

Andrew S. Tanenbaum

Moderne Betriebssysteme

2. überarbeitete Auflage

übersetzt von Prof. Dr. Uwe Baumgarten
Technische Universität München



ein Imprint der Pearson Education Deutschland GmbH

Deadlocks

Computersysteme enthalten zahlreiche Ressourcen, die nur ein Prozess zur gleichen Zeit benutzen kann. Typische Beispiele sind Drucker, Bandlaufwerke oder Einträge in Systemtabellen. Wenn zwei Prozesse gleichzeitig denselben Drucker benutzen, wird er wahrscheinlich nur Unbrauchbares drucken. Wenn zwei Prozesse gleichzeitig in denselben Eintrag einer Dateizuordnungstabelle schreiben, wird das Dateisystem inkonsistent. Folglich müssen alle Betriebssysteme die Fähigkeit haben, Ressourcen (zeitweise) einem einzigen Prozess zuzuordnen.

Viele Anwendungen verlangen nicht nur auf eine Ressource den alleinigen Zugriff, sondern gleich auf mehrere. Nehmen wir zum Beispiel an, zwei Prozesse wollen beide ein Dokument einscannen und es dann auf eine CD brennen. Prozess A reserviert sich zunächst den Scanner. Prozess B läuft etwas anders ab und verlangt zunächst den CD-Brenner. Als Nächstes versucht A, sich den CD-Brenner zu reservieren, was aber fehlschlägt, weil B ihn noch nicht freigegeben hat. Dummerweise verlangt B nun den Scanner, statt den CD-Brenner freizugeben. Zu diesem Zeitpunkt sind beide Prozesse blockiert und daran wird sich auch so schnell nichts ändern. Eine solche Situation wird **Deadlock** genannt.

Deadlocks können auch über mehrere Rechner verteilt sein, beispielsweise im lokalen Netz einer Firma, in dem Geräte wie Drucker und Scanner von jedem Benutzer an jedem Rechner benutzt werden können. Wenn diese Ressourcen von entfernten Rechnern (also von einem beliebigen Arbeitsplatz aus) reserviert werden können, kann dies zu derselben Art von Deadlocks führen, wie sie eben beschrieben wurden.

In komplizierteren Situationen können sich Deadlocks ergeben, an denen drei, vier oder noch mehr Geräte und Benutzer beteiligt sind.

Deadlocks können nicht nur durch die Reservierung von E/A-Geräten¹, sondern auch in vielen anderen Situationen entstehen. In einer Datenbankanwendung könnte ein Programm zum Beispiel die Datensätze, an denen es arbeitet, sperren, um Überschneidungen bei zeitkritischen Abläufen zu vermeiden. Wenn Prozess A den Datensatz R1 und Prozess B den Datensatz R2 sperrt und anschließend jeder Prozess versucht, den Datensatz des anderen zu sperren, entsteht ebenfalls ein Deadlock. Deadlocks können also sowohl durch Hardware- als auch durch Softwareressourcen entstehen.

In diesem Kapitel werden wir das Thema Deadlocks genauer untersuchen. Wir werden sehen, wie sie entstehen, und einige Möglichkeiten untersuchen, sie zu vermeiden oder zu verhindern. Obwohl wir uns hier auf Deadlocks im Umfeld von Betriebssystemen konzentrieren, kommen Deadlocks auch in Datenbanksystemen und in vielen anderen Gebieten der Informatik vor. Der Stoff in diesem Kapitel lässt sich also auf ein weites Feld von Mehrprozesssystemen anwenden.

¹ E/A wird zwecks Erleichterung des Leseflusses im folgenden als Abkürzung für Ein-/Ausgabe benutzt. Im englischen wird dazu auch I/O für In-/Output gebraucht.

Über Deadlocks existiert bereits eine Menge Literatur. Einen Überblick geben zwei Bibliographien, die in der *Operating Systems Review* erschienen sind (Newton, 1979 und Zobel, 1983). Diese Artikel sind zwar alt, aber immer noch nützlich, da die meiste Forschung über Deadlocks vor 1980 unternommen wurde.

3.1 Ressourcen

Deadlocks entstehen, wenn Prozessen das alleinige Zugriffsrecht auf Geräte, Dateien oder Ähnliches erteilt wird. Um Deadlocks so allgemein wie möglich zu behandeln, nennen wir die reservierten Objekte **Ressourcen**. Eine Ressource kann ein physisches Gerät sein (z. B. ein Laufwerk), aber auch eine Informationseinheit (z. B. ein gesperrter Datensatz). Normalerweise hat ein Computer viele verschiedene Ressourcen, die reserviert werden können. Von einigen Ressourcen können mehrere identische Instanzen existieren, z. B. drei Diskettenlaufwerke. Wenn mehrere Kopien einer Ressource zur Verfügung stehen, kann jede von ihnen benutzt werden, um eine Anfrage nach dieser Ressource zu erfüllen.

Kurz gesagt ist eine Ressource etwas, das nur von genau einem Prozess gleichzeitig benutzt werden kann.

3.1.1 Unterbrechbare und ununterbrechbare Ressourcen

Es gibt zwei Arten von Ressourcen: unterbrechbare und ununterbrechbare. Eine **unterbrechbare Ressource**² kann dem Prozess, der sie besitzt, ohne unerfreuliche Nebenwirkungen entzogen werden. Ein Beispiel hierfür ist Arbeitsspeicher. Stellen wir uns ein System vor, das über 32 MB Speicher und einen Drucker verfügt. Auf dem System laufen zwei 32 MB große Prozesse, die beide etwas ausdrucken wollen. Prozess *A* reserviert sich den Drucker und beginnt mit der Berechnung der Daten, die ausgedruckt werden sollen. Bevor er damit fertig ist, überschreitet er sein Rechenzeitquantum und wird ausgelagert. Jetzt wird Prozess *B* ausgeführt und versucht ohne Erfolg, den Drucker zu reservieren. Wir befinden uns nun in einer möglichen Deadlock-Situation, da *A* den Drucker und *B* den Speicher besitzt. Keiner der beiden kann ohne die Ressource des anderen weitermachen. Zum Glück ist es möglich, *B* den Speicher zu entziehen (präemptiv), indem man *B* auslagert und *A* einlagert. Jetzt kann *A* seine Berechnung beenden, die Daten ausdrucken und dann den Drucker freigeben. So entsteht kein Deadlock.

Im Gegensatz dazu kann eine **ununterbrechbare Ressource**³ ihrem gegenwärtigen Besitzer nicht entzogen werden, ohne dass dessen Ausführung fehlschlägt. Einem Prozess einen CD-Brenner während des Schreibvorgangs zu entziehen, führt zu einer zerstörten CD. CD-Brenner sind also ununterbrechbare Ressourcen, d.h., sie können einem Prozess nicht jederzeit entzogen werden.

Generell haben Deadlocks immer mit ununterbrechbaren Ressourcen zu tun. Mögliche Deadlocks, an denen unterbrechbare Ressourcen beteiligt sind, können normalerweise aufgelöst werden, indem man unterbrechbare Ressourcen neu zuteilt, also werden wir uns hier auf die ununterbrechbaren Ressourcen konzentrieren.

² engl.: preemptable resource

³ engl.: nonpreemptable resource

Die Benutzung einer Ressource besteht aus den folgenden abstrakten Teilschritten:

1. Die Ressource anfordern.
2. Die Ressource benutzen.
3. Die Ressource freigeben.

Wenn eine Ressource gerade besetzt ist, muss der Prozess, der sie anfordert, warten. Einige Betriebssysteme blockieren einen Prozess automatisch, wenn eine Ressourcen-Anforderung fehlschlägt, und wecken ihn wieder auf, sobald sie frei wird. Andere geben einen Fehlercode zurück und der Prozess muss von sich aus kurz warten und es dann nochmal versuchen.

Ein Prozess, dem eine Ressource verweigert wurde, sitzt normalerweise in einer engen Schleife, die die Ressource anfordert, kurz wartet und sie dann erneut anfordert. Obwohl dieser Prozess technisch gesehen nicht blockiert ist, kann man ihn mit gutem Gewissen als blockiert ansehen, da er keine sinnvolle Arbeit leisten kann. Im Folgenden werden wir annehmen, dass ein Prozess blockiert wird, wenn ihm eine Ressource verweigert wird.

Wie eine Ressource genau angefordert wird, hängt stark vom jeweiligen System ab. Einige Systeme stellen einen `request`-Systemaufruf zur Verfügung, mit dem Prozesse ausdrücklich Ressourcen anfordern können. In anderen Systemen stellt das Betriebssystem spezielle Dateien zur Verfügung, die nur ein Prozess gleichzeitig öffnen kann. Der Zugriff funktioniert dann einfach mit dem üblichen `open`-Systemaufruf. Wenn die Datei schon von einem anderen Prozess geöffnet wurde, wird der aufrufende Prozess blockiert, bis der Besitzer sie schließt.

3.1.2 Ressourcenanforderung

Bei manchen Arten von Ressourcen, wie etwa den Datensätzen einer Datenbank, müssen sich die Prozesse selbst um die Zuteilung der Ressourcen kümmern. Eine Möglichkeit der Ressourcenverwaltung durch die Benutzer ist, jeder Ressource ein Semaphore zuzuordnen. Die Semaphore werden alle mit 1 initialisiert. Ein Mutex pro Ressource funktioniert genauso gut. Die vorher erwähnten drei Schritte werden dann folgendermaßen implementiert: Die Ressource wird durch eine `Down`-Operation auf dem Semaphore reserviert, anschließend benutzt und schließlich durch eine `Up`-Operation wieder freigegeben. Diese Schritte werden in Abbildung 3.1(a) dargestellt.

Es kann vorkommen, dass ein Prozess zwei oder mehrere Ressourcen gleichzeitig benötigt. Diese können dann wie in Abbildung 3.1(b) angefordert werden. Wenn der Prozess mehrere Ressourcen benötigt, fordert er sie einfach eine nach der anderen an.

So weit, so gut. Solange nur ein Prozess beteiligt ist, funktioniert alles hervorragend. Natürlich gibt es eigentlich auch gar keinen Grund, Ressourcen formell anzufordern, wenn die Konkurrenz fehlt.

Nehmen wir also an, es gibt zwei Prozesse *A* und *B* und zwei Ressourcen. Abbildung 3.2 zeigt zwei mögliche Szenarien. In Abbildung 3.2(a) fordern beide Prozesse die Ressourcen in derselben Reihenfolge an, in Abbildung 3.2(b) ist die Reihenfolge verschieden. Der Unterschied scheint vernachlässigbar, ist es aber nicht.

Einer der Prozesse in 3.2(a) wird die erste Ressource anfordern, bevor es der andere Prozess tut. Dieser Prozess wird dann auch die zweite Ressource bekommen und kann

Abbildung 3.1 Jede Ressource wird durch ein Semaphor geschützt. (a) Eine Ressource. (b) Zwei Ressourcen.

<pre>typedef int semaphore; semaphore resource_1; void process_A (void) { down(&resource_1); use_resource_1(); up(&resource_1); }</pre> <p style="text-align: center;">(a)</p>	<pre>typedef int semaphore; semaphore resource_1; semaphore resource_2; void process_A (void) { down(&resource_1); down(&resource_2); use_both_resources(); up(&resource_2); up(&resource_1); }</pre> <p style="text-align: center;">(b)</p>
---	---

Abbildung 3.2 (a) Deadlock-freier Code. (b) Code mit einem möglichen Deadlock.

<pre>typedef int semaphore; semaphore resource_1; semaphore resource_2; void process_A (void) { down(&resource_1); down(&resource_2); use_both_resources(); up(&resource_2); up(&resource_1); }</pre> <pre>void process_B (void) { down(&resource_1); down(&resource_2); use_both_resources(); up(&resource_2); up(&resource_1); }</pre> <p style="text-align: center;">(a)</p>	<pre>typedef int semaphore; semaphore resource_1; semaphore resource_2; void process_A (void) { down(&resource_1); down(&resource_2); use_both_resources(); up(&resource_2); up(&resource_1); }</pre> <pre>void process_B (void) { down(&resource_2); down(&resource_1); use_both_resources(); up(&resource_1); up(&resource_2); }</pre> <p style="text-align: center;">(b)</p>
--	--

seine Arbeit beenden. Wenn der andere Prozess versucht, Ressource 1 zu bekommen, bevor sie wieder frei ist, wird er einfach blockiert, bis der erste Prozess sie freigibt.

In Abbildung 3.2(b) ist die Situation anders. Es kann passieren, dass einer der Prozesse rechtzeitig beide Ressourcen reserviert und seine Arbeit beenden kann, während der andere Prozess blockiert ist. Es könnte aber genauso gut sein, dass Prozess A Ressource 1 und Prozess B Ressource 2 reserviert. Dann werden beide blockiert, sobald sie die andere Ressource anfordern. Keiner der beiden wird jemals wieder aufwachen; die Situation ist ein Deadlock.

Man sieht also, dass ein scheinbar kleiner Unterschied im Programmierstil (welche Ressource zuerst angefordert wird) den Unterschied zwischen einem Programm ausmachen kann, das fehlerfrei läuft und einem, das auf schwer erklärliche Weise fehlschlägt. Weil Deadlocks so leicht entstehen können, wurde eine Menge Forschungsarbeit in Methoden gesteckt, sie zu bekämpfen.

