

Mag. iur. Dr. techn. Michael Sonntag

KV Betriebssysteme

Synchronisation 2: Deadlocks

Institut für Informationsverarbeitung und
Mikroprozessortechnik (FIM)
Johannes Kepler Universität Linz, Österreich

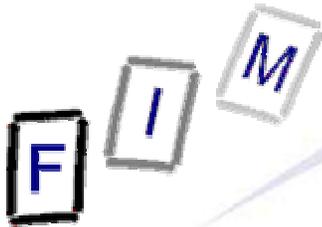
E-Mail: sonntag@fim.uni-linz.ac.at
<http://www.fim.uni-linz.ac.at/staff/sonntag.htm>

F I M

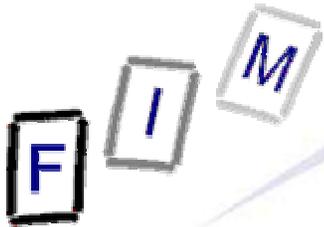
Fragen?

Bitte gleich stellen!

Wiederholung: Sequentielle Prozesse

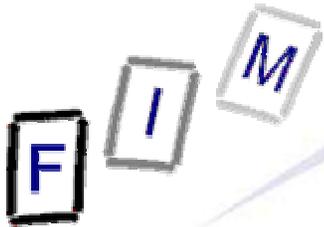


- Zu jedem beliebigen Zeitpunkt t wird genau eine einzige Instruktion ausgeführt
 - Hängt ab vom Abstraktionsgrad
 - Beispiel: Pipelining in der CPU
 - » Jeder Befehl wird in kleinere Teile zerlegt
 - » Diese werden aber streng hintereinander ausgeführt
 - Mehrere Pipelines in einer CPU: Siehe Multiprozessorrechner!
 - » Die CPU stellt per Hardware sicher, dass der Ablauf immer einer sequentiellen Ausführung entspricht
 - » Dies bedeutet, dass ev. Ergebnisse verworfen werden!
- Also immer nur **eine** Instruktion, eine **nach** der anderen, und nachdem die vorige (zumindest logisch) **komplett abgeschlossen** wurde!
 - Dann gibt es keine Synchronisationsprobleme!



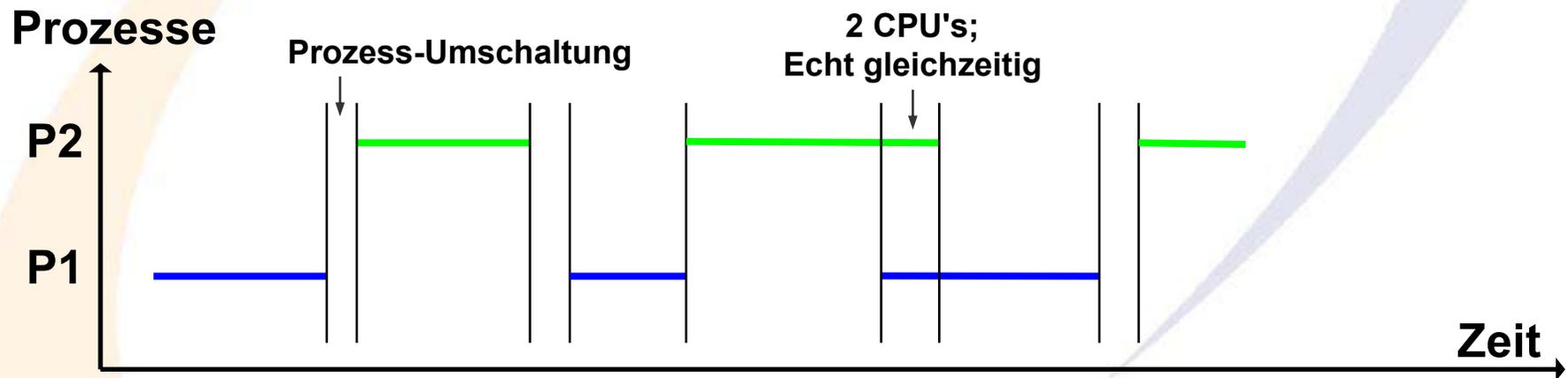
Wiederholung: Parallele Prozesse

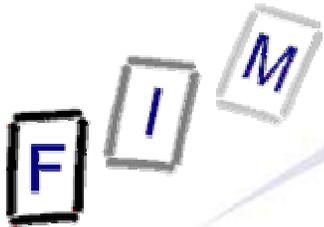
- Basiert auf zumindest zwei sequentiellen Prozessen
- Parallelität: Es gibt zumindest eine winzige Überschneidung
 - Zumindest eine Instruktion des zweiten Prozesses wird ausgeführt bevor der erste Prozess beendet ist
 - » Praxis: Erste Instruktion des zweiten Prozesses nach der ersten und vor der letzten Instruktion des ersten Prozesses
- **Keine** Annahmen über die Reihenfolge der Instruktionen
- Unabhängig von der Anzahl der CPU's, Rechner, ...
 - Eine CPU: Zeitliche Verzahnung der Prozesse
 - » Siehe Scheduling: Time slicing, ...
 - Mehrere CPU's: "Echt" gleichzeitig
- "Unabhängige Prozesse": Keine gemeinsamen Daten



