
`receive(mutex,msg)`
- Austrittsaktion: `send(mutex,msg)`

◆ löst Produzent / Konsumenten Problem:

- 2 Mailboxen: **P2C** als Puffer, **C2P** als Kapazitätsbeschränkung
- produce: `receive(C2P,msg)`,
`send(P2C,data)`
- consume: `receive(P2C,data)`,
`send(C2P,msg)`



Zwischenfazit: Wechselseitiger Ausschluß

Parallele Prozesse / Threads

=> Zugriff auf geteilte Ressourcen kritisch

=> Lösungen für wechselseitigen Ausschluß erforderlich

- reine SW Lösungen: Dekker/Dijkstra, Peterson wg Fehlerträchtigkeit und Busy Waiting ungünstig
- Semaphor-Konzept: ok, aber fehlerträchtig durch verteilte inkrement / dekrement Operationen
- Monitor-Konzept: ok
- Message passing: günstig auch im Hinblick auf verteilte Applikationen

Haben wir damit alle Probleme gelöst ?

◆ Verklemmungen (Deadlocks)

- Was kennzeichnet Verklemmungen ?
- Maßnahmen: 1) verhindern 2) vermeiden 3) erkennen & beseitigen

◆ Verhungern (Starvation)

- Faire Verteilung von Ressourcen

Deadlock = Verklemmung

Dauerhafte Blockierung einer (Teil-) Menge von Prozessen durch Warten auf Ressourcen oder Nachrichten

Deadlocks sind unerwünscht

-> Anforderung an OS

Deadlocks bei Prozessen / Threads

zu verhindern:

- Bedingungen an die Ressourcenvergabe sind so gestaltet, dass es grundsätzlich nicht zu Deadlocks kommen kann.

oder zu vermeiden:

- Ressourcen werden durch das Betriebssystem situativ nur so vergeben, dass keine Deadlocks auftreten.

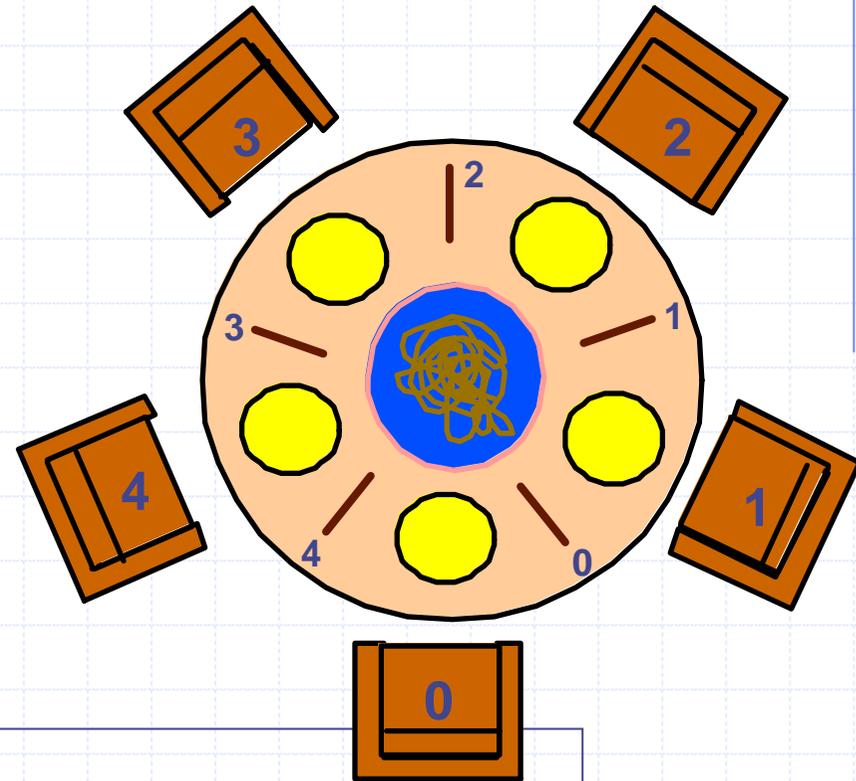
oder zu entdecken und beseitigen:

- falls Deadlocks auftreten, diese zu entdecken und zu beseitigen.