Dieses Kapitel beschäftigt sich mit Deadlocks und der Frage, was man gegen sie tun kann.

3.2 Einführung in Deadlocks

Formal kann man den Begriff Deadlock so definieren:

Eine Menge von Prozessen befindet sich in einem Deadlock-Zustand, wenn jeder Prozess aus der Menge auf ein Ereignis wartet, das nur ein anderer Prozess aus der Menge auslösen kann.

Alle Prozesse warten und werden deshalb die Ereignisse, auf die die anderen Prozesse warten, niemals auslösen. Keiner der Prozesse wird jemals aufwachen. Für dieses Modell nehmen wir an, dass jeder Prozess nur aus einem einzigen Thread besteht und dass keine Unterbrechungen möglich sind, die einen Prozess aufwecken könnten. Diese letzte Bedingung ist nötig, da sonst ein beteiligter Prozess geweckt werden könnte, zum Beispiel durch einen Alarm, und dann Ereignisse auslösen könnte, die andere Prozesse aufwecken.

Meistens ist das Ereignis, auf das ein Prozess wartet, die Freigabe einer Ressource, die momentan ein anderer Prozess aus der Menge belegt. Mit anderen Worten, jeder Prozess in der Menge wartet auf eine Ressource, die ein anderer Prozess aus der Menge blockiert. Keiner der Prozesse kann weiterlaufen, Ressourcen freigeben oder aufgeweckt werden. Wie viele Prozesse an dem Deadlock beteiligt sind, ist unwichtig, ebenso die Anzahl und Art der Ressourcen, die reserviert und angefordert werden. Diese Aussage gilt für jede Art von Ressourcen, einschließlich Hardware- und Softwareressourcen.

3.2.1 Voraussetzungen für Deadlocks

Nach Coffman et al. (1971) müssen für einen Deadlock folgende vier Voraussetzungen erfüllt sein:

1. Wechselseitiger Ausschluss: Jede Ressource ist entweder verfügbar oder genau einem Prozess zugeordnet.
2. Hold-and-wait-Bedingung: Prozesse, die schon Ressourcen reserviert haben, können noch weitere Ressourcen anfordern.
3. Ununterbrechbarkeit: Ressourcen, die einem Prozess bewilligt wurden, können diesem nicht gewaltsam wieder entzogen werden. Der Prozess muss sie explizit freigeben.
4. Zyklische Wartebedingung: Es muss eine zyklische Kette von Prozessen geben, von denen jeder auf eine Ressource wartet, die dem nächsten Prozess in der Kette gehört.

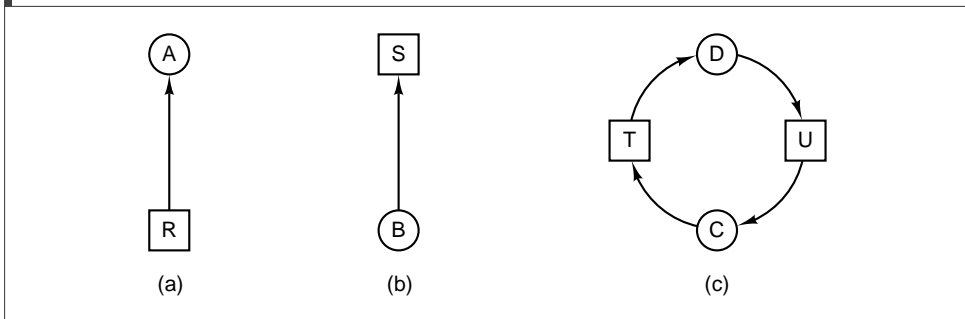
Alle vier Bedingungen müssen gleichzeitig erfüllt sein, damit ein Deadlock entstehen kann. Wenn eine fehlt, ist ein Deadlock unmöglich. Es ist erwähnenswert, dass jede die-

ser Bedingungen einem Grundsatz bei der Ressourcenverwaltung entspricht. Kann eine Ressource mehr als einem Prozess gleichzeitig zugeteilt werden? Kann ein Prozess, der schon eine Ressource besitzt, noch eine zweite anfordern? Kann eine Ressource einem Prozess entzogen werden? Kann es zu zyklischen Wartebedingungen kommen? Später werden wir sehen, wie man Deadlocks verhindern kann, indem man versucht, einige dieser Bedingungen unerfüllbar zu machen.

3.2.2 Modellierung von Deadlocks

Holt (1972) hat gezeigt, wie diese vier Bedingungen mit Hilfe von gerichteten Graphen modelliert werden können. Die Graphen haben zwei Arten von Knoten: Prozesse, die als Kreise dargestellt werden, und Ressourcen, dargestellt als Quadrate. Eine Kante von einem Ressourcenknoten (Quadrat) zu einem Prozessknoten (Kreis) bedeutet, dass die Ressource von dem Prozess angefordert wurde und dass er sie nun belegt. Abbildung 3.3(a) zeigt einen solchen **Belegungs-Anforderungs-Graphen**⁴, in dem Prozess *A* die Ressource *R* belegt.

Abbildung 3.3 Belegungs-Anforderungs-Graphen. (a) *A* belegt *R*. (b) *B* wartet auf *S*. (c) Deadlock.



Eine Kante von einem Prozess zu einer Ressource bedeutet, dass der Prozess auf die Ressource wartet. In Abbildung 3.3(b) wartet Prozess *B* auf die Ressource *S*.

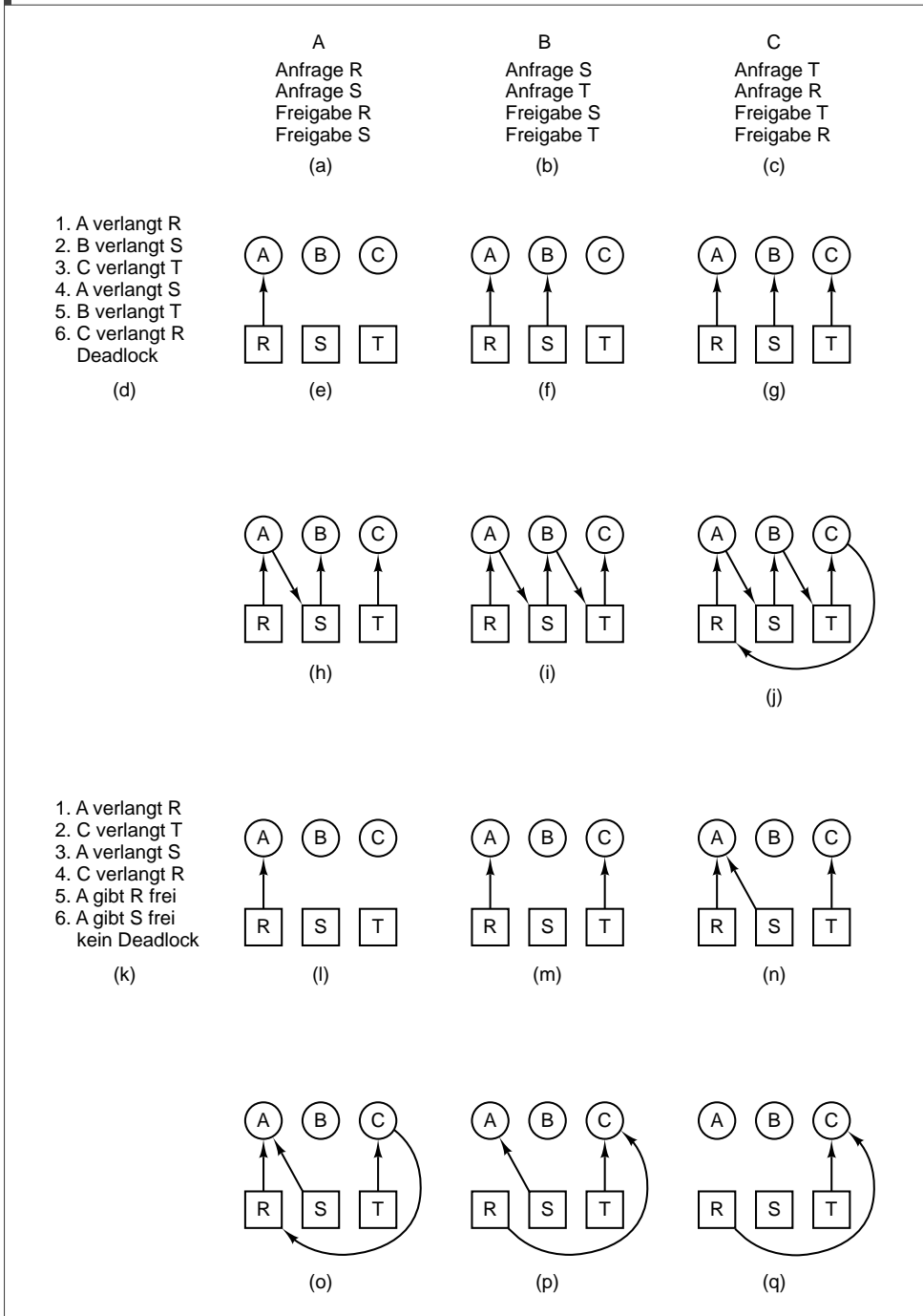
Abbildung 3.3(c) zeigt einen Deadlock: Prozess *C* wartet auf die Ressource *T*, die gerade von Prozess *D* belegt wird. Prozess *D* wird die Ressource *T* nicht freigeben, weil er auf die Ressource *U* wartet, die wiederum von *C* belegt wird. Beide Prozesse warten ewig.

Ein Zyklus im Graphen bedeutet, dass ein Deadlock vorliegt, an dem die Prozesse und Ressourcen im Zyklus beteiligt sind (vorausgesetzt, es gibt nur eine Ressource von jeder Art). In diesem Beispiel ist der Zyklus *C-T-D-U-C*.

Sehen wir uns jetzt ein Beispiel an, wie die Belegungs-Anforderungs-Graphen benutzt werden können. Nehmen wir an, wir haben drei Prozesse, *A*, *B* und *C*, und drei Ressourcen, *R*, *S* und *T*. Abbildung 3.4(a) bis (c) zeigt, wie die Ressourcen reserviert und freigegeben werden. Das Betriebssystem kann zu jedem Zeitpunkt jeden beliebigen nicht blockierten Prozess ausführen. Es könnte also beispielsweise Prozess *A* ausführen, bis er

⁴ engl.: resource allocation graph.

Abbildung 3.4 Ein Beispiel für einen Deadlock und wie er vermieden werden kann.



beendet ist, dann B und schließlich C . Diese Reihenfolge führt nicht zum Deadlock, da es keine Konkurrenz um die Ressourcen gibt, allerdings gibt es auch keine parallele Ausführung der Prozesse. Zusätzlich zum Anfordern und Freigeben von Ressourcen führen die Prozesse noch Berechnungen und E/A-Operationen aus. Wenn die Prozesse sequentiell ausgeführt werden, gibt es keine Möglichkeit, dass ein Prozess die CPU benutzt, während ein anderer auf eine E/A-Unterbrechung wartet. Sequentielle Ausführung ist also nicht unbedingt optimal. Andererseits ist Shortest-Job-First besser als Round Robin, solange keiner der Prozesse E/A-Operationen ausführt, deshalb könnte es unter Umständen die beste Alternative sein.

Nehmen wir ab jetzt an, dass die Prozesse sowohl Berechnungen als auch E/A-Operationen ausführen, so dass Round Robin ein sinnvoller Scheduling-Algorithmus ist. Eine mögliche Reihenfolge von Anforderungen wird in Abbildung 3.4(d) gezeigt. Abbildung 3.4(e) bis (j) zeigt die sechs zugehörigen Graphen. Nach der vierten Anforderung blockiert A und wartet darauf, dass S frei wird (Abbildung 3.4(h)). In den nächsten zwei Schritten blockieren B und C , was schließlich zu einem Zyklus und der Deadlock-Situation aus Abbildung 3.4(j) führt.

Allerdings kann sich das Betriebssystem, wie bereits erwähnt, die Reihenfolge der Ausführung aussuchen. Insbesondere kann es die Zuteilung einer freien Ressource verweigern, falls dies zu einem Deadlock führen könnte, und den Prozess stattdessen blockieren (d.h. ihm einfach keine Rechenzeit zuteilen). Falls das Betriebssystem in Abbildung 3.4 den drohenden Deadlock erkennt, könnte es B blockieren, anstatt ihm S zu bewilligen. Dann würden nur C und A ausgeführt und wir würden die Anforderungen und Freigaben aus Abbildung 3.4(k) erhalten, statt denen aus 3.4(d). Bei dieser Reihenfolge entstehen die Belegungs-Anforderungs-Graphen aus Abbildung 3.4(l) bis (q), die nicht zum Deadlock führen.

Nach dem Schritt (q) kann B dann die Ressource S zugeteilt bekommen, weil A fertig ist und C schon alles hat, was er braucht. Auch wenn B später T anfordert und blockiert wird, entsteht kein Deadlock, weil C zu Ende laufen kann und schließlich T freigibt.