Wiederholung: Interleaving

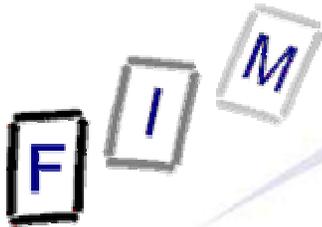
- Verzahnung von Prozessen
→ Wie diese erfolgt ist nicht festgelegt, es muss sich nur zumindest einmal eine geringfügige Überschneidung ergeben!
- Beispiel:





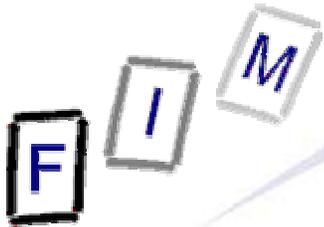
Einschub: Spin Locks (1)

- Endlosschleife bis die Bedingung wahr wird
 - Eine Möglichkeit für die Implementation von Synchronisation
- Pseudocode:
 - `while(!Bedingung) ;`
- Bedingung: Die Prüfung muß eine **unteilbare Aktion** sein!
 - Darin enthaltene Variablen dürfen während der Prüfung **nicht** verändert werden
 - Es darf währenddessen also **kein** Interleaving geben!
- Nachteil: Die CPU ist dauernd beschäftigt!
- Vorteil: Keine Prozessumschaltung nötig!
 - Daher nur bei Multiprozessorrechnern sinnvoll, oder wenn die Bedingung extern ist (Netzwerk, Disk)!



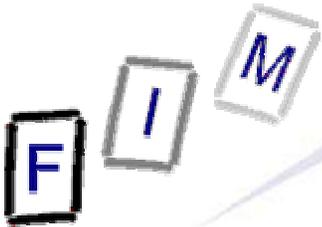
Einschub: Spin Locks (2)

- Anwendung von Spin-Locks:
 - Wenn Locks nur sehr kurz gehalten werden (z. B. im Kernel)
 - Nützlich insbesondere auf Geräten mit mehreren CPUs
 - » Hier gar keine Taskumschaltung nötig, da eine andere CPU den Wert ändert und so den Lock beendet!
- Realisierung:
 - Sperren der Interrupts und Konfiguration des Kernel, sodass kein Taskswitch auftritt
 - Bei mehreren CPUs Hardware-Unterstützung nötig
- Kann es auch bei Spin-Locks zu Deadlocks kommen?
 - Natürlich, kein Unterschied zu anderen Synchronisations-Mechanismen; hier wird eben nur Busy-waiting durchgeführt!



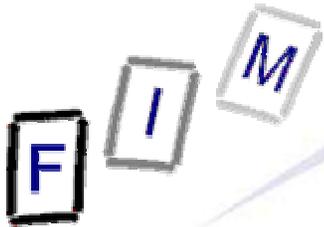
Wiederholung: "Busy waiting"

- Je nach Implementierung von critical regions (siehe vorige Übung!) kann es zu "busy waiting" kommen
- Dies bedeutet, dass das Programm in einer Schleife (in der sonst nichts passiert!) dauernd läuft, bis es seinen Ablauf fortsetzen darf
 - `while(!proceed) ;`
// OS setzte irgendwann Variable "proceed" auf "true"
- Dies ist nicht ideal, da Rechenzeit (time slices, ...) verbraucht wird, ohne für das "eigentliche" Ergebnis benötigt zu werden
- Besser sind daher Methoden, welche den Zustand des Prozesses auf "waiting" setzen, wodurch er automatisch beim Scheduling ausgelassen wird!



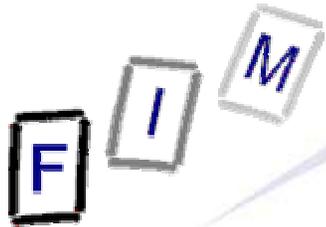
Bedeutung auf Einzelrechnern

- Auf jedem Rechner laufen heute viele Anwendungsprogramme gleichzeitig, aber nur wenige arbeiten auf dieselben Daten im Speicher: Eher geringe Bedeutung
 - **Sehr hohe Bedeutung jedoch bei Multithreaded-Programmen!**
- Aber viele Prozesse arbeiten auf denselben logischen Daten (z.B. auf der Festplatte): Hier besitzt die Synchronisation (in Form von Locking) große Bedeutung!
 - **Datenbanken: Transaktionen**
- Unerlässlich ist Synchronisation im inneren des Betriebssystems: Sehr viele Dinge laufen gleichzeitig ab und alle bearbeiten interne Datenstrukturen
 - **Beispiele: Netzwerke, Festplatten, IPC, ...**