Achtung: gängige Praxis in Betriebssystemen ist jedoch **Ignorieren** d.h. keine automatische Lösung umgesetzt

Ein Klassiker: Dining Philosophers Problem

5 Philosophen verbringen ihr Leben mit Essen und Denken in stetigem Wechsel. Am gemeinsamen Tisch finden sich jedoch nur 5 Gabeln, von denen jeder Philosoph jeweils die Gabel zu seiner rechten und zu seiner linken Seite zum Essen verwendet.



Prozesse: Philosophen

Ressourcen: Gabeln

Besonderheit: 2 Ressourcen je Aktion erforderlich

Gesucht: Algorithmus mit 5 Threads (Philosophen) und 5 geteilten Objekten (Gabeln) der unbegrenzten Ablauf ohne Deadlocks und Starvation erlaubt.

Dining Philosophers

Deadlock tritt auf, wenn

- jeder Philosoph Gabeln nacheinander aufnimmt
- alle Philosophen bei der Gabelaufnahme die gleiche Reihenfolge wählen
- und alle Philosophen 1 Gabel aufgenommen haben.

Starvation kann auftreten, wenn

- jeder Philosoph beide Gabeln gleichzeitig aufnimmt oder wartet bis beide frei sind.

Lösung ohne Deadlock & Starvation:

1. Anzahl Philosophen am Tisch wird auf 4 begrenzt
2. ... und natürliche existieren weitere Lösungen

Dining Philosophers sind eine interessante Programmieraufgabe zur Anwendung von Threads und Monitoren!

Bedingungen für Deadlocks

1. Mutual exclusion

- maximal 1 Prozeß darf eine Ressource zu einem Zeitpunkt nutzen

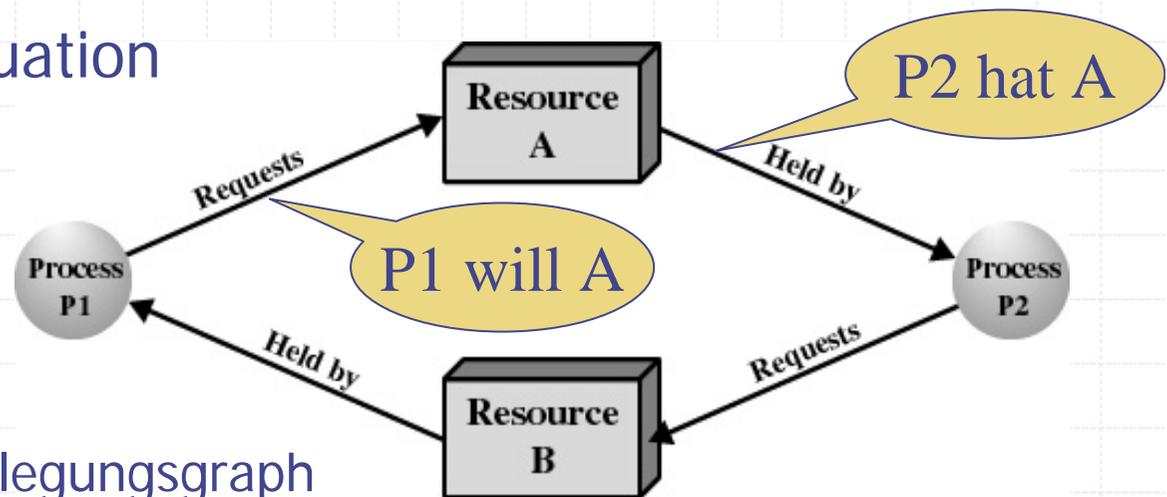
2. Halten-und-Warten

- ein Prozeß allokiert nacheinander Ressourcen, ggfs wartet er auf Ressourcen, jedoch ohne bereits allokierte Ressourcen freizugeben

3. Keine Unterbrechung und zwangsweise Ressourcenfreigabe bei einem Prozeß

4. Zyklische Wartesituation

Bedingungen 1-4 sind notwendig & hinreichend!