Später in diesem Kapitel sehen wir uns einen Algorithmus an, der die Zuteilungsentscheidungen so trifft, dass keine Deadlocks entstehen. Im Augenblick sollte man sich nur merken, dass Belegungs-Anforderungs-Graphen eine Möglichkeit darstellen, zu entscheiden, ob eine gegebene Sequenz von Anforderungen und Freigaben zu einem Deadlock führt.

Wir führen einfach Schritt für Schritt die Anforderungen und Freigaben aus und untersuchen den Graphen nach jedem Schritt auf Zyklen. Wenn wir einen Zyklus finden, befinden wir uns in einem Deadlock, sonst nicht.

Wir haben uns zwar auf den Fall einer einzigen Ressource pro Klasse beschränkt, das Modell kann aber auf die Behandlung von mehreren Ressourcen derselben Klasse verallgemeinert werden (Holt, 1972).

Grundsätzlich gibt es vier verschiedene Strategien, Deadlocks zu behandeln:

1. Das ganze Problem ignorieren. Wenn man so tut, als ob es nicht existiert, glaubt es das vielleicht auch.
2. Erkennen und Beheben. Deadlocks zulassen, erkennen und etwas dagegen unternehmen.

3. Dynamische Verhinderung durch vorsichtiges Ressourcenmanagement.
4. Vermeidung von Deadlocks. Eine der vier notwendigen Bedingungen muss prinzipiell unerfüllbar werden.

In den nächsten vier Abschnitten werden wir jede dieser Methoden einzeln untersuchen.

3.3 Der Vogel-Strauß-Algorithmus

Der einfachste Ansatz ist der so genannte Vogel-Strauß-Algorithmus:⁵ Den Kopf in den Sand stecken und so tun, als gäbe es gar kein Problem⁶. Die Meinungen über diese Strategie sind geteilt. Mathematiker finden sie völlig indiskutabel und meinen, Deadlocks müssten um jeden Preis verhindert werden. Ingenieure fragen, wie oft das Problem auftritt, wie oft das System aus anderen Gründen abstürzt und wie schwerwiegend ein Deadlock ist. Wenn ein Deadlock nur durchschnittlich alle fünf Jahre vorkommt, das System aber durch Hardwareausfälle und Fehler im Compiler und Betriebssystem einmal pro Woche abstürzt, sind die meisten Ingenieure nicht bereit, größere Einbußen an Geschwindigkeit oder Bequemlichkeit hinzunehmen, um Deadlocks zu vermeiden.

Um diesen Gegensatz noch deutlicher zu machen, sollte erwähnt werden, dass in den meisten Betriebssystemen Deadlocks auftreten können, die nicht bemerkt und schon gar nicht automatisch behoben werden.

Die Gesamtzahl der Prozesse, die auf einem System laufen, wird durch die Anzahl der Einträge in der Prozesstabelle bestimmt. Einträge in der Prozesstabelle sind also endliche Ressourcen. Wenn ein `fork`-Aufruf fehlschlägt, weil die Prozesstabelle voll ist, wäre eine vernünftige Reaktion, eine zufällige Zeit zu warten und es dann noch einmal zu probieren.

Nehmen wir nun an, dass ein UNIX-System 100 Einträge in der Prozesstabelle hat. Es laufen zehn Programme gleichzeitig, von denen jedes zwölf Kindprozesse starten möchte. Nachdem die Prozesse jeweils neun Kindprozesse gestartet haben, laufen die zehn ursprünglichen Prozesse und zusätzlich 90 Kindprozesse. Damit ist die Prozesstabelle voll. Jeder der zehn ursprünglichen Prozesse sitzt nun in einer Endlosschleife fest und versucht erfolglos, weitere Kinder zu erzeugen — ein Deadlock. Die Wahrscheinlichkeit, dass diese Situation auftritt, ist sehr gering, aber nicht gleich Null. Sollte man also Prozesse und den `fork`-Systemaufruf abschaffen, um das Problem zu umgehen?

Die maximale Anzahl von offenen Dateien ist in ähnlicher Weise durch die Größe der I-Node-Tabelle beschränkt, also entsteht ein ähnliches Problem, wenn sie voll ist. Eine weitere endliche Ressource ist der Platz auf der Platte für ausgelagerte Prozesse. Tatsächlich ist fast jede Tabelle im Betriebssystem eine endliche Ressource. Sollten wir sie alle abschaffen, nur weil es passieren kann, dass eine Menge von n Prozessen jeweils einen Anteil von $1/n$ belegen und dann noch einen weiteren Eintrag anfordern?

Die meisten Betriebssysteme, darunter UNIX und Windows, ignorieren das Problem mit der Begründung, dass die meisten Benutzer einen gelegentlichen Deadlock der Einschränkung auf einen einzigen Prozess, eine einzige offene Datei usw. vorziehen. Wenn

⁵ engl.: ostrich algorithm.

⁶ Strauße können in Wirklichkeit 60 km/h schnell laufen und mit einem Tritt Löwen töten, die von gigantischen Brathähnchen träumen.

Deadlocks ohne Nachteil beseitigt werden könnten, gäbe es keine Diskussion. Das Problem ist aber, dass der Preis hoch ist. Insbesondere müssten Prozesse unbequemen Einschränkungen unterworfen werden, wie wir bald sehen werden.

Wir sind also in einer Zwickmühle zwischen Bequemlichkeit und Korrektheit und die Frage, was für wen wichtiger ist, wird heiß diskutiert.

Unter diesen Bedingungen sind allgemeingültige Lösungen kaum zu finden.

3.4 Deadlocks erkennen und beheben

Eine zweite Art, an das Problem heranzugehen, ist das Erkennen und Beheben von Deadlocks. Bei dieser Technik versucht das System nicht, das Auftreten von Deadlocks zu verhindern. Stattdessen werden Deadlocks zugelassen und das System versucht, sie zu erkennen und anschließend etwas dagegen zu unternehmen. In diesem Abschnitt lernen wir einige Möglichkeiten kennen, Deadlocks zu erkennen und zu beheben.

3.4.1 Deadlock-Erkennung bei einer Ressource pro Typ

Fangen wir mit dem einfachsten Fall an: Es gibt nur eine Ressource in jeder Klasse. Ein solches System könnte zum Beispiel einen Scanner und einen Plotter haben, also höchstens eine Ressource in jeder Klasse. Mit anderen Worten, wir ignorieren Systeme mit zwei Plottern. Solche Systeme werden wir später mit einer anderen Methode behandeln.

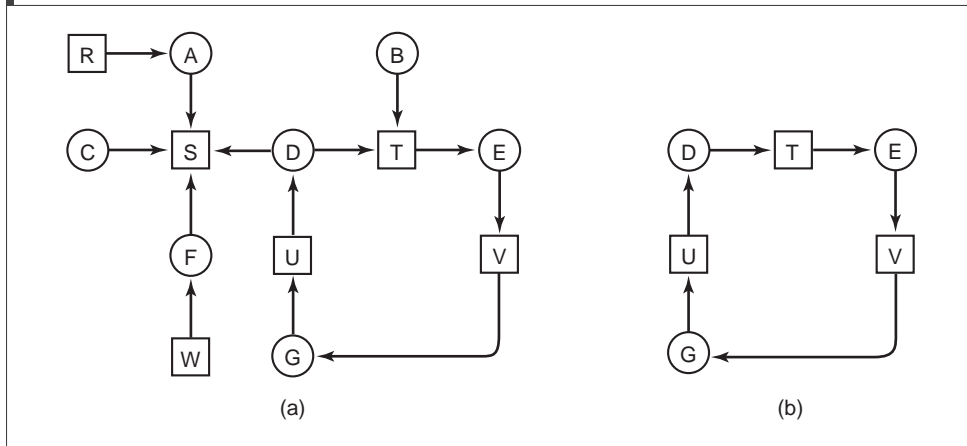
Für Systeme mit einer Ressource pro Typ können wir einen Belegungs-Anforderungs-Graphen wie in Abbildung 3.3 konstruieren. Wenn dieser Graph ein oder mehrere Zyklen enthält, befindet sich das System in einem Deadlock. Wenn kein Zyklus existiert, gibt es auch keinen Deadlock.

Stellen wir uns nun ein etwas komplexeres System als die bisher behandelten vor, eines mit sieben Prozessen, *A* bis *G*, und sechs Ressourcen, *R* bis *W*. Das System befindet sich in folgendem Zustand:

1. *A* belegt *R* und verlangt *S*.
2. *B* belegt nichts, verlangt aber *T*.
3. *C* belegt nichts, verlangt aber *S*.
4. *D* belegt *U* und verlangt *S* und *T*.
5. *E* belegt *T* und verlangt *V*.
6. *F* belegt *W* und verlangt *S*.
7. *G* belegt *V* und verlangt *U*.

Die Frage lautet: „Ist dieses System in einem Deadlock-Zustand, und wenn ja, wer ist daran beteiligt?“

Um sie zu beantworten, konstruieren wir den Graphen aus Abbildung 3.5(a). Wie man durch Hinsehen erkennt, enthält der Graph einen Zyklus (Abbildung 3.5(b)). Dieser Zyklus entspricht einem Deadlock, an dem die Prozesse *D*, *E* und *G* beteiligt sind. Die Prozesse *A*, *C* und *F* sind nicht beteiligt, da *S* nacheinander jedem Prozess zugeteilt werden kann.

Abbildung 3.5 (a) Ein Belegungs-Anforderungs-Graph. (b) Ein Zyklus aus (a).

Obwohl es relativ einfach ist, einen Deadlock und die beteiligten Prozesse in einem einfachen Graphen durch Hinsehen zu erkennen, ist für die Anwendung in echten Systemen ein formaler Algorithmus nötig. Es sind viele Algorithmen bekannt, die Zyklen in gerichteten Graphen finden können. Der folgende Algorithmus ist ein einfaches Beispiel. Er terminiert, sobald er einen Zyklus gefunden hat oder wenn kein Zyklus existiert. Als einzige Datenstruktur benutzt er eine Liste von Knoten L . Während der Algorithmus abläuft, markiert er die bereits untersuchten Kanten, um weitere Überprüfungen zu verhindern. Der Algorithmus besteht aus den folgenden Schritten:

1. Für alle Knoten K im Graphen führe die folgenden Schritte aus. Benutze K als Startknoten.
2. Lösche alle Kantenmarkierungen und L .
3. Hänge den aktuellen Knoten an das Ende von L und überprüfe, ob er in L zweimal vorkommt. Wenn ja, enthält L den gesuchten Zyklus und der Algorithmus terminiert.
4. Überprüfe, ob vom aktuellen Knoten unmarkierte Kanten wegführen. Wenn ja, gehe zu Schritt 5, sonst gehe zu Schritt 6.
5. Wähle zufällig eine wegführende Kante und markiere sie. Folge der Kante zum neuen aktuellen Knoten und gehe zu Schritt 3.
6. Wir sind in einer Sackgasse. Wenn der aktuelle Knoten der Startknoten ist, enthält der Graph keine Zyklen und der Algorithmus terminiert. Ansonsten lösche den aktuellen Knoten aus L , gehe zurück zum vorherigen Knoten, d.h. zu dem, der zuletzt der aktuelle Knoten war, mache ihn wieder zum aktuellen Knoten und gehe zu Schritt 3.

Der Algorithmus nimmt sich jeden Knoten im Graphen vor und behandelt ihn als Wurzel eines möglichen Baums. Der Baum wird dann mit Tiefensuche durchlaufen. Wenn er auf einen schon besuchten Knoten stößt, hat er einen Zyklus gefunden. Falls alle Kanten, die von einem Knoten wegführen, markiert sind, springt er zum vorherigen Knoten zurück. Wenn er zum Startknoten zurückspringt und nicht mehr weiter kann, ist der von diesem Knoten aus erreichbare Teilgraph zyklensfrei. Wenn dies für alle Knoten gilt, ist der Graph selbst zyklensfrei und das System enthält keine Deadlocks.

Um den Algorithmus in Aktion zu erleben, wenden wir ihn auf den Graphen in Abbildung 3.5(a) an. Die Reihenfolge der Knoten ist beliebig, also können wir sie von links nach rechts und von oben nach unten durchlaufen. Der erste Knoten ist also R , dann kommen A, B, C, S, T, D, E, F usw. Wenn wir auf einen Zyklus stoßen, terminiert der Algorithmus.

Wir beginnen bei R und initialisieren L mit der leeren Liste. Dann hängen wir R an die Liste an und folgen der einzig möglichen Verbindung nach A . Nachdem wir A zur Liste hinzugefügt haben ist $L = [R, A]$.

Von A gehen wir weiter nach S , so dass $L = [R, A, S]$. S ist eine Sackgasse, also müssen wir zurück nach A . Von A führen keine neuen Kanten weg, also gehen wir zurück nach R . Da auch von A keine neuen Kanten wegführen, ist damit unsere Behandlung von R beendet.

Nun wenden wir den Algorithmus auf A an und setzen L wieder auf die leere Liste. Auch diese Suche ist schnell zu Ende und wir machen mit B weiter. Von B aus folgen wir den Kanten bis D . Die Liste enthält nun $[B, T, E, V, G, U, D]$. An dieser Stelle müssen wir zufällig eine der zwei Kanten wählen, die von D wegführen. Wenn wir S wählen, sind wir in einer Sackgasse und gehen zurück zu D . Beim zweiten Versuch bleibt uns nur T und wir erhalten $L = [B, T, E, V, G, U, D, T]$. Damit haben wir den Zyklus gefunden und der Algorithmus terminiert.

