1 Einführung

Prozesse müssen ständig mit anderen Prozessen kommunizieren. In einer Shell-Pipe zum Beispiel muss die Ausgabe des ersten Prozesses an den zweiten Prozess weitergereicht werden und so weiter die Reihe durch. Damit gibt es die Notwendigkeit für eine Kommunikation zwischen Prozessen, vorzugsweise in einer gut strukturierten Weise, die keine Unterbrechungen verwendet. In den folgenden Kapiteln werden wir uns einige Punkte in Zusammenhang mit dieser **Interprozesskommunikation** oder kurz **IPC** ansehen.

Ganz kurz gesagt, gibt es hier drei Punkte. Der erste deutet auf Obiges hin: wie ein Prozess Informationen zu einem anderen weiterreichen kann. Der zweite hat damit zu tun, sicherzustellen, dass zwei oder mehr Prozesse sich nicht gegenseitig in die Quere kommen, wenn kritische Aktivitäten anliegen (nehmen wir an, zwei Prozesse versuchen jeweils, das letzte MB an Speicher zu bekommen). Der dritte betrifft den sauberen Ablauf, wenn Abhängigkeiten vorliegen: Falls Prozess *A* Daten erzeugt und Prozess *B* diese ausgibt, muss *B* so lange warten, bis *A* Daten erzeugt hat, bevor sie ausgegeben werden. Wir werden alle drei Themen betrachten, beginnend im nächsten Abschnitt.

Es ist wichtig zu erwähnen, dass zwei dieser Punkte ebenso gut für Threads gelten. Das erste — Informationen weiterreichen — ist für Threads unkompliziert, da sie ja einen gemeinsamen Adressraum benutzen (Threads, die in verschiedenen Adressräumen kommunizieren müssen, fallen unter die Überschrift der kommunizierenden Prozesse). Die anderen zwei allerdings — sich nicht in die Quere kommen und richtiger Ablauf — gelten ebenso gut für Threads. Es bestehen die gleichen Probleme und es gelten die gleichen Lösungen. Weiter unten werden wir das Problem in Zusammenhang mit Prozessen diskutieren, aber behalten Sie im Hinterkopf, dass die gleichen Probleme und Lösungen auch für Threads gelten.

Erklären was eine Shell-Pipe ist.
Bsp: dir | sort | print
Dir: Filenamen auflisten; sort: Namen sortieren; print namen drucken;

IPC löst 2 grundsätzliche Probleme (Aufgabenstellungen)

1. Wie wird der Zugriff von paralellen Tasks auf gemeinsame Variablen und Speicher geregelt.
2. Wie kann ein Task einen 2-ten benachrichtigen und ihm Daten zur Weiterverarbeitung übergeben.

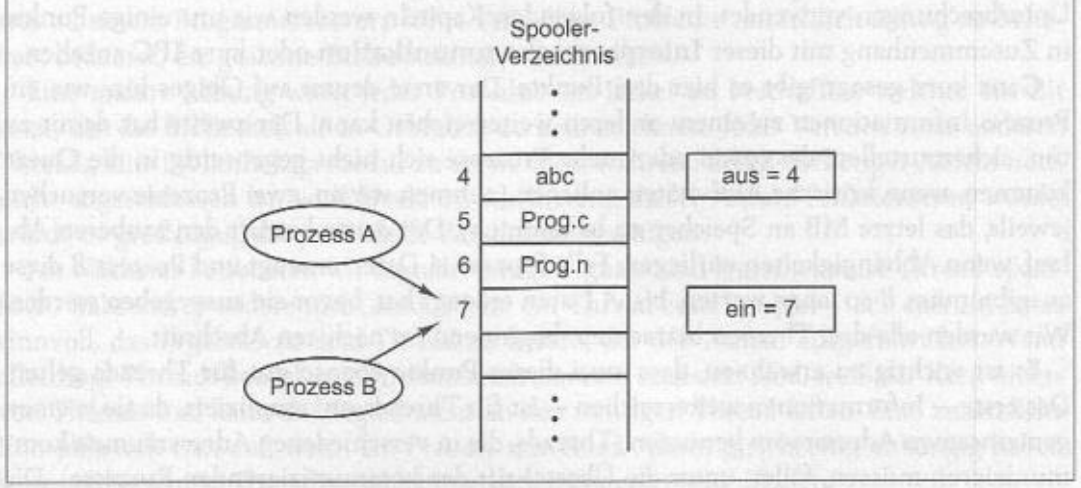
2 Race Conditions

In einigen Betriebssystemen können Prozesse, die miteinander arbeiten, einen gemeinsamen Speicher benutzen, in den jeder schreiben und lesen kann. Der gemeinsame Speicherplatz könnte im Hauptspeicher liegen (möglicherweise eine Kerndatenstruktur) oder es könnte eine gemeinsame Datei sein; der Ort des gemeinsamen Speichers ändert nicht die Natur der Kommunikation oder der auftretenden Probleme. Um zu sehen, wie Interprozesskommunikation arbeitet, betrachten wir ein einfaches, aber häufiges Beispiel: ein Drucker-Spooler. Wenn ein Prozess eine Datei drucken möchte, trägt er den Dateinamen in einen speziellen Spooler-Ordner ein. Ein anderer Prozess, der **Drucker-Daemon**, überprüft zyklisch, ob es irgendwelche Dateien zu drucken gibt, und wenn es welche gibt, druckt er diese aus und entfernt danach ihren Namen wieder aus dem Ordner.

Stellen Sie sich vor, unser Spooler-Ordner hat eine sehr große Anzahl von Einträgen, mit 0, 1, 2, ... nummeriert, jeder kann einen Dateinamen aufnehmen. Stellen Sie sich weiterhin vor, es gibt zwei gemeinsame Variablen, *out*, welche auf die nächste zu druckende Datei zeigt, und *in*, welche auf den nächsten freien Eintrag im Ordner zeigt. Diese zwei Variablen könnten gut in einer unter allen Prozessen verfügbaren Zwei-Wort-Datei gehalten werden. Unter gewissen Umständen sind die Einträge 0 bis 3 leer (die Dateien wurden bereits ausgedruckt) und die Einträge 4 bis 6 sind belegt (mit Namen von Dateien, die gedruckt werden sollen). Prozess *A* und *B* entscheiden mehr oder we-

niger gleichzeitig, dass sie eine Datei zum Ausdruck einreihen wollen. Diese Situation wird in Abbildung 2.18 dargestellt.

Abbildung 2.18 Zwei Prozesse möchten auf gemeinsam genutzten Speicher gleichzeitig zugreifen.



Gemäß Murphys Gesetz[5] könnte Folgendes passieren. Der Prozess *A* liest *in* aus und speichert den Wert, 7, in einer lokalen Variablen, genannt *next_free_slot*. Genau dann tritt eine Unterbrechung durch die Uhr auf und die CPU entscheidet, dass der Prozess *A* nun lange genug gelaufen ist, und wechselt deshalb zum Prozess *B*. Der Prozess *B* liest ebenfalls *in* aus und bekommt auch eine 7 geliefert. Er speichert sie auch in *seiner* lokalen Variablen *next_free_slot*. Unter diesen Umständen glauben beide Prozesse, dass der nächste verfügbare Eintrag 7 ist.

Prozess *B* läuft nun weiter. Er speichert den Namen seiner Datei im Eintrag 7 und aktualisiert *in* auf 8. Dann ist er fertig und tut andere Dinge.

Prozess *A* läuft schließlich wieder, und zwar ab der Stelle, an der die Unterbrechung stattfand. Er schaut in *next_free_slot*, findet dort eine 7 vor und schreibt seinen Dateinamen in den 7. Eintrag, wobei er den Namen, den Prozess *B* gerade dort abgelegt hat, löscht. Dann berechnet er *next_free_slot* + 1, was 8 ergibt, und setzt *in* auf 8. Der Spooler-Ordner ist nun intern konsistent, so dass der Drucker-Daemon nichts Falsches bemerkt, aber Prozess *B* wird niemals eine Ausgabe erhalten. Der Benutzer *B* wird jahrelang im Druckerraum herumhängen, wehmütig auf eine Ausgabe hoffend, die niemals kommen wird. Situationen wie diese, in denen zwei oder mehr Prozesse einen gemeinsamen Speicher lesen oder beschreiben und das Endergebnis davon abhängt, wer wann genau läuft, werden **Race Conditions** genannt. Programme, die Race Conditions enthalten, zu debuggen, macht überhaupt keinen Spaß. Die Ergebnisse der meisten Testläufe sind in Ordnung, bis auf Ausnahmen, in denen etwas Eigenartiges und Unerklärliches geschieht.

5 Wenn etwas schiefgehen kann, wird es auch schiefgehen.

2.1 Besseres Beispiel von HL

Der obestehende Buchauszug ist als Einführung in die Problematik der RaceConditions gut geeignet.
Bessere Beispiele zu RaceConditions findet man in:
Siehe 5. Beispiele zu Race-Conditions

3 Kritische Abschnitte

2.3.2 Kritische Abschnitte

Wie vermeiden wir Race Conditions? Der Schlüssel zur Vermeidung von Ärger, hier und in vielen anderen Situationen, mit gemeinsam genutztem Speicher, Dateien und anderen gemeinsam genutzten Dingen, ist es, einen Weg zu finden, der es einem Prozess verbietet, gleichzeitig mit einem anderen die gemeinsam genutzten Daten zu lesen oder zu beschreiben. Mit anderen Worten, was wir brauchen, ist **wechselseitiger Ausschluss**. Das bedeutet, dass ein Prozess eine gemeinsam genutzte Variable oder Datei nutzen kann, ohne dass ein anderer Prozess das Gleiche tun kann. Die obige Schwierigkeit trat deshalb auf, weil der Prozess *B* anfing, eine der gemeinsam genutzten Variablen zu verwenden, bevor Prozess *A* damit fertig war. Die Wahl entsprechend primitiver Operationen, um wechselseitigen Ausschluss zu erzielen, ist eine der großen Entwurfsfragen in jedem Betriebssystem, und ein Thema, welches wir im Detail in den folgenden Kapiteln untersuchen werden.

Das Problem, Race Conditions zu vermeiden, kann auch auf einem abstrakten Weg ausgedrückt werden. Ein Teil der Zeit, die ein Prozess damit beschäftigt ist, interne Berechnungen und andere Dinge zu tun, führt zu keinen Race Conditions. Allerdings muss ein Prozess manchmal auf gemeinsam genutzten Speicher oder Dateien zugreifen oder andere kritische Dinge machen, die dann zu diesen Wettläufen führen können. Die Teile des Programms, in denen auf gemeinsam genutzten Speicher zugegriffen wird, nennt man **kritische Region** oder **kritischen Abschnitt**. Wenn wir die Angelegenheiten so anordnen könnten, dass niemals zwei Prozesse gleichzeitig in ihren kritischen Regionen sind, ließen sich Race Conditions vermeiden.