Ressourcen-Belegungsgraph

1. OS Strategie: Deadlockverhinderung

1 der 4 Deadlockbedingungen wird durch OS aufgehoben
=> Deadlocks können nicht entstehen

1.1 Prozesse müssen alle Ressourcen zusammen anfordern (one-shot-allocation)

- pro: für bestimmte Prozesse ok
- contra: ineffizient, verzögert Prozeßerzeugung, zukünftiger Bedarf muß bei Erzeugung bekannt sein

1.2 Unterbrechung von Prozessen mit Ressourcenfreigabe

- falls Prozeß Ressourcen allokiert hat und bei weiteren Anfragen blockiert, muß er alle Ressourcen freigeben
- falls ein Prozeß eine bereits belegte Ressource nutzen will, zwingt das OS (je nach Priorität) den besitzenden Prozeß zur Aufgabe seiner Ressourcen
- pro: bei einfachem Sichern des Ressourcenstatus und wieder Aufsetzen ok
- contra: mehr Unterbrechungen als erforderlich, zyklisches Wiederholen von Abläufen möglich falls Prozesse gleicher Priorität auftreten

1. OS Strategie: Deadlockverhinderung

1.3 Keine zyklischen Wartesituationen durch eine lineare Ordnung auf den Ressourcen (hierarchische Belegung)

- definiere Ordnung auf Ressourcen, d.h. durchnummerieren:
1,2,3,...
- Prozesse dürfen Ressourcen nur in aufsteigender Ordnung allokalieren
- contra: ineffizient, unnötig vielen Prozessen werden Ressourcen vorenthalten

2. OS Strategie: Deadlock Vermeidung

- ◆ weniger restriktive Strategie, erfordert Entscheidungsalgorithmus zur Ressourcenvergabe, so daß Deadlocks vermieden werden.
- ◆ Advance Claim Verfahren:
 - Vor.: zukünftiger Ressourcenbedarf jedes Prozesses bekannt
 - Zustand wird durch aktuelle Ressourcenbelegung beschrieben.
 - OS erfüllt zusätzlichen Ressourcenbedarf nur, wenn deadlockfreie Abarbeitung aller Prozesse möglich ist.
 - Variante 1) Starte neuen Prozeß nur, wenn Maximalanforderungen aller Prozesse simultan erfüllt werden können.
 - Variante 2) Erfülle zusätzlichen Ressourcenbedarf nur, wenn sequentielle Abarbeitungsfolge für alle Prozesse möglich bleibt.
= Systemzustand ist **sicher**.

Banker's Algorithmus:

- ◆ Erfülle Anforderung nur dann, wenn der Nachfolgezustand ebenfalls sicher ist.

Bankers Algorithmus Bestimmung sicherer Zustände

Beispiel: initialer Zustand

Beobachtung: P2 kann komplett durchgeführt werden, weil die maximale Anforderung (+ 1 R3) noch verfügbar ist.

Da P2 nach Terminierung alle Ressourcen freigibt, stehen dann mehr Ressourcen zur Verfügung als im aktuellen Zustand und die Anzahl Prozesse ist geringer geworden.

Wir iterieren den Vorgang also bis alle Prozesse terminiert sind und damit ein sicherer Zustand vorliegt.

Ressourcen insgesamt

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
9	3	6

Resource Vector

R1	R2	R3
0	1	1

Available Vector

4 Prozesse
3 Ressourcen
maximale Anforderungen

Aktuell
belegte
Ressourcen

Noch verfügbare
Ressourcen

Bankers Algorithmus

Bestimmung sicherer Zustände

2. Schritt: Zustand nach Durchf. von P2, dann P1, P3, P4, Zustand sicher!

Die Auswahl des Prozesses ist unkritisch, da ablauffähige Prozesse bzgl der Sequenz nicht in Konflikt stehen.

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

R1	R2	R3
6	2	3

Available Vector

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

R1	R2	R3
7	2	3

Available Vector

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

R1	R2	R3
9	3	4

Available Vector

Erkennung unsicherer Zustände

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
9	3	6

Resource Vector

R1	R2	R3
1	1	2

Available Vector

Aktueller Zustand ist sicher: durch Abarbeitung in sequentieller Reihenfolge P2, P3, P4, P1 sind alle Anforderungen ohne Deadlock zu befriedigen.

Darf Anfrage von P1 für zusätzliche Ressource R1 und R3 bejaht werden ?