Bedeutung in Netzwerken

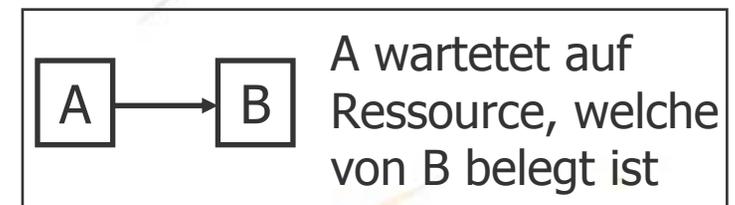
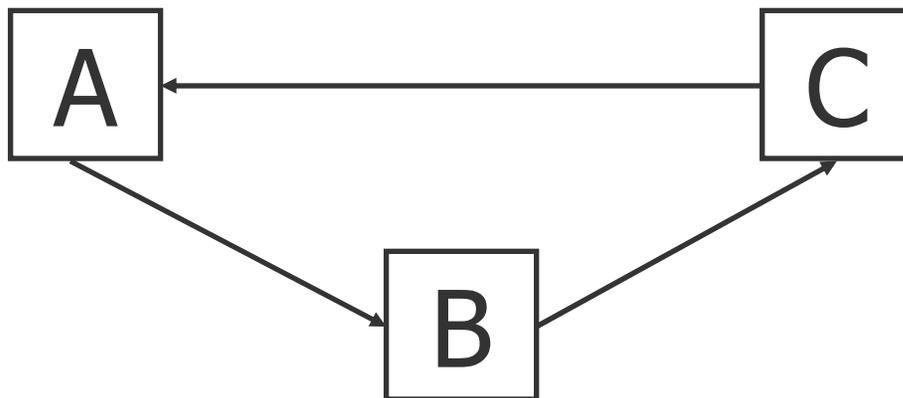
- Programme die auf ≥ 2 Rechner laufen haben das Problem, dass es keinen gemeinsamen Takt geben kann
 - » Innerhalb eines Multiprozessor-Rechners noch möglich!
 - Kommunizieren die Programme miteinander, so ist Synchronisation unvermeidbar: Es kann **nie** garantiert werden wann genau Daten einlangen!
- Hier gibt es zwar keinen gemeinsamen Speicher, aber die Programme arbeiten immer auf gemeinsame Daten (sonst würden sie ja nicht kommunizieren!)
 - Auch wenn diese Daten ev. mehrfach existieren
- Synchronisation ist in Netzwerken viel komplexer und wird hier nicht behandelt
 - Siehe LVA Netzwerke und Verteilte Systeme!

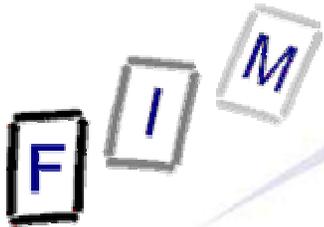


Trotzdem Probleme? Synchronisation ist kein Allheilmittel!

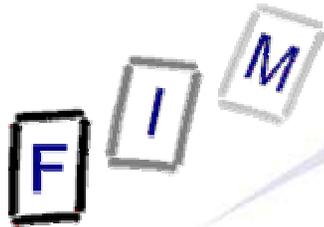
- Weitere Probleme existieren:
 - Deadlocks: A wartet auf B und B wartet auf A
 - » Beide hängen (in Semaphor, Monitor, etc.) für alle Ewigkeit fest
 - Livelocks: A signalisiert an B und B signalisiert an A
 - » Es wird im Kreis synchronisiert, ohne eigentliche Arbeit zu tun
 - » Eher selten!
 - Starvation: Ein Thread hat bei der Zuteilung immer (oder zumindest sehr lange) Pech und erhält keine Zuteilung
 - » Siehe "Fair Semaphor" in Java 1.5 als Abhilfe!
 - Ignorieren: Ein Thread greift einfach ohne Synchronisation zu
 - » Programmierfehler (oder mangelnde Dokumentation)!
 - Andere Probleme: Siehe vorige Übung (z.B. "lost update")
- Synchronisation kann diese Probleme **nicht** beheben!
 - Sicherheitsmechanismen, Scheduling, **Programmierer**, ...