Obwohl dieser Algorithmus alles andere als optimal ist, zeigt er, dass Algorithmen zur Deadlock-Erkennung existieren. Ein besserer Algorithmus findet sich in (Even, 1979).

3.4.2 Deadlock-Erkennung bei mehreren Ressourcen pro Typ

Wenn es von einer Ressource mehrere Kopien gibt, ist ein anderer Ansatz zur Deadlock-Erkennung nötig. Wir beschreiben nun einen Matrix-basierten Algorithmus zur Erkennung von Deadlocks in einer Menge von n Prozessen P_1 bis P_n . Die Anzahl der Ressourcenklassen sei m , mit E_i Ressourcen der Klasse i ($1 \leq i \leq m$). E heißt **Ressourcenvektor**⁷ und gibt die Anzahl der Ressourcen an, die von jeder Klasse insgesamt verfügbar sind. Falls zum Beispiel die Bandgeräte Klasse 1 sind, bedeutet $E_1 = 2$, dass das System zwei Bandgeräte besitzt.

Zu jedem Zeitpunkt sind einige Ressourcen belegt. Der **Ressourcenrestvektor**⁸ A enthält für jede Ressource i die Anzahl der freien Instanzen A_i . Wenn z. B. beide Bandlaufwerke belegt sind, ist $A_1 = 0$.

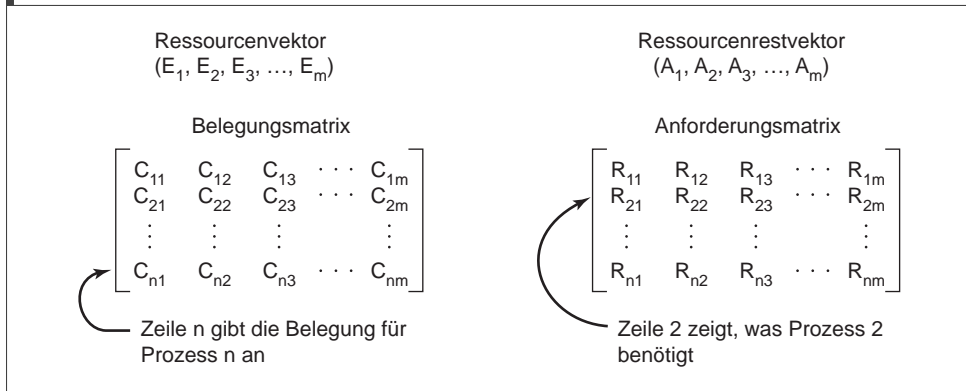
Zusätzlich brauchen wir noch zwei Matrizen: die **Belegungsmatrix**⁹ C und die **Anforderungsmatrix**¹⁰ R . Die i -te Zeile von C enthält die Anzahl der Ressourcen, die der Prozess P_i von jeder Klasse belegt. C_{ij} ist also die Anzahl der Ressourcen der Klasse j , die Prozess i belegt. Analog ist R_{ij} die Anzahl der Ressourcen der Klasse j , die Prozess i gerne hätte. Abbildung 3.6 zeigt die vier beschriebenen Datenstrukturen.

⁷ engl.: existing resource vector

⁸ engl.: available resource vector

⁹ engl.: allocation matrix

¹⁰ engl.: request matrix

Abbildung 3.6 Die vier Datenstrukturen für den Deadlock-Erkennungsalgorithmus.

Da jede Ressource entweder belegt oder frei ist, gilt die folgende Invariante für jede Ressourcenverteilung:

$$\sum_{i=1}^n C_{ij} + A_j = E_j$$

Mit anderen Worten, für jede Ressourcenklasse ist die Summe der von allen Prozessen belegten Ressourcen und der freien Ressourcen gleich der Gesamtzahl der Ressourcen in dieser Klasse. Der Deadlock-Erkennungsalgorithmus basiert auf dem Vergleich von Vektoren. Wir definieren die Relation \leq auf der Menge der Vektoren so, dass $A \leq B$ genau dann, wenn jedes Element von A kleiner oder gleich dem entsprechenden Element von B ist, wenn also $A_i \leq B_i$ für alle $1 \leq i \leq n$.

Zu Beginn des Algorithmus ist jeder Prozess unmarkiert. Wenn ein Prozess markiert wird, bedeutet das, er kann zu Ende laufen, ist also an keinem Deadlock beteiligt. Wenn der Algorithmus terminiert, ist jeder nicht markierte Prozess an einem Deadlock beteiligt.

Der Deadlock-Erkennungsalgorithmus läuft folgendermaßen ab:

1. Suche einen unmarkierten Prozess P_i , für den die i -te Zeile von R kleiner oder gleich A ist.
2. Wenn ein solcher Prozess existiert, addiere die i -te Zeile von C zu A , markiere den Prozess und gehe zu Schritt 1.
3. Andernfalls beende den Algorithmus.

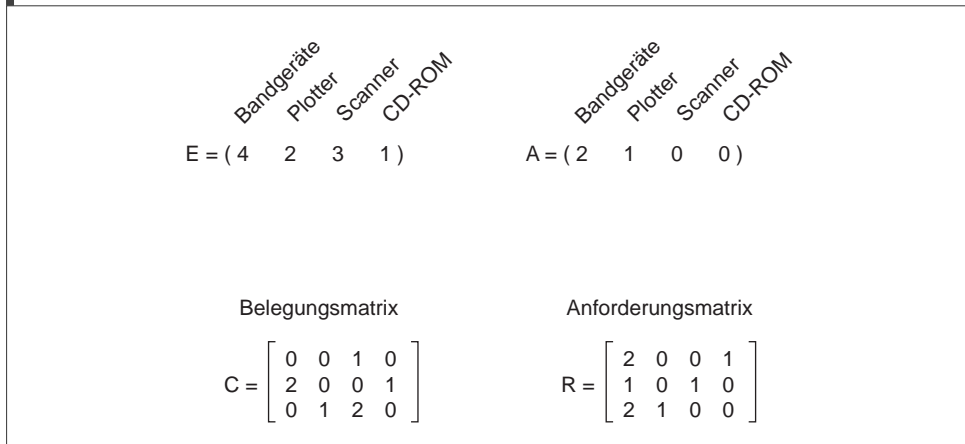
Wenn der Algorithmus beendet ist, sind alle nicht markierten Prozesse an einem Deadlock beteiligt.

In Schritt 1 sucht der Algorithmus nach einem Prozess, der zu Ende laufen kann. Dieser Algorithmus zeichnet sich dadurch aus, dass die momentan verfügbaren Ressourcen für seine Anforderungen ausreichen. Der gewählte Prozess wird dann bis zu seinem Ende ausgeführt und die Ressourcen, die er belegt, werden zu den anderen verfügbaren Ressourcen addiert. Anschließend wird der Prozess als beendet markiert. Wenn schließlich alle Prozesse beendet werden können, ist keiner von ihnen an einem Deadlock beteiligt.

Wenn aber einige von ihnen niemals ausgeführt werden können, befinden sie sich in einem Deadlock. Obwohl der Algorithmus nichtdeterministisch ist (da er die Prozesse in jeder möglichen Reihenfolge ausführen kann), ist das Ergebnis immer dasselbe.

Abbildung 3.7 zeigt ein Beispiel, an dem wir den Algorithmus ausprobieren können. In diesem Beispiel haben wir drei Prozesse und vier Ressourcenklassen, die als Bandlaufwerke, Plotter, Scanner und CD-ROM-Laufwerke bezeichnet sind. Die Namen der Ressourcenklassen sind beliebig austauschbar.

Abbildung 3.7 Ein Beispiel für den Deadlock-Erkennungsalgorithmus.



Prozess 1 belegt einen Scanner. Prozess 2 belegt zwei Bandlaufwerke und ein CD-Laufwerk. Prozess 3 belegt einen Plotter und zwei Scanner. Die R -Matrix zeigt, welche Ressourcen jeder Prozess noch zusätzlich benötigt.

Um unseren Algorithmus anzuwenden, suchen wir zunächst nach einem Prozess, dessen Anforderungen erfüllt werden können. Der erste Prozess kann nicht zufrieden gestellt werden, weil kein CD-Laufwerk frei ist. Der zweite fällt auch aus, weil es keinen freien Scanner gibt. Zum Glück verlangt der dritte Prozess nur verfügbare Ressourcen. Er kann also ausgeführt werden und gibt schließlich alle seine Ressourcen frei, so dass

$$A = (2, 2, 2, 0)$$

Jetzt kann Prozess 2 weiterlaufen. Nachdem er seine Ressourcen freigegeben hat, ist

$$A = (4, 2, 2, 1)$$

Nun kann auch der letzte Prozess ausgeführt werden, es gibt also keinen Deadlock.

Betrachten wir nun eine kleine Variation von Abbildung 3.7. Nehmen wir an, Prozess 2 braucht neben den zwei Bandlaufwerken und dem Plotter noch ein CD-Laufwerk. Nun kann keine der Forderungen mehr erfüllt werden, das System befindet sich in einem Deadlock.

Nachdem wir nun wissen, wie man Deadlocks erkennt, stellt sich die Frage, wann man die Überprüfung durchführen soll. Eine Möglichkeit wäre, bei jeder einzelnen Ressour-

cenanforderung nach Deadlocks zu suchen. So würde man Deadlocks zwar frühestmöglich erkennen, aber es würde möglicherweise zu viel Prozessorzeit verbraucht.

Alternativ könnte man alle k Minuten nach Deadlocks suchen oder nur dann, wenn die Prozessornutzung unter eine gewisse Grenze fällt. Der Grund für diese letzte Strategie ist, dass die CPU oft leer läuft, wenn genügend Prozesse an einem Deadlock beteiligt sind, weil dann nur wenige Prozesse ausführbar sind.

3.4.3 Beheben von Deadlocks

Nehmen wir an, unser Deadlock-Erkennungsalgorithmus war erfolgreich und hat einen Deadlock gefunden. Was nun? Gesucht ist eine Möglichkeit, den Deadlock zu beheben und das System wieder in Gang zu bringen. In diesem Abschnitt werden wir Möglichkeiten behandeln, Deadlocks zu beheben, wenn auch keine von ihnen besonders vielversprechend ist.

Behebung durch Unterbrechung

In einigen Fällen kann es möglich sein, einem Prozess eine Ressource zeitweise zu entziehen und sie einem anderen Prozess zu geben. Oft geht dies nur manuell, insbesondere bei Mainframes mit Batch-Betrieb. Um beispielsweise einen Laserdrucker seinem Besitzer zu entziehen, könnte der Operator die bereits ausgedruckten Blätter beiseite legen und den Prozess suspendieren (als nicht ausführbar markieren). Jetzt kann der Drucker einem anderen Prozess zugeteilt werden. Später können die gedruckten Blätter wieder in den Ausgabeschacht zurückgelegt und der erste Prozess wieder gestartet werden.

Ob es möglich ist, einem Prozess eine Ressource zu entziehen, sie einem anderen Prozess zur Verfügung zu stellen und sie dann dem ersten Prozess wiederzugeben, ohne dass dieser es bemerkt, hängt stark von der Art der Ressource ab. Deadlocks auf diese Art zu beheben, ist häufig schwierig oder sogar unmöglich. Die Auswahl des Prozesses, der suspendiert wird, hängt hauptsächlich davon ab, ob der Prozess Ressourcen belegt, die leicht entzogen und wieder zurückgegeben werden können.

Behebung durch teilweise Wiederholung (Rollback)

Wenn Systemdesigner und Operatoren wissen, dass Deadlocks wahrscheinlich sind, können sie dafür sorgen, dass der Zustand eines Prozesses in regelmäßigen Abständen (an so genannten **Checkpoints**) in eine Datei geschrieben wird¹¹, so dass der Prozess später von diesem Punkt aus neu gestartet werden kann. Der Checkpoint enthält nicht nur den Speicherinhalt des Prozesses, sondern auch Informationen über die momentan belegten Ressourcen. Diese Methode ist am wirkungsvollsten, wenn neue Checkpoints die alten nicht überschreiben, so dass sich zu einem Prozess, während er abläuft, eine Folge von Checkpoint-Dateien ansammelt.

Wenn ein Deadlock erkannt wird, ist es einfach, die benötigten Ressourcen zu ermitteln. Zur Behebung wird dann ein Prozess, der eine benötigte Ressource besitzt, zu einem Checkpoint zurückgesetzt, an dem er diese Ressource noch nicht reserviert hatte. Die

¹¹ engl.: checkpointed

Arbeit, die der Prozess seit diesem Checkpoint geleistet hat, geht verloren (Drucker- ausgaben müssen z. B. weggeworfen werden, da der Prozess den Ausdruck wiederholen wird). Die freigewordene Ressource wird nun einem der an dem Deadlock beteiligten Prozesse zugeteilt. Wenn der zurückgesetzte Prozess versucht, die Ressource wiederzu- bekommen, muss er warten, bis sie wieder frei ist.

Behebung durch Prozessabbruch

Das härteste, aber einfachste Verfahren, einen Deadlock zu beheben, ist der Abbruch eines oder mehrerer Prozesse. Dabei kann es sich um einen Prozess aus dem Zyklus handeln. Mit etwas Glück können die anderen Prozesse dann weiterlaufen. Wenn nicht, können weitere Prozesse abgebrochen werden, bis der Zyklus gebrochen ist.

Alternativ kann das Opfer auch ein Prozess sein, der nicht am Zyklus beteiligt ist. Bei diesem Ansatz muss sehr sorgfältig ein Prozess ausgewählt werden, der Ressourcen belegt, die ein Prozess aus dem Zyklus benötigt. Beispielsweise könnte ein Prozess einen Drucker belegen und auf einen Plotter warten, während ein anderer Prozess einen Plotter belegt und auf einen Drucker wartet. Beide Prozesse sitzen in einem Deadlock fest. Zur gleichen Zeit könnte ein dritter Prozess fröhlich weiterlaufen und sowohl einen Plotter als auch einen Drucker belegen. Wenn man diesen Prozess nun abbricht, gibt er den Drucker und den Plotter frei und der Deadlock der ersten zwei Prozesse ist behoben.

Wenn möglich, sollte man zunächst einen Prozess abbrechen, der ohne unangenehme Nebenwirkungen neu gestartet werden kann. Der Ablauf eines Compilers kann beispiels- weise immer wiederholt werden, weil er nur eine Quelldatei liest und eine Objektdatei erzeugt. Wenn er auf halbem Weg unterbrochen wird, hat der erste Ablauf keinen Ein- fluss auf den zweiten.

Im Gegensatz dazu kann ein Prozess, der in eine Datenbank schreibt, meist nicht prob- lemlos neu ausgeführt werden. Wenn der Prozess z. B. zu einem Datensatz 1 addiert, würden ein teilweiser und ein vollständiger Ablauf möglicherweise 2 addieren, was zu einem falschen Ergebnis führt.

3.5 Deadlock-Verhinderung

Bei der Deadlock-Erkennung haben wir stillschweigend angenommen, dass ein Prozess alle Ressourcen, die er braucht, auf einmal anfordert (die R -Matrix aus Abbildung 3.6). In den meisten Systemen werden Ressourcen aber eine nach der anderen angefordert. Das System darf einem Prozess nur dann eine Ressource zuteilen, wenn dies ungefährlich ist. Es stellt sich also die Frage, ob ein Algorithmus existiert, der Deadlocks zuverlässig vermeiden kann, indem er immer die richtige Entscheidung trifft. Die Antwort lautet „ja, aber“. Man kann Deadlocks verhindern, aber nur, wenn bestimmte Informationen im Voraus zur Verfügung stehen. In diesem Abschnitt suchen wir nach Möglichkeiten, Deadlocks durch vorsichtige Zuteilung von Ressourcen zu verhindern.

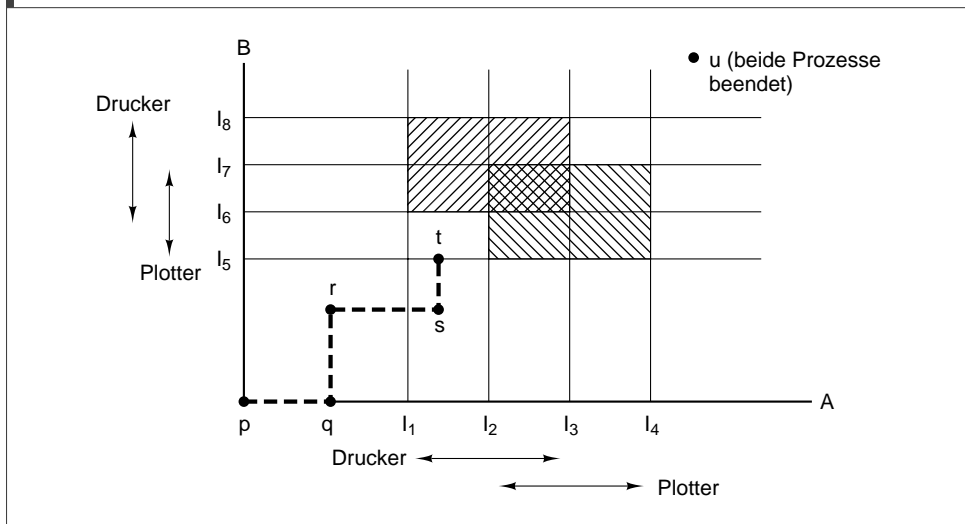
3.5.1 Ressourcenspur

Die wichtigsten Algorithmen zur Deadlock-Verhinderung basieren auf dem Konzept der sicheren Zustände. Bevor wir uns den Algorithmen selbst zuwenden, schweifen wir etwas

ab und stellen das Konzept der Sicherheit in anschaulicher und leicht verständlicher Weise vor. Obwohl sich der grafische Ansatz nicht direkt in einen Algorithmus umsetzen lässt, wird das Problem dadurch anschaulicher und intuitiv zugänglicher.

In Abbildung 3.8 sehen wir ein Modell, das zwei Prozesse und zwei Ressourcen behandelt, beispielsweise einen Drucker und einen Plotter. Die horizontale Achse repräsentiert die Anzahl der Anweisungen, die von Prozess *A* ausgeführt werden. Die vertikale Achse repräsentiert die Anzahl der Anweisungen, die von Prozess *B* ausgeführt werden. Am Punkt I_1 verlangt Prozess *A* einen Drucker, am Punkt I_2 benötigt er einen Plotter. Der Drucker und der Plotter werden jeweils bei I_3 und I_4 wieder freigegeben. Prozess *B* benötigt den Plotter von I_5 bis I_7 und den Drucker von I_6 bis I_8 . Jeder Punkt im Diagramm repräsentiert einen gemeinsamen Zustand der beiden Prozesse. Solange noch keiner der Prozesse Anweisungen ausgeführt hat, ist der Zustand im Ursprung *p*. Wenn der Scheduler zunächst *A* zur Ausführung auswählt, kommen wir zum Punkt *q*. Hier hat *A* schon eine Anzahl von Anweisungen ausgeführt, *B* aber noch keine. Am Punkt *q* wird die Spur senkrecht, was bedeutet, dass nun *B* ausgeführt wird. Mit nur einem Prozessor verlaufen die Pfade immer horizontal oder vertikal, niemals diagonal. Außerdem geht die Bewegung immer nach rechts oder nach oben, niemals nach links oder unten, da Prozesse nicht rückwärts laufen können. Sobald *A* auf dem Weg von *r* nach *s* die Linie bei I_1 überschreitet, reserviert er sich den Drucker. Am Punkt *t* reserviert sich *B* den Plotter.

Abbildung 3.8 Eine Ressourcenspur für zwei Prozesse.



Die schraffierten Bereiche sind besonders interessant. Der von links unten nach rechts oben schraffierte Bereich enthält die Zustände, in denen beide Prozesse den Drucker belegen. Da dies unmöglich ist, ist dieser Bereich unerreichbar. Mit dem anders schraffierten Bereich verhält es sich ähnlich. Hier belegen beide Prozesse den Plotter und der Bereich ist ebenfalls unerreichbar.

Falls das System in das Rechteck eintritt, das links und rechts von I_1 und I_2 und oben und unten von I_6 und I_5 begrenzt wird, ist ein Deadlock unvermeidlich, sobald es den

Schnittpunkt von I_2 und I_6 erreicht. An diesem Punkt verlangt A den Plotter und B den Drucker und beide sind schon vergeben.

Das gesamte Rechteck ist unsicher und darf nicht betreten werden. Am Punkt t ist die einzig sichere Möglichkeit, Prozess A bis I_4 auszuführen. Nach I_4 ist jede beliebige Spur gleich gut.

Wichtig ist hier die Tatsache, dass Prozess B am Punkt t eine Ressource anfordert. Das System muss entscheiden, ob er sie bekommt oder nicht. Wenn B die Ressource bekommt, tritt das System in einen unsicheren Bereich ein und es entsteht schließlich ein Deadlock. Um einen Deadlock zu vermeiden, sollte B blockiert werden, bis A den Plotter angefordert und wieder freigegeben hat.

3.5.2 Sichere und unsichere Zustände

Die Algorithmen zur Verhinderung von Deadlocks, die wir behandeln werden, benutzen die Informationen aus Abbildung 3.6. Zu jedem Zeitpunkt wird der aktuelle Zustand durch E , A , C und R bestimmt. Ein Zustand heißt **sicher**, wenn kein Deadlock vorliegt, und es eine Scheduling-Reihenfolge gibt, die nicht zum Deadlock führt, selbst wenn alle Prozesse sofort ihre maximale Anzahl an Ressourcen anfordern. Dieses Konzept lässt sich am einfachsten an einem Beispiel mit nur einer Ressourcenklasse veranschaulichen. Abbildung 3.9(a) zeigt einen Zustand, in dem Prozess A drei Instanzen der Ressource belegt, später aber möglicherweise bis zu neun Instanzen benötigt. B belegt momentan zwei Instanzen und benötigt später insgesamt vier. C hat ebenfalls zwei, braucht aber später noch bis zu fünf weitere Instanzen. Insgesamt existieren zehn Instanzen der Ressource. Sieben sind schon belegt, also sind momentan noch drei verfügbar.

Abbildung 3.9 Nachweis, dass der Zustand in (a) sicher ist.

nutzt max.			nutzt max.			nutzt max.			nutzt max.			nutzt max.		
A	3	9	A	3	9	A	3	9	A	3	9	A	3	9
B	2	4	B	4	4	B	0	–	B	0	–	B	0	–
C	2	7	C	2	7	C	2	7	C	7	7	C	0	–
Frei: 3			Frei: 1			Frei: 5			Frei: 0			Frei: 7		
(a)			(b)			(c)			(d)			(e)		

Der Zustand in Abbildung 3.9(a) ist sicher, weil es eine Folge von Zuteilungen gibt, so dass alle Prozesse zu Ende laufen können. Der Scheduler könnte zunächst ausschließlich B ausführen, bis dieser die zwei weiteren Instanzen der Ressource reserviert hat (Abbildung 3.9(b)). Sobald B beendet ist, befinden wir uns im Zustand 3.9(c). Nun kann der Scheduler C ausführen, was zu dem Zustand in Abbildung 3.9(d) führt. Wenn C beendet ist, erhalten wir den Zustand in Abbildung 3.9(e). Jetzt sind die sechs Instanzen, die A benötigt, frei und A kann ebenfalls zu Ende laufen. Der Zustand in Abbildung 3.9(a) ist also sicher, weil das System durch vorsichtiges Scheduling einen Deadlock vermeiden kann.

Gehen wir jetzt von demselben Anfangszustand in Abbildung 3.10(a) aus, mit dem Unterschied, dass A noch eine weitere Ressource zugeteilt bekommt, so dass sich der

Zustand in Abbildung 3.10(b) ergibt. Können wir eine Reihenfolge finden, die sicher funktioniert? Versuchen wir es. Der Scheduler könnte Prozess *B* ausführen, bis dieser alle seine Ressourcen anfordert (Abbildung 3.10(c)).

Abbildung 3.10 Nachweis, dass der Zustand in (b) unsicher ist.

nutzt max.			nutzt max.			nutzt max.			nutzt max.		
A	3	9	A	4	9	A	4	9	A	4	9
B	2	4	B	2	4	B	4	4	B	–	–
C	2	7	C	2	7	C	2	7	C	2	7
Frei: 3			Frei: 2			Frei: 0			Frei: 4		
(a)			(b)			(c)			(d)		

Sobald *B* beendet ist, haben wir die Situation aus Abbildung 3.10(d). An diesem Punkt stecken wir fest. Wir haben nur vier freie Instanzen der Ressource und jeder der aktiven Prozesse braucht fünf. Es gibt keine Reihenfolge, die garantiert, dass alle Prozesse zu Ende laufen können. Die Zuteilungsentscheidung von Abbildung 3.10(a) nach 3.10(b) führte das System von einem sicheren Zustand in einen unsicheren. In Abbildung 3.10(b) zunächst *A* oder *C* auszuführen, hätte auch nicht geholfen. Im Nachhinein betrachtet, hätte *A* die Ressource nicht bekommen dürfen.

Es sollte erwähnt werden, dass ein unsicherer Zustand noch kein Deadlock-Zustand ist. Vom Zustand in Abbildung 3.10(b) aus könnte das System noch eine Weile weiterlaufen. Tatsächlich könnte sogar ein Prozess beendet werden. Außerdem ist es möglich, dass *A* eine Ressource freigibt, bevor er versucht, weitere zu reservieren. So könnte *B* beendet werden und ein Deadlock wäre gänzlich vermieden. Der Unterschied zwischen einem sicheren und einem unsicheren Zustand ist, dass das System in einem sicheren Zustand *garantieren* kann, dass alle Prozesse zu Ende laufen; in einem unsicheren Zustand ist eine solche Garantie unmöglich.

3.5.3 Der Bankier-Algorithmus für eine einzelne Ressource

Ein Scheduling-Algorithmus, der Deadlocks verhindern kann, geht auf Dijkstra (1965) zurück und ist als **Bankier-Algorithmus**¹² bekannt. Der Algorithmus ist eine Erweiterung des Deadlock-Erkennungsalgorithmus, den wir in Abschnitt 3.4.1 kennen gelernt haben. Er ist der Art nachempfunden, wie ein kleiner Bankier die Kreditwünsche einer Gruppe von Kunden behandeln könnte.

Der Algorithmus überprüft bei jedem Kreditantrag, ob der Kredit zu einem unsicheren Zustand führt. Wenn der Kredit zu einem sicheren Zustand führt, wird er bewilligt, ansonsten wird er abgelehnt.

Abbildung 3.11(a) zeigt vier Kunden, *A*, *B*, *C* und *D*, von denen jeder ein bestimmtes Maximum an Krediteinheiten hat (z. B. 1 Einheit = 1K Dollar). Der Bankier weiß, dass nicht alle Kunden sofort ihr Maximum ausschöpfen werden, und reserviert deshalb nur zehn Einheiten statt 22.

¹² engl.: banker's algorithm

Abbildung 3.11 Drei Belegungszustände: (a) Sicher. (b) Sicher. (c) Unsicher.

<table style="margin: auto; border-collapse: collapse;"> <tr><td colspan="3" style="text-align: center; padding: 2px;">nutzt max.</td></tr> <tr><td style="border: 1px solid black; padding: 2px;">A</td><td style="border: 1px solid black; padding: 2px;">0</td><td style="border: 1px solid black; padding: 2px;">6</td></tr> <tr><td style="border: 1px solid black; padding: 2px;">B</td><td style="border: 1px solid black; padding: 2px;">0</td><td style="border: 1px solid black; padding: 2px;">5</td></tr> <tr><td style="border: 1px solid black; padding: 2px;">C</td><td style="border: 1px solid black; padding: 2px;">0</td><td style="border: 1px solid black; padding: 2px;">4</td></tr> <tr><td style="border: 1px solid black; padding: 2px;">D</td><td style="border: 1px solid black; padding: 2px;">0</td><td style="border: 1px solid black; padding: 2px;">7</td></tr> </table> <p style="text-align: center; margin-top: 5px;">Frei: 10 (a)</p>	nutzt max.			A	0	6	B	0	5	C	0	4	D	0	7	<table style="margin: auto; border-collapse: collapse;"> <tr><td colspan="3" style="text-align: center; padding: 2px;">nutzt max.</td></tr> <tr><td style="border: 1px solid black; padding: 2px;">A</td><td style="border: 1px solid black; padding: 2px;">1</td><td style="border: 1px solid black; padding: 2px;">6</td></tr> <tr><td style="border: 1px solid black; padding: 2px;">B</td><td style="border: 1px solid black; padding: 2px;">1</td><td style="border: 1px solid black; padding: 2px;">5</td></tr> <tr><td style="border: 1px solid black; padding: 2px;">C</td><td style="border: 1px solid black; padding: 2px;">2</td><td style="border: 1px solid black; padding: 2px;">4</td></tr> <tr><td style="border: 1px solid black; padding: 2px;">D</td><td style="border: 1px solid black; padding: 2px;">4</td><td style="border: 1px solid black; padding: 2px;">7</td></tr> </table> <p style="text-align: center; margin-top: 5px;">Frei: 2 (b)</p>	nutzt max.			A	1	6	B	1	5	C	2	4	D	4	7	<table style="margin: auto; border-collapse: collapse;"> <tr><td colspan="3" style="text-align: center; padding: 2px;">nutzt max.</td></tr> <tr><td style="border: 1px solid black; padding: 2px;">A</td><td style="border: 1px solid black; padding: 2px;">1</td><td style="border: 1px solid black; padding: 2px;">6</td></tr> <tr><td style="border: 1px solid black; padding: 2px;">B</td><td style="border: 1px solid black; padding: 2px;">2</td><td style="border: 1px solid black; padding: 2px;">5</td></tr> <tr><td style="border: 1px solid black; padding: 2px;">C</td><td style="border: 1px solid black; padding: 2px;">2</td><td style="border: 1px solid black; padding: 2px;">4</td></tr> <tr><td style="border: 1px solid black; padding: 2px;">D</td><td style="border: 1px solid black; padding: 2px;">4</td><td style="border: 1px solid black; padding: 2px;">7</td></tr> </table> <p style="text-align: center; margin-top: 5px;">Frei: 1 (c)</p>	nutzt max.			A	1	6	B	2	5	C	2	4	D	4	7
nutzt max.																																															
A	0	6																																													
B	0	5																																													
C	0	4																																													
D	0	7																																													
nutzt max.																																															
A	1	6																																													
B	1	5																																													
C	2	4																																													
D	4	7																																													
nutzt max.																																															
A	1	6																																													
B	2	5																																													
C	2	4																																													
D	4	7																																													

In dieser Analogie sind die Kunden Prozesse, die Krediteinheiten könnten z. B. Bandlaufwerke sein und der Bankier ist das Betriebssystem.

Die Kunden kümmern sich um ihre Geschäfte und beantragen gelegentlich einen Kredit (d.h., sie fordern Ressourcen an). Zu einem bestimmten Zeitpunkt ist der Zustand der in Abbildung 3.11(b). Dieser Zustand ist sicher, weil der Bankier alle Anforderungen außer die von C verzögern kann. Nachdem C beendet ist, gibt er alle vier Ressourcen frei. Mit vier freien Ressourcen kann der Bankier dann entweder D oder B die benötigten Kredite bewilligen usw.

Stellen wir uns vor, was passieren würde, wenn B in Abbildung 3.11(b) noch eine weitere Einheit bewilligt würde. In diesem Fall hätten wir den unsicheren Zustand in Abbildung 3.11(c). Wenn alle Kunden auf einmal ihr Maximum ausschöpfen wollten, könnte der Bankier keinen der Anträge bewilligen und wir hätten einen Deadlock. Ein unsicherer Zustand führt nicht *unbedingt* zum Deadlock, da ein Kunde nicht notwendigerweise sein Maximum ausschöpft, aber auf dieses Verhalten kann sich der Bankier nicht verlassen.

Der Bankier-Algorithmus prüft jede Anforderung, sobald sie auftritt, und stellt fest, ob sich ein sicherer Zustand ergibt. Wenn sich ein sicherer Zustand ergibt, wird die Anforderung bewilligt, ansonsten wird sie auf später verschoben. Um festzustellen, ob ein Zustand sicher ist, überprüft der Algorithmus, ob noch genug Ressourcen übrig sind, um einen anderen Kunden zufriedenzustellen. Wenn ja, wird angenommen, dass dieser Kunde seine Kredite zurückzahlt, und der Kunde, der nun am nächsten an seinem Limit ist, wird überprüft. Wenn schließlich alle zurückgezahlt sind, ist der Zustand sicher und der Antrag kann bewilligt werden.

3.5.4 Der Bankier-Algorithmus für mehrere Ressourcen

Der Bankier-Algorithmus kann auf mehrere Ressourcenklassen verallgemeinert werden. Abbildung 3.12 zeigt, wie es geht.

In Abbildung 3.12 sehen wir zwei Matrizen. Die linke zeigt, wie viele Instanzen aus jeder Ressourcenklasse jeder der fünf Prozesse gerade belegt. Die rechte Matrix zeigt, wie viele Ressourcen jeder Prozess noch benötigt, bevor er beendet werden kann.

Diese Matrizen sind gerade C und R aus Abbildung 3.6. Wie im Fall mit einer Ressourcenklasse muss jeder Prozess vor seiner Ausführung angeben, wie viele Ressourcen

Abbildung 3.12 Der Bankier-Algorithmus für mehrere Ressourcenklassen.

	Prozess	Bandgeräte	Plotter	Scanner	CD-ROM
A	3	0	1	1	
B	0	1	0	0	
C	1	1	1	0	
D	1	1	0	1	
E	0	0	0	0	

Zugewiesene Ressourcen

	Prozess	Bandgeräte	Plotter	Scanner	CD-ROM
A	1	1	0	0	
B	0	1	1	2	
C	3	1	0	0	
D	0	0	1	0	
E	2	1	1	0	

Benötigte Ressourcen

$E = (6342)$
 $P = (5322)$
 $A = (1020)$

er insgesamt benötigt, damit das System zu jeder Zeit die rechte der beiden Matrizen berechnen kann.

Die drei Vektoren rechts in der Abbildung enthalten jeweils die insgesamt vorhandenen Ressourcen E , die belegten Ressourcen P und die verfügbaren Ressourcen A . Aus E sehen wir, dass das System insgesamt sechs Bandlaufwerke, drei Plotter, vier Scanner und zwei CD-ROM-Laufwerke besitzt. Davon sind im Augenblick fünf Bandlaufwerke, drei Plotter, zwei Scanner und zwei CD-ROM-Laufwerke belegt. Diese Zahlen kann man leicht berechnen, indem man die Spalten der linken Matrix aufaddiert. Der Ressourcenrestvektor ist ganz einfach die Differenz zwischen den insgesamt vorhandenen und den belegten Ressourcen jeder Klasse.

Jetzt können wir den Algorithmus angeben, der überprüft, ob ein Zustand sicher ist.

1. Suche eine Zeile aus R , die kleiner oder gleich A ist. Wenn es keine solche Zeile gibt, kann kein Prozess beendet werden und das System wird in einen Deadlock laufen.
2. Nimm an, dass der Prozess, der der gewählten Zeile entspricht, alle nötigen Ressourcen reserviert und seine Ausführung beendet. Markiere den Prozess als beendet und addiere seine Ressourcen zu A .
3. Wiederhole Schritt 1 und 2, bis entweder alle Prozesse markiert sind oder ein Deadlock auftritt. Im ersten Fall ist der Zustand sicher, im zweiten unsicher.

Wenn im ersten Schritt mehrere Prozesse zur Auswahl stehen, ist es unwichtig, welcher zuerst ausgeführt wird: Die verfügbaren Ressourcen können sich nur vermehren oder schlimmstenfalls gleich bleiben.

Kommen wir nun zu dem Beispiel aus Abbildung 3.12 zurück. Der gezeigte Zustand ist sicher. Nehmen wir an, dass Prozess B jetzt einen Scanner verlangt, der ihm bewilligt werden kann, weil der Zustand sicher bleibt (Prozess D kann beendet werden, dann A und E , dann der Rest).

Stellen wir uns vor, dass E den letzten Scanner verlangt, nachdem B einen der zwei übrigen Scanner reserviert. Wenn E den letzten Scanner bekommt, wäre der Restvektor auf $(1\ 0\ 0\ 0)$ reduziert, was zu einem Deadlock führen würde. Offensichtlich muss das System E den Scanner vorläufig verweigern.

Der Bankier-Algorithmus wurde zuerst von Dijkstra im Jahre 1965 veröffentlicht. Seitdem wurde er in fast allen Büchern über Betriebssysteme eingehend behandelt. Zahllose Artikel sind über die verschiedensten Aspekte des Algorithmus erschienen. Unglücklicherweise besaßen nur sehr wenige Autoren den Mut, darauf hinzuweisen, dass der Algorithmus zwar in der Theorie wunderschön, in der Praxis aber im Grunde nutzlos ist, weil Prozesse nur selten im Voraus wissen, wie viele Ressourcen sie brauchen werden. Außerdem ist die Anzahl der Prozesse nicht fest, sondern ändert sich ständig, wenn sich neue Benutzer ein- und ausloggen. Abgesehen davon können Ressourcen, die bisher verfügbar waren, plötzlich verschwinden (z. B. können Bandlaufwerke ausfallen). Daher wird der Bankier-Algorithmus wenn überhaupt nur von sehr wenigen Systemen benutzt, um Deadlocks zu verhindern.

3.6 Deadlock-Vermeidung

Nachdem wir nun wissen, dass Deadlock-Verhinderung im Grunde unmöglich ist, weil dazu Informationen über zukünftige Ressourcenanforderungen nötig sind, die nicht bekannt sind, stellt sich die Frage: Was unternehmen eigentlich reale Systeme gegen Deadlocks? Die Antwort steckt in den schon bekannten vier Voraussetzungen für Deadlocks, die von Coffman et al. (1971) formuliert wurden. Wenn wir sicherstellen können, dass zumindest eine dieser Voraussetzungen niemals erfüllt ist, werden Deadlocks prinzipiell unmöglich (Havender 1968).

3.6.1 Angriff auf den wechselseitigen Ausschluss

Nehmen wir uns zunächst den wechselseitigen Ausschluss vor. Wenn keine Ressource jemals nur einem Prozess zugeteilt wäre, könnten niemals Deadlocks entstehen. Ebenso klar ist aber, dass man nicht zwei Prozessen gleichzeitig erlauben darf, auf demselben Drucker zu drucken. Das Ergebnis wäre Chaos. Durch Spooling können mehrere Prozesse gleichzeitig Ausgaben erzeugen. In diesem Modell reserviert nur der Drucker-Dämon den Drucker. Da der Drucker-Dämon niemals andere Ressourcen verlangt, können wir Deadlocks für den Drucker ausschließen.

Leider lässt sich Spooling nicht für alle Arten von Ressourcen verwenden (Prozessstaben eignen sich dafür z. B. eher schlecht). Außerdem kann die Konkurrenz um Plattenplatz im Spooling-Verzeichnis auch wieder zu Deadlocks führen. Was würde wohl passieren, wenn zwei Prozesse jeweils die Hälfte des verfügbaren Spooling-Speichers mit ihrer Ausgabe füllen würden und noch nicht fertig wären? Wenn der Dämon schon zu drucken beginnt, bevor die gesamte Ausgabe vorliegt, könnte der Drucker stundenlang leer laufen, bis es dem Prozess einfällt, die zweite Hälfte seiner Daten zu produzieren. Aus diesem Grund sind Drucker-Dämonen meist so programmiert, dass sie erst mit dem Drucken beginnen, wenn die gesamte Ausgabedatei vorliegt. In diesem Fall haben wir aber zwei Prozesse, die beide nur einen Teil ihrer Ausgabe beendet haben und nicht weiterlaufen können. Keiner der beiden wird jemals fertig werden und wir haben einen Deadlock.

Trotzdem ist dies der Ansatz zu einer Idee, die sich häufig anwenden lässt: Ressourcen nur zuzuteilen, wenn es unbedingt nötig ist, und dafür zu sorgen, dass so wenig Prozesse wie möglich die Ressource selbst anfordern.

3.6.2 Angriff auf die hold-and-wait-Bedingung

Die zweite Voraussetzung von Coffman et al. sieht etwas vielversprechender aus. Wenn wir verhindern, dass Prozesse auf Ressourcen warten, während sie andere Ressourcen belegen, gibt es keine Deadlocks mehr. Ein mögliches Vorgehen ist, zu verlangen, dass jeder Prozess alle benötigten Ressourcen im Voraus anfordert. Wenn die Ressourcen verfügbar sind, werden sie ihm zugeteilt und er kann ausgeführt werden. Wenn ein oder mehrere Ressourcen belegt sind, werden keine Ressourcen reserviert und der Prozess muss warten.

Ein offensichtliches Problem bei diesem Ansatz ist, dass die meisten Prozesse nicht im Voraus wissen, wie viele Ressourcen sie benötigen werden. Wenn sie es wüssten, könnte man ja den Bankier-Algorithmus verwenden. Außerdem werden die Ressourcen auf diese Art nicht effizient genutzt. Nehmen wir als Beispiel einen Prozess, der Daten von einem Band liest, dann eine Stunde rechnet und anschließend das Ergebnis ausdruckt. Wenn alle Ressourcen im Voraus angefordert werden müssen, belegt der Prozess den Drucker für eine zusätzliche Stunde.

Trotzdem verlangen die Betriebssysteme einiger Großrechner, dass der Benutzer in die erste Zeile eines Auftrags die benötigten Ressourcen einträgt. Das System belegt die Ressourcen dann sofort und gibt sie erst wieder frei, wenn der Auftrag erledigt ist. Diese Methode belastet die Programmierer und verschwendet Ressourcen, aber man muss zugeben, dass sie Deadlocks vermeidet.

Eine etwas andere Art, der hold-and-wait-Bedingung beizukommen, sieht vor, von einem Prozess zu verlangen, vor einer Anforderung alle seine Ressourcen kurzzeitig freizugeben und dann alles auf einmal zu reservieren.

3.6.3 Angriff auf die Ununterbrechbarkeit

Der Angriff auf die dritte Bedingung (Ununterbrechbarkeit) verspricht noch weniger Erfolg als der auf die zweite. Einem Prozess gewaltsam einen Drucker zu entziehen, auf dem er gerade etwas ausdruckt, ist bestenfalls schwierig und schlimmstenfalls unmöglich.

3.6.4 Angriff auf die zyklische Wartebedingung

Eine letzte Bedingung bleibt noch übrig. Die zyklische Wartebedingung kann auf mehrere Arten beseitigt werden. Eine Möglichkeit ist es, jedem Prozess nur eine Ressource gleichzeitig zu erlauben. Wenn er eine zweite benötigt, muss er zunächst die erste wieder freigeben. Für einen Prozess, der eine größere Datei von einem Bandlaufwerk auf die Festplatte kopieren will, ist diese Bedingung nicht annehmbar.

Eine andere Möglichkeit ist es, die Ressourcen wie in Abbildung 3.13(b) durchnummerieren. Jeder Prozess kann jetzt Ressourcen anfordern, wann immer er will, allerdings muss er es in aufsteigender Reihenfolge tun. Ein Prozess darf zuerst einen Plotter und

dann ein Bandlaufwerk anfordern, aber nicht zuerst einen Plotter und dann einen Scanner.

Abbildung 3.13 (a) Numerisch geordnete Ressourcen. (b) Ein Belegungs-Anforderungs-Graph.



Mit dieser Regel bleibt der Belegungs-Anforderungs-Graph immer zyklensfrei. Überlegen wir uns dies für den Fall von zwei Prozessen (Abbildung 3.13(b)). Ein Deadlock ist nur möglich, wenn A die Ressource j verlangt und B die Ressource i . Angenommen, i und j sind verschiedene Ressourcen und haben deshalb verschiedene Nummern. Wenn $i > j$ ist, darf A j nicht verlangen, weil er schon eine Ressource mit einer höheren Nummer besitzt. Wenn $i < j$ ist, darf B i nicht verlangen, weil er schon eine Ressource mit einer höheren Nummer besitzt. In beiden Fällen ist ein Deadlock unmöglich.

Die gleiche Überlegung gilt für mehrere Prozesse. Zu jedem Zeitpunkt hat eine der reservierten Ressourcen die höchste Nummer. Der Prozess, der diese Ressource belegt, wird niemals eine Ressource verlangen, die schon vergeben ist. Er wird entweder zu Ende laufen oder schlimmstenfalls eine Ressource verlangen, deren Nummer noch höher ist. Am Ende gibt er die belegten Ressourcen frei und ein anderer Prozess hat nun die Ressource mit der höchsten Nummer. Es existiert also eine Reihenfolge, in der alle Prozesse zu Ende laufen können, und es gibt keine Deadlocks.

Eine kleine Variante von diesem Algorithmus ist es, die Forderung fallen zu lassen, Ressourcen in streng aufsteigender Reihenfolge anzufordern. Stattdessen wird nur noch verlangt, dass kein Prozess eine Ressource anfordert, die eine kleinere Nummer hat als die schon reservierten Ressourcen. Wenn ein Prozess 9 und 10 reserviert und sie anschließend wieder freigibt, fängt er eigentlich wieder von vorne an. Es gibt also keinen Grund, warum er nicht als Nächstes 1 anfordern sollte.

Obwohl das Durchnummerieren der Ressourcen das Problem der Deadlocks beseitigt, könnte es unmöglich sein, eine Ordnung zu finden, mit der jeder zufrieden ist. Es gibt so viele verschiedene Anwendungsmöglichkeiten und Ressourcen, z. B. Prozesstabelleneinträge, gesperrte Datensätze und andere abstrakte Ressourcen, dass keine Ordnung für jeden funktionieren würde.

Abbildung 3.14 fasst die verschiedenen Ansätze zur Deadlock-Vermeidung noch einmal zusammen.

Abbildung 3.14 Die verschiedenen Ansätze zur Deadlock-Vermeidung.

Zustand	Versuch
Wechselseitig	Alles spoolen
Hold-and-Wait	Ressourcen zu Beginn anfordern
Ununterbrechbar	Ressourcen wegnehmen
Zyklisches Warten	Ressourcen nummerieren

3.7 Sonstige Punkte zu Deadlocks

In diesem Abschnitt behandeln wir einige Punkte, die mit Deadlocks verwandt sind, darunter Two-Phase Locking, Deadlocks ohne Ressourcen und Verhungern.

3.7.1 Two-Phase Locking

Obwohl weder das Verhindern noch das Vermeiden von Deadlocks im allgemeinen Fall besonders vielversprechend sind, gibt es viele hervorragende Algorithmen für spezielle Anwendungen. In vielen Datenbanksystemen kommt es zum Beispiel vor, dass ein Prozess mehrere Datensätze sperrt, um sie dann zu bearbeiten. Wenn mehrere Prozesse gleichzeitig auf die Datenbank zugreifen, ist die Gefahr von Deadlocks relativ hoch.

In diesem Fall wird häufig so genanntes **Two-Phase Locking** (zweistufiges Sperren) eingesetzt. In der ersten Phase versucht der Prozess, alle Datensätze, die er bearbeiten will, der Reihe nach zu sperren. Wenn das funktioniert, beginnt er mit der zweiten Phase, in der er die Datensätze bearbeitet und wieder freigibt. In der ersten Phase wird keine wirkliche Arbeit erledigt.

Wenn der Prozess in der ersten Phase auf einen gesperrten Datensatz stößt, gibt er alle Datensätze frei und fängt nochmals von vorne an. Man könnte sagen, dass dies so ähnlich ist, als ob der Prozess alle benötigten Ressourcen im Voraus reserviert. Zumindest reserviert er sie, bevor etwas passiert, was nicht mehr rückgängig gemacht werden kann. Es gibt auch Varianten, bei denen der Prozess nicht von vorne anfängt, falls er auf eine Sperre trifft. In diesem Fall sind Deadlocks möglich.

Diese Strategie ist allerdings nicht immer anwendbar. In Echtzeit- oder Prozesskontrollsystemen kann man beispielsweise nicht einfach einen Prozess abbrechen und neu starten, nur weil eine Ressource belegt ist. Genauso wenig kann man einen Prozess neu starten, der schon Nachrichten über das Netz geschickt, Dateien verändert oder irgendwas anderes gemacht hat, was nicht ohne Nebeneffekte wiederholt werden kann.

Der Algorithmus funktioniert nur in Situationen, in denen der Programmierer dafür gesorgt hat, dass das Programm in der ersten Phase jederzeit unterbrochen und neu gestartet werden kann. Für viele Anwendungen ist das unmöglich.

3.7.2 Deadlocks ohne Ressourcen

Bisher haben wir uns auf Deadlocks im Zusammenhang mit Ressourcen konzentriert. Ein Prozess braucht eine Ressource, die ein anderer belegt, und muss warten, bis dieser sie freigibt. Deadlocks können aber auch in anderen Situationen vorkommen, darunter solche, an denen gar keine Ressourcen beteiligt sind.

Beispielsweise kann ein Deadlock entstehen, wenn zwei Prozesse jeweils darauf warten, dass der andere etwas unternimmt. Semaphoren sind in diesem Zusammenhang gefährlich. In Kapitel 2 haben wir einige Beispiele gesehen, in denen Prozesse eine down-Operation auf zwei Semaphoren ausführen mussten. Falls ein Prozess die Operationen in der falschen Reihenfolge ausführt, können Deadlocks entstehen.

3.7.3 Verhungern

Ein eng mit Deadlocks verwandtes Problem ist das so genannte **Verhungern**¹³. In einem dynamischen System kommt es ständig zu Ressourcenanforderungen. Das System entscheidet nach einer gewissen Strategie, welcher Prozess zu welchem Zeitpunkt welche Ressource bekommt. Diese Strategie kann absolut vernünftig erscheinen, aber doch dazu führen, dass ein Prozess niemals ablaufen kann, obwohl er nicht in einem Deadlock steckt.

Nehmen wir als Beispiel die Zuteilung eines Druckers. Angenommen, das System versucht mit irgendeinem Algorithmus sicherzustellen, dass durch die Druckerzuteilung keine Deadlocks entstehen. Wenn nun mehrere Prozesse gleichzeitig den Drucker verlangen, wer sollte ihn bekommen?

Eine Möglichkeit wäre, den Prozess mit der kleinsten Datei zuerst drucken zu lassen (falls diese Information bekannt ist). Dieser Prozess erzeugt eine maximale Anzahl von glücklichen Kunden und scheint fair zu sein. Nehmen wir jetzt an, dass der Drucker ständig zu tun hat und ein Prozess eine riesige Datei ausdrucken will. Nach jedem Druckauftrag sieht sich das System um und wählt den Prozess mit dem kürzesten Auftrag. Solange der Strom von kurzen Dateien nicht abreißt, wird der Prozess mit der riesigen Datei niemals den Drucker bekommen. Der Prozess verhungert, d.h., seine Ausführung wird unendlich aufgeschoben, obwohl er sich in keinem Deadlock befindet.

Das Verhungern lässt sich durch eine FCFS-Strategie (first come, first serve) verhindern. Bei diesem Ansatz wird immer der Prozess ausgewählt, der am längsten wartet. Irgendwann wird jeder Prozess einmal der, der am längsten wartet, und bekommt also die benötigte Ressource.

3.8 Forschung über Deadlocks

Wenn in den frühen Jahren der Betriebssysteme ein Gebiet gnadenlos untersucht wurde, dann waren es Deadlocks. Der Grund dafür ist, dass Deadlocks ein hübsches kleines Problem aus der Graphentheorie sind, in das sich ein mathematisch orientierter Doktorand bequem drei bis vier Jahre lang verbeißen kann. Alle möglichen Algorithmen wurden

¹³ engl.: starvation.

ausgeklügelt, einer exotischer und unpraktischer als der andere. Inzwischen ist die Deadlock-Forschung weitgehend beendet, nur hin und wieder erscheint noch ein Artikel (z. B. Karacali et al., 2000). Die wenigsten Systeme versuchen, Deadlocks zu erkennen oder zu verhindern, und wenn sie es tun, benutzen sie eine der Methoden aus diesem Kapitel.

Verteilte Deadlocks sind allerdings immer noch ein aktuelles Thema, das hier jedoch nicht behandelt wird, weil es erstens nicht zum Thema dieses Buches gehört und zweitens nicht einmal im entferntesten praktisch anwendbar ist. Verteilte Deadlocks sind hauptsächlich eine Arbeitsbeschaffungsmaßnahme für arbeitslose Graphentheoretiker.

3.9 Zusammenfassung

Deadlocks können in jedem Betriebssystem ein Problem sein. Sie entstehen, wenn in einer Gruppe von Prozessen jeder Prozess das alleinige Zugriffsrecht auf eine Ressource besitzt und jeder noch eine weitere Ressource benötigt, die einem anderen Prozess aus der Gruppe gehört. Alle Prozesse in der Gruppe sind blockiert und können niemals weiterlaufen. Deadlocks können verhindert werden, indem das System mitverfolgt, welche Zustände sicher und unsicher sind. Ein sicherer Zustand bedeutet, dass es eine Reihenfolge von Ereignissen gibt, die garantiert, dass alle Prozesse zu Ende laufen können. Ein unsicherer Zustand bietet keine solche Garantie. Der Bankier-Algorithmus verhindert Deadlocks, indem er Ressourcenanforderungen nur erfüllt, wenn sie zu einem sicheren Zustand führen.

Deadlocks können von vornherein vermieden werden, indem das System so entworfen wird, dass Deadlocks niemals auftreten können. Wenn z. B. jeder Prozess nur eine Ressource gleichzeitig belegen darf, wird die zyklische Wartebedingung unerfüllbar und Deadlocks sind unmöglich. Deadlocks können auch vermieden werden, indem man alle Ressourcen durchnummeriert und verlangt, dass jeder Prozess sie nur in streng aufsteigender Reihenfolge anfordert. Verhungern kann durch eine FCFS-Strategie vermieden werden.

ÜBUNGEN

1. Denken Sie sich ein Beispiel für einen Deadlock in der Politik aus.
2. Die Studenten in einem Computerraum arbeiten an einzelnen Rechnern. Um Dateien auszudrucken, senden sie sie an einen Server, der sie in einem Spooling-Verzeichnis zwischenspeichert. Unter welchen Bedingungen können Deadlocks auftreten, wenn der Plattenplatz auf dem Server beschränkt ist? Wie könnten die Deadlocks verhindert werden?
3. Welche Ressourcen sind in der vorigen Frage unterbrechbar und welche sind ununterbrechbar?
4. In Abbildung 3.1 werden die Ressourcen in der umgekehrten Reihenfolge freigegeben, in der sie reserviert wurden. Wäre es genauso gut, sie in einer anderen Reihenfolge freizugeben?
5. Abbildung 3.3 illustriert das Konzept eines Belegungs-Anforderungs-Graphen. Gibt es auch illegale Graphen, d.h. Graphen, die unserem Modell der Ressourcennutzung widersprechen? Wenn ja, geben Sie ein Beispiel.
6. Beim Vogel-Strauß-Algorithmus wurde die Möglichkeit erwähnt, dass die Prozesstabelle oder andere Systemtabellen voll sind und keine Einträge mehr reserviert werden können. Können Sie sich vorstellen, wie ein Administrator eine solche Situation beheben könnte?
7. Sehen wir uns Abbildung 3.4 noch einmal an. Nehmen wir an, C verlangt im Schritt (o) die Ressource S anstatt R . Ergibt sich dadurch ein Deadlock? Was ist, wenn er S und R verlangt?
8. An einer Kreuzung steht an allen vier Zufahrten ein Stoppschild. Nach den Verkehrsregeln hat immer der von rechts Kommende Vorfahrt. Wenn aber gleichzeitig aus allen Richtungen Autos kommen, ist diese Regel nicht anwendbar. Zum Glück sind Autofahrer manchmal intelligenter als Computer und das Problem wird normalerweise dadurch gelöst, dass ein Fahrer einen von links kommenden Fahrer herüberwinkt. Gibt es eine Analogie zwischen diesem Verhalten und einer der Vorgehensweisen, einen Deadlock zu beheben, aus Abschnitt 3.4.3?
9. Angenommen, in Abbildung 3.6 sei $C_{ij} + R_{ij} > E_j$ für irgendein i . Was bedeutet dies für alle Prozesse, die ohne Deadlock zu Ende laufen?
10. In Abbildung 3.8 verlaufen alle Spuren horizontal oder vertikal. Können Sie sich eine Situation vorstellen, in der sie auch diagonal verlaufen können?
11. Kann das Schema aus Abbildung 3.8 auch verwendet werden, um das Problem der Deadlocks für drei Prozesse und drei Ressourcen zu veranschaulichen? Wenn ja, was müsste man ändern? Wenn nicht, warum funktioniert es nicht?
12. Theoretisch könnte man die Graphen von Ressourcenspuren verwenden, um Deadlocks zu verhindern. Das Betriebssystem könnte die unsicheren Bereiche durch geschicktes Scheduling umgehen. Geben Sie ein Beispiel für ein Problem, das sich dabei in der Praxis stellen könnte.

13. Sehen Sie sich Abbildung 3.11(b) noch einmal gut an. Führt es zu einem sicheren oder unsicheren Zustand, wenn D noch eine weitere Ressource verlangt? Was wäre, wenn C die Ressource verlangen würde?
14. Kann sich ein System in einem Zustand befinden, der weder sicher noch ein Deadlock ist? Wenn ja, geben Sie ein Beispiel. Wenn nicht, beweisen Sie, dass jeder Zustand entweder sicher oder ein Deadlock ist.
15. Ein System hat zwei Prozesse und drei identische Ressourcen. Jeder Prozess braucht maximal zwei Ressourcen. Ist ein Deadlock möglich? Begründen Sie Ihre Antwort.
16. Betrachten wir noch einmal das vorige Problem, aber jetzt mit p Prozessen und r Ressourcen, von denen jeder Prozess höchstens m benötigt. Welche Bedingung muss gelten, damit Deadlocks unmöglich sind?
17. Führt es zu einem Deadlock, wenn Prozess A in Abbildung 3.12 das letzte Bandlaufwerk verlangt?
18. Ein Computer hat sechs Bandlaufwerke, um die sich n Prozesse streiten. Jeder Prozess braucht bis zu zwei Laufwerke. Für welche n kann es keinen Deadlock geben?
19. Ein System mit m Ressourcenklassen und n Prozessen benutzt den Bankier-Algorithmus. Wenn m und n gegen unendlich gehen, ist die Rechenzeit für eine Zustandsüberprüfung von der Ordnung $m^a n^b$. Welche Werte haben a und b ?
20. Ein System hat vier Prozesse und fünf reservierbare Ressourcen. Die folgende Tabelle zeigt, welche Ressourcen belegt sind und wie viele Ressourcen maximal benötigt werden:

	<i>Belegt</i>	<i>Maximal</i>	<i>Verfügbar</i>
Prozess A	1 0 2 1 1	1 1 2 1 3	0 0 x 1 1
Prozess B	2 0 1 1 0	2 2 2 1 0	
Prozess C	1 1 0 1 0	2 1 3 1 0	
Prozess D	1 1 1 1 0	1 1 2 2 1	

Was ist das kleinste x , für das der Zustand sicher ist?

21. Ein verteiltes System, das Mailboxen zur Kommunikation verwendet, hat zwei IPC-Aufrufe, `send` und `receive`. Beim Aufruf von `receive` muss der Absenderprozess angegeben werden. `receive` blockiert, wenn keine Nachricht von dem angegebenen Prozess angekommen ist, auch wenn Nachrichten von anderen Prozessen in der Mailbox warten. Es gibt keine gemeinsamen Ressourcen, aber Prozesse müssen aus anderen Gründen häufig kommunizieren. Überlegen Sie, ob Deadlocks möglich sind.
22. Die Prozesse A und B möchten beide die Datensätze 1, 2 und 3 in einer Datenbank sperren. Wenn beide sie in der Reihenfolge 1, 2, 3 anfordern, ist kein Deadlock möglich. Wenn B sie aber in der Reihenfolge 3, 2, 1 anfordert, kann ein Deadlock auftreten. Es gibt 3! oder sechs Möglichkeiten für jeden Prozess,

- drei Ressourcen zu reservieren. Welche Teilmenge aller Kombinationen ist garantiert Deadlock-frei?
23. Betrachten wir noch einmal das vorige Beispiel, aber diesmal mit Two-Phase-Locking. Sind dadurch Deadlocks ausgeschlossen? Gibt es andere Nachteile?
 24. In einem elektronischen Überweisungssystem laufen Hunderte von identischen Prozessen. Jeder Prozess liest die Überweisungssumme und die beiden Kontonummern von einem Eingabegerät. Anschließend sperrt er beide Konten, überweist das Geld und gibt die Konten wieder frei. Die Gefahr von Deadlocks in diesem System ist durch die vielen parallelen Prozesse sehr hoch. Lassen Sie sich eine Strategie einfallen, um Deadlocks zu verhindern. Dabei darf kein Konto freigegeben werden, bevor die Überweisung beendet ist. (Mit anderen Worten, eine Lösung, die ein Konto sperrt und es sofort wieder freigibt, falls das zweite gesperrt ist, gilt nicht.)
 25. Eine Möglichkeit, Deadlocks zu vermeiden, ist, die hold-and-wait-Bedingung unerfüllbar zu machen. Im Text haben wir die Möglichkeit vorgeschlagen, dass ein Prozess, bevor er eine Ressource anfordert, alle seine Ressourcen freigeben muss (wenn möglich). Die Gefahr dabei ist, dass er die neue Ressource bekommt, aber eine der anderen verliert. Wie könnte man diese Strategie verbessern?
 26. Ein Student, der an einem Projekt über Deadlocks arbeitet, hat folgende geniale Idee, um Deadlocks zu beseitigen: Wenn ein Prozess eine Ressource anfordert, gibt er ein Zeitlimit an. Wenn die Ressource nicht verfügbar ist, wird ein Timer gestartet. Nach überschreiten des Zeitlimits wird der Prozess freigegeben und darf weiterlaufen. Stellen Sie sich vor, Sie wären der Professor. Benoten Sie den Vorschlag und begründen Sie Ihre Entscheidung.
 27. Aschenputtel und der Prinz lassen sich scheiden. Für die Gütertrennung haben sie sich auf folgenden Algorithmus geeinigt. Jeden Morgen schickt jeder von ihnen einen Brief an den Anwalt des anderen, der Anspruch auf einen Gegenstand erhebt. Es dauert einen Tag, bis ein Brief ankommt, deshalb haben sie festgelegt, dass sie, wenn beide am selben Tag denselben Gegenstand verlangen, am nächsten Tag einen Brief schicken, der die Forderung rückgängig macht. Unter anderem besitzen die beiden einen Hund, Woofer, seine Hundehütte, einen Kanarienvogel, Tweeter, und dessen Käfig. Der Hund hängt sehr an seiner Hütte, ebenso wie der Vogel an seinem Käfig, deshalb hat man sich geeinigt, dass eine Gütertrennung ungültig ist, wenn sie ein Tier von seiner Behausung trennt. In diesem Fall fängt die Trennung wieder von vorne an. Aschenputtel und der Prinz wollen beide dringend den Hund haben. Damit sie (getrennt) in Urlaub fahren können, hat jeder von ihnen seinen PC programmiert, um die Verhandlungen zu führen. Bei ihrer Rückkehr verhandeln die Computer immer noch. Warum? Ist ein Deadlock möglich? Ist Verhungern möglich?
 28. Ein Anthropologie-Student mit Nebenfach Informatik will in einem Forschungsprojekt herausfinden, ob man afrikanischen Pavianen beibringen kann,

Deadlocks zu vermeiden. Er sucht eine tiefe Schlucht und spannt ein Seil, so dass sich die Paviane darüber hangeln können. Solange alle in die gleiche Richtung hangeln, können mehrere Paviane die Schlucht gleichzeitig überqueren. Wenn Paviane von Osten und von Westen gleichzeitig mit der Überquerung beginnen, entsteht ein Deadlock (die Paviane hängen in der Mitte fest), weil sie nicht übereinander klettern können. Wenn ein Pavian die Schlucht überqueren will, muss er erst feststellen, ob kein anderer Pavian gerade in der Gegenrichtung unterwegs ist. Schreiben Sie ein Programm, das Semaphoren benutzt, um Deadlocks zu vermeiden. Kümmern Sie sich nicht darum, dass eine Folge von ostwärts hangelnden Pavianen die im Osten wartenden für unbestimmte Zeit aufhalten könnte.

29. Wiederholen Sie die vorige Aufgabe und vermeiden Sie diesmal das Verhungern. Wenn ein Pavian, der nach Osten will, sieht, dass gerade Paviane nach Westen unterwegs sind, wartet er, bis das Seil frei ist. Es dürfen aber keine neuen Paviane von Osten auf das Seil, bis mindestens einer die Schlucht von Westen überquert hat.
30. Schreiben Sie eine Simulation des Bankier-Algorithmus. Das Programm soll immer wieder alle Kunden nach ihren Kreditwünschen fragen und berechnen, ob der Kredit zu einem sicheren oder unsicheren Zustand führen würde. Speichern Sie die Anforderungen und Entscheidungen in einer Logdatei.