Erkennung unsicherer Zustände

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
0	1	1

Available Vector

Zustand ist unsicher, weil keine Reihenfolge von P1, P2, P3, P4 existiert, die mit bestehenden Ressourcen durchführbar ist - unter der pessimistischen Annahme sofortiger maximaler Allokation.

OS würde Ressourcen nicht zuweisen!

Achtung: unsicherer Zustand führt nur dann sicher zu Deadlock, wenn Allokationen ohne vorherige Freigabe erfolgen!

Deadlock Vermeidung

Bankers Algorithmus rechnet bei einer Ressourcenanforderung den Folgezustand aus und prüft dann, ob der Folgezustand sicher ist, d.h. ob eine Sequenz existiert, so daß alle Prozesse terminieren können. Nur bei sicherem Folgezustand, wird die Ressourcenanfrage befürwortet.

Restriktionen

- Maximale Ressourcenanforderungen müssen im voraus bekannt sein.
- Prozesse müssen unabhängig sein, keine Synchronisation/Reihenfolgebedingungen untereinander
- Anzahl Ressourcen muß fest sein
- Kein Prozeß darf terminieren ohne Ressourcen freizugeben

◆ Vorteile:

- keine Unterbrechung,
- keine erneute Berechnung (Wiederholung),
- weniger restriktiv als Verhinderung

3. OS Strategie Deadlockerkennung

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request Matrix Q



	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation Matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource Vector