Auch wenn dies Race Conditions vermeidet, reicht es nicht aus, parallele Prozesse richtig und effizient zusammenarbeiten zu lassen, wenn sie gemeinsam genutzte Daten verwenden. Wir müssen vier Bedingungen einhalten, um eine gute Lösung zu bekommen:

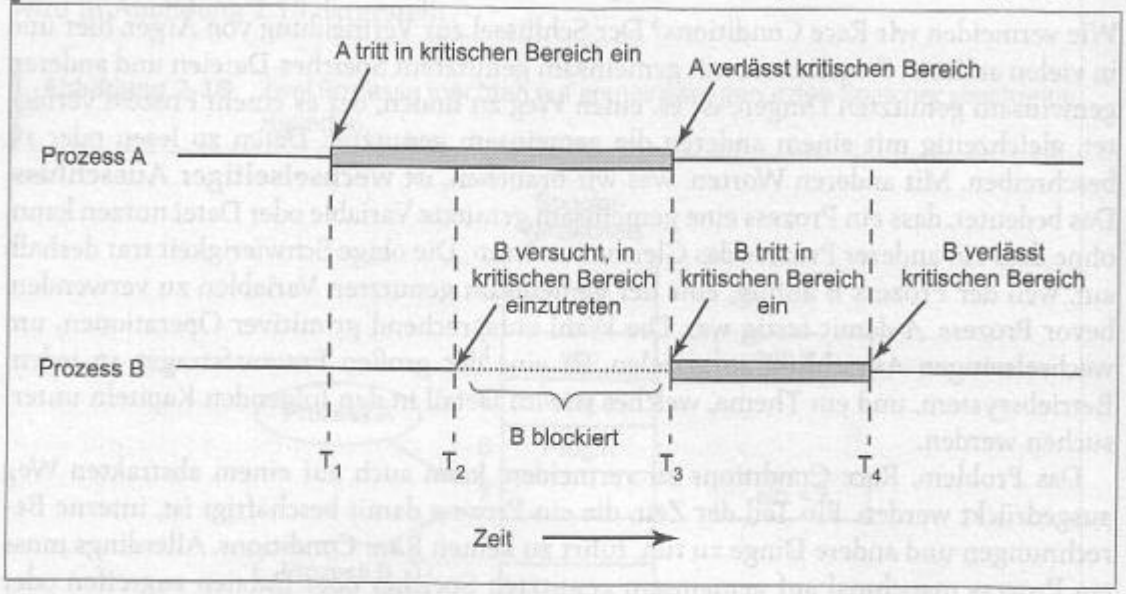
1. Keine zwei Prozesse dürfen gleichzeitig in ihren kritischen Regionen sein.
2. Es dürfen keine Annahmen über Geschwindigkeit und Anzahl der CPUs gemacht werden.
3. Kein Prozess, der außerhalb seiner kritischen Region läuft, darf andere Prozesse blockieren.
4. Kein Prozess sollte ewig darauf warten müssen, in seine kritische Region einzutreten.

Auf eine abstrakte Weise zeigen wir das gewünschte Verhalten in Abbildung 2.19. Hier tritt Prozess *A* in seine kritische Region zum Zeitpunkt T_1. Ein bisschen später, zum Zeitpunkt T_2, versucht der Prozess *B*, in seine kritische Region einzutreten, scheitert aber, da sich ein anderer Prozess bereits in seiner kritischen Region befindet und wir das nun mal nur einem erlauben. Folglich wird *B* vorübergehend schlafen gelegt, bis zum Zeitpunkt T_3, wenn *A* seine kritische Region verlässt und *B* sofort erlaubt wird einzutreten.

2.3.3 Wechselseitiger Ausschluss mit aktivem Warten

In diesem Abschnitt werden wir verschiedene Vorschläge prüfen, um wechselseitigen Ausschluss zu erreichen. Während ein Prozess damit beschäftigt ist, in seiner kritischen

Abbildung 2.19 Wechselseitiger Ausschluss unter Verwendung von kritischen Regionen.



Region den gemeinsam genutzten Speicher zu aktualisieren, kann also kein anderer Prozess in *seine* kritische Region eintreten und Ärger verursachen.

Unterbrechungen ausschalten

Die einfachste Lösung besteht darin, jeden Prozess alle Unterbrechungen ausschalten zu lassen, gerade nachdem er in seine kritische Region eingetreten ist, und sie wieder einzuschalten, kurz bevor er sie verlässt. Mit ausgeschalteten Unterbrechungen kann keine Unterbrechung durch die Uhr auftreten. Die CPU wird immer nur durch ein Ereignis vom Taktgeber oder andere Unterbrechungen von Prozess zu Prozess weitergeschaltet. Mit ausgeschalteten Unterbrechungen wird die CPU nicht zu einem anderen Prozess wechseln. Damit kann der Prozess, wenn einmal die Unterbrechungen abgeschaltet sind, den gemeinsam genutzten Speicher überprüfen und aktualisieren, ohne Angst haben zu müssen, dass ein anderer Prozess dazwischenkommt.

Dieser Ansatz ist im Allgemeinen unattraktiv, weil es unklug wäre, einem Benutzerprozess die Macht zu geben, Unterbrechungen abzuschalten. Nehmen Sie an, einer von ihnen tut es und schaltet sie nie wieder ein? Das könnte das Ende des Systems bedeuten. Darüber hinaus betrifft die Abschaltung der Unterbrechungen, falls das System ein Mehr-Prozessor-System mit zwei oder mehr CPUs ist, nur diejenige CPU, die die `disable`-Anweisung ausgeführt hat. Die anderen werden weiter laufen und können auf den gemeinsam genutzten Speicher zugreifen.

Auf der anderen Seite ist es für den Kern oft günstig, selbst die Unterbrechungen für ein paar Anweisungen auszuschalten, während er Variablen oder Listen aktualisiert. Falls eine Unterbrechung aufgetreten ist, während zum Beispiel die Liste mit bereiten Prozessen inkonsistent war, könnten Race Conditions auftreten. Daher ist der Schluss zu ziehen: Unterbrechungen abzuschalten, ist oft eine nützliche Technik innerhalb des Betriebssystems selbst, ist aber nicht als ein allgemeiner gegenseitiger Ausschlussmechanismus für Benutzerprozesse geeignet.

3.1 Unterbrechungen ausschalten Kommentar

- funktioniert
- gibt den Benutzer (C Programmierer) aber eine Möglichkeit in die Hand mit der er das ganze System lahmlegen kann => unbrauchbar auf einem PC Betriebssystem.
- Ist aber auf einem uC ohne Betriebssystem aber die beste (und einzige) Möglichkeit
  wenn schon gezielt ausschalten nur TimerISR, nur serielle ISR

Variablen sperren

Lassen Sie uns als zweiten Versuch nach einer Softwarelösung suchen. Betrachten wir die Verwendung einer einzelnen gemeinsam genutzten (Sperr-)Variablen, initialisiert mit 0. Wenn ein Prozess in seine kritische Region eintreten möchte, fragt er zuerst die Sperre ab. Wenn die Sperre 0 ist, setzt der Prozess sie auf 1 und betritt die kritische Region. Falls die Sperre bereits 1 ist, wartet der Prozess einfach, bis sie 0 wird. Somit bedeutet eine 0, dass kein Prozess in seiner kritischen Region ist, und eine 1 bedeutet, dass irgendein Prozess in seiner kritischen Region ist.

Leider enthält diese Idee genau den gleichen verhängnisvollen Fehler, den wir im Spooler-Ordner gesehen haben. Nehmen Sie an, ein Prozess liest die Sperre aus und sieht, dass sie 0 ist. Bevor er die Sperre auf 1 setzen kann, wird ein weiterer Prozess schedult, der die Sperre auf 1 setzt. Wenn der erste Prozess wieder läuft, setzt auch er die Sperre auf 1 und es werden zwei Prozesse gleichzeitig in ihren kritischen Regionen ausgeführt.

Nun könnten Sie annehmen, dass wir um dieses Problem herumkommen, indem wir zuerst den Sperr-Wert auslesen und ihn dann nochmals, kurz bevor wir hineinschreiben, überprüfen, aber das hilft nicht. Dann tritt die Race Condition auf, wenn der zweite Prozess die Sperre modifiziert, kurz nachdem der erste Prozess seine zweite Überprüfung abgeschlossen hat.

Strikter Wechsel

Eine dritte Annäherung an das Problem des wechselseitigen Ausschlusses ist in Abbildung 2.20 zu sehen. Dieser Programmauszug ist, wie nahezu alle anderen in diesem Buch, in C geschrieben. C wurde hier gewählt, weil reale Betriebssysteme fast immer in C geschrieben werden (oder ab und zu in C++), aber

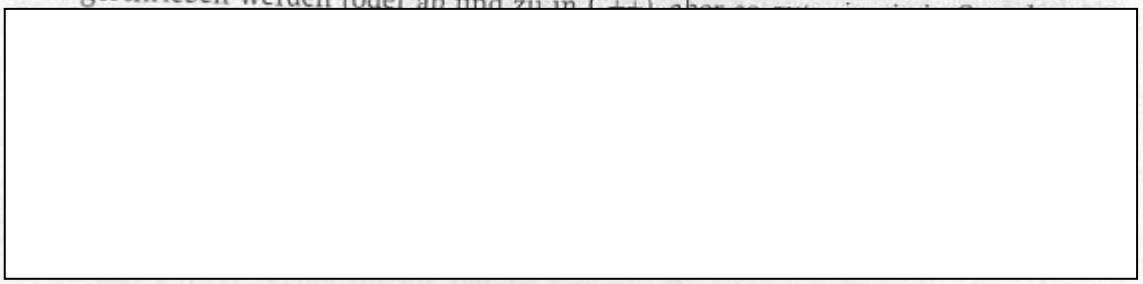


Abbildung 2.20 Eine vorgeschlagene Lösung für das Problem der kritischen Region. (a) Prozess 0. (b) Prozess 1. Stellen Sie in beiden Fällen sicher, dass die While-Anweisungen durch ein Semikolon abgeschlossen sind.

```
while (TRUE) {
    while (turn != 0)   /* Schleife */;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)   /* Schleife */;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

In Abbildung 2.20 verfolgt die ganzzahlige Variable *turn*, initialisiert mit 0, wer an der Reihe ist, in die kritische Region einzutreten und den gemeinsam genutzten Speicher zu überprüfen oder zu aktualisieren. Anfänglich überprüft der Prozess 0 *turn*, stellt fest, dass es 0 ist und betritt die kritische Region. Prozess 1 stellt auch fest, dass der Wert 0 ist und sitzt deshalb in einer Schleife fest, die ständig *turn* untersucht, um zu sehen, wann der Wert 1 wird. Ständiges Überprüfen einer Variable, bis irgendein Wert erscheint, nennt man **aktives Warten**. Es sollte für gewöhnlich vermieden werden, da es CPU-Zeit verschwendet. Nur wenn zu erwarten ist, dass die Wartezeit kurz sein wird, wird aktives Warten verwendet. Eine Sperre, die aktives Warten verwendet, wird **Spinlock** genannt.

Wenn Prozess 0 die kritische Region verlässt, setzt er *turn* auf 1, um es dem Prozess 1 zu erlauben, in seine kritische Region einzutreten. Nehmen Sie an, dass Prozess 1 seine kritische Region schnell beendet, so dass beide Prozesse sich in ihren nicht kritischen Regionen befinden, mit *turn* auf 0 gesetzt. Nun führt Prozess 0 seine gesamte Schleife schnell aus, verlässt seine kritische Region und setzt *turn* auf 1. An diesem Punkt ist *turn* 1 und beide Prozesse führen ihre nicht kritischen Regionen aus.

Plötzlich beendet Prozess 0 seinen nicht kritischen Bereich und kehrt zurück an den Anfang seiner Schleife. Unglücklicherweise ist es ihm nicht erlaubt, in seine kritische Region einzutreten, weil *turn* 1 ist und der Prozess 1 mit seiner nicht kritischen Region beschäftigt ist. Er hängt in seiner `while`-Schleife, bis Prozess 1 *turn* auf 0 setzt. Dieses „an die Reihe nehmen" stellt aber keine gute Idee dar, wenn ein Prozess sehr viel langsamer als der andere ist.

Diese Situation verletzt die Bedingung 3, welche oben gefordert wird: Prozess 0 wird von einem Prozess blockiert, ohne dass dieser in seiner kritischen Region ist. Gehen wir zu dem oben diskutierten Drucker-Ordner zurück, falls wir jetzt die kritische Region mit dem Lesen und Schreiben des Drucker-Ordners verbinden, könnte es Prozess 0 nicht erlaubt werden, eine andere Datei zu drucken, weil Prozess 1 noch etwas anderes tut.

In der Tat erfordert diese Lösung, dass zwei Prozesse sich streng darin abwechseln, ihre kritischen Regionen zu betreten, wie zum Beispiel beim Spooling von Dateien. Keinem würde es erlaubt sein, zwei Dateien hintereinander einzuordnen. Obwohl dieser Algorithmus alle Race Conditions vermeiden würde, ist es kein ernsthafter Anwärter für eine Lösung, weil er Bedingung 3 verletzt.

3.2 Variablen Sperren Kommentar

Wie im Buchtext erklärt enthält dieser Ansatz RaceConditions im Sperralgorithmus.
Die RaceConditions sind mehr oder weniger dieselben wie die in **Kap. 5** erklärten.

3.3 Strikter Wechsel Kommentar

- Funktioniert so schlecht und recht
- Bedingung 3 der oben genannten 4 Bedingungen wird verletzt
- Beim aktiven Warten verschwendet der wartende Thread sinnlos Prozessorzeit.

3.4 Sperrvariablen mithilfe der TSL Anweisung

Die TSL-Anweisung

Betrachten wir jetzt einen Vorschlag, der ein wenig Hilfe von der Hardware fordert. Viele Computer, speziell jene, die mit mehreren Prozessoren arbeiten, haben eine Anweisung

```
TSL RX,LOCK
```

(Test and Set Lock), die wie folgt funktioniert. Sie liest den Inhalt des Speicherworts *lock* ins Register RX und speichert dann einen Wert ungleich Null an die Speicheradresse von *lock*. Das Lesen und das Schreiben des Wortes sind garantiert unteilbare Operationen — kein anderer Prozess kann auf das Speicherwort zugreifen, bis die Anweisung beendet ist. Wenn die CPU die TSL-Answeisung ausführt, wird der Speicherbus gesperrt, um anderen CPUs den Zugriff auf den Speicher so lange zu verbieten, bis er fertig ist.

Um die TSL-Anweisung zu verwenden, werden wir eine gemeinsam genutzte Variable, *lock*, verwenden, um den Zugriff auf gemeinsam genutzten Speicher zu koordinieren. Wenn *lock* 0 ist, könnte jeder Prozess sie auf 1 setzen, indem er die TSL-Anweisung benutzt, und dann den gemeinsam genutzten Speicher lesen oder beschreiben. Wenn dies erledigt ist, setzt der Prozess *lock* mit Hilfe einer gewöhnlichen move-Anweisung zurück auf 0.

Wie lassen sich mit dieser Anweisung zwei Prozesse davon abhalten, gleichzeitig in ihre kritischen Regionen einzutreten? Die Lösung sehen Sie in Abbildung 2.22. Diese zeigt ein vier Anweisungen enthaltendes Unterprogramm in einer fiktiven (aber typischen) Assembler-Sprache. Die erste Anweisung kopiert den alten Wert von *lock* in das Register und setzt dann *lock* auf 1. Dann wird der alte Wert mit 0 verglichen. Falls er nicht Null ist, war die Sperre bereits gesetzt, so dass das Programm einfach wieder zurück zum Anfang geht und ihn nochmal prüft. Früher oder später wird er 0 werden (wenn der gerade in seiner kritischen Region laufende Prozess mit seiner kritischen Region fertig ist), und das Unterprogramm kehrt zurück, mit der gesetzten Sperre. Die Sperre zu lösen, ist einfach. Das Programm speichert einfach eine 0 in *lock*. Es werden keine speziellen Anweisungen benötigt.

Abbildung 2.22 Eintreten und Verlassen einer kritischen Region unter Verwendung der TSL-Anweisung.

```
enter_region:
        TSL_REGISTER_LOCK          | kopiere Sperrvariable und sperre mit 1
        CMP REGISTER, #0           | war die Sperrvariable 0?
        JNE enter_region           | wenn nicht, dann war gesperrt, Schleife
        RET                        | Rücksprung, nicht in kritischen Bereich

leave_region:
        MOVE_LOCK, #0              | speicher 0 in Sperrvariable
        RET                        | Rücksprung
```

Eine Lösung für das Problem der kritischen Regionen ist jetzt einfach. Bevor in die kritische Region eingetreten wird, ruft ein Prozess *enter_region* auf, welches aktiv wartet, bis die Sperre frei ist; dann erwirbt es die Sperre und kehrt zurück. Nach der kritischen Region ruft der Prozess *leave_region* auf, welches eine 0 in *lock* speichert. Wie mit allen Lösungen, die auf kritischen Regionen basieren, muss der Prozess *enter_region* und *leave_region* zur richtigen Zeit aufrufen, damit die Methode funktioniert. Falls eine Methode mogelt, wird der wechselseitige Ausschluss scheitern.

3.5 Sperrvariablen mithilfe der TSL Anweisung Kommentar

- Die Maschinensprachanweisung JNE bedeutet:
  Springe wenn das Ergebniss des Compare Befehls ungleich war
- Algorithmus bitte anhand des obigen Buchtextes selbst durchüberlegen.
- Saubere Lösung zur Vermeidung von RaceConditions in kritischen Abschnitten.
- Ist für Merhprozessorsysteme geeignet.
- Verwendet aktives Warten was nicht immer schön und gewollt ist.

Threads dürfen den kritischen Abschnitt nur betreten wenn `locVar==0` ist.
Konnte ein Thread den kritischen Abschnitt erfolgreich betreten setzt er `locVar=1` um den kritischen Abschnitt für weitere Threads zu sperren.
Das Setzen und Lesen von `locVar` ist durch die TSL-Anweisung ein unteilbarer Vorgang und damit gegenüber der Lösung Variablen-Sperren sicher.

Eigene Assemblerversion für Sperrvariablen und TSL:

```
int lockVar; // deklaration der Sperrvariablen

enter_region:
   TSL AX, lockVar; // Inhalt von lockVar nach AX kopieren und lockVar=1 setzen
   CMP AX, 0; // war lockVar==0
   JNE enter_region; // wenn nicht dann war gesperrt => in der Schleife bleiben
   RET; // rücksprung kritischer Bereich kann vom Aufrufer betreten werden

leave_region:
   MOV lockVar, 0; // speichere 0 in die Sperrvariable
   RET; // rücksprung zum Aufrufer
```

Verwendung aus der Sicht eines C-Programms:

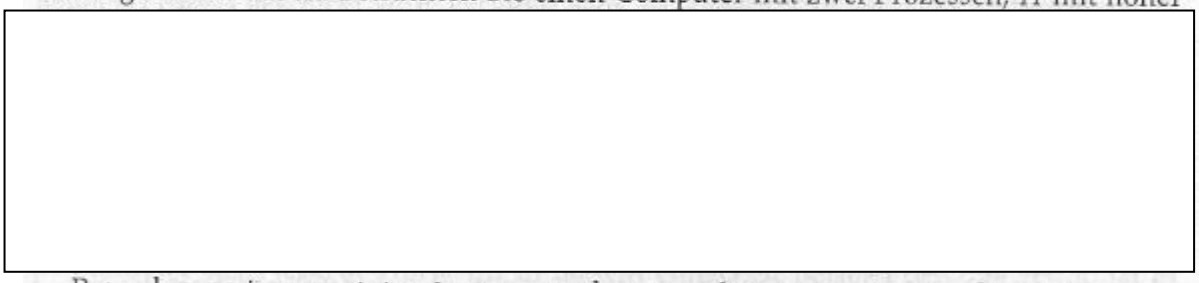
```
enter_region();
  // code zur mannipulation von gemeinsamen Daten
leave_region();
```

3.6 Sleep und Wakeup

2.3.4 Sleep and Wakeup

Beide Lösungen, die von Peterson und die mit `TSL`, sind korrekt, aber beide haben den Nachteil, dass aktives Warten erforderlich ist. Im Wesentlichen gehen die Lösungen wie folgt vor: Wenn ein Prozess in eine kritische Region eintreten möchte, überprüft er, ob der Eintritt erlaubt ist. Falls das nicht der Fall ist, hängt der Prozess so lange in einer kleinen Schleife, bis er eintreten darf.

Dieser Ansatz verschwendet nicht nur CPU-Zeit, sondern er kann auch Überraschungseffekte haben. Betrachten Sie einen Computer mit zwei Prozessen, *H* mit hoher



Betrachten wir nun einige Interprozesskommunikation-Primitiven, die sperren, anstatt CPU-Zeit zu verschwenden, wenn es ihnen nicht erlaubt ist, ihre kritische Region zu betreten. Eine der einfachsten ist das Paar `sleep` und `wakeup`. `sleep` ist ein Systemaufruf, der den Aufrufer veranlasst zu sperren. Dieser Zustand bleibt so lange bestehen, bis ein weiterer Prozess ihn wieder aufweckt. Der `wakeup`-Aufruf hat nur einen Parameter, den aufzuweckenden Prozess. Alternativ haben `sleep` und `wakeup` beide einen Parameter, eine Speicheradresse, die benötigt wird, damit die `sleeps` mit den `wakeups` zusammenpassen.

3.7 Das Erzeuger Verbraucher Problem

Als ein Beispiel für die Verwendung dieser Primitiven betrachten wir das **Erzeuger-Verbraucher**-Problem (auch bekannt als das **beschränkte Puffer**-Problem). Zwei Prozesse benutzen einen gemeinsamen allgemeinen Puffer mit fester Größe. Einer von ihnen, der Erzeuger, legt Informationen in den Puffer, der andere, der Verbraucher, nimmt sie heraus. (Das Problem lässt sich auch so verallgemeinern, dass man *m* Erzeuger und *n* Verbraucher hat, aber wir wollen nur den Fall mit einem Erzeuger und einem Verbraucher berücksichtigen, weil diese Annahme die Lösung vereinfacht.)

Ein Problem entsteht, wenn der Erzeuger eine neue Nachricht in den Puffer legen möchte, der aber schon voll ist. Die Lösung ist, den Erzeuger schlafen zu legen und wieder aufzuwecken, wenn der Verbraucher eine oder mehrere Nachrichten entfernt hat. Ähnlich geht der Verbraucher schlafen, wenn er eine Nachricht aus dem Puffer entfernen möchte und bemerkt, dass der Puffer leer ist, bis der Erzeuger etwas in den Puffer legt und ihn aufweckt.

Dieser Ansatz klingt einfach genug, aber er führt zur gleichen Art von Race Conditions, wie wir sie früher beim Spooler-Ordner gesehen haben. Um die Anzahl der Nachrichten im Puffer zu verfolgen, brauchen wir eine Variable *count*. Falls die maximale Anzahl an Nachrichten, die der Puffer aufnehmen kann, *N* beträgt, prüft der Code des Erzeugers zuerst, ob *count* *N* ist. Falls dem so ist, legt sich der Erzeuger schlafen; falls nicht, fügt der Erzeuger eine Nachricht hinzu und erhöht *count* um eins.

Der Code des Verbrauchers ist ähnlich: zuerst prüfen, ob *count* 0 ist. Falls dem so ist, legt er sich schlafen; falls sie nicht Null ist, entfernt er eine Nachricht und verringert den Zähler um eins. Jeder der Prozesse prüft auch, ob der andere aufgeweckt werden sollte, und falls dem so ist, weckt er ihn auch auf. Der Code für beide, Erzeuger und Verbraucher, ist in Abbildung 2.23 gezeigt.

Abbildung 2.23 Das Erzeuger-Verbraucher-Problem mit einer verhängnisvollen Race Condition.

```
#define N 100                                   /* Anzahl der Plätze im Puffer */
int count = 0;                                  /* Anzahl der Elemente im Puffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* Endlosschleife */
          item = produce_item();                /* erzeuge nächstes Element */
          if (count == N) sleep();              /* wenn Puffer voll, schlafe */
          insert_item(item);                    /* schreibe Element in Puffer */
          count = count + 1;                    /* erhöhe Anzahl der Elemente */
          if (count == 1) wakeup(consumer);     /* war der Puffer leer? */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                             /* Endlosschleife */
          if (count == 0) sleep();              /* wenn Puffer voll, schlafe */
          item = remove_item();                 /* hole Element aus Puffer */
          count = count - 1;                    /* erniedrige Anzahl der Elemente */
          if (count == N - 1) wakeup(producer); /* war der Puffer voll? */
          consume_item(item);                   /* gebe Element aus */
     }
}
```

Um Systemaufrufe wie `sleep` und `wakeup` in C auszudrücken, werden wir sie als Aufrufe von Bibliotheksroutinen darstellen. Sie sind zwar nicht Bestandteil der Standard-C-Bibliothek, aber vermutlich auf jedem System vorhanden, das über diese Systemaufrufe verfügt. Die Prozeduren *insert_item* und *remove_item*, welche nicht dargestellt sind, verwalten die Buchführung, um Nachrichten in den Puffer zu legen und Nachrichten aus dem Puffer herauszunehmen.

Lassen Sie uns nun zu den Race Conditions zurückkommen. Diese können auftreten, da der Zugriff auf *count* nicht zwingend ist. Die folgende Situation könnte möglicherweise auftreten. Der Puffer ist leer und der Verbraucher hat gerade *count* gelesen, um zu sehen, ob er 0 ist. In diesem Augenblick entscheidet der Scheduler, den Verbraucher vorübergehend zu stoppen, und startet den Erzeuger. Der Erzeuger fügt eine Nachricht in den Puffer, erhöht *count* um eins und bemerkt, dass er jetzt 1 ist. Überzeugt davon, dass

count gerade 0 war und der Verbraucher offensichtlich schlafen muss, ruft der Erzeuger *wakeup* auf, um den Verbraucher zu wecken.

Leider schläft der Verbraucher noch nicht wirklich, so dass das Aufweck-Signal verloren geht. Wenn der Verbraucher das nächste Mal läuft, wird er den Wert von *count*, den er gerade gelesen hat prüfen, stellt fest, dass er 0 war und geht schlafen. Früher oder später wird der Erzeuger den Puffer auffüllen und auch schlafen gehen. Damit schlafen beide für immer.

Das wesentliche Problem hier ist, dass ein Weckruf, der an einen (noch) nicht schlafenden Prozess geschickt wird, verloren geht. Falls diese Aufrufe nicht verlorgen gehen würden, würde alles funktionieren. Eine schnelle Korrektur wäre ein Ändern der Regeln derart, dass ein **Weckruf-Warte-Bit** in das Bild eingefügt wird. Wenn ein Weckruf an einen Prozess, der noch wach ist, gesendet wird, ist dieses Bit gesetzt. Später, wenn der Prozess versucht schlafen zu gehen, wird das Weckruf-Warte-Bit gelöscht, falls es noch gesetzt ist, aber der Prozess bleibt noch wach. Das Weckruf-Warte-Bit stellt eine Art „Sparschwein" für Aufweck-Signale dar.

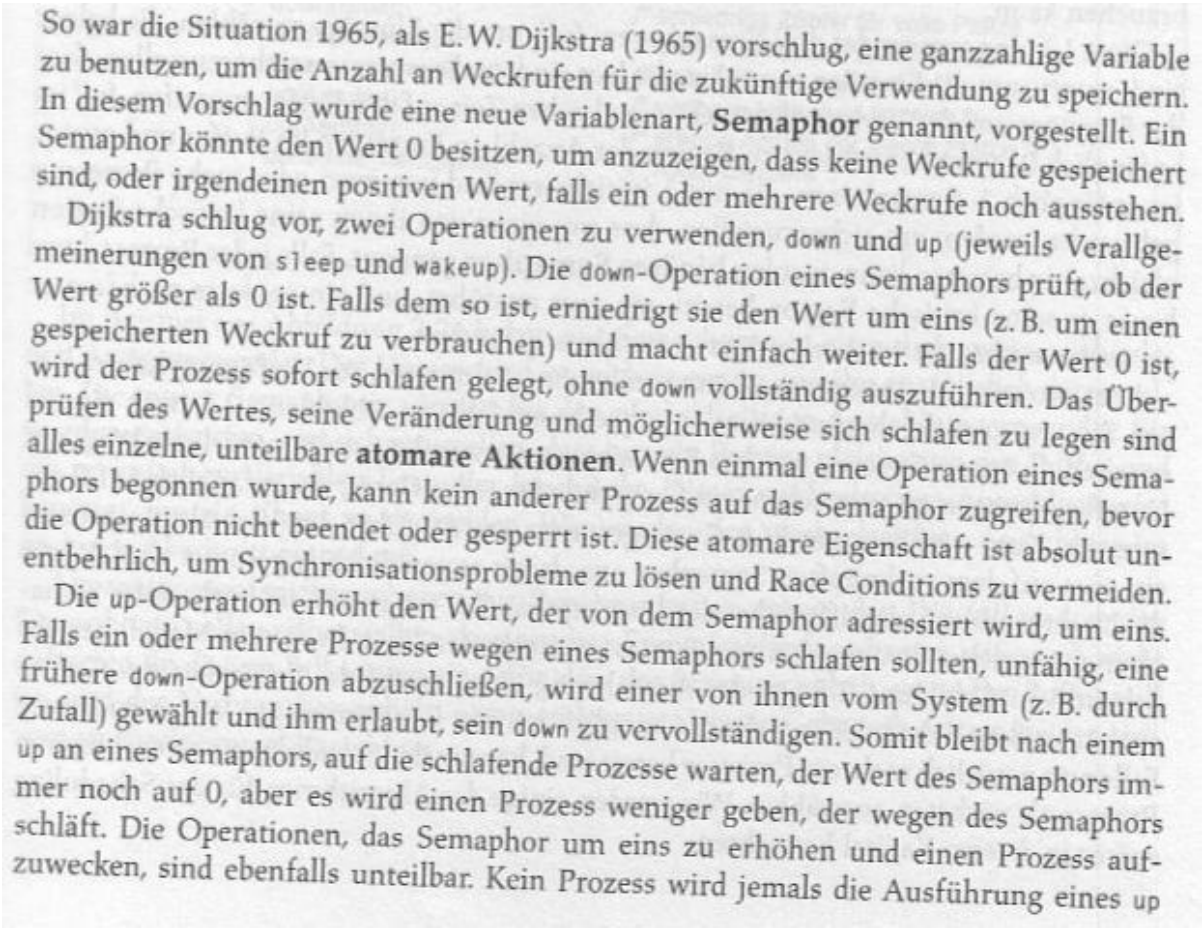
Während das Weckruf-Warte-Bit in diesem einfachen Beispiel den Tag rettet, ist es einfach, Beispiele zu konstruieren, mit drei oder mehr Prozessen, in denen ein Weckruf-Warte-Bit zu wenig ist. Wir könnten eine weitere Korrektur vornehmen und ein zweites Weckruf-Warte-Bit hinzufügen oder auch 8 oder 32 davon, aber das Problem ist im Prinzip immer noch vorhanden.

3.8 Erzeuger Verbraucher Problem Kommentar

Das oben beschriebene Erzeuger Verbraucher Problem ist ein Standartbeispiel an dem Synchronisationsmechnismen getestet werden. Es gibt viele Aufgabenstellungen in HW/SW Architekturen die mit dem Erzeuger Verbraucher Problem beschrieben werden können.

- Zwischenpuffern von Daten aus UserTasks in Richtung eines Kommunikationskanals (RS232)
- Zwischenpuffern von Werten aus der Datenerfassung auf einer Datenerfassungskarte in Richtung Visualisierung und Archivierung auf einem PC.
- Die Kernaufgabe solcher Zwischenpufferungen ist es, das die Echtzeit Erfassungstask ihre Daten ohne aufgehalten zu werden in den Zwischenpuffern ablegen können.
  Es dürfen bei der Zwischenpufferung keine Daten verloren gehen und die Daten müssen in der Reihenfolge vom Empfäger entnommen werden können in der Sie vom Sender in den Puffer hineingestellt wurden.

3.9 Semaphoren



So war die Situation 1965, als E. W. Dijkstra (1965) vorschlug, eine ganzzahlige Variable zu benutzen, um die Anzahl an Weckrufen für die zukünftige Verwendung zu speichern. In diesem Vorschlag wurde eine neue Variablenart, **Semaphor** genannt, vorgestellt. Ein Semaphor könnte den Wert 0 besitzen, um anzuzeigen, dass keine Weckrufe gespeichert sind, oder irgendeinen positiven Wert, falls ein oder mehrere Weckrufe noch ausstehen.

Dijkstra schlug vor, zwei Operationen zu verwenden, `down` und `up` (jeweils Verallgemeinerungen von `sleep` und `wakeup`). Die `down`-Operation eines Semaphors prüft, ob der Wert größer als 0 ist. Falls dem so ist, erniedrigt sie den Wert um eins (z. B. um einen gespeicherten Weckruf zu verbrauchen) und macht einfach weiter. Falls der Wert 0 ist, wird der Prozess sofort schlafen gelegt, ohne `down` vollständig auszuführen. Das Überprüfen des Wertes, seine Veränderung und möglicherweise sich schlafen zu legen sind alles einzelne, unteilbare **atomare Aktionen**. Wenn einmal eine Operation eines Semaphors begonnen wurde, kann kein anderer Prozess auf das Semaphor zugreifen, bevor die Operation nicht beendet oder gesperrt ist. Diese atomare Eigenschaft ist absolut unentbehrlich, um Synchronisationsprobleme zu lösen und Race Conditions zu vermeiden.

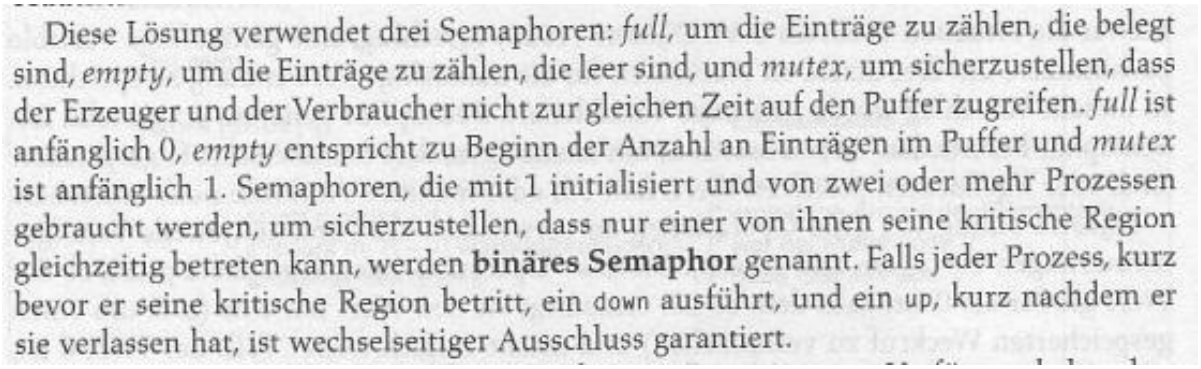
Die `up`-Operation erhöht den Wert, der von dem Semaphor adressiert wird, um eins. Falls ein oder mehrere Prozesse wegen eines Semaphors schlafen sollten, unfähig, eine frühere `down`-Operation abzuschließen, wird einer von ihnen vom System (z. B. durch Zufall) gewählt und ihm erlaubt, sein `down` zu vervollständigen. Somit bleibt nach einem `up` an eines Semaphors, auf die schlafende Prozesse warten, der Wert des Semaphors immer noch auf 0, aber es wird einen Prozess weniger geben, der wegen des Semaphors schläft. Die Operationen, das Semaphor um eins zu erhöhen und einen Prozess aufzuwecken, sind ebenfalls unteilbar. Kein Prozess wird jemals die Ausführung eines `up`

Semaphore sind die universellsten Synchronistaionsprimitve. Mit Semaphoren können gegenseitiger Ausschluss und das Benachrichtigen von einem Thread zu einem anderen realisiert werden.
Die Operationen auf Semaphore sind sogenannte **atomare** oder unteilbare Operationen.
Führt ein Thread eine atomare Operation aus also z.B. `sema_up()` so kann er innerhalb der Operation von keinem anderen Thread unterbrochen werden.

`sema_up(aSema)`
: aSema wird um 1 erhöht. Sollten Threads durch den Aufruf von `sema_down(aSema)` blokiert sein so wird einer der blokierten Threads aufgeweckt.

`sema_down(aSema)`
: Wenn der Semaphorzähler größer 0 ist kann der aufrufende Thread weiterlaufen. Ansonsten wird der Aufrufende Thread blockiert bis ein anderer Thread den Semaphorzähler mit `sema_up()` auf größer 0 setzt.

3.10 Das Erzeuger Verbraucher Problem mit Semaphoren



Diese Lösung verwendet drei Semaphoren: *full*, um die Einträge zu zählen, die belegt sind, *empty*, um die Einträge zu zählen, die leer sind, und *mutex*, um sicherzustellen, dass der Erzeuger und der Verbraucher nicht zur gleichen Zeit auf den Puffer zugreifen. *full* ist anfänglich 0, *empty* entspricht zu Beginn der Anzahl an Einträgen im Puffer und *mutex* ist anfänglich 1. Semaphoren, die mit 1 initialisiert und von zwei oder mehr Prozessen gebraucht werden, um sicherzustellen, dass nur einer von ihnen seine kritische Region gleichzeitig betreten kann, werden **binäres Semaphor** genannt. Falls jeder Prozess, kurz bevor er seine kritische Region betritt, ein `down` ausführt, und ein `up`, kurz nachdem er sie verlassen hat, ist wechselseitiger Ausschluss garantiert.

Abbildung 2.24 Das Erzeuger-Verbraucher-Problem unter Verwendung von Semaphoren.

```
#define N 100                                   /* Anzahl der Plätze im Puffer */
typedef int semaphore;                          /* Semaphoren sind spezielle Integerwerte */
semaphore mutex = 1;                            /* steuert den Zugriff auf kritische Bereiche */
semaphore empty = N;                            /* zählt leere Pufferplätze */
semaphore full = 0;                             /* zählt volle Pufferplätze */

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();                  /* TRUE ist 1 */
        down(&empty);                           /* erzeuge Element */
        down(&mutex);                           /* erniedrige Zähler für leere Plätze */
        insert_item(item);                      /* trete in kritischen Bereich ein */
        up(&mutex);                             /* lege Element in Puffer */
        up(&full);                              /* verlasse kritischen Bereich */
    }                                           /* erhöhe Anzahl der vollen Plätze */
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);                            /* Endlosschleife */
        down(&mutex);                           /* erniedrige Zähler für volle Plätze */
        item = remove_item();                   /* trete in kritischen Bereich ein */
        up(&mutex);                             /* hole Element aus Puffer */
        up(&empty);                             /* verlasse kritischen Bereich */
        consume_item(item);                     /* erhöhe Anzahl der leeren Plätze */
    }                                           /* nutze Element */
}
```

3.11 Semaphore für gegenseitigen Ausschluss und für die Benachrichtigung

Im Beispiel aus Abbildung 2.24 haben wir Semaphoren in eigentlich zwei verschiedenen Arten verwendet. Der Unterschied ist wichtig genug, um ihn ausführlich darzustellen. Die *mutex* Semaphoren werden für den wechselseitigen Ausschluss verwendet. Sie wurden entworfen, um sicherzustellen, dass nur ein Prozess gleichzeitig den Puffer und die verknüpften Variablen liest oder beschreibt. Dieser wechselseitige Ausschluss wird benötigt, um ein Chaos zu vermeiden. Wir werden den wechselseitigen Ausschluss im nächsten Abschnitt betrachten.

Alternativ werden Semaphoren zur Synchronisation verwendet. Die *full* und *empty* Semaphoren sollen sicherstellen, dass gewisse Ereignisabläufe auftreten oder auch nicht auftreten. In diesem Fall sorgen sie dafür, dass der Erzeuger anhält, wenn der Puffer voll ist, und der Verbraucher anhält, wenn er leer ist. Dieser Gebrauch unterscheidet sich vom wechselseitigen Ausschluss.

3.12 Mutexe

Benötigt man die Möglichkeit eines Semaphors zu zählen nicht, wird manchmal eine vereinfachte Version eines Semaphors, Mutex genannt, verwendet. Mutexe dienen nur der Verwaltung des gegenseitigen Ausschlusses von irgendeiner gemeinsam genutzten Ressource oder eines Codestücks. Sie sind einfach und effizient zu realisieren, was sie besonders in Thread-Paketen nützlich macht, die komplett im Benutzeradressraum realisiert sind.

Ein **Mutex** ist eine Variable, die zwei Zustände annehmen kann: nicht gesperrt oder gesperrt. Folglich wird nur 1 Bit benötigt, um sie darzustellen. In der Praxis wird häufig eine Ganzzahl verwendet, bei der 0 nicht gesperrt bedeutet und alle anderen Werte gesperrt bedeuten. Zwei Prozeduren werden mit Mutexen verwendet. Wenn ein Thread (oder ein Prozess) Zugang zu einer kritischen Region braucht, ruft er *mutex_lock* auf. Falls der Mutex gerade nicht gesperrt ist (was bedeutet, dass die kritische Region verfügbar ist), ist der Aufruf erfolgreich und dem aufrufenden Thread steht es frei, in die kritische Region einzutreten.

Falls der Mutex jedoch bereits gesperrt ist, wird der aufrufende Thread so lange gesperrt, bis der Thread in der kritischen Region fertig ist und *mutex_unlock* aufruft. Wenn mehrere Threads wegen des Mutex gesperrt sind, wird einer von ihnen per Zufall ausgewählt und ihm erlaubt, die Sperre zu erwerben.

Da Mutexe so einfach sind, können sie leicht im Benutzeradressraum realisiert werden, wenn eine TSL-Anweisung vorhanden ist. Der Code für *mutex_lock* und *mutex_unlock* für den Gebrauch mit Benutzeradressraum-Thread-Paketen wird in Abbildung 2.25 gezeigt.

Abbildung 2.25 Realisierung von *mutex_lock* und *mutex_unlock*.

```
mutex_lock:
        TSL REGISTER, MUTEX     | kopiere Mutex in Register, Mutex = 1
        CMP REGISTER, #0        | war Mutex Null?
        JZE ok                  | wenn Null, Mutex war belegt, Rücksprung
        CALL thread_yield       | Mutex belegt; führe anderen Thread aus
        JMP mutex_lock          | versuche es später
ok:     RET                     | Rücksprung, in kritischen Bereich eingetreten

mutex_unlock:
        MOVE MUTEX, #0          | speichere 0 im Mutex
        RET                     | Rücksprung
```

Der Code von *mutex_lock* ähnelt dem Code von *enter_region* aus Abbildung 2.22, abgesehen von einem entscheidenden Unterschied. Wenn *enter_region* daran scheitert, in die kritische Region einzutreten, hört er nicht auf, die Sperre immer wieder zu prüfen (aktives Warten). Die Zeit läuft schließlich ab und irgendein anderer Prozess wird zur Ausführung ausgewählt. Früher oder später kommt der Prozess mit der Sperre dran und gibt diese wieder frei.

Mit Threads ist die Situation anders, weil es keinen Taktgeber gibt, der Threads anhält, die zu lange laufen. Folglich wird ein Thread, der eine Sperre durch aktives Warten

4 Typische Aufgabenstellungen für Thread und Prozess Synchronisation

Erzeuger Verbraucher Problem:

Wurde in Kap. 3 erklärt.

Leser Schreiber Problem:

Bsp. Gleichzeitiges Abfragen und Aktualisieren einer Datenbank (Wikipedia) übers Internet.
Ein Artikel kann zu einem gegebenen Zeitpunkt nur von einem Benutzer geschrieben werden aber von mehreren Benutzern gelesen werden.

5 Beispiele zu Race-Conditions

5.1 Abwechselndes Beschreiben eines Puffers

2 Threads (Prozesse) sollen abwechselnd konsistente Daten in einen Puffer (eine Ablage) schreiben.
Es soll nur sicher gestellt werden, daß auch beide Threads ihre Daten ablegen können und das keine Daten in der Ablage überschrieben werden.

Siehe Skizze des Puffers an der Tafel

Thread1:	**Thread2:**

`while(1) {`
 	
`  data=get_from_adc1();`
`  // int_off()`
`  buff[idx]=data;`
`  idx++;`
`  // in_on()`
`}`
	`while(1) {`
	`  data=get_from_adc2();`
	`  // int_off()`
	`  buff[idx]=data;`
	`  idx++;`
	`  // int_on()`
	`}`

Das Problem gilt auch für z.B. *1 Interruptroutine* und *1 main()* auf einem Mikrokontroller.
Main() und *Interruptroutine* auf einem Mikrokontroller verhalten sich zueinander so ähnlich wie 2 Threads mit dem folgenden Unterschied:
Nur die *ISR* kann *main()* unterbrechen, *main()* kann die *ISR* nicht unterbrechen.
Das ist bei 2 Threads anders, hier kann jeder den anderen unterbrechen.

Für die Race-Condition betrachten wir einmal nur wie idx++ in Maschinenbefehle übersetzt wird:

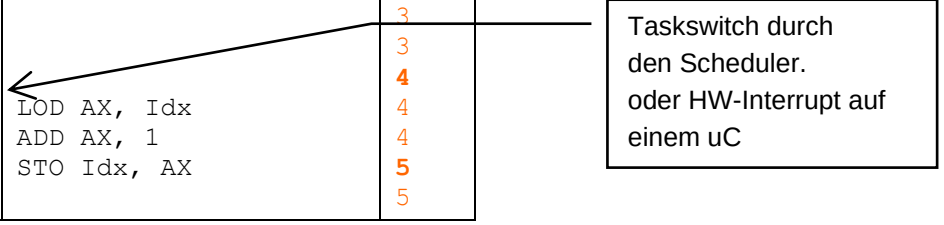
```
LOD AX, Idx ; Variable Idx in das AX-Register laden
ADD AX, 1   ; 1 zum Inhalt von AX dazuzählen
STO Idx, AX ; Inhalt von AX in die Variable zurückschreiben
```

Gutfall auf einer 1 Prozessor Maschine:

Thread1:	**Thread2:**	Idx

`LOD AX, Idx`
`ADD AX, 1`
`STO Idx, AX`
		3
		3
		3

	`LOD AX, Idx`
	`ADD AX, 1`
	`STO Idx, AX`
		4
		4
		4
		5
		5



Schlechtfall auf einer 1 Prozessor Maschine: Tafel

Thread1:	**Thread2:**	Idx
		3
LOD AX, Idx		3
ADD AX, 1		3
	LOD AX, Idx	3
STO Idx, AX		4
	ADD AX, 1	4
	STO Idx, AX	**4**
		4

Schlechtfall auf einer 2 Prozessor Maschine (Core2Duo): Tafel

5.2 Producer Consumer unsynchronisiert über Ringpuffer

Skizze des Puffers auf der Tafel.

Producer:	**Consumer:**

`while(1) {`
`  while(itemCount>=MAX_ITEMS)`
`  ; //warten bis platz im buffer ist`
`  val=get_from_adc();`
`  buff[wrIdx]=val;`
`  wrIdx++; // wrap at buffer_end`
`  itemCount++;`
`}`
	`while(1) {`
	`  while(itemCount==0)`
	`  ; //warten bis Items im buffer sind`
	`  val=buff[rdIdx];`
	`  add2curve(val);`
	`  rdIdx++; // wrap at buffer_end`
	`  itemCount--;`
	`}`

`add2curve(val);` fügt einen Messwert zu einer Kurve von Online-Messwerten hinzu.

Wo können Race-Conditions auftreten ?
`itemCount++` und `itemCount–` können sich wie unten gezeigt in die Quere kommen:

		itemCount
		5
`LOD AX,itemCount;`		5
	`LOD AX,itemCount;`	5
`ADD AX,1;`		5
`STO itemCount,AX;`		**6**
	`SUB AX,1;`	5
	`STO itemCount,AX;`	5
		4

Je nachdem wie der Scheduler die Rechenzeit verteilt ist ItemCount==4 oder ItemCount==6.
itemCount müsste aber 5 sein

Testfragen: Gibt es Race-Conditions an den folgenden Stellen ?

1. `while(itemCount>=MAX_ITEMS)`
2. `while(itemCount==0)`
3. `wrIdx++;`
4. `rdIdx--;`

Die Antworten müssen begründet werden.

1. Nein weil itemCount in diesem Fall vom Prod nur gelesen und vom Cons nur geschrieben wird
2. Nein weil itemCount in diesem Fall vom Cons nur gelesen und vom Prod nur geschrieben wird
3. Nein wrIdx wird nur vom Producer verwendet => kein concurrent use
4. Nein rdIdx wird nur vom Consumer verwendet => kein concurrent use

6 Synchronistaionsmechanismen in C#

In C# kann jedes C# objekt als Synchronistaionsobjekt verwendet werden.
Mann kann sich das so vorstellen, daß auf der Ebene der CLR für jedes C# objekt ein Semaphor existiert.
Die Synchronisationsfunktionen welche auf ein C# objekt angewendet werden können
sind in der Klasse `Monitor` zusammengefasst.
Monitor bietet Funktionen für den gegenseitigen Ausschluss und für das Benachrichtigen von Threads.
CLR heißt ausgeschrieben **C**ommon **L**aguage **R**untime und ist das Laufzeitsystem von C#
Eine kurze und gute Übersicht über Threadprogrammierung in C# ist in **BS_Threads_Sync_in_C#.ppt** zu finden.

Mutex Funktionen von Monitor:

```
public static class Monitor
{
  static void Enter(object obj);
  static void Exit(object obj);
}
```

Signalisierungs Funktionen von Monitor:

```
public static class Monitor
{
  static void Pulse(object obj); // signal senden

  static bool Wait(object obj);  // auf das signal blockierend warten
}
```

6.1 Verwendung der Mutex-Funktionen von Monitor in Mutex1.cs

In diesem Beispiel wird das *objekt mutex* verwendet um wechselseitigen Ausschluss beim Befüllen eines Arrays zu erreichen.
Code für einen der beiden Threads welche mit wechselseitigen Ausschluss das Array befüllen:

```
void ThrFuncA()
 {
   int num2 = 0;
   while (true)
   {
     Monitor.Enter(mutex); // kein anderer Thread kann FillArray() jetzt aufrufen
       FillArray("A", num2++);
     Monitor.Exit(mutex); // FillArray() kann von anderen Threads wieder aufger. werden
   }
 }
```

Die CodeSequenz `Monitor.Enter(mutex); Monitor.Exit(mutex);` kann kompakter und eleganter auch so formuliert werden:

```
void ThrFuncB()
{
  int num2 = 0;
  while (true)
  {
    lock (mutex)
    {
      FillArray("B", num2++);
     }
  }
}
```

6.2 Einseitige Synchronisation (Signalisierung) im Beispiel SignalingDemo.cs

`ReceiverThread()` wird durch `SenderThread()` zyklisch immer wieder aufgeweckt.
zum Benachrichtigen wird `Monitor.Wait(sigObject)` und `Monitor.Pulse(sigObject)` verwendet.
`Monitor.Wait(sigObject)` und `Monitor.Pulse(sigObject)` können nur aufgerufen werden wenn exclusiver Zugriff auf `sigObject` besteht, wenn also vorher `lock(sigObject)` aufgerufen wurde.

6.3 Einseitige Synchronisation (Signalisierung) im Beispiel ThreadDemo3.cs

Das Hauptprogamm main() soll an einem objekt (finishedSig) warten bis einer der beiden Threads MAX_CNT erreicht hat.

Warten in main():

```
// Main wird hier blockiert ( bleibt stehen ) bis ThrFuncA() oder ThrFuncB()
// einen Puls auf finishedSig geben ( finishedSig signalisieren )
Monitor.Enter(finishedSig);
Monitor.Wait(finishedSig);
// Wait(finishedSig) darf nur aufgerufen werden wenn vorher durch Enter(finishedSig)
// exklusiever Zugriff auf finishedSig erlangt wurde.
// innerhalb von Wait(finishedSig) wird Exit(finishedSig) implizit aufgerufen.
```

Signalisieren in einem der Threads:

```
// ein Signal auf finishedSig geben
Monitor.Enter(finishedSig);
Monitor.Pulse(finishedSig);
// innerhalb von Pulse(finishedSig) wird Exit(finishedSig) implizit aufgerufen.
```

7 Verzeichniss der C# Beispiele

ThreadDemo1.cs: Verwendung von Threads in C#

ThreadDemo2.cs: Ist die ThreadDemo1.cs mit verhübschter Ausgabe auf der Konsole.

ThreadDemo3.cs: Visualisierung von: Wie verteilt der Scheduler die Rechenzeit auf 2-Threads

SignalingDemo.cs: Einfaches Beispiel zur einseitigen Synchronistaion (Signalisierung)
`ReceiverThread()` wird durch `SenderThread()` zyklisch immer wieder aufgeweckt.
Zum Benachrichtigen `wird Monitor.Wait(sigObject) und Monitor.Pulse(sigObject)` verwendet
Hausübung1: Auf 2 bzw. N Receiver erhöhen und überprüfen ob die auch schön Round-Robin drankommen

Mutex1.cs: Ein Array soll von 2 Producer Threads abwechselnd konsistent mit 10 Dateneinträgen befüllt werden. Ohne Synchronisation (lock mutex) werden die Einträge der Threads im Array vermischt.
Das Befüllen der einzelnen Einträge im Array wird durch printf's mitgeloggt.

TwoProducers.cs: Ähnliche Aufgabenstellung wie bei Mutex1.cs.
Das befüllte Array wird jetzt allerdings durch einen eigenen `Consumer-Thread` ausgelesen.
Skizze des Zusammenspiels der Threads, Puffer und Sync-Objekte an der Tafel
`consSig` weckt den Consumer auf wenn `buffer` fertig befüllt ist.
`mutex` regelt den wechselseitigen Zugriff von `ProducerA` und `ProducerB` auf den DatenBuffer (buffer)
Wird `mutex` nicht verwendet liest der Consumer einen inkonsistenten Datenstand aus dem DatenBuffer (buffer)
Hausübung2: einen 3'ten Producer hinzufügen

ProdCons1_Comment.cs: Die C# Version von Producer/Consumer unter Verwendung der C# Synchronisationsmechanismen.

ProdCons2.cs: Wenn der Buffer leer ist wartet der Consumer bis wieder mind. 5 Items im Buffer sind bevor er wieder mit dem Konsumieren anfängt.

RequestResponse1_Template.cs: Template für die Hausübung zu Request/Response
1 Thread Requester() sendet Anfragen über die Request-Mailbox an den Responder-Thread.
Der Responder-Thread sendet Antworten über die Response-Mailbox an den Requester() zurück.
Lösung in RequestResponse1.cs

Hausübung4.cs: 3-Threads sollen einander ringförmig benachrichtigen (aufwecken).
Skizze an der Tafel. 1. Wie starten damit es keinen DeadLock gibt ?? 2. Wie starten damit es einen DeadLock gibt ?? Im 3ten muß ein sleep sein damit das Ganze nicht zu schnell läuft.
Das ganze kann von main() aus mit Taste wieder abgebrochen werden.

Hausübung5.cs: 2 bis N-Threads in einer Form-Anwendung starten.
Die Threads führen `sleep(rnd())` aus. Der Fortschritt der Threads soll in Bargraphs visualisiert werden.
Die Visualisierung soll in der Form über Timer angetrieben paralell laufen und die Threadcounter anzeigen.
Über ein Mutex welches die Visualisierung hin und wieder sperrt (Checkbox) können die Threads blockiert und entblokiert werden.
Noch besser die Threads werden von der Visualisierung mit `suspend()/resume()` gesteuert.

Hausübung6.cs: Die SemaDemo.cs erweitern.
1 SenderThread und 3 ReceiverThreads
Der SenderThread ruft `sema.Release(3);` auf.
Überprüfen ob mit dem Aufruf auch alle 3 Receiver aufgewekt werden.

Hausübung7.cs: Die speisenden Philosophen

8 Funktionsweise eines Ringpuffers

Skizze mit Puffer, Variablen und Zahlen an der Tafel.

C# Code: aus dem ProdCons_Commented.cs hier hereinkopieren.

```
class RBuffer
{
  int[] buffer;
  int wrIdx, rdIdx, itemCount;

  public RBuffer(int aSize)
  {
    buffer = new int[aSize];
    wrIdx = rdIdx = itemCount = 0;
  }

  public int getCount()
  {
    return itemCount;
  }

  public bool isEmpty()
  {
    return itemCount == 0;
  }

  public bool isFull()
  {
    return itemCount == buffer.Length;
  }

  public void put(int aVal)
  {
    if (itemCount >= buffer.Length)
      return;
    buffer[wrIdx] = aVal; wrIdx++;
    if (wrIdx >= buffer.Length)
      wrIdx = 0;
    itemCount++;
  }

  public int get()
  {
    if (itemCount == 0)
      return 0;
    int ret = buffer[rdIdx]; rdIdx++;
    if (rdIdx >= buffer.Length)
      rdIdx = 0;
    itemCount--;
    return ret;
  }
}
```

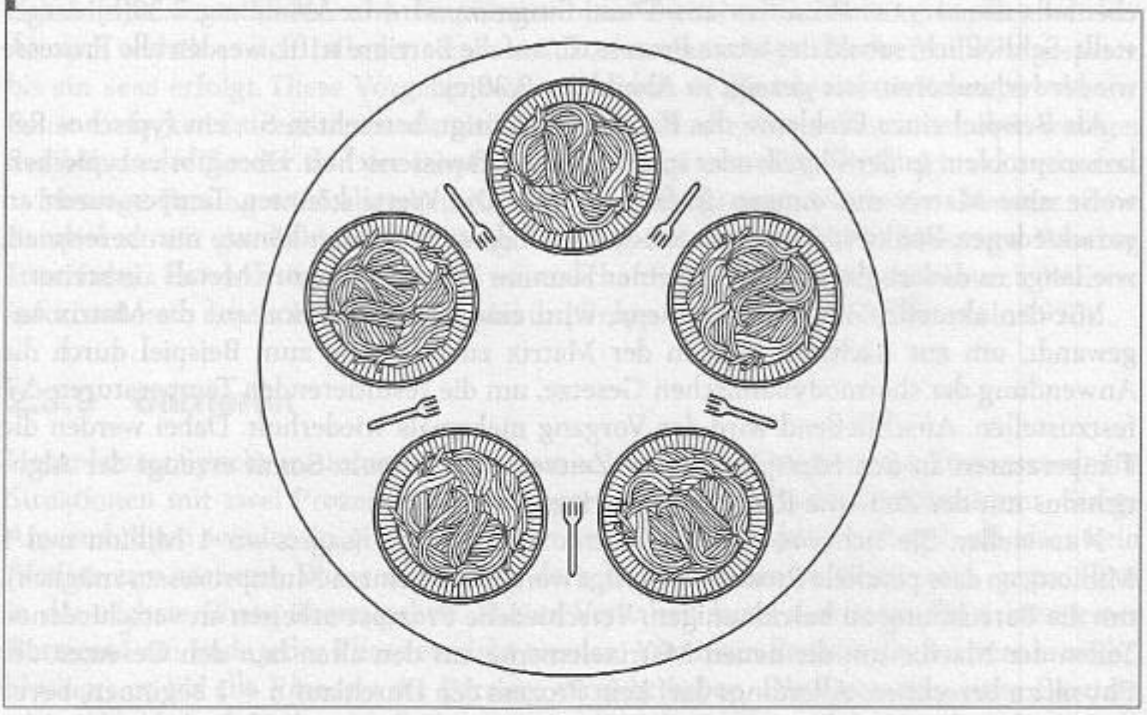
9 Klassische Probleme der Interprozesskommunikation

9.1 Das Problem der speisenden Philosophen

1965 veröffentlichte und löste Dijkstra ein Synchronisationsproblem, dass er **das Problem der speisenden Philosophen** nannte. Seitdem fühlte sich jeder, der eine weitere Synchronisationsprimitive erfand, verpflichtet zu zeigen, wie elegant sich das Problem der speisenden Philosophen damit lösen lässt. Das Problem kann ziemlich einfach dar-

gestellt werden. Fünf Philosophen sitzen um einen runden Tisch. Jeder Philosoph hat einen Teller voll Spagetti vor sich. Die Spagetti sind so schlüpfrig, dass jeder Philosoph zwei Gabeln braucht, um sie zu essen. Zwischen den Tellern liegt jeweils eine Gabel. Abbildung 2.31 zeigt diese Situation.

Abbildung 2.31 Mittagessen bei den Philosophen.



Das Leben eines Philosophen besteht abwechselnd aus Zeiten des Denkens und Zeiten des Essens. (Dies ist sogar für Philosophen eine Art Abstraktion, aber die anderen Aktivitäten sind hier unwichtig.) Wenn ein Philosoph Hunger bekommt, versucht er, seine linke und rechte Gabel zu bekommen. Wenn es ihm gelingt, zwei Gabeln zu bekommen, isst er eine Zeit lang, legt die Gabeln zurück und denkt wieder weiter. Die Hauptfrage ist: Kann man ein Programm für die Philosophen schreiben, das erwartungsgemäß funktioniert, aber niemals hängenbleibt? (Es wurde darauf hingewiesen, dass die Zwei-Gabel-Forderung etwas konstruiert ist; vielleicht sollten wir vom italienischen Essen zu chinesischem Essen wechseln, mit Reis als Ersatz für die Spagetti und Stäbchen für die Gabeln.)

Abbildung 2.32 zeigt die offensichtliche Lösung. Die Prozedur *take_fork* wartet, bis die gewünschte Gabel verfügbar ist, um sie dann zu greifen. Leider ist die offensichtliche Lösung falsch. Nehmen Sie an, dass alle fünf Philosophen ihre linken Gabeln gleichzeitig nehmen. Keinem wird es möglich sein, seine rechte Gabel zu nehmen, und es wird einen Deadlock geben.

Nun können wir das Programm verändern, so dass es prüft, ob die rechte Gabel frei ist, nachdem die linke Gabel genommen wurde. Falls nicht, legt der Philosoph seine linke Gabel zurück, wartet eine Weile und wiederholt den gesamten Vorgang. Dieser Vorschlag scheitert auch, allerdings aus einem anderen Grund. Mit ein wenig Pech können alle Philosophen ihren Algorithmus gleichzeitig starten, ihre linken Gabeln aufnehmen und

Abbildung 2.32 Keine Lösung für das Problem der speisenden Philosophen.

```
#define N 5                                     /* Anzahl der Philosophen */

void philosopher(int i)
{
        while (TRUE) {
                think();                        /* Philosoph denkt nach */
                take_fork(i);                   /* nimm linke Gabel */
                take_fork((i+1) % N);           /* nimm rechte Gabel, % ist der Modulooperator */
                eat();                          /* lecker, Spagetti */
                put_fork(i);                    /* lege linke Gabel zurück */
                put_fork((i+1) % N);            /* lege rechte Gabel zurück */
        }
}
```

feststellen, dass die rechte Gabel nicht verfügbar ist, dann ihre Gabeln wieder hinlegen und warten, ihre linken Gabeln wieder gleichzeitig aufnehmen und so weiter. Eine Situation wie diese, in der alle Programme uneingeschränkt weiterlaufen, aber keine Fortschritte machen, bezeichnet man als **Aushungern**.

Nun könnten Sie denken: „Falls die Philosophen einfach eine zufällige anstatt die gleiche Zeit warten, nachdem sie bei dem Versuch gescheitert sind, eine Gabel zu bekommen, ist die Möglichkeit sehr gering, dass alle im Gleichschritt weitermachen werden." Diese Beobachtung ist richtig und in fast allen Anwendungen ist es auch kein Problem, es zu einem späteren Zeitpunkt wieder zu versuchen. Im populären Ethernet LAN zum Beispiel, wenn zwei Rechner gleichzeitig ein Datenpaket senden, wartet jeder eine zufällige Zeitspanne und versucht es dann wieder; praktisch funktionert diese Lösung wunderbar. Ein paar Anwendungen würden allerdings eine Lösung bevorzugen, die immer funktioniert und nicht aufgrund einer unwahrscheinlichen Serie von Zufallsfaktoren scheitern kann. Denken Sie beispielsweise an die Sicherheitskontrolle in einem Kernkraftwerk.

Eine Verbesserung zu Abbildung 2.32, ohne Deadlock und ohne Aushungern, sieht vor, dass die fünf Anweisungen, die nach dem *think* folgen, durch ein binäres Semaphor geschützt werden. Bevor ein Philosoph anfängt, eine Gabel zu nehmen, würde er ein down auf einen *mutex* ausführen. Nach dem Zurücklegen der Gabeln würde er ein up auf *mutex* ausführen. Aus theoretischer Sicht ist diese Lösung angemessen. Aus praktischer Sicht hat sie ein Performance-Problem: Es kann immer nur ein Philosoph essen. Mit fünf verfügbaren Gabeln sollte es möglich sein, zwei Philosophen gleichzeitig das Essen zu erlauben.

Die Lösung, die in Abbildung 2.33 dargestellt ist, weist keine Deadlocks auf und erlaubt ein Maximum an Parallelität für eine beliebige Anzahl an Philosophen. Sie benutzt ein Feld, *stat*, um zu verfolgen, ob ein Philosoph gerade isst, denkt oder hungrig ist (versucht, Gabeln zu bekommen). Ein Philosoph könnte nur in den essenden Zustand übergehen, wenn kein Nachbar am Essen ist. Die Nachbarn des Philosophen *i* sind durch die Makros *LEFT* und *RIGHT* definiert. Mit anderen Worten, falls *i* gleich 2 ist, sind *LEFT* gleich 1 und *RIGHT* gleich 3.

Das Programm verwendet ein Feld von Semaphoren, eine pro Philosoph, so dass hungrige Philosophen blockieren können, falls die benötigten Gabeln in Gebrauch sind. Beach-

Abbildung 2.33 Eine Lösung für das Problem der speisenden Philosophen.

```
#define N                5                  /* Anzahl der Philosophen */
#define LEFT             (i+N–1)%N          /* Anzahl der linken Nachbarn */
#define RIGHT            (i+1)%N            /* Anzahl der rechten Nachbarn */
#define THINKING         0                  /* Philosoph denkt nach */
#define HUNGRY           1                  /* Philosoph versucht 2 Gabeln zu bekommen */
#define EATING           2                  /* Philosoph isst */
typedef int semaphore;                      /* Semaphoren sind Integer */
int state[N];                               /* Feld mit dem Status jedes Philosophen */
semaphore mutex = 1;                        /* Wechselseitiger Ausschluß im kritischen Bereich */
semaphore s[N];                             /* ein Semaphor pro Philosoph */

void philosopher (int i)                    /* i: Nummer des Philosophen, von 0 bis N–1 */
{
        while (TRUE) {                      /* Endlosschleife */
                think();                    /* Philosoph denkt nach */
                take_forks(i);              /* Nimm 2 Gabeln oder blockiere */
                eat();                      /* lecker, Spagetti! */
                put_forks(i);               /* lege beide Gabeln zurück auf den Tisch */
        }
}

void take_forks (int i)                     /* i: Nummer des Philosophen, von 0 bis N–1 */
{
        down(&mutex);                       /* Eintritt in den kritischen Bereich */
        state[i] = HUNGRY;                  /* Merken, dass Philosoph i hungrig ist */
        test(i);                            /* versuche 2 Gabeln zu bekommen */
        up(&mutex);                         /* verlassen des kritischen Bereichs */
        down(&s[i]);                        /* blockiere, wenn Gabeln nicht erhalten */
}

void put_forks (i)                          /* i: Nummer des Philosophen, von 0 bis N–1 */
{
        down(&mutex);                       /* Eintritt in den kritischen Bereich */
        state[i] = THINKING;                /* Philosoph ist mit dem Essen fertig */
        test(LEFT);                         /* kann der linke Nachbar essen? */
        test(RIGHT);                        /* kann der rechte Nachbar essen? */
        up(&mutex);                         /* verlassen des kritischen Bereichs */
}

void test (i)                               /* i: Nummer des Philosophen, von 0 bis N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

ten Sie, dass jeder Prozess die Prozedur *philosopher* in seinem Hauptcode laufen lässt, aber die anderen Prozeduren, *take_forks*, *put_forks* und *test*, gewöhnliche Prozeduren und keine eigenständigen Prozesse sind.

9.2 Das Leser Schreiber Problem

Das Problem der speisenden Philosophen ist nützlich, um Prozesse zu veranschaulichen, die um den exklusiven Zugriff auf eine begrenzte Anzahl Ressourcen konkurrieren, wie E/A-Geräte. Ein weiteres berühmtes Problem ist das Leser-Schreiber-Problem (Courtois et al., 1971), welches den Zugriff auf eine Datenbank modelliert. Stellen Sie sich zum Beispiel ein Reservierungssystem einer Fluggesellschaft vor, mit vielen konkurrierenden Prozessen, die Lese- und Schreibrecht wünschen. Es ist akzeptabel, wenn mehrere Prozesse gleichzeitig die Datenbank auslesen, aber wenn ein Prozess die Datenbank aktualisiert (beschreibt), darf kein anderer Prozess auf die Datenbank zugreifen, nicht einmal zum Lesen. Die Frage ist, wie programmiert man die Leser und Schreiber? Eine Lösung ist in Abbildung 2.34 dargestellt.

Abbildung 2.34 Eine Lösung für das Leser-Schreiber-Problem.

```
typedef int semaphore;                  /* Raten Sie mal */
semaphore mutex = 1;                    /* Kontrolle für den Zugriff auf `rc´ */
semaphore db = 1;                       /* Kontrolle zum Zugriff auf die Datenbank */
int rc = 0;                             /* Anzahl der lesebereiten Prozesse */

void reader (void)
{
        while (TRUE) {                  /* Endlosschleife */
                down(&mutex);           /* exklusiven Zugriff auf `rc´ */
                rc = rc + 1;            /* ein Leser mehr */
                if (rc == 1) down(&db); /* wenn es der erste Leser ist ... */
                up(&mutex);             /* exklusiven Zugriff auf `rc´ freigeben */
                read_data_base();       /* Zugriff auf die Daten */
                down(&mutex);           /* exklusiven Zugriff auf `rc´ */
                rc = rc – 1;            /* ein Leser weniger */
                if (rc == 0) up(&db);   /* wenn es der letzte Leser war ... */
                up(&mutex);             /* exklusiven Zugriff auf `rc´ freigeben */
                use_data_read();        /* unkritischer Bereich */
        }
}

void writer (void)
{
        while (TRUE) {                  /* Endlosschleife */
                think_up_data();        /* unkritischer Bereich */
                down(&db);              /* exklusiven Zugriff */
                write_data_base();      /* schreibe die neuen Daten */
                up(&db);                /* exklusiven Zugriff freigeben */
        }
}
```

In dieser Lösung führt der erste Leser, der Zugriff auf die Datenbank bekommt, ein `down` auf das Semaphor *db* aus. Nachfolgende Leser machen nichts anderes, als den Zähler, *rc*, zu erhöhen. Wenn die Leser fertig sind, erniedrigen sie den Zähler und der Letzte führt ein `up` auf das Semaphor aus. Das erlaubt es dem blockierten Schreiber, falls es einen gibt, einzutreten. Die hier vorgestellte Lösung enthält implizit eine subtile Entscheidung,

die genauer betrachtet werden soll. Nehmen Sie an, dass ein weiterer Leser zugreift, während der Leser die Datenbank benutzt. Da es kein Problem ist, zwei Leser gleichzeitig zu haben, wird der zweite Leser zugelassen. Ein dritter und auch nachfolgende Leser dürfen zugreifen.

Nun stellen Sie sich vor, dass ein Schreiber zugreifen will. Der Schreiber kann nicht auf die Datenbank zugreifen, da Schreiber einen exklusiven Zugriff benötigen, somit wird der Schreiber schlafen gelegt. Später treten weitere Leser auf. Solange mindestens ein Leser aktiv ist, werden nachfolgende Leser zugelassen. Solange also ein ständiger Nachschub an Lesern vorliegt, können diese sofort zugreifen, wenn sie den Zugriff benötigen. Der Schreiber wird so lange schlafend bleiben, bis kein Leser mehr da ist. Falls ein neuer Leser, sagen wir alle 2 Sekunden, ankommt und jeder Leser 5 Sekunden braucht, um seine Arbeit zu verrichten, wird der Schreiber niemals an die Reihe kommen.

Um diese Situation zu verhindern, könnte das Programm etwas anders geschrieben werden: Sobald ein Leser erscheint und ein Schreiber wartet, wird der Leser hinter den Schreiber gestellt, anstatt gleich zugelassen zu werden. Auf diese Weise braucht der Schreiber, wenn er zugreifen will, nur auf Leser zu warten, die noch fertig werden müssen, und nicht auf nach ihm ankommende Leser. Der Nachteil dieser Lösung ist, dass sie weniger Parallelität und somit weniger Geschwindigkeit erreicht. Courtois et al. stellt eine Lösung vor, die den Schreibern den Vorzug gibt. Für eine genauere Beschreibung verweisen wir auf seine Arbeit.