- Bei einem Deadlock wartet aus einer Menge an Prozessen jeder Prozess auf eine Ressource, die irgendein anderer Prozess dieser Menge bereits besitzt
 - Kann NUR bei Synchronisation auftreten
 - » Kein Warten auf Ressourcen → Keine Deadlocks
 - » Keine Synchronisation → Keine Deadlocks



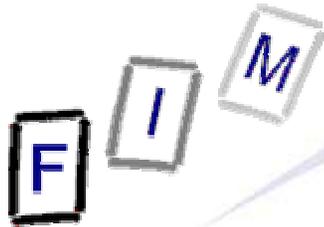


- 4 Bedingungen sind nötig (Coffman'sche Bedingungen)
 - Mutual exclusion: Die Ressource kann nicht gemeinsam genutzt werden; d. h. zu jedem Zeitpunkt maximal ein Prozess
 - » Beispiele: CPU, Drucker, Datenbus, ...
 - » Gegenbeispiele: Bildschirm (teilw.), Maus
 - Hold and wait: Während des Wartens auf weitere Ressource wird bereits belegte Ressource nicht freigegeben
 - » Beispiel: Semaphore
 - » Gegenbeispiel: Monitor (nur eigener!)
 - No preemption: Ein Prozess muß eine Ressource freiwillig zurückgeben; sie kann ihm nicht entzogen werden
 - » Beispiel: Windows <=95; Datei
 - » Gegenbeispiel: Windows >=NT, Linux; Bildschirm-Vordergrund
 - Circular wait: Prozesse warten "im Kreis".
 - » D. h. A wartet auf B, B wartet auf C und C wartet auf A



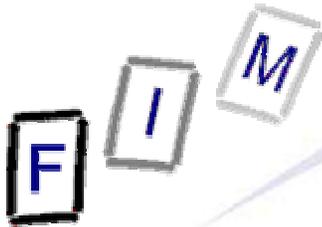
Deadlock – Behandlung: 3 grundsätzliche Lösungen

- Vermeidung: Wir stellen sicher, dass es von vornherein nie zu einem Deadlock kommen kann
 - Dies bedeutet einen dauernden zusätzlichen Aufwand
 - "Sichere" Lösung
 - Verschiedene Lösungen möglich
 - » Für jede der Coffmanschen Bedingungen eine!
- Erkennung und Behebung: Wir beobachten das System und wenn ein Deadlock auftritt, wird dieser aufgelöst.
 - Beobachtung bedeutet einen dauernden Zusatzaufwand.
 - Auflösung kann problematisch sein
 - » Z.B. Datei "wegnehmen" durch externes Schließen
 - » Prozess beenden und neu starten (Daten- und Arbeitsverlust)



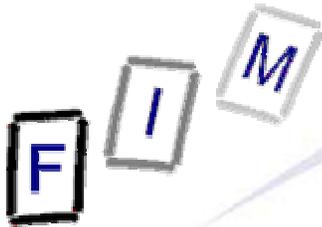
Deadlock – Behandlung: 3 grundsätzliche Lösungen

- Ignorieren: Das Problem wird nicht behandelt/gelöst
 - Deadlocks sind sehr selten. Falls sie doch auftreten, muss der Benutzer/Administrator/... das Problem **selbst** und **irgendwie** erkennen lösen.
 - Kein zusätzlicher Aufwand im System
 - » Dafür extern: Bei dem, der Erkennen/Auflösen muß!
 - Üblicher Ansatz für Betriebssysteme
 - » Windows, Linux, ...
 - » Anderes nur für spezielle Systeme bzw. besondere Programme
 - Hochverfügbarkeit in parallelen/verteilten Systemen
 - Entweder der Anwender oder die ausgeführten Programme selbst müssen das Problem lösen
 - Auch für sonstige Probleme (Livelock, Abstürze, ...) nötig



Deadlock - Vermeidung

- Mutual exclusion
 - Lässt sich nur bei teilbaren Ressourcen beseitigen
 - Beispiel: Read-only Dateien können von beliebig vielen Prozessen gleichzeitig gelesen werden, daher keine Synchronisation nötig (bzw. nur Sicherstellung dass alle Prozesse nur lesen, aber keiner schreibt)
 - Keine allgemeine Lösung, da sehr oft unvermeidbar!
- Hold and wait
 - Wenn ein Prozess auf eine Ressource wartet, darf er keine einzige andere besitzen.
 - » Eine Möglichkeit: Zu Beginn jeder Aktion müssen alle benötigten Ressourcen gleichzeitig reserviert werden. Er erhält entweder alle oder muss warten, bis alle gleichzeitig verfügbar.
 - Problem: Ressourcenauslastung schlecht, Starvation möglich



Deadlock - Vermeidung

- No preemption

→ Hauptproblem: Nicht alle Ressourcen können einfach entzogen werden (z. B. Datei, Drucker, ...).

- » Gute Lösung, falls es möglich ist!

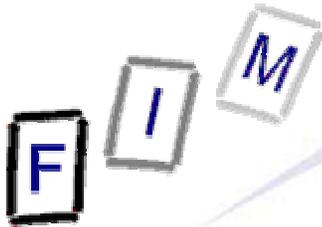
- » Beispiel: CPU (Scheduling), Speicher (≈Virtual Memory), ...

 - Zustandsspeicherung und Wechsel leicht und schnell möglich

 - Keine Daten gehen dabei verloren

→ Beispiel: Wenn ein Prozess Ressource nicht erhalten kann und warten muss, werden **alle** Ressourcen die er besitzt **automatisch** freigegeben und zur Liste der Benötigten hinzugefügt

- » Der Prozess muss dann warten, bis alle bisherigen und die neue verfügbar sind



Deadlock - Vermeidung

- Circular wait

→ Ressourcen ordnen: Nur höhere als die höchste besessene kann reserviert werden

» Praktische Bedeutung für Programmierer: Ressource immer in der gleichen Reihenfolge reservieren

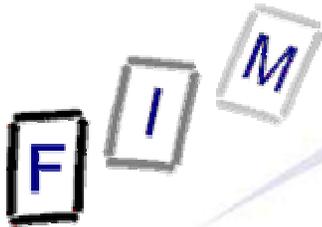
- Einfache Lösung und funktioniert garantiert!

- Nur möglich, bei einheitlichen Programmen (=1 Programmierer)

 - » Reine Konvention, keinerlei Prüfung möglich!

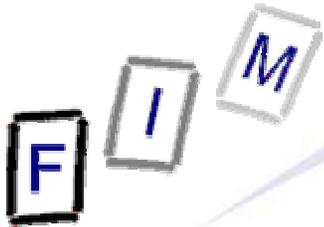
→ Problem: Wie stellt man eine solche Liste auf? Programme sind sehr stark eingeschränkt bei der Ressourcen-Reservierung!

» Automatisch von geringer Bedeutung!



Deadlock - Vermeidung

- Safe state: Ein "sicherer Zustand" liegt dann vor, wenn man die Ressourcen in zumindest **einer** Ordnung den Prozessen so zuteilen kann, dass schließlich **alle** terminieren können
 - Entweder die Ressourcen sind frei, oder sie werden von irgendeinem Prozess **vor** ihm in der Liste belegt
 - Dadurch wird circular wait vermieden!
- Liegt kein sicherer Zustand vor, **kann** es zu einem Deadlock kommen (**muss** aber nicht!)
- Ergebnis: Ein Prozess muss ev. warten, obwohl die gerade benötigte Ressource frei ist, falls dadurch ein Wechsel in einen unsicheren Zustand erfolgen würde!
 - **Schlechtere Ressourcenauslastung**



Deadlock - Vermeidung

- Ressourcen-Belegungs-Graph

→ 2 Arten von Knoten:

» Prozesse 

» Ressourcen 

→ 3 Arten von Kanten:

» $P \rightarrow R$: Prozess wartet auf Ressource

– Warte-Kante 

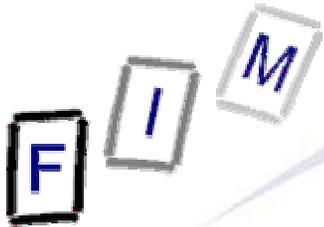
» $R \rightarrow P$: Ressource ist Prozess P zugeordnet

– Belegungs-Kante 

» $P \rightarrow R$: Prozess wird in Zukunft vielleicht auf Ressource zugreifen

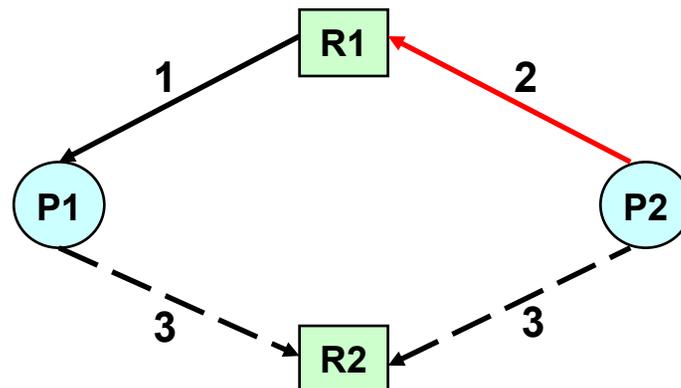
– Reservierungs-Kante 

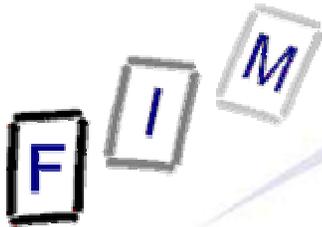
- Ein Prozess muss beim Start **alle** ev. benötigten Ressourcen anmelden: Reservierungs-Kanten eintragen!



Deadlock - Vermeidung

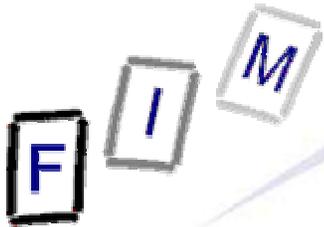
- Beispiel:
 - ① P1 besitzt R1
 - ② P2 wartet auf R1
 - ③ P1 und P2 belegen ev. später einmal R2 (oder auch nicht)
- Kein Kreis enthalten





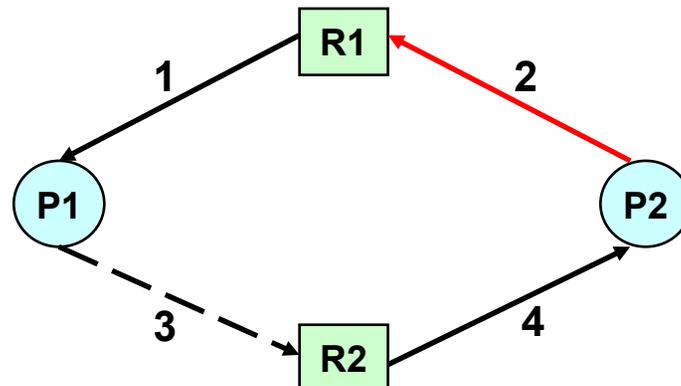
Deadlock - Vermeidung

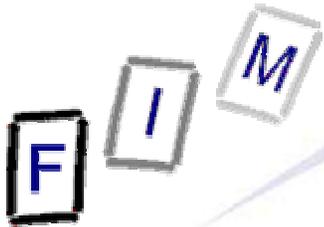
- Ressourcen-Belegungs-Graph
 - Befindet sich im Graph ein Kreis von durchgezogenen Kanten, so liegt eventuell ein Deadlock vor:
 - » Existiert die Ressource nur einmal, **ist** es ein Deadlock!
 - » Existiert die Ressource mehrfach, **kann** es ein Deadlock sein!
 - Resource zuteilen:
 - » Reservierungs- in Belegungs-Kante umwandeln
 - Resource freigegeben:
 - » Belegungs- zurück in Reservierungs-Kante umwandeln
- Sicherer Zustand: Kein Kreis im Graph
- Unsicherer Zustand:
 - Kreis (ev. mit Reservierungs-Kanten!) im Graph!
 - Nicht notwendig ein Deadlock!
 - Grund: Wir können nicht verhindern, dass ein Prozess auf eine angemeldete Ressource warten möchte: Dann wäre Deadlock!



Deadlock - Vermeidung

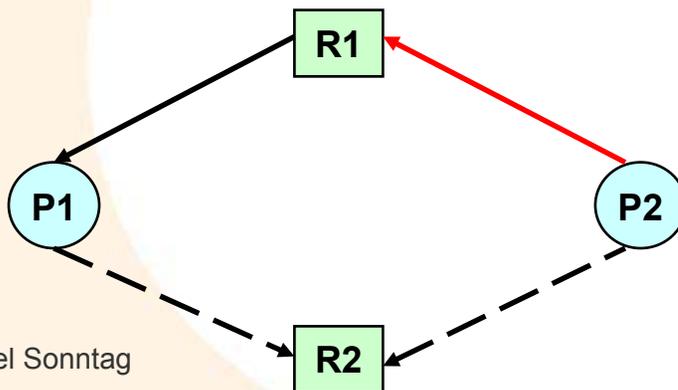
- Beispiel (wie oben):
 - ① P1 besitzt R1
 - ② P2 wartet auf R1
 - ③ P1 und P2 belegen ev. später einmal R2 (oder auch nicht)
 - ④ P2 möchte R2 jetzt belegen (NEU)
- Ergebnis (falls wir das durchführen würden):
 - Kreis im Graph (1→3→4→2→1), daher unsicherer Zustand!
 - Diese Anforderung darf **NICHT** erfüllt werden!





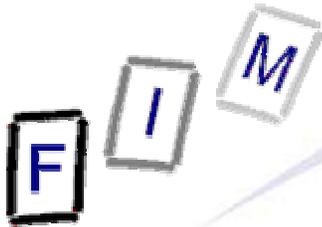
Deadlock - Erkennung

- Algorithmus für **einfach** vorkommende Ressourcen
 - Funktioniert nur für Ressourcen, welche nur ein **einziges Mal** vorkommen (nur **eine Instanz** existiert)
 - Ressourcen-Belegungs-Graph immer aktuell halten
 - » Wer wartet auf irgendeine Ressource, die ein Prozess besitzt?
 - Vereinfachung möglich: Ressourcen können weggelassen werden!
 - Regelmäßig im Graph nach Kreisen suchen
 - » Dies bedeutet eine Zeitkomplexität von $O(\text{Prozessanzahl}^2)$
 - Zusätzlicher Overhead für mitspeichern und regelm. prüfen



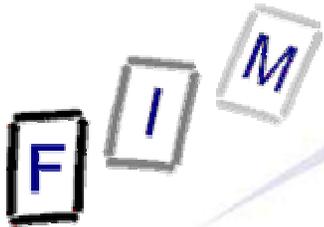
kann vereinfacht werden zu:



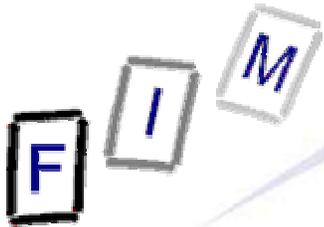


Deadlock - Erkennung

- Wie oft bzw. wann soll der Algorithmus zur Deadlock-Erkennung aufgerufen werden?
 - Wie oft wird ein Deadlock auftreten (Durchschnittlich)?
 - Wie viele Prozesse werden vom Deadlock betroffen sein?
- Eine Möglichkeit: Wenn bei einer Ressourcenanforderung gewartet werden muss
 - SEHR hoher Aufwand!
- Besser: Regelmäßig (alle x Sek.) / besondere Zeitpunkte
 - Geringe CPU-Auslastung
 - » Sehr gut bei starker Auslastung: Deadlocks ergeben Reduktion!
 - Problem: Hier können wir nicht mehr sagen, **welcher Prozess** die Ursache war!
 - » Aber ist das überhaupt von Bedeutung? → Warten im Kreis!



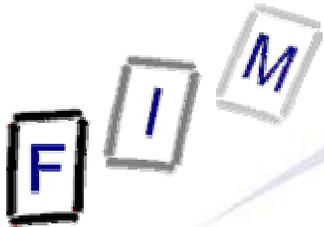
- Vier grundsätzliche Lösungsmöglichkeiten:
 - Benutzer verständigen und diesen das Problem lösen lassen
 - » Keine ideale Lösung, besonders nicht für Arbeitsplatzrechner (Benutzer wird sich kaum auskennen)
 - » Gute Lösung für große Systeme, falls Auswahl des "Opfers" schwierig/teuer ist
 - Neue Ressourcen zur Verfügung stellen
 - » Meist unpraktisch und/oder unmöglich (z. B. Datei verdoppeln)
 - Einen/Mehrere Prozesse beenden / zurückrollen
 - » Praktisch bedeutsam: Datenbank-Transaktionen
 - Eine/mehrere Ressourcen entziehen
 - » Siehe oben: Oft sehr schwierig/unmöglich
- Wenn wir schon Erkennung betreiben, dann können wir auch gleich die Behandlung übernehmen!



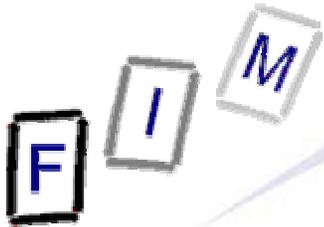
Deadlock – Behebung: Prozessabbruch

- Die Auswahl des/der Opfer(s) kann schwierig sein:
 - **Alle** Prozesse im Deadlock beenden: Nicht optimal!
 - Nur **einen** beenden und prüfen, ob noch immer Deadlock
 - » Aufwendig; wo anfangen?
 - Starvation?
 - Man bräuchte eine Kostenfunktion, bei welchem Prozess es am "billigsten" ist, ihn zu beenden
 - » Prozesspriorität
 - » Wie lang er bereits ausgeführt wurde
 - » Wie lang er noch laufen wird (→ Page Replacement: Orakel!)
 - » Wie weit er zurück mus (Neustart / Wiederaufsetzen möglich?)
 - » Welche Ressourcen er besitzt
 - » Welche Ressourcen er noch benötigen wird
 - » Welche anderen Prozesse noch terminiert werden müssen
 - » ...

Deadlock – Behebung: Prozessabbruch

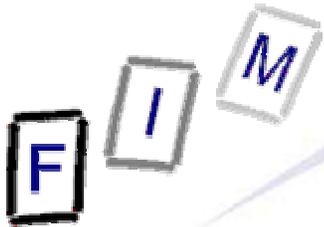


- Folgen:
 - Der Prozess muss abgebrochen werden
 - **Alle** Änderungen die der Prozess vorgenommen hat, müssen rückgängig gemacht werden
 - » Analog zum Abbruch einer Transaktion!
 - Der Prozess muss neu gestartet werden
- Problem: Wiederherstellung
 - Wie setzen wir Dateien, Drucker, etc. zurück?
 - Was ist mit Daten, die der Prozess über Netzwerke verschickt hat? Auch diese müssten "zurückgeholt" werden!
 - » Abhilfe: Verteilte Transaktionen