R1	R2	R3	R4	R5
0	0	0	0	1

Available Vector

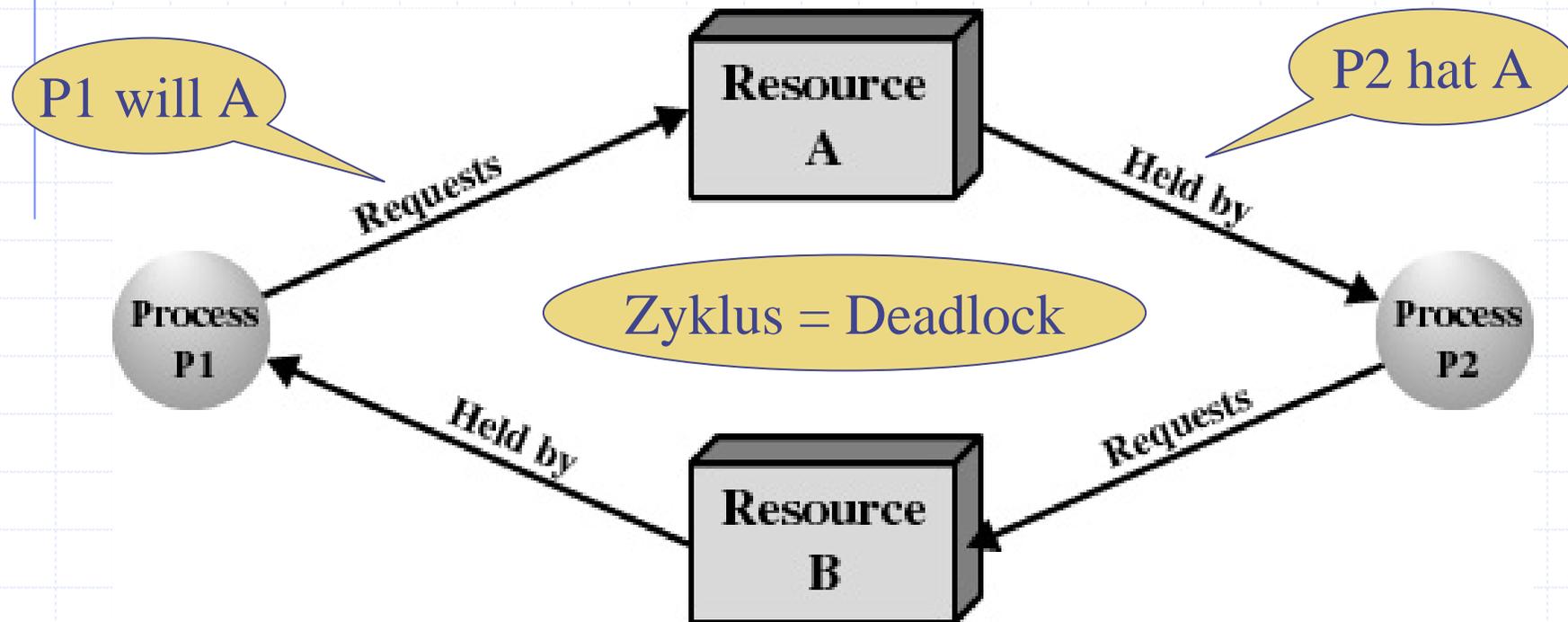
Neu: nur aktuelle Anforderungen

Erkennungsverfahren sucht nach Menge wechselseitig blockierter Prozesse. Annahme: nach Erfüllung der Anforderungen werden Ressourcen durch den Prozeß freigegeben. Durchführbare Prozesse (P3) oder unkritische Prozesse (ohne Ressourcen, P4) werden markiert (iterativ wg Annahme!). Prozesse, die am Schluß noch unmarkiert sind, befinden sich in einem Deadlock.

Alternative: Zyklensuche im Ressourcen-Allokations Graph

Ressourcen-Allokations Graph

- ◆ Prozeß- > Ressource: "fordert an"
- ◆ Ressource- > Prozeß: "wird belegt von"



Deadlockbeseitigung

◆ Prozesse im Deadlock

- alle Prozesse abbrechen (abort) und aus Zwischenspeicherung Prozesse an Checkpoint wiederaufsetzen und fortführen (Risiko: kann zum gleichen Deadlock führen)
- schrittweise Prozesse abbrechen, bis Deadlock aufgehoben ist
- schrittweise Ressourcen zwangsweise entziehen, bis Deadlock aufgehoben

◆ Was tun mit derart behandelten Prozessen ?

- Roll-back oder Restart

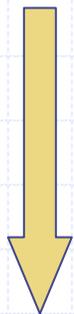
◆ Welche Prozesse sollten zuerst ausgewählt werden ?

- Kostenkriterien
 - ◆ geringste verbrauchte Prozessorzeit
 - ◆ geringste Anzahl produzierter Zeilen Ausgabe
 - ◆ größte geschätzte Restprozessorzeit
- Bedeutung des Prozesses
 - ◆ geringste Anzahl bisher belegter Ressourcen
 - ◆ geringste Priorität

Zwischenfazit Deadlocks

Für Betriebssysteme ist keine Ideallösung bekannt, Silberschatz et al schlagen eine Kombination vor

- ◆ gruppieren Ressourcen (Swap space, Prozeß Ressourcen, Hauptspeicher, interne Ressourcen)
- ◆ bilde lineare Ordnung zur Verhinderung von Deadlocks zwischen Ressourcengruppen
- ◆ innerhalb der Gruppe: gruppenspezifischer Algorithmus



- Swap space: Prozeß muß alle Anforderungen zugleich stellen
- Prozeß Ressourcen: Vermeidungsstrategie oder lin. Ordnung
- Hauptspeicher: Unterbrechung -> Swapping
- interne Ressourcen: lineare Ordnung

Lineare Ordnung

Für Anwendungsprogramme mit Prozessen & Threads lassen sich einige dieser Techniken jedoch einsetzen, insbesondere die hierarchische Belegung.

Zusammenfassung

- ◆ Nebenläufige Prozesse / Threads ist bzgl Ressourcennutzung kritisch
 - exklusive Nutzung -> wechselseitiger Ausschluß
 - Maßnahmen für den wechselseitigen Ausschluß
 - ◆ Algorithmen, Semaphore, Monitore, Message Passing
- ◆ Wechselseitiger Ausschluß und Synchronisation von Prozessen kann zu Deadlocks und Starvation führen
 - Charakterisierung von Deadlocks
 - Maßnahmen
 - ◆ verhindern (Vorbedingung aufheben)
 - ◆ vermeiden (Entscheidungsalgorithmus)
 - ◆ erkennen & beseitigen (Detektionsalgorithmus und erzwungener Ressourcenentzug / Terminierung)
- ◆ häufige Lösung in Betriebssystemen: Ignorieren, in der Hoffnung, dass Deadlocks relativ selten auftreten
- ◆ Probleme aus Betriebssystemen überschneiden sich mit Problemen im Design paralleler Programme