# 2 Prozesse

In früheren Zeiten waren die Rechner für eine Hauptaufgabe zu einem Zeitpunkt bestimmt; alle Programme wurden zu einem Paket geschnürt und liefen nacheinander durch (**Stapelverarbeitung** oder *Batch*-Betrieb). Üblicherweise gibt es heutzutage aber nicht nur ein Programm auf einem Rechner, sondern mehrere (**Mehrprogrammbetrieb**, *multi-tasking*). Auch gibt es nicht nur einen Benutzer (*single user*), sondern mehrere (**Mehrbenutzerbetrieb**, *multi-user*). Um Konflikte zwischen ihnen bei der Benutzung des Rechners zu vermeiden, muß die Verteilung der Betriebsmittel auf die Programme geregelt werden.

Dies spart außerdem noch Rechnerzeit und erniedrigt damit die Bearbeitungszeiten. Beispielsweise kann die Zuteilung des Hauptprozessors (*Central Processing Unit* CPU) für den Ausdruck von Text parallel zu einer Textverarbeitung so geregelt werden, daß die Textverarbeitung die CPU in der Zeit erhält, in der der Drucker ein Zeichen ausdruckt. Ist dies erledigt, schiebt der Prozessor ein neues Zeichen dem Drucker nach und arbeitet dann weiter an der Textverarbeitung.

Zusätzlich zu jedem Programm muß also gespeichert werden, welche Betriebsmittel es benötigt: Speicherplatz, CPU-Zeit, CPU-Inhalt etc. Die gesamte Zustandsinformation der Betriebsmittel für ein Programm wird als eine Einheit angesehen und als **Prozeß** (*task*) bezeichnet (Abb. 2.1).

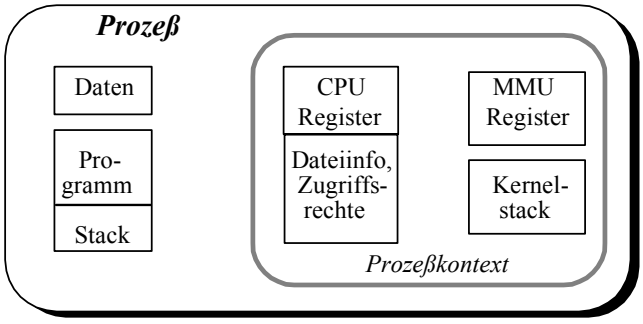



**Abb. 2.1** *Zusammensetzung der Prozeßdaten*

Ein Prozeß kann auch einen anderen Prozeß erzeugen, wobei der erzeugende Prozeß als **Elternprozeß** und der erzeugte Prozeß als **Kindsprozeß** bezeichnet wird.

Ein Mehrprogrammsystem (*multiprogramming system*) erlaubt also das „gleichzeitige“ Ausführen mehrerer Programme und damit mehrerer Prozesse (Mehrprozeßsystem, *multi-tasking system*). Ein Programm (**Job**) kann dabei auch selbst mehrere Prozesse erzeugen.

**Beispiel** *UNIX*

Die Systemprogramme werden in UNIX als Bausteine angesehen, die beliebig miteinander zu neuen, komplexen Lösungen kombiniert werden können. Die Unabhängigkeit der Prozesse und damit auch der Jobs erlaubt nun in UNIX, mehrere Jobs gleichzeitig zu starten. Beispielsweise kann man das Programm *cat*, das mehrere Dateien aneinander hängt, das Programm *pr*, das einen Text formatiert, und das Programm *lpr*, das einen Text ausdruckt, durch die Eingabe

```
cat Text1 Text2  |  pr  |  lpr
```

miteinander verbinden: Die Texte „`Text1`“ und „`Text2`“ werden aneinandergehängt, formatiert und dann ausgedruckt. Der Interpreter (*shell*), dem der Befehl übergeben wird, startet dazu die drei Programme als drei eigene Prozesse, wobei das Zeichen „|“ ein Umlenken der Ausgabe des einen Programms in die Eingabe des anderen veranlaßt. Gibt es mehrere Prozessoren im System, so kann jedem Prozessor ein Prozeß zugeordnet werden und die obige Operation tatsächlich parallel ablaufen. Ansonsten bearbeitet der eine Prozessor immer nur ein Stück eines Prozesses und schaltet dann um zum nächsten.

Zu einem diskreten Zeitpunkt ist bei einem Einprozessorsystem nur immer ein Prozeß aktiv; die anderen sind blockiert und warten. Dies wollen wir näher betrachten.

## 2.1 Prozeßzustände

Zusätzlich zu dem Zustand „aktiv“ (*running*) für den einen, aktuellen Prozeß müssen wir noch unterscheiden, worauf die anderen Prozesse warten. Für jede der zahlreichen Ereignismöglichkeiten gibt es meist eine eigene Warteschlange, in der die Prozesse einsortiert werden.
Ein blockierter Prozeß kann darauf warten,

- aktiv den Prozessor zu erhalten, ist aber sonst bereit (*ready*),
- eine Nachricht (*message*) von einem anderen Prozeß zu erhalten,
- ein Signal von einem Zeitgeber (*timer*) zu erhalten,

- Daten eines Ein/Ausgabegeräts zu erhalten (*io*).

Üblicherweise ist der *bereit*-Zustand besonders ausgezeichnet: Alle Prozesse, die Ereignisse erhalten und so entblockt werden, werden zunächst in die *bereit*-Liste (*ready-queue)* verschoben und erhalten dann in der Reihenfolge den Prozessor. Die Zustände und ihre Übergänge sind in Abb. 2.2 skizziert.

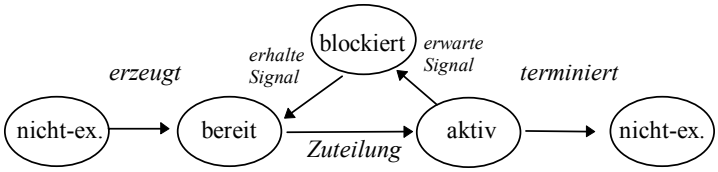



**Abb. 2.2** *Prozeßzustände und Übergänge*

Alle Zustände enthalten eine oder mehrere Warteschlangen (Listen), in die die Prozesse mit diesem Zustand eingetragen werden. Es ist klar, daß ein Prozeß immer nur in einer Liste enthalten sein kann.

Programme und damit die Prozesse existieren nicht ewig, sondern werden irgendwann erzeugt und auch beendet. Dabei verwalten die Prozesse aus Sicherheitsgründen sich nicht selbst, sondern das Einordnen in die Warteschlangen wird von einer besonderen Instanz des Betriebssystems, dem **Scheduler**, nach einer Strategie geplant. Bei einigen Betriebssystemen gibt es darüber hinaus eine eigene Instanz, den **Dispatcher**, der das eigentliche Überführen von einem Zustand in den nächsten bewirkt. Das Einordnen in eine Warteschlange, die Zustellung der Signale und das Abspeichern der Prozeßdaten werden also von einer zentralen Instanz erledigt, die der Benutzer nicht direkt steuern kann. Statt dessen werden über die Betriebssystemaufrufe die Wünsche der Prozesse angemeldet, denen im Rahmen der Betriebsmittelverwaltung vom Scheduler mit Rücksicht auf andere Benutzer entsprochen wird.

Im Unterschied zu dem Maschinencode werden die Zustandsdaten der Hardware (CPU, FPU, MMU), mit denen der Prozeß arbeitet, als **Prozeßkontext** bezeichnet, s. Abb. 2.1. Der Teil der Daten, der bei einem blockierten Prozeß den letzten Zustand der CPU enthält und damit wie ein Abbild der CPU ist, kann als *virtueller Prozessor* angesehen werden und muß bei einer Umschaltung zu einem anderen Prozeß bzw. Kontext (*context switch*) neu geladen werden.

Die verschiedenen Betriebssysteme differieren in der Zahl der Ereignisse, auf die gewartet werden kann, und der Anzahl und Typen von Warteschlangen, in denen gewartet werden kann. Sie unterscheiden sich auch darin, welche Strategien sie für das Erzeugen und Terminieren von Prozessen sowie die Zuteilung und Einordnung in Wartelisten vorsehen.

### 2.1.1 Beispiel UNIX

Im UNIX-Betriebssystem gibt es sechs verschiedene Zustände: die drei oben erwähnten *running* (SRUN), *blocked* (SSLEEP) und *ready* (SWAIT) sowie *stopped* (SSTOP), was einem Warten auf ein Signal des Elternprozesses bei der Fehlersuche (*tracing* und *debugging*) entspricht. Außerdem existieren noch die zusätzlichen Zwischenzustände *idle* (SIDL) und *zombi* (SZOMB), die beim Erzeugen und Terminieren eines Prozesses entstehen. Die Zustandsübergänge haben dabei die in Abb. 2.3 gezeigte Gestalt.

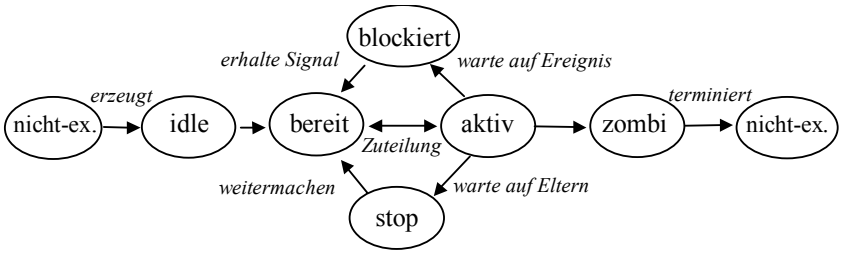



**Abb. 2.3** *Prozeßzustände und Übergänge bei UNIX*

Der Übergang von einem Zustand in einen nächsten wird durch Anfragen (Systemaufrufe) erreicht. Ruft beispielsweise ein Prozeß die Funktion `fork()` auf, so wird vom laufenden Prozeß (von sich selbst) eine Kopie gezogen und in die *bereit*-Liste eingehängt. Damit gibt es nun zwei fast identische Prozesse, die beide aus dem `fork()`-Aufruf zurückkehren. Der Unterschied zwischen beiden liegt im Rückgabewert der Funktion: Der Elternprozeß erhält die Prozeßkennzahl PID des Kindes; das Kind erhält PID=0 und erkennt daran, daß es der Kindsprozeß ist und den weiteren Programmablauf durch Abfragen anders gestalten kann. Es kann z. B. mit `exec('program')` bewirken, daß das laufende Programm mit Programmcode aus der Datei „`program`“ überladen wird. Alle Zeiger und Variablen werden initialisiert (z. B. der Programmzähler auf die Anfangsadresse des Programms gesetzt) und der fertige Prozeß in die *bereit*-Liste eingehängt. Damit ist im Endeffekt vom Elterprozeß ein völlig neues Programm gestartet worden.

Der Elternprozeß hat dann die Möglichkeit, auf einen `exit()`-Systemaufruf und damit auf das Ende des Kinds mit einem `waitpid(PID)` zu warten. In Abb. 2.4 ist der Ablauf einer solchen Prozeßerzeugung gezeigt.

Man beachte, daß der Kindsprozeß den `exit()`-Aufruf im obigen Beispiel nur dann erreicht, wenn ein Fehler bei `exec()` auftritt; z. B. wenn die Datei „`program`“ nicht existiert, nicht lesbar ist usw. Ansonsten ist der nächste Programmbefehl nach `exec()` (im *user mode*) identisch mit dem ersten Befehl des Programms „`program`“. Der Kindsprozeß beendet sich erst, wenn in „`program`“ selbst ein `exit()`-Aufruf erreicht wird.

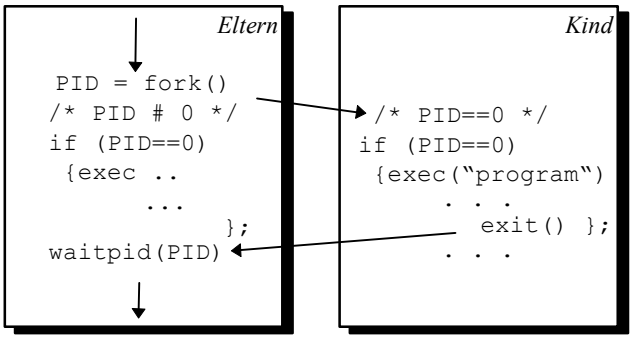



**Abb. 2.4** *Erzeugung und Vernichten eines Prozesses in UNIX*

Mit diesen Überlegungen wird folgendes Beispiel für einen Prozeß für Benutzereingaben am Terminal (*shell*) klar. Der Code ist allerdings nur das Grundgerüst der in UNIX tatsächlich für jeden Benutzer gestarteten *shell.*

**Beispiel shell**

```
LOOP
  Write(prompt);                    (* tippe z. B. '>' *)
  ReadLine(command, params);        (* lese strings, getrennt durch Leertaste *)
  pid := fork();                    (* erzeuge Kopie dieses Prozesses *)
  IF (pid=0)
   THEN execve(command,params,0)          (* Kind: überlade mit Programm *)
   ELSE waitpid(-1, status,0) (*Eltern: warte auf Ausführungsende vom Kind*)
  END;
END;
```

Alle Prozesse in UNIX stammen direkt oder indirekt von einem einzigen Prozeß ab, dem Init-Prozeß mit PID=1. Ist beim „Ableben“ eines Kindes kein Elternprozeß mehr vorhanden, so wird statt dessen „init“ benachrichtigt. In der Zeit zwischen dem `exit()`-Systemaufruf und dem Akzeptieren der Nachricht darüber bei dem Elternprozeß gelangt der Kindsprozeß in einen besonderen Zustand, genannt „Zombi“, s. Abb. 2.1.

Der interne Prozeßkontext ist in zwei Teile geteilt: einer, der als Prozeßeintrag (*Prozeßkontrollblock* PCB) in einer speicherresidenten Tabelle (*process table*) steht, für das Prozeßmanagement wichtig und deshalb immer präsent ist, und ein zweiter (*user structure*), der nur wichtig ist, wenn der Prozeß aktiv ist, und der deshalb auf den Massenspeicher mit dem übrigen Code und den Daten ausgelagert werden kann.

Die beiden Teile sind im wesentlichen

- *speicherresidente Prozeßkontrollblöcke* PCB der Prozeßtafel
  - Scheduling-Parameter
  - Speicherreferenzen: Code-, Daten-, Stackadressen im Haupt- bzw. Massenspeicher
  - Signaldaten: Masken, Zustände
  - Verschiedenes: Prozeßzustand, erwartetes Ereignis, Timerzustand, PID, PID der Eltern, User/Group-IDs
- *auslagerbarer Benutzerkontext (swappable user structure)*
  - Prozessorzustand: Register, FPU-Register, …
  - Systemaufruf: Parameter, bisherige Ergebnisse, …
  - Dateiinfo-Tabelle (file descriptor table)
  - Benutzungsinfo: CPU-Zeit, max. Stackgröße, …
  - Kernel-Stack: Stackplatz für Systemaufrufe des Prozesses

Im Unterschied zu dem PCB, den der Prozeß nur indirekt über Systemaufrufe abfragen und ändern kann, gestatten spezielle UNIX-Systemaufrufe, die *user structure* direkt zu inspizieren und Teile davon zu verändern.

### 2.1.2 Beispiel Windows NT

Da in Windows NT verschiedene Arten von Prozessen unterstützt werden müssen, deren vielfältige Ausprägungen sich nicht behindern dürfen, wurde nur für eine einzige, allgemeine Art von Prozessen, den sog. *thread objects*, ein Prozeßsystem geschaffen. Die speziellen Ausprägungen der OS/2-Objekte, POSIX-Objekte und Windows32-Objekte sind dabei in den Objekten gekapselt und spielen bei den Zustandsänderungen keine Rolle. Der vereinfachte Graph der Zustandsübergänge ist in Abb. 2.5 gezeigt.

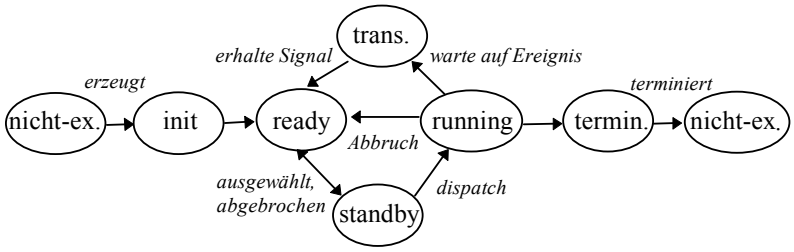



**Abb. 2.5** *Prozeßzustände in Windows NT*

Die Prozeßerzeugung in Windows NT ist etwas komplexer als in UNIX, da als Vorgabe mehrere Prozeßmodelle erfüllt werden mußten. Dazu wurden die speziellen Ausprägungen in den Subsystemen gekapselt. Zur Erzeugung von Prozessen gibt es nur einen einzigen Systemaufruf `NtCreateProcess()`, bei dem neben der Initialisierung durch Code auch der Elternprozeß angegeben werden kann. Auf diesem bauen alle anderen, subsystemspezifischen Aufrufvarianten auf, die von den Entwicklern benutzt werden.

Dies ermöglicht beispielsweise den POSIX-`fork()`-Mechanismus. Dazu ruft das POSIX-Programm (POSIX-Prozeß) über die API (*Application Programming Interface*) den `fork()`-Befehl auf. Dies wird in eine Nachricht umgesetzt und über den Kern an das POSIX-Subsystem geschickt, s. Abb. 1.9. Dieses wiederum ruft `NtCreateProcess()` auf und gibt als ElternPID das POSIX-Programm an. Der vom Kern zurückgegebene Objektschlüssel (*object handle*) wird dann vom POSIX-Subsystem verwaltet; alle Systemaufrufe vom POSIX-Prozeß, die als Nachrichten beim POSIX-Subsystem landen, werden dort mit Hilfe von NT-Systemaufrufen erledigt und die Ergebnisse im POSIX-Format an den aufrufenden Prozeß zurückgegeben. Analog gilt dies auch für die Prozeßaufrufe der anderen Subsysteme.

### 2.1.3 Leichtgewichtsprozesse

Der Speicherbedarf eines Prozesses ist meist sehr umfangreich. Er enthält nicht nur wenige Zahlen, wie Prozeßnummer und CPU-Daten, sondern auch beispielsweise alle Angaben über offene Dateien sowie den logischen Status der benutzten Ein- und Ausgabegeräte sowie den Programmcode und seine Daten. Dies ist bei sehr vielen Prozessen meist mehr, als in den Hauptspeicher paßt, so daß bis auf wenige Daten die Prozesse auf den Massenspeicher (Festplatte) ausgelagert werden. Da beim Prozeßwechsel der aktuelle Speicher ausgelagert und der alte von der Festplatte wieder restauriert werden muß, ist ein Prozeßwechsel eine „schwere“ Systemlast und dauert relativ lange.

Bei vielen Anwendungen werden keine völlig neuen Prozesse benötigt, sondern nur unabhängige Codestücke (*threads*), die im gleichen Prozeßkontext agieren, beispielsweise Prozeduren eines Programms. In diesem Fall spricht man auch von **Coroutinen**. Ein typisches Anwendungsbeispiel ist das parallele Ausführen unterschiedlicher Funktionen bei Texteditoren, bei denen gleichzeitig zur Eingabe der Zeichen auch bereits der Text auf korrekte Rechtschreibung überprüft wird.

Die Verwendung von *threads* ermöglicht es, innerhalb eines Prozesses ein weiteres Prozeßsystem aus sog. **Leichtgewichtsprozessen** (LWP) zu schaffen. In der einfachsten Form übergeben sich die Prozesse gegenseitig explizit die Kontrolle („Coroutinen-Konzept“). Dies bedingt, daß die anderen Prozesse bzw. ihre Namen (ID) den aufrufenden Prozessen auch bekannt sind. Je mehr Prozesse erzeugt werden, desto schwieriger wird dies. Aus diesem Grund kann man auch hier einen Scheduler einführen, der die Kontrolle immer erhält und sie an einen Prozeß aus

seiner *bereit*-Liste weitergibt. Wird dies nicht vom Anwender programmiert, sondern ist bereits im Betriebssystem über Systemaufrufe enthalten, so werden die *thread*-Prozesse durch die Umschaltzeiten der Systemaufrufe etwas schwergewichtiger.

Jeder Prozeß muß seine eigenen Daten unabhängig vom anderen halten. Dies gilt auch für Leichtgewichtsprozesse, auch wenn sie die gleichen Dateien und den gleichen Speicher (genauer: den gleichen virtuellen Adreßraum, s. Kapitel 3) mit den anderen Leichtgewichtsprozessen teilen. Dazu wird meist sein Stack verwendet, der für jeden Prozeß extra als Platz bei der Erzeugung reserviert wird. Im Unterschied zu den „echten" Prozessen benötigen Leichtgewichtsprozesse deshalb nur wenige Kontextdaten, die beim Umschalten geändert werden müssen. Die wichtigsten sind das Prozessorstatuswort PS und der Stackpointer SP. Selbst der Programmzähler PC kann auf dem Stack abgelegt sein, so daß er nicht explizit übergeben werden muß. Effiziente Implementierungen des Umschaltens in Assemblersprache machen diese Art von Prozeßsystemen sehr schnell.

### 2.1.4 Beispiel UNIX

In UNIX sind die Leichtgewichtsprozesse (*threads*) als Benutzerbibliothek in C oder C++ implementiert, siehe UNIX-Manual Kapitel 3. Je nach Implementierung gibt es einfache Systeme mit direkter Kontrollübergabe (Coroutinen-Konzept) oder komplexere mit einem extra Scheduler und all seinen Möglichkeiten, siehe Abschnitt 2.2.

Der Vorteil einer Implementierung von *threads* als Bibliothek (z. B. Northrup 1996) besteht in einer sehr schnellen Prozeßumschaltung (*lightweight threads*), da die Mechanismen des Betriebssystemaufrufs und seiner Dekodierung nach Dienstnummer und Parametern nicht durchlaufen werden müssen. Der Nachteil besteht darin, daß ein *thread*, der auf ein Ereignis (z. B. I/O) wartet, den gesamten Prozeß mit allen *threads* blockiert. Im oben erwähnten Beispiel eines Texteditors würde der Teil, der die Rechtschreibüberprüfung durchführt, während einer Tipp-Pause blockiert werden; die Vorteile der Benutzung von *threads* (parallele Ausführung und Ausnutzung freier CPU-Zeit) wären dahin. Eine solche *thread*-Bibliothek eignet sich deshalb eher für unabhängige Programmstücke, die keine Benutzereingabe benötigen.

Der Aufruf `vfork()` in UNIX, der einen echten, neuen Prozeß bei altem Prozesskontext schafft, ist nicht identisch mit einem *thread*; er übernimmt den Kontrollfluß des Elternprozesses und blockiert ihn damit. In LINUX ist er identisch mit einem `fork()`.

Es gibt Versuche, die *threads* zu normieren, um Programmportierungen zu erleichtern (siehe IEEE 1992). In den neuen UNIX-Spezifikationen für ein 64-Bit UNIX (UNIX-98,) sind sie deshalb enthalten und gehören fest zu „UNIX".

### 2.1.5 Beispiel Windows NT

In Windows NT sind im Unterschied zu UNIX die LWP als Betriebssystemaufrufe implementiert. Zwar dauert damit die Umschaltung länger (*heavyweight threads*), aber zum einen hat dies den Vorteil, daß der Systemprogrammierer eine feste, verbindliche Schnittstelle hat, an die er oder sie sich halten kann. Es erleichtert die Portierung von Programmen, die solche LWP benutzen, und verhindert die Versuchung, eigene abweichende Systeme zu entwickeln, wie dies für UNIX lange Zeit der Fall war. Zum anderen hat der BS-Kern damit auch die Kontrolle über die LWP, was besonders wichtig ist, wenn in Multiprozessorsystemen die LWP echt parallel ausgeführt werden oder für I/O nur ein *thread* eines Prozesses blockiert werden soll, wie z.B. bei unserem Texteditor.

Da ein solcher Schwergewichts-*thread* wieder zu unnötigen Verzögerungen bei unkritischen Teilen innerhalb einer Anwendung führt, wurden in Windows NT ab Version 4 sogenannte *fibers* eingeführt. Dies sind parallel ablaufende Prozeduren, die nach dem Coroutinen-Konzept funktionieren: Die Abgabe der Kontrolle von einer *fiber* an eine andere innerhalb eines *threads* erfolgt freiwillig; ist der *thread* blockiert, so sind alle *fibers* darin ebenfalls blockiert. Dies erleichtert auch die Portierung von Programmen auf UNIX-Systeme.

### 2.1.6 Aufgaben

**Aufgabe 2.1-1 (Betriebsarten):**

Nennen Sie einige Betriebsarten eines Betriebssystems (inklusive der drei wichtigsten).

**Aufgabe 2.1-2 (Prozesse)**

a) Erläutern Sie nochmals die wesentlichen Unterschiede zwischen Programm, Prozeß und *thread*.
b) Wie sehen im UNIX-System des Bereichsrechners ein Prozeßkontrollblock (PCB) und die *user structure* aus? Inspizieren Sie dazu die *include files* /include/sys/proc.h und /include/sys/user.h und charakterisieren Sie grob die Einträge darin.

**Aufgabe 2.1-3 (Prozesse)**

a) Welche Prozeßzustände durchläuft ein Prozeß?
b) Was sind typische Wartebedingungen und Ereignisse?
c) Ändern Sie die Zustandsübergänge im Zustandsdiagramm von Abb. 2.2 durch ein abweichendes, aber ebenso sinnvolles Schema. Begründen Sie Ihre Änderungen.

**Aufgabe 2.1-4 (UNIX-Prozesse)**

Wie ließe sich in UNIX mit der Sprache C oder MODULA-2 ein Systemaufruf „ExecuteProgram(prg)“ als Prozedur mit Hilfe der Systemaufrufe `fork()` und `waitpid()` realisieren?

**Aufgabe 2.1-5 (threads)**

Realisieren Sie zwei Prozeduren als Leichtgewichtsprozesse, die sich wechselseitig die Kontrolle übergeben. Der eine Prozeß gebe „Ha“ und der andere „tschi!“ aus.

In MODULA-2 ist die Verwendung von Prozeduren als LWP mit Hilfe der Konstrukte NEWPROCESS und TRANSFER leicht möglich. Ein LWP ist in diesem Fall eine **Coroutine**, die parallel zu einer anderen ausgeführt werden kann. Dabei gelten folgende Schnittstellen:

```
PROCEDURE NEWPROCESS( Prozedurname: PROCEDURE;
                      Stackadresse: ADDRESS;
                      Stacklänge:   CARDINAL;
                      Coroutine:    ADDRESS )
```

Die Prozedur `NEWPROCESS` wird nur einmal aufgerufen und initialisiert die Datenstrukturen für die als Coroutine zu verwendende Prozedur. Dazu verwendet sie einen (vorher zu reservierenden) Speicherplatz.
Die Coroutinenvariable vom Typ `ADDRESS` hat dabei die Funktion eines Zeigers in diesen Speicherbereich, in dem alle Registerinhalte gerettet werden, bevor die Kontrolle einer anderen Coroutine übergeben wird.

```
PROCEDURE TRANSFER  ( QuellCoroutine,
                      ZielCoroutine: ADDRESS)
```

Bei der Prozedur `TRANSFER` wird die Prozessorkontrolle von einer Coroutine `QuellCoroutine` zu einer Coroutine `ZielCoroutine` übergeben.

## 2.2 Prozeßscheduling

Gibt es in einem System mehr Bedarf an Betriebsmitteln, als vorhanden sind, so muß der Zugriff koordiniert werden. Eine wichtige Rolle spielt dabei der oben erwähnte Scheduler und seine Strategie beim Einordnen der Prozesse in Warteschlangen. Betrachten wir dazu die Einprozessorsysteme, auf denen voneinander unabhängige Prozesse hintereinander (*sequentiell*) ablaufen.

In einem normalen Rechensystem können wir zwei Arten von Scheduling-Aufgaben unterscheiden: das Planen der Jobausführung (**Langzeitscheduling**, Jobscheduling) und das Planen der aktuellen Prozessorzuteilung (**Kurzzeitscheduling**). Beim Langzeitscheduling wird darauf geachtet, daß nur so viele

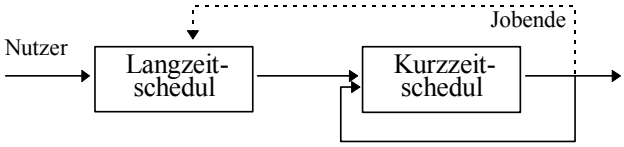



**Abb. 2.6** *Langzeit- und Kurzzeitscheduling*

Benutzer mit ihren Jobs neu ins System kommen (*log in*) dürfen, wie Benutzer sich abmelden (*log out*). Sind es zu viele, so muß der Zugang gesperrt werden, bis die Systemlast erträglich wird.
Ein Beispiel dafür ist die Zugangskontrolle von Benutzern über das Netz zu Dateiservern (ftp-server, www-server). Eine weitere Aufgabe für Langzeitscheduling ist das Ausführen von nicht-interaktiv ablaufenden Jobs („Batch-Jobs“) zu bestimmten Zeiten, z. B. nachts.

Die meiste Arbeit wird allerdings in das Kurzzeitscheduling, die Strategie zur Zuweisung des Prozessors an Prozesse, gesteckt. Im folgenden betrachten wir einige der gängigsten Strategien dafür.

### 2.2.1 Zielkonflikte

Alle Schedulingstrategien versuchen, gewisse Ziele zu verwirklichen. Die gängigsten Ziele sind:

- *Auslastung der CPU*

  Ist die CPU das Betriebsmittel, das am wenigsten vorhanden ist, so wollen wir den Gebrauch möglichst effizient gestalten. Ziel ist die 100%ige **Auslastung** der CPU, normal sind 40%–90%.

- *Durchsatz*

  Die Zahl der Jobs pro Zeiteinheit, der **Durchsatz** (*throughput*), ist ein weiteres Maß für die Systemauslastung und sollte maximal sein.

- *Faire Behandlung*

  Kein Job sollte dem anderen bevorzugt werden, wenn dies nicht ausdrücklich vereinbart wird (**fairness**). Dies bedeutet, daß jeder Benutzer im Mittel den gleichen CPU-Zeitanteil erhalten sollte.

- *Ausführungszeit*

  Die Zeitspanne vom Jobbeginn bis zum Jobende, die **Ausführungszeit** (*turnaround tim*e), enthält alle Zeiten in Warteschlangen, der Ausführung (**Bedienzeit**) und der Ein- und Ausgabe. Natürlich sollte sie minimal sein.

- *Wartezeit*

  Der Scheduler kann von der gesamten Ausführungszeit nur die **Wartezeit** *(waiting time)* in der *bereit*-Liste beeinflussen. Für die Schedulerstrategie kann man sich also als Ziel darauf beschränken, die Wartezeit zu minimieren.

- *Antwortzeit*

  In interaktiven Systemen empfindet der Benutzer es als besonders unangenehm, wenn nach einer Eingabe die Reaktion des Rechners lange auf sich warten läßt. Unabhängig von der gesamten Ausführungszeit des Jobs sollte besonders die Zeit zwischen einer Eingabe und der Übergabe der Antwortdaten an die Ausgabegeräte, die **Antwortzeit** (*response time*), auch minimal werden.

Die Liste der möglichen Zielvorgaben ist weder vollständig noch konsistent. Beispielsweise benötigt jede Prozeßumschaltung einen Wechsel des Prozeßkontextes (*context switch*). Werden die kurzen Prozesse bevorzugt, so verkürzt sich zwar die Antwortzeit mit der Ablaufzeit zwischen zwei Eingaben und der Durchsatz steigt an, aber der relative Zeitanteil der langen Prozesse wird geringer; sie werden benachteiligt und damit die Fairneß verletzt. Wird andererseits die Auslastung erhöht, so verlängern sich die Antwortzeiten bei interaktiven Jobs.

Dies kann man analog dazu auch im täglichen Leben sehen: Werden bei einer Autovermietung bestimmte Kunden trotz großen Andrangs bevorzugt bedient, so müssen andere Kunden länger warten. Werden die Mietwagen gut ausgelastet, so muß ein neu hinzukommender Kunde meist warten, bis er einen bekommt; für eine kurze Reaktionszeit müssen ausreichend viele Wagen eines Modells zur Verfügung stehen.

Da sich für jeden Benutzerkreis die Zielvorgaben verändern können, gibt es keinen idealen Schedulingalgorithmus für jede Situation. Aus diesem Grund ist es sehr sinnvoll, die Mechanismen des Scheduling (Umhängen von Prozessen aus einer Warteschlange in die nächste etc.) von der eigentlichen Schedulingstrategie und ihren Parametern abzutrennen. Erzeugt beispielsweise ein Datenbankprozeß einige Hilfsprozesse, deren Charakteristika er kennt, so sollte es möglich sein, die Schedulingstrategie der Kindsprozesse durch den Elternprozeß zu beeinflussen. Die BS-Kern-eigenen, internen Scheduling- und Dispatchingmechanismen sollten dazu über eine genormte Schnittstelle (Systemaufrufe) benutzt werden; die Schedulingstrategie selbst aber sollte vom Benutzer programmiert werden können.

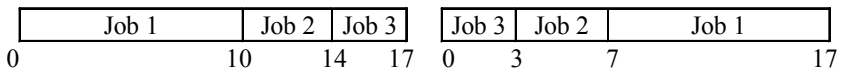
### 2.2.2 Non-preemptives Scheduling

Im einfachsten Fall können die Prozesse so lange laufen, bis sie von sich aus den Aktivzustand verlassen und auf ein Ereignis (I/O, Nachricht etc.) warten, die Kontrolle an andere Prozesse abgeben oder sich selbst beenden: Sie werden nicht vorzeitig unterbrochen (**non-preemptive scheduling**). Diese Art von Scheduling ist bei allen Systemen sinnvoll, bei denen man genau weiß, welche Prozesse existie-

ren und welche Charakteristika sie haben. Ein Beispiel dafür ist das oben erwähnte Datenbankprogramm, das genau weiß, wie lange eine Transaktion normalerweise dauert. In diesem Fall kann man ein Leichtgewichtsprozeßsystem zur Realisierung verwenden. Für diese Art von Scheduling gibt es die folgenden, am häufigsten verwendeten Strategien:

- *First Come First Serve (FCFS)*

  Eine sehr einfache Strategie besteht darin, die Prozesse in der Reihenfolge ihres Eintreffens in die Warteschlange einzusortieren. Damit kommen alle Tasks an die Reihe, egal wieviel Zeit sie verbrauchen. Die Implementierung dieser Strategie mit einer FIFO Warteschlange ist sehr einfach.
  Die Leistungsfähigkeit dieses Algorithmus ist allerdings auch sehr begrenzt. Nehmen wir an, wir haben 3 Jobs der Längen 10, 4 und 3, die fast gleichzeitig eintreffen. Diese werden nach FCFS eingeordnet und bearbeitet. In Abb. 2.7 (a) ist dies gezeigt. Die Ausführungszeit (*turnaround time*) von Job 1 ist in diesem Beispiel 10, von Job 2 ist sie 14 und 17 von Job 3, so daß die mittlere Ausführungszeit (10+14+17):3=13,67 beträgt. Ordnen wir allerdings die Jobs so an, daß die kürzesten zuerst bearbeitet werden, so ist für unser Beispiel in (b) die mittlere Ausführungszeit (3+7+17):3=9 und damit kürzer. Dieser Sachverhalt führt uns zur nachfolgenden Strategie.




(a) FCFS-Reihenfolge (b) SJN-Reihenfolge

**Abb. 2.7** *Jobreihenfolgen*

- *Shortest Job First (SJF)*

  Der Prozeß mit der (geschätzt) kürzesten Bedienzeit wird allen anderen vorgezogen. Diese Strategie vermeidet zum einen die oben beschriebenen Nachteile von FCFS. Zum anderen bevorzugt sie stark interaktive Prozesse, die wenig CPU-Zeit brauchen und meist in Ein-/Ausgabeschlangen auf den Abschluß der parallel ablaufenden Aktionen der Ein-/Ausgabekanäle (DMA) warten, so daß die mittlere Antwortzeit gering gehalten wird.

  Man kann zeigen, daß SJF die mittlere Wartezeit eines Jobs für eine gegebene Menge von Jobs minimiert, da beim Vorziehen des kurzen Jobs seine Wartezeit stärker absinkt, als sich die Wartezeit des langen Jobs erhöht. Aus diesem Grund ist es die Strategie der Wahl, wenn keine anderen Gesichtspunkte dazukommen.

  Ein Problem dieser Strategie besteht darin, daß bei großem Zustrom von kurzen Prozessen ein bereiter Prozeß mit großen CPU-Anforderungen, obwohl

nicht blockiert, nie die CPU erhält. Dieses als **Verhungern** (**starvation)** bekannte Problem tritt übrigens auch in anderen Situationen auf.

- *Highest Response Ratio Next (HRN)*

  Hier werden die Jobs mit großem Verhältnis (Antwortzeit/Bedienzeit) bevorzugt bearbeitet, wobei für Antwortzeit und Bedienzeit die Schätzungen verwendet werden, die auf vorher gemessenen Werten basieren. Diese Strategie zieht Jobs mit kurzer Bedienzeit vor, begrenzt aber auch die Wartezeit von Jobs mit langen Bedienzeiten, weil bei einer Benachteiligung auch deren Antwortzeit zunimmt.

- *Priority Scheduling (PS)*

  Jedem Prozeß wird initial beim Start eine **Priorität** zugeordnet. Kommt ein neuer Prozeß in die Warteschlange, so wird er so einsortiert, daß die Prozesse mit höchster Priorität am Anfang der Schlange stehen; die mit geringster am Ende. Sind mehrere Prozesse gleicher Priorität da, so muß innerhalb dieser Prozesse die Reihenfolge nach einer anderen Strategie, z. B. FCFS, entschieden werden.

  Man beachte, daß auch hier wieder benachteiligte Jobs verhungern können. Bei Prioritäten läßt sich dieses Problem dadurch umgehen, daß die Prioritäten nicht fest sind, sondern dynamisch verändert werden. Erhält ein Prozeß in regelmäßiger Folge zusätzliche Priorität, so wird er irgendwann die höchste Priorität haben und so ebenfalls die CPU erhalten.

Die Voraussetzung von *SJF* und *HRN* (einer überschaubaren Menge von Prozessen mit bekannten Charakteristika) wird durch die Tatsache in Frage gestellt, daß die Ausführungszeiten von Jobs sehr uneinheitlich sind und häufig wechseln. Der Nutzen der Strategien bei sehr uneinheitlichen Systemen ist deshalb begrenzt. Anders ist dies im Fall von häufig auftretenden, gut überschaubaren und bekannten Jobs wie sie beispielsweise in Datenbanken oder Prozeßsystemen (Echtzeitsysteme) vorkommen. Hier lohnt es sich, die unbekannten Parameter ständig neu zu schätzen und so das Scheduling zu optimieren.

Die ständige Anpassung dieser Parameter (wie Ausführungszeit und Bedienzeit) an die Realität kann man mit verschiedenen Algorithmen erreichen. Einer der bekanntesten besteht darin, für einen Parameter *a* eines Prozesses zu jedem Zeitpunkt *t* den gewichteten Mittelwert aus dem aktuellen Wert $b_t$ und dem früheren Wert a(t) zu bilden:

$$a(t) = (1-\alpha)\, a(t-1) + \alpha\, b_t$$

Es ergibt sich hier die Reihe

$$\begin{aligned} a(0) &\equiv b_0 \\ a(1) &= (1-\alpha)\, b_0 + \alpha b_1 \\ a(2) &= (1-\alpha)^2 b_0 + (1-\alpha)\, \alpha b_1 + \alpha b_2 \end{aligned}$$

$$a(3) = (1-\alpha)^3 b_0 + (1-\alpha)^2 \alpha b_1 + (1-\alpha)\,\alpha b_2 + \alpha b_3$$
$$\ldots$$
$$a(n) = (1-\alpha)^n b_0 + (1-\alpha)^{n-1} \alpha b_1 + \ldots + (1-\alpha)^{n-i} \alpha b_i + \ldots + \alpha b_n$$

Wie man sieht, schwindet mit $\alpha<1$ der Einfluß der frühen Messungen exponentiell, der Parameter *altert*. Dieses Prinzip findet man bei vielen adaptiven Verfahren. Die geringere Gewichtung der früheren Messungen bewirkt, daß sich Verhaltensänderungen des Prozesses auch in den aktuellen Parameterschätzungen wiederfinden. Betrachtet man die Folge der Parameterwerte als Ereignisse einer stochastischen Variablen, so heißt dies, daß die Verteilung der Variablen nicht konstant ist, sondern sich mit der Zeit verändert. Ein solcher adaptiver Algorithmus ermittelt also nicht einfach den Mittelwert der Messungen (erwarteter Parameterwert, siehe Aufgabe 2.2-1), sondern schätzt den aktuellen Stand einer zeitabhängigen Verteilungsfunktion.

Für den Fall $\alpha=½$ läßt sich der Algorithmus besonders schnell implementieren: Die Division durch zwei entspricht gerade einer Shift-Operation der Zahl nach rechts um eine Stelle.

Die adaptive Schätzung der Parameter eines Prozesses für einen Schedulingalgorithmus (*adaptive Prozessorzuteilung*) muß für jeden Prozeß extra durchgeführt werden; in unserem Beispiel müßte *a* also einen doppelten Index bekommen: einen für die Parameternummer pro Prozeß und einen für die Prozeßnummer. Die Schätzmethode für die Parameter ist unabhängig vom Algorithmus, der für das Scheduling verwendet wird; aus diesem Grund gilt das obige nicht nur für die Algorithmen des non-preemptiven Scheduling, sondern auch für die Verfahren des preemptiven Scheduling des nächsten Abschnitts.

### 2.2.3 Preemptives Scheduling

In einem normalen Mehrbenutzersystem, bei dem die verschiedenen Jobs von verschiedenen Benutzern gestartet werden, gibt es zwangsläufig Ärger, wenn ein Job alle anderen blockiert. Hier ist ein anderes Scheduling erforderlich, bei dem jeder Job vorzeitig unterbrochen werden kann: das **preemptive Scheduling**.

Eine der wichtigsten Maßnahmen dieses Verfahrens besteht darin, die verfügbare Zeitspanne für das Betriebsmittel, meist die CPU, in einzelne, gleich große Zeitabschnitte (**Zeitscheiben**) aufzuteilen. Wird ein Prozeß bereit, so wird er in die Warteschlange an einer Stelle einsortiert. Zu Beginn einer neuen Zeitscheibe wird der Dispatcher per Zeitinterrupt aufgerufen, der bisher laufende Prozeß wird abgebrochen und wie ein neuer *bereit*-Prozeß in die Warteschlange eingereiht. Dann wird der Prozeß am Anfang der Schlange in den Aktivzustand versetzt. Dies ist in Abb. 2.8 dargestellt. Die senkrechten Striche symbolisieren die Prozesse, die von links in das „Gefäß“, die Warteschlange, hineingeschoben werden. Die Bearbeitungseinheit, hier der Prozessor, ist als Ellipse visualisiert. Nach einem Abbruch wird der Prozeß an einen Platz zwischen die anderen Prozesse in der Warteschlange eingereiht.

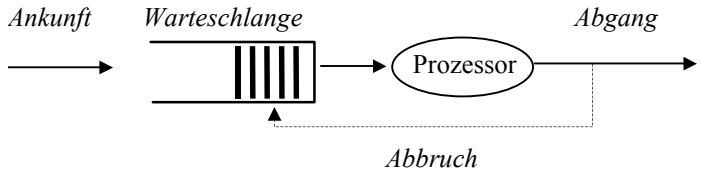



**Abb. 2.8** *Preemptives Scheduling*

Für die Wahl dieses Platzes gibt es verschiedene Strategien:

- *Round Robin (RR)*

  Die einfachste Strategie beim Zeitscheibenverfahren ist die FCFS-Strategie mit der FIFO-Schlange. Die Kombination von FCFS und Zeitscheibenverfahren ist auch als der **Round-Robin**-Algorithmus bekannt. Die analytische Untersuchung zeigt, daß hier die Antwortzeiten proportional zu den Bedienzeiten sind und unabhängig von der Verteilung der Bedienzeiten nur von der mittleren Bedienzeit abhängen.

  Es ist klar, daß die Leistungsfähigkeit des RR stark von der Größe der Zeitscheibe abhängt. Wählt man die Zeitscheibe unendlich groß, so wird nur noch der einfache FCFS Algorithmus ausgeführt. Wählt man die Zeitscheibe dagegen sehr klein (z. B. gerade eine Instruktion), so erhalten alle *n* Jobs jeweils 1/n Prozessorleistung; der Prozessor teilt sich in *n* virtuelle Prozessoren auf. Dies funktioniert allerdings nur dann so, wenn der Prozessor sehr schnell gegenüber der Peripherie (Speicher) ist und die Prozeßumschaltung durch Hardwaremechanismen sehr schnell bewirkt wird (z. B. bei der CDC6600) und praktisch keine Zeit beansprucht. Auf Standardsystemen ist dies aber meist nicht der Fall. Hier geschieht die Umschaltung des Prozeßkontextes durch Softwaremechanismen und benötigt extra Zeit, in der Größenordnung von 10-100 Mikrosekunden. Wählen wir die Zeitscheibe zu klein, so wird das Verhältnis Arbeitszeit/Umschaltzeit zu klein, der Durchsatz verschlechtert sich, und die Wartezeiten steigen an. Im Extremfall schaltet der Prozessor nur noch um und bearbeitet den Job nie.

  Für ein vernünftiges Verhalten zwischen FCFS und Dauerumschalten ist die Kenntnis verschiedener Parameter nötig. Eine bewährte Daumenregel besteht darin, die Zeitscheibe größer zu machen als der mittlere CPU-Zeitbedarf zwischen zwei I/O-Aktivitäten (*cpu-burst*) von 80% der Jobs. Dies entspricht meist einem Wert von ca. 100 ms.

- *Dynamic Priority Round Robin (DPRR)*

  Das RR-Scheduling läßt sich noch durch eine Vorstufe, einer prioritätsgeführten Warteschlange, für die Jobs ergänzen. Die Priorität der Prozesse in der Vorstufe wächst nach jeder Zeitscheibe so lange, bis sie die Schwellenpriorität des eigentlichen RR-Verfahrens erreicht haben und in die Hauptschlange ein-

sortiert werden. Damit wird eine unterschiedliche Bearbeitung der Jobs nach Systemprioritäten erreicht, ohne das RR-Verfahren direkt zu verändern.

- *Shortest Remaining Time First*

  Unsere SJF-Strategie zum Einsortieren in die Schlange bedeutet hier, daß derjenige Job bevorteilt wird, der die kleinste noch verbleibende Bedienzeit vorweist.

Eine weitere Gruppe von Algorithmen erhalten wir, wenn wir zum Abbrechen anstelle der Zeitscheibe die Prioritäten verwenden. Das **Priority Scheduling** bedeutet im preemptiven Fall, daß der laufende Prozeß durch einen neu hinzukommenden Prozeß (z. B. aus einer I/O-Warteschlange) höherer Priorität verdrängt werden kann und wieder in die *bereit*-Liste einsortiert wird.

Auch eine Kombination beider Verfahren ist üblich. Dabei wird die FCFS-Strategie der Round-Robin-Warteschlange auf Prioritäten umgestellt.

### 2.2.4 Multiple Warteschlangen und multiple Scheduler

In einem modernen Einprozessorsystem gibt es zwar nur einen Hauptprozessor, aber fast alle schnelleren Ein- und Ausgabegeräte verfügen über einen Controller, der unabhängig vom Hauptprozessor Daten aus dem Hauptspeicher auf den Massenspeicher und umgekehrt schaffen kann (*Direct Memory Access DMA*). Diese DMA-Controller wirken damit wie eigene, spezialisierte Prozessoren und können als eigene, unabhängige Betriebsmittel betrachtet werden. Es ist deshalb sinnvoll, für jeden Ein-/Ausgabekanal eine eigene Warteschlange einzurichten, die von dem DMA-Controller bedient wird. Das gesamte Dispatching besteht also aus einem Umhängen von Jobs von einer Warteschlange in die nächste, wobei kurze CPU-Aktivitäten (*CPU bursts*) dazwischenliegen. In Abb. 2.9 ist ein solches System von Warteschlangen gezeigt.

Eine weitere Variante besteht darin, daß wir nicht nur eine Art von Jobs haben, sondern mehrere unterschiedlicher Priorität und deshalb für jede Jobart eine eigene Warteschlange eröffnen (**Multi-level-Scheduling**).

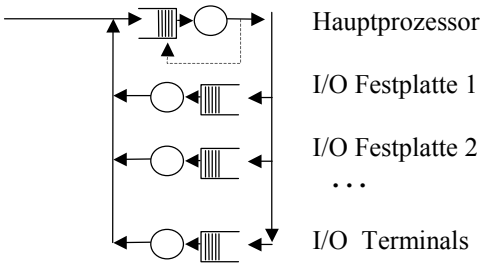



**Abb. 2.9** *Scheduling mit multiplen Warteschlangen*

Durch die unterschiedlichen Prioritäten sind die Warteschlangen in einer bestimmten Reihenfolge angeordnet, die die Abarbeitungsreihenfolge bestimmt: Ist die erste Schlange höchster Priorität abgearbeitet, so folgt die zweite usw. Da ständig neue Jobs hinzukommen, wechselt die Abarbeitungsreihenfolge auch ständig. In Abb. 2.10 ist dies an einem 4-Ebenen-System visualisiert. Können die Jobs bei längerer Wartezeit in eine Warteschlange höherer Ordnung überwechseln, so spricht man von *multi-level feedback* Scheduling.

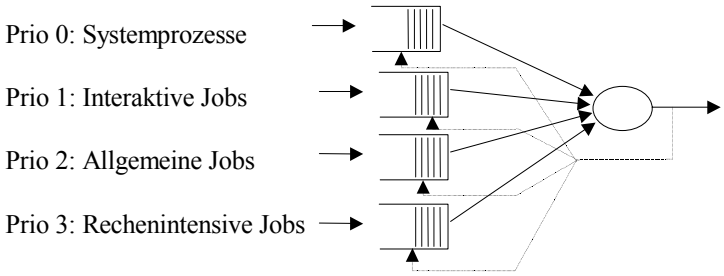



**Abb. 2.10** *Multi-level Scheduling*

Bei unseren Schedulingalgorithmen haben wir bisher die meistvorhandene Situation vernachlässigt, daß nicht alle Prozesse im Hauptspeicher gleichzeitig verfügbar sein können. Um trotzdem ein Scheduling der Prozesse durchführen zu können, werden nur die wichtigsten Daten eines Prozesses, zusammengefaßt im **Prozeßkontrollblock** PCB, im Hauptspeicher gehalten; alle anderen Daten werden auf den Massenspeicher verlagert (**geswappt**). Wird ein Prozeß aktiviert, so muß er erst durch Kopieren (*swapping*) vom Massenspeicher in den Hauptspeicher geholt und dann erst ausgeführt werden. Dies erfordert erhebliche, zusätzliche Zeit beim Wechsel des Prozeßkontextes und erhöht die Bearbeitungsdauer. Am besten ist es dagegen, jeweils den „richtigen“ Prozeß bereits im Hauptspeicher zu haben. Wie soll dazu der Übergang eines Prozesses vom Massenspeicher zum Hauptspeicher geregelt werden?

Eine Lösung für dieses Problem ist die Einführung eines zweiten Schedulers, der nur für das Ein- und Auslagern der Prozesse verantwortlich ist. Der erste Scheduler managt die Zuordnung der Prozesse zum Betriebsmittel „Prozessor“ und ist kurzzeitig tätig; der zweite Scheduler ist ein Mittel- bzw. Langzeitscheduler (ähnlich wie in Abb. 2.6) und regelt die Zuordnung der Prozesse zum Betriebsmittel „Hauptspeicher“, indem er die Größe des vom Prozeß belegten Hauptspeicherbereichs reguliert. Er wird in größeren Zeitabständen aufgerufen als der Kurzzeitscheduler. Beide verfügen über eigene Warteschlangen und regeln Zugang und Ordnung dafür. Strategien für einen solchen Scheduler werden wir in Abschn. 3.3 kennenlernen.

### 2.2.5 Scheduling in Echtzeitbetriebssystemen

Es gibt eine Reihe von Computersystemen, die als **Echtzeitsysteme** (**real time systems**) bezeichnet werden. Was fällt unter diese Bezeichnung? Eine intuitive Auffassung davon betrachtet alle Systeme, die schnell reagieren müssen, als Echtzeitsysteme. Was heißt aber „schnell"? Dies können wir genauer fassen: ein System, das Jobs ausführt, die expliziten Zeitbedingungen gehorchen müssen. Aber auch dies ist noch nicht präzise genug. Fast jedes System muß Zeitbedingungen gehorchen: Ein Editor sollte nicht länger als zwei Sekunden benötigen, um ein Zeichen auf dem Bildschirm sichtbar einzufügen; eine Bank sollte eine Überweisung innerhalb einer Woche durchführen, um Ärger zu vermeiden. Im Unterschied zu diesen auch als „*Soft*-Echtzeitsysteme" bezeichneten Systemen, bei denen die Zeitschranken nur „weich" und unspezifiziert sind und eine Nichterfüllung keine schweren Konsequenzen nach sich zieht, sind es die „*Hard*-Echtzeitsysteme" oder auch kurz nur „Echtzeitsysteme" genannten Rechner, die in Kernkraftwerken, Flugzeugen und Fahrzeugsteuerungen eingesetzt werden. Ein solches Echtzeitsystem ist ein System, das explizite endliche Zeitgrenzen für die Prozesse erfüllen muß, um ernsthafte Konsequenzen, z. B. einen Mißerfolg oder Ausfall des Systems zu verhindern. Dabei bezeichnen wir mit „Mißerfolg" jedes Systemverhalten, das die formale Systemspezifikation nicht mehr erfüllt.

Die Schedulingalgorithmen dafür orientieren sich an der Art der Prozesse. Typisch für Echtzeitsysteme ist die Situation, daß die zeitkritischen *tasks* zu genau festgesetzten Zeiten immer wiederkehren und damit sowohl in der Häufigkeit ihres Auftretens als auch in der Dauer, Betriebsmittelbelegung etc. vorhersehbar sind und es sich deshalb lohnt, einen festen Schedul dafür zu konstruieren.

**Beispiel** *Periodische Tasks*

Ein Flugzeug (z. B. Airbus A-340) wird von Rechnern gesteuert (*flight-by-wire*). Zur Steuerung benötigen die Rechner verschiedene Flugdaten, die in unterschiedlichen Intervallen ermittelt und verarbeitet werden müssen: die Beschleunigungswerte in x-, y- und z-Richtung alle 5 ms, die drei Werte der Drehungen alle 40 ms, die Temperatur alle Sekunde und die absolute Position zur Kontrolle alle 10 Sekunden. Der Display auf den Monitorschirmen wird jede Sekunde aktualisiert.

Die wichtigsten Schedulingstrategien für Echtzeitsysteme sind (s. Laplante 1993):

- *Polled Loop*

  Der Prozessor führt eine Schleife aus, bei der er immer wiederkehrend die Geräte abfragt, ob neue Daten vorhanden sind. Wenn ja, so verarbeitet er sie sofort. Diese Strategie eignet sich für Einzelgeräte, versