Referat "Deadlocks" im Proseminar "Computer Science Unplugged"
Dozent Prof. M. Hofmann
Referent Margit Jahn, 37111744
Datum 10.12.2004

Deadlocks

Referat "Deadlocks" im Proseminar "Computer Science Unplugged"
Dozent Prof. M. Hofmann
Referent Margit Jahn, 37111744
Datum 10.12.2004

Deadlocks

1.Grundlagen

1.1. Einführung

Eine Menge von Prozessen ist in einem Deadlock, wenn jeder Prozess auf eine andere Ressource wartet, die schon von einem anderen Prozess belegt ist. Ein Programmbeispiel soll diese Definitionsidee untermauern:

Void process_A(void) {
down (&resource_1);

Unterbrechung
⇨ Prozess A muss CPU wegen Ablauf der Zeitscheibe zurück-geben
⇨ Prozess B gelangt in die CPU

Void process_B(void) {
down (&resource_2);
down (&resource_1);

Unterbrechung
⇨ Ressource 1 belegt
⇨ Prozess gibt CPU freiwillig zurück
⇨ Prozess A wird in die CPU geschedult

down (&resource_1);

Unterbrechung
⇨ Ressource 2 belegt
⇨ Prozess gibt CPU freiwillig zurück
⇨ Prozess B wird in die CPU geschedult

Die folgenden Programmteile kommen nicht mehr zur Ausführung:

use_resource_1();
use_resource_2();
up (&resource_1);
up (&resource_2) }

use_resource_2();
use_resource_1();
up (&resource_2);
up (&resource_1) }

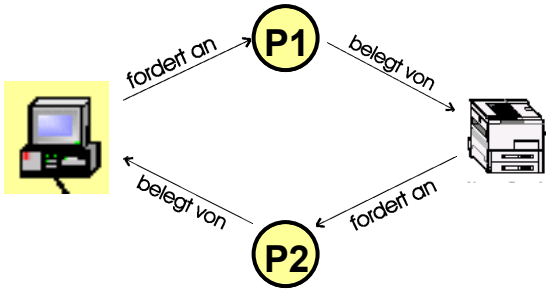
Wie wir an diesem Beispiel gut sehen können, entsteht ein Deadlock immer dann, wenn zwei Prozesse gegenseitig aufeinander wegen einer Ressource zyklisch Warten.
Aber was ist eigentlich eine Ressource?

1.2. Definition des Ressourcen-Begriffes

Nach Wikipedia wird der Begriff Ressourcen folgendermaßen definiert: „Ressourcen sind Systemelemente, die von Prozessen zur korrekten Ausführung benötigt werden.“
Man unterscheidet Softwarekomponenten wie beispielsweise Daten oder Programmkonstrukte und Hardwarekomponenten wie Drucker, Monitor, etc. Ressourcen können in Klassen zusammengefasst werden, wie dies in einem Büro z.B. in einem Druckerpool geschieht.
Ihre Nutzung ist exklusiv, d.h. sie werden ausschließlich von einem Prozess genutzt oder mehrfach wie dies bei Spoolern geschieht.
Hat ein Prozess eine Ressource so kann sie ihm entweder entzogen werden (preemtive) oder sie darf ihm nicht entzogen werden, sondern kann nur freiwillig von ihm zurückgegeben werden (non-preemptive). Diese Eigenschaften bezeichnet man auch als unterbrechbar und ununterbrechbar.
Deadlocks können <u>nur</u> bei ununterbrechbaren Ressourcentypen entstehen.

1.3. Definition des Deadlock-Begriffes

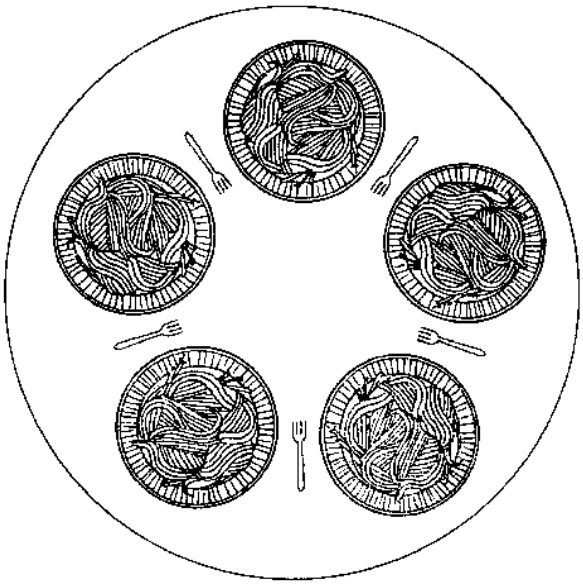
Ein Deadlock - auch Verklemmung genannt – ist in der Informatik ein Zustand von Prozessen, bei dem mindestens zwei Prozesse untereinander auf Ressourcen warten, die dem jeweils anderen Prozess zugeteilt sind.
Beispielsweise ist dem Prozess 1 der Bildschirm zugeteilt worden, wobei er gleichzeitig den Drucker benötigt. Auf der Gegenseite ist der Drucker dem Prozess 2 zugeteilt, der wiederum den Bildschirm fordert:



1.4. Das „Dining-Philosopher-Problem“

Eines der bekanntesten Beispiele für einen Deadlock ist das „Dining Philosopher“- Problem von Dijkstra :

An einem Tisch sitzen 5 Philosophen de dinieren wollen. Zwischen je zwei Tellern befindet sich eine Gabel . Um nun Essen zu können muss zuerst die rechte und dann die linke Gabel aufgehoben werden.
Auf Grund der Prämisse, dass erst nach erhalt der linken die rechte Gabel aufgehoben werden darf wird es zwangsläufig zu einem Deadlock kommen, da die zweite Gabel immer vom rechts dinierenden Philosophen aufgenommen wurde.
Wie lässt sich nun das Problem lösen ?
1)Eine Möglichkeit besteht in der folgenden Abänderung der Prämissen : Einer der Philosophen hebt zuerst die rechte Gabel auf ehe er die linke nimmt. Dadurch wird die zyklische Wartebedingung aufgehoben.
2) Manchen Philosophen darf die Gabel aus der Hand genommen werden.
3) Es werden zwei Gabeln zwischen die Teller gelegt.
4) Die Philosophen dürfen mit nur einer Gabel essen.



1.5.Voraussetzungen nach Coffman

Die für das „Dining Philosopher“ - Problem definierten Vorraussetzungen wurden 1971 von Coffman als allgemeine Vorraussetzungen für Deadlocks folgendermaßen definiert :

1) Mutual Exclusion (Wechselseitiger Ausschluss)
Nur eine Instanz kann einen Prozess gleichzeitig belegen

2) Hold and Wait

Ein Prozess mit mindestens einer Ressource wartet auf eine Ressource die von einem anderen Prozess gehalten wird.

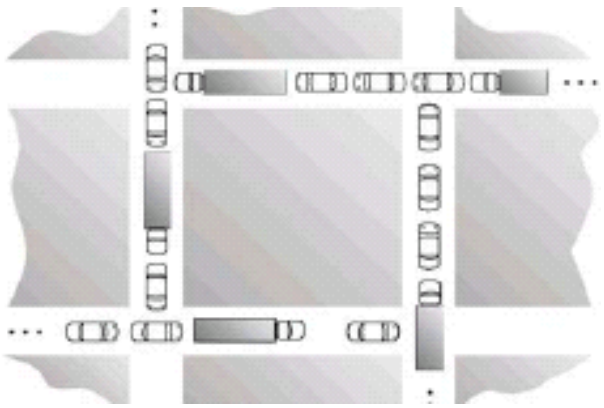
3) Non-preemptive Algorithmus (Ununterbrechbarkeit)

Gehaltene Ressourcen können nicht nur freiwillig aufgegeben werden

4) Circular Waiting (Zyklisches Warten)

Jeder Prozess wartet in einer Kette auf den nächsten.

Nur wenn alle diese vier Voraussetzungen erfüllt sind, ist ein Deadlock aufgetreten.



2. Umgang mit Deadlocks

2.1.Vogel-Strauß-Allgorithmus

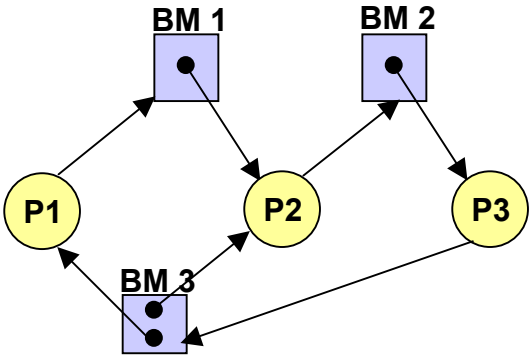
Der einfachste Ansatz beim Umgang mit Deadlocks ist der Vogel-Strauß-Algorithmus, d.h. die Ignorierung des Problems. Dieser Ansatz, der für Mathematiker indiskutabel ist, wird dennoch unter Informatikern und Ingeneuren in Erwägung gezogen.
Die Gründe dafür sind einfach:
Die Wahrscheinlichkeit für ein Deadlock ist wesentlich geringer als die für Compiler- oder Betriebssystemfehler.
Die Behandlung von Deadlocks jedoch führt zu einer nicht unbeträchtlichen Geschwindigkeitseinbuße und ist daher wesentlich störender als die meisten Deadlocks, die oft für den Benutzer unerkannt bleiben.
Außerdem müssen Prozesse für die Behandlung von Deadlocks stark eingeschränkt werden.
Es handelt sich also beim Umgang mit Deadlocks, um die Frage dem Komfort oder der Korrektheit den Vorzug zu geben.
Daher wird in den meisten Betriebssystemen der Vogelstrauß-Algorithmus angewandt.

2.2. Deadlock - Erkennung

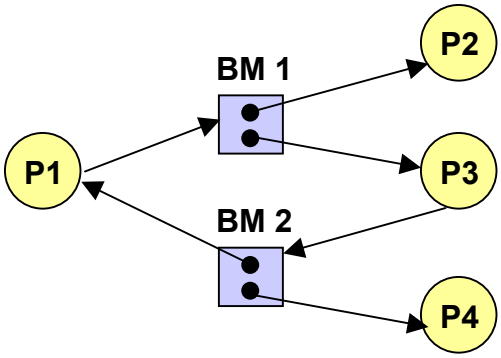
Grundsätzlich unterscheidet man drei Algorithmen zur Erkennung von Deadlocks den Ressourcen-Allokations-Graph, den Wait-for-Graph und den matrixbasierten Algorithmus. Der erste Algorithmus erkennt hierzu die zyklischen Wartebedingungen von Prozessen durch Darstellung von allen Ressourcen und Prozessen in einem Graphen. Der zweite stellt eine Vereinfachung des ersten dar. Wenn jede Ressource ausschließlich einmal vorhanden ist, so kann nämlich die Abhängigkeit der Prozesse dargestellt werden ohne Angabe, um welche Ressource es sich handelt.

2.2.1. Ressourcenallokationsgraph

Der Ressourcen-Allokations-Graph arbeitet mit Hilfe der oben erklärten Vorraussetzungen von Coffman. Besonders nutzt er die Vorraussetzung des zyklischen warten. In einem gerichteten Graphen werden alle Ressourcenklassen (R) und Prozesse (P) angetragen. Entsteht hierbei ein Zyklus, der nicht durch eine zusätzliche Ressourceninstanz gelöst werden kann, so ist ein Deadlock vorhanden.



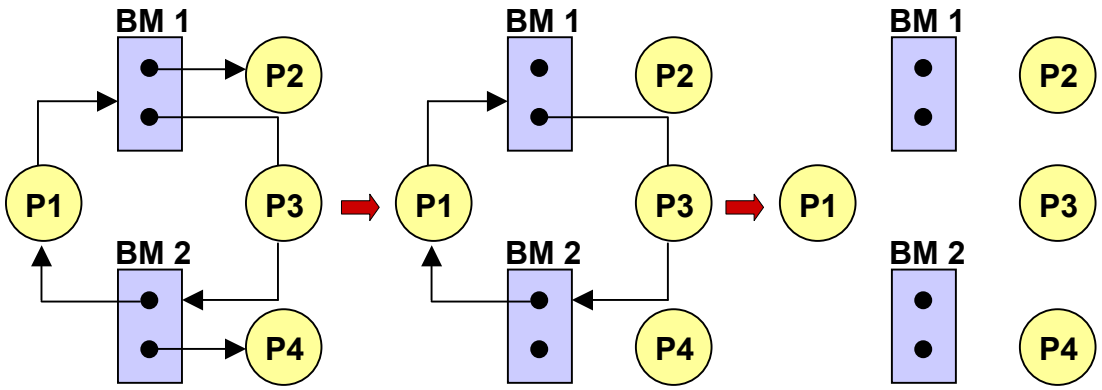
(a) Deadlocksituation



(b) per Ressourceninstanz lösbarer Zyklus

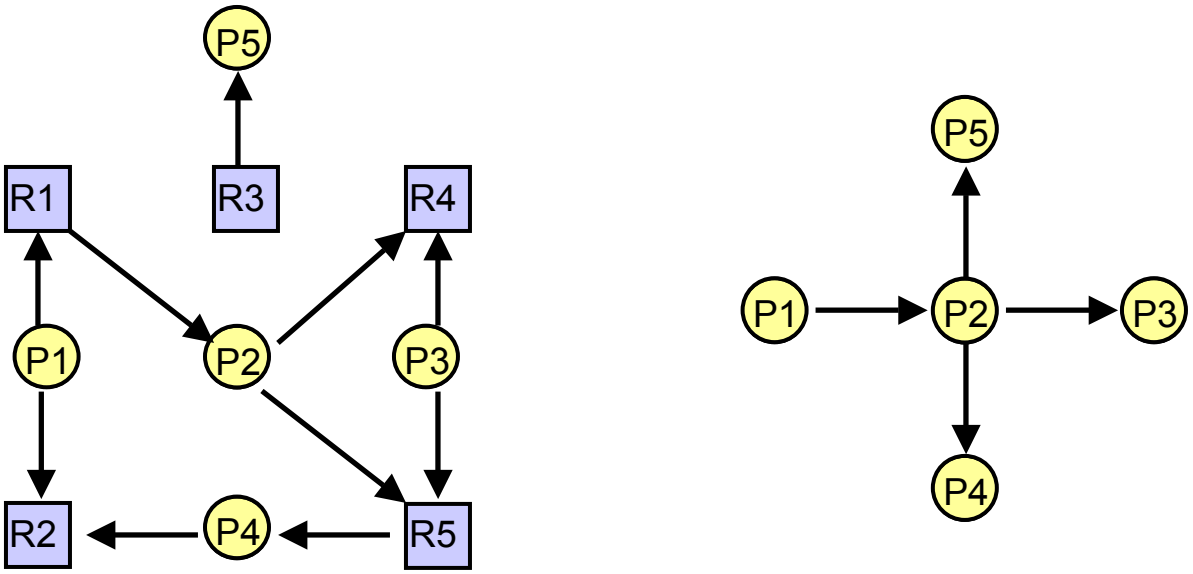
2.2.1.1.Reduzierung eines Resourcen-Allokations-Graphen

Wie wir aus Beispiel (b) ersehen können Ressourcen-Allokations-Graphen reduziert werden, um so scheinbare Deadlocksituationen zu lösen.
Wir lassen Prozess 2 und 4 abarbeiten. Es wird deutlich, dass hier keine Deadlocksituation vorliegt, denn alle Prozesse lassen sich durch die folgenden Schritte aus dem Betriebsmittelgraphen reduzieren:



2.2.2. Wait-for Graph

Grundsätzlich unterscheidet sich der Wait-for-Graph von dem eben kennen gelernten Ressourcenallokationsgraphen dadurch das jede der am Deadlock beteiligten Ressourcen nur einmal vorhanden ist. Dadurch entsteht eine Abhängigkeit zwischen dem Prozess, der eine Ressource hält und demjenigen der diese Ressource benötigt. Es reicht daher aus diese Abhängigkeit als Kanten zwischen den Prozessen darzustellen.



2.2.3. Matrixbasierter Algorithmus

Da der Ressourcenallokationsgraphen sehr unübersichtlich ist, wird zur Erkennung bei mehreren Ressourcen häufig die Darstellung als Matrix-Algorithmus gewählt.
Ebenfalls werden die Ressourcen in Klassen m mit ihrer Anzahl E_i eingeteilt. Ein Ressourcenvektor E gibt die maximale Anzahl der einzelnen Ressourcen an. Ein Ressourcenrestvektor A enthält die Anzahl der noch zu vergebenden Instanzen A_i. In einer Belegungsmatrix C wird angegeben welche Ressourcen die einzelnen Prozesse schon halten Die Anforderungsmatrix R gibt Auskunft über die noch benötigten Ressourcen. Da alle Ressourcen als frei (in A) oder gebunden (in C) vorkommen ergibt sich die folgende Invariante :

$$\sum_{i=1}^{n} Cij + Aj = Ej$$

Es ergibt sich somit folgende Ausgangssituation :

$\mathbf{E} = (E_1, E_2, E_3, E_4, \ldots, E_m)$

(a) Ressourcenvektor

$\mathbf{A} = (A_1, A_2, A_3, A_4, \ldots, A_m)$

(b) Anforderungsvektor

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & C_{14} & \ldots & C_{1m} \\ C_{21} & C_{22} & C_{23} & C_{24} & \ldots & C_{2m} \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ C_{n1} & C_{n2} & C_{n3} & C_{n4} & \ldots & C_{nm} \end{bmatrix}$$

(c) Belegungsmatrix

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & R_{14} & \ldots & R_{1m} \\ R_{21} & R_{22} & R_{23} & R_{24} & \ldots & R_{2m} \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ R_{n1} & R_{n2} & R_{n3} & R_{n4} & \ldots & R_{nm} \end{bmatrix}$$

(d) Anforderungsmatrix

Zur Verdeutlichung hier ein konkretes Beispiel :

	Bandgerät	Plotter	Scanner	CD-ROM
E =	4	2	3	1

(a) Ressourcenvektor

	Bandgerät	Plotter	Scanner	CD-ROM
A =	2	1	0	0

(b) Anforderungsvektor

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 2 & 1 & 2 & 0 \end{bmatrix}$$

(c) Belegungsmatrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

(d) Anforderungsmatrix

2.3. Behebung von Deadlocks

2.3.1. Behebung durch temporären Ressourcenentzug

Ein im Deadlock befindlicher Prozess erhält die benötigte Ressource, indem einem anderen Prozess zeitweise diese Ressource entzogen wird.

2.3.2. Behebung durch Prozessabbruch

Der Deadlock wird beseitigt, indem ein verklemmter Prozess oder einen Prozess, der - ohne direkt am Deadlock beteiligt zu sein - das benötigtes Betriebsmittel besitzt, abgebrochen wird. Diese Methode ist jedoch sehr rabiat und deshalb nicht immer ratsam.

2.3.3. Checkpoint-Verfahren

In regelmäßigen Abständen wird der Zustand des Prozesses sowie die Informationen über Ressourcen gesichert als sog. Checkpoint. Entsteht ein Deadlock wird ein Prozess mit den benötigten Ressourcen auf einen Checkpoint ohne Ressource zurückgesetzt.

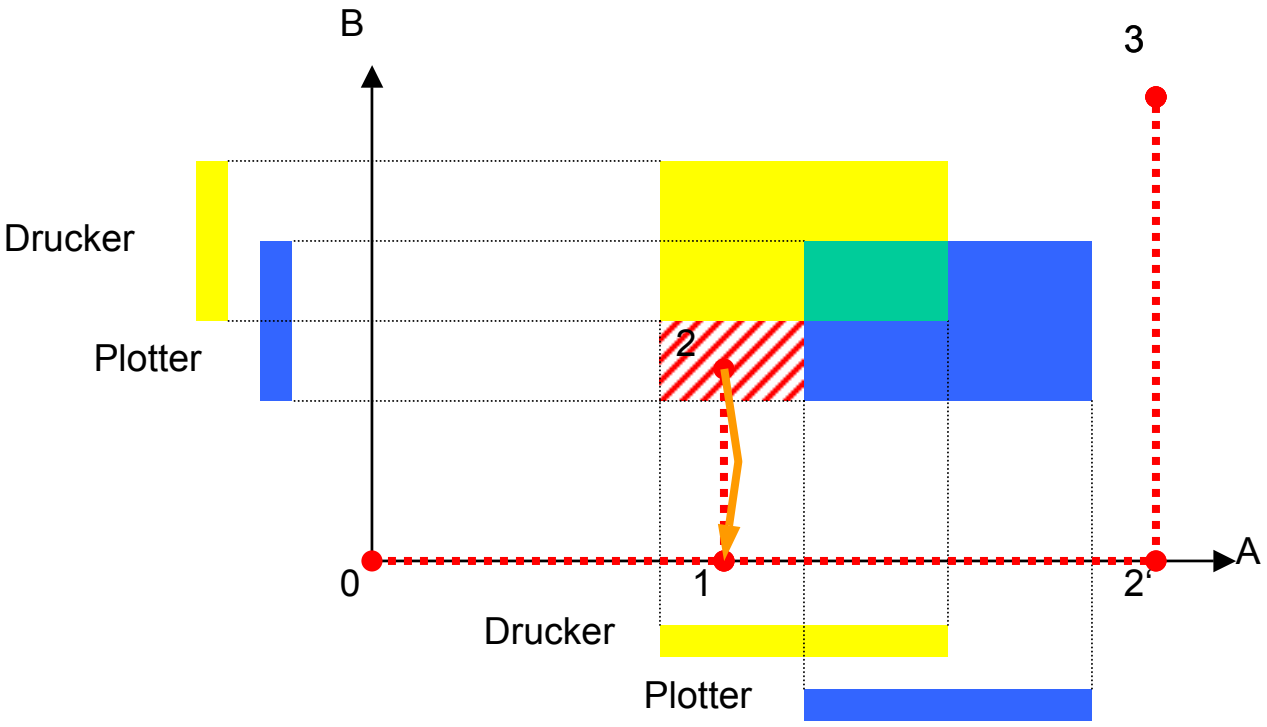
2.3.4.Journaling

Das Journalingverfahren ist eine Art des Rollbackverfahrens. Im Gegensatz zum Checkpoint-Vorgehen werden nur noch die Änderungen zwischen zwei Checkpoints gesichert. Dies spart Speicherplatz, erhöht jedoch den Verwaltungsaufwand.

2.3. Verhinderung von Deadlocks

2.3.1. Das Prozessfortschrittsdiagramm

Befinden sich nur zwei Prozesse im Deadlock so kann man mit Kenntnis des Programmablaufs , d.h. mit dem Wissen, wann welcher Prozess welche Ressource benötigt, einen Deadlock verhindern. Dazu trägt man in ein Koordinatensystem an die x- Achse den Zeitablauf von Prozess 1 an die y-Achse den Zeitablauf von Prozess 2 an. Die Bereiche, in denen beide Prozesse gleichzeitig auf die selbe Ressource zugreifen, werden als unmögliche Bereiche gekennzeichnet. Der Schedulingalgorithmus, der als Linie dargestellt wird, ist dann unsicher, wenn jeder der Prozesse eine Ressource hält, die der andere kurze Zeit später anfordert.



Befindet sich ein Schedulingalgorithmus in einem solchen unsicheren Bereich kann nur noch ein Rollbackverfahren (von 2 nach 1) helfen einen Deadlock zu verhindern.

2.4.2. Sichere - Unsichere Zustände

Wie wir bereits am Prozessfortschrittsdiagramm gesehen haben unterscheiden wir sichere und unsichere Zustände.
Dabei definieren wir einen sicheren Zustand, wenn kein Deadlock vorliegt und es eine Schedulingreihenfolge gibt, die nicht zum Deadlock führt, selbst wenn die maximale Anzahl an Ressourcen angefordert wurde.

Beispiel für einen sicheren Zustande:

	nutzt	max.
A	3	9
B	2	4
	2	7

(a) frei : 3

	nutzt	max.
A	3	9
B	4	4
	2	7

(b) frei : 1

	nutzt	max.
A	3	9
B	0	-
	2	7

(c) frei : 5

	nutzt	max.
A	3	9
B	0	-
	7	7

(d) frei : 0

	nutzt	max.
A	3	9
B	0	-
	0	-

(e) frei : 7

Gibt es eine Schedulingreihenfolge, die zum Deadlock führt so befinden wir uns in einem unsicheren Zustand.

	nutzt	max.
A	3	9
B	2	4
C	2	7

	nutzt	max.
A	4	9
B	2	4
C	2	7

Unsicherer Zustand

	nutzt	max.
A	4	9
B	4	4
C	2	7

	nutzt	max.
A	4	9
B	-	
C	2	7

Deadlock !

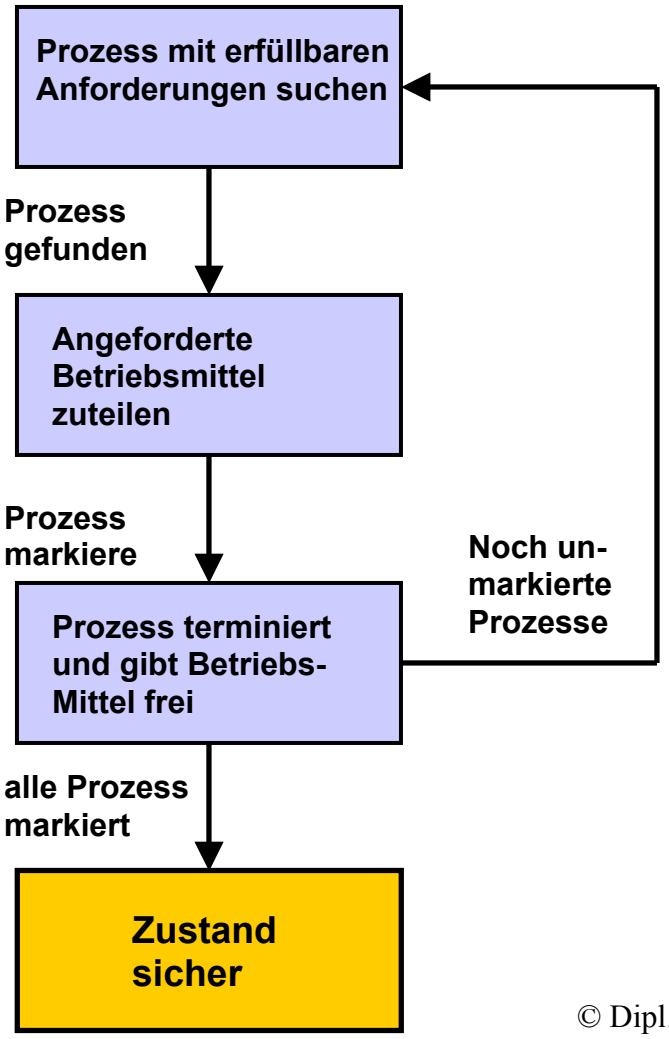
2.4.3. Bankier-Algorithmus

2.4.3.1. Datenstrukturen

Bei der Sicherheitsüberprüfung eines Systemzustandes prüft der Bankier-Algorithmus, ob mit den verfügbaren Betriebsmitteln alle aktuellen Prozesse ihre Arbeit beenden können. Dafür verwendet er den oben vorgestellten Matrix-basierten Algorithmus mit Belegungsmatrix, Anforderungsmatrix, Ressourcenvektor und Ressourcenrestvektor.

2.4.3.2 Schritte des Bankieralgorithmus

Die Abarbeitung des Bankier-Algorithmus findet in 3 Phasen rekursiv statt..
Phase 1
Suche nach Prozess, mit erfüllbarer Restanforderung in der Anforderungsmatrix
Setzen der Einträge in der Anforderungsmatrix
auf 0.
Phase 2
Auswahl und Abgleich des Betriebsmittelrestvektor und der Belegungsmatrix.
Phase 3
Freigabe der Betriebsmittel, d.h. Addition der Betriebsmittel zum Betriebsmittelrestvektor und Setzen der Einträge in der Belegungsmatrix auf 0.



2.5.Deadlockvermeidung

Generell kann ein Deadlock vermieden werden wenn eine der vier Vorraussetzungen nach Coffman ausgeschlossen werden kann, d.h. eine der vier Vorraussetzungen muss ausgeschlossen werden.

2.5.1. Ausschluss der Mutual-Exclusion-Bedingung

Um dieses Prinzip anzuwenden, wird das Problem der Ressourcenexklusivität umgangen, indem ein Spooler alle Aufträge zwischenspeichert und eine Warteliste erstellt. Erst nach Eingang des kompletten Datensatzes wird der Auftrag nacheinander an die Ressource weitergegeben. Der doppelte Zugriff auf die Ressource wird somit ausgeschlossen, da nur der Spooler mit der Ressource im Kontakt steht. Ein Beispiel hierfür wäre ein Druckerspooler. Dieses Prinzip ist sehr gut, kann jedoch nicht für alle Ressourcen wie beispielsweise bei der Speicherplatzreservierung Anwendung finden.

2.5.2. Ausschluss der Hold & Wait - Bedingung

Um das Halten und Warten von Ressourcen zu umgehen, müssen alle Ressourcen vor der Ausführung des Prozesses reserviert werden - siehe hierzu auch Bankier-Algorithmus – oder Ressourcen müssen wieder freigegeben werden bis alle reservierbar sind.
Der Nachteil bei dieser Methode ist jedoch ein großer Verwaltungsaufwand sowohl für die Reservierung der Ressourcen als auch für eventuelle Fehlermeldungen, falls nicht alle Ressourcen zur Verfügung stehen. Außerdem muss der Ressourcenbedarf im vorhinein bekannt sein. Schließlich kann es zu langen Wartezeiten auf die Ressourcen kommen. Dennoch findet das Prinzip Anwendung in Betriebssystemen von Großrechnern und in Datenbanksystemen, wo das Prinzip Anwendung im Two-Phase-Locking findet.

2.5.3. Ausschluss der Non-Preemptive-Bedingung

Ein Deadlock kann verhindert werden, indem Prozesse unterbrochen werden dürfen, um damit die gehaltenen Ressourcen freizugeben. Die dabei zu unterbrechenden Prozesse sind entweder direkt an Deadlock beteiligt oder sie halten Ressourcen die ein im Deadlock befindlicher Prozess benötigt.
Leider ist dieses Prinzip nur unzureichend anwendbar, denn die Entziehung von Ressourcen ist oft schwierig und sehr aufwendig oder sogar unmöglich. So kann der Vorgang des Brennens einer CD nicht einfach vom Betriebssystem unterbrochen werden weil ein weiterer Prozess auch Daten auf eine CD schreiben will. Dennoch findet dieses Prinzip Anwendung in Echtzeitsystemen.

2.5.4. Ausschluss der Circular-Wait-Bedingung

Es gibt zwei Vorgehensweisen, um das Vorkommen zyklischen Wartens zu verhindern. Vorgehensweise 1. erlaubt einem Prozess maximal eine Ressource zu halten. Will er eine weitere so muss die erste zurückgegeben werden. Bei der zweiten Vorgehensweise werden die Ressourcen hierarchisch durchnummeriert. Ein Prozess muss Ressourcen in dieser hierarchischen Weise anfordern, d.h. er kann nur Ressourcen mit niedrigerer beziehungsweise höherer Nummer, je nachdem welche Hierarchiestruktur vom Betriebssystem vorgegeben wird, anfordern.
Das Prinzip scheint auf den ersten Blick recht gut, jedoch wird es schwierig wenn nicht sogar unmöglich sein eine geeignete Hierarchie für die unterschiedlichen Ressourcen zu finden die allen Anforderungen gerecht wird. Außerdem ist eine solche Struktur mit einem enormen Verwaltungsaufwand verbunden. Prozesse, die Ressourcen aus dem unteren Bereich der Hierarchie anfordern können verhungern.

3. Forschung über Deadlocks

Besonders in den frühen Jahren der Betriebssysteme war das Thema „Deadlock“ sehr beliebt. Heute ist es jedoch weitreichend ausgeschöpft und die Forschung beschäftigt sich vorwiegend mit Deadlocks in verteilten Systemen.
Auf der Betriebsystemebene wird vorzugsweise das Vogelstraußprinzip angewandt, da größere Arbeits- und Hauptspeicher, sowie schnellere Busse und CPUs die Wahrscheinlichkeit des Auftretens eines Deadlocks stark gemindert haben. Außerdem ziehen die Ingeneure den Komfort eines schnelleren Rechners der absoluten Korrektheit von Programmen, wie sie Mathematiker fordern würden, vor.
Programmiersprachenkonstrukte wie beispielsweise in Java sleep und notify all, mindern zusätzlich die Wahrscheinlichkeit von Deadlocks .
Vor allem im Bereich Datenbanksysteme findet das Prinzip des Journaling Anwendung.
Gerade in verteilten Systemen jedoch wird die Thematik von Deadlocks wieder interessant und daher sind die eben vorgestellten Konzepte mit Sicherheit noch nicht in ihrer Allgemeinheit überholt .

4. Literaturnachweis

Als Grundlage für diese Ausarbeitung diente folgende Literatur :

- A.S. Tanenbaum, „Moderne Betriebssysteme“,
  Prentice Hall, 2.überarbeitete Auflage, 2002
- E.G.Coffman;P.J. Denning ,“Operating Systems Theory”,
  London, Prentice Hall International, 1973
- J.L. Peterson; A.Silberschatz „Operating Systems Concepts“,
  2.Auflage, Bonn, Reading;Mass., Addison-Wesley Verlag 1983
- H. J. Siegert; U. Baumgarten: *Betriebssysteme - Eine Einführung*.
  4.überarbeitete Auflage, München; Wien: R. Oldenbourg Verlag, 1998.
- Gabi Dreo-Rodosek, Skript zur Vorlesung Informatik III, WS 2002/03
  Ludwig-Maximilian-Universität, München

Bilder :

www-sst.informatik.tu-cottbus.de
www.epfl.ch/teaching/programming00_01/pictures/dp.gif