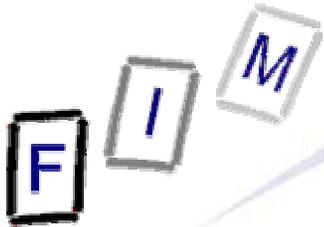
Deadlock – Behebung: Ressourcenentzug

- Anstatt Prozesse zu beenden, kann man auch einzelne Ressourcen entziehen und neu zuteilen
- Auch hier ist die Auswahl des/der Opfer(s) schwierig!
 - Wo anfangen?
 - Auch hier wieder eine Kostenfunktion nötig:
 - » Welche Prozesse sind betroffen?
 - » Welche Ressourcen und wie viele werden entzogen?
 - » Welche Kosten hat die Benutzung schon verursacht?
 - » Welche Kosten würde sie noch verursachen? (→ Orakel!)
 - Starvation?
 - Wie geht es mit Prozessen weiter, die Ressourcen verlieren?
 - » Sind die Programme darauf eingerichtet, irgendwo Ressourcen zu verlieren anstatt nur an bestimmten Stellen?



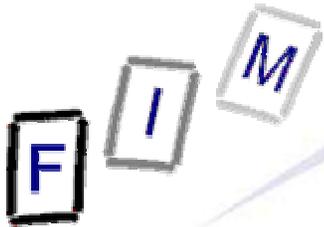
Deadlock – Behebung: Ressourcenentzug

- Folgen von Ressourcenentzug:
 - Muss der Prozess deswegen beendet werden (siehe vorher!)?
 - Wo kann der Prozess wiederaufsetzen?
 - » Muss der Prozess Wiederaufsetzpunkte bereitstellen bzw. dies explizit berücksichtigen, oder macht dies das Betriebssystem?
 - Transaktionen helfen hier nicht unbedingt!
 - » Nur, wenn jeweils der Gesamtzustand des Prozesses von der Transaktion umfasst ist!
 - Wie bringen wir die Ressourcen wieder in einen korrekten Zustand (= wie zum letzten Wiederaufsetzpunkt)?
 - » Drucker, Dateien, etc. sind hier schon wieder ein Problem!
 - Ev. Abhilfe: Dateisystem auf Datenbankbasis
 - » Siehe Windows Longhorn!



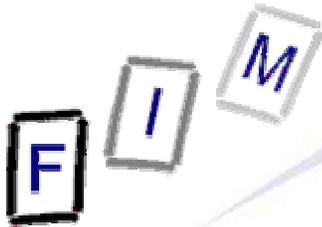
Deadlock Erkennung und Behebung

- Zusammenfassung: Erkennung ist nicht besonders schwer, aber die Behebung dafür umso mehr!
 - Sehr viele Probleme, für die es keine eindeutige oder auch nur allgemeingültige (=immer mögliche) Lösung gibt
- Daher: Deadlocks grundsätzlich vermeiden
 - Eine der verschiedenen Strategien verwenden
 - Funktioniert nur, wenn sich alle Prozesse daran halten oder das Betriebssystem dies durchsetzt
 - Für den Rest: Der Benutzer soll es erkennen und beheben!
- Ausnahme: Datenbanksysteme
 - Hier ist Abbruch einer Transaktion immer möglich
 - » Schon im Konzept enthalten
 - Programme müssen aber damit umgehen können!



Deadlocks und Java

- In Java gibt es keine Ansätze zur Deadlock-Behandlung
- Der Benutzer ist ausschließlich selbst verantwortlich
 - Bei einem Deadlock muss diesen der Anwender erkennen und auch selbst eingreifen!
 - » Außer: Programmierer hat zusätzliches selbst eingebaut
- Lösung:
 - Bei der Programmierung von Synchronisation immer auch an Deadlocks denken und entsprechend programmieren
 - "synchronized" Methode: Ungefährlich, solange keine "synchronized(obj) {}" Blöcke verwendet werden
 - » Wenn, dann immer in der gleichen Schachtelung verwenden
 - `Synchronized void A() { synchronized(b) { synchronized (c) { ... }}}`



Deadlocks und Java

- Lösung:

- Relativ "ungefährlich": Mutex (lock & unlock)

- » Solange immer paarweise, d. h. kein "return" dazwischen!

- » Auch hier wieder auf die selbe Reihenfolge achten, sobald mehrere Locks verwendet werden!

- Semaphor: Je nach Verwendung; Vorsicht!

- Wait & notify: **Sehr gefährlich**

- » Werden meist getrennt eingesetzt (d.h. "return" dazwischen)

- » Zufällig notify vor wait: Deadlock!

- Generell:

- Wenn unterstützt, versuchen alle Ressourcen auf einmal (in einer **einzig** Anweisung!) zu reservieren

- » Aber nur bei kleinen/kurzen Codestücken sinnvoll

- Ansonsten feste Reihenfolge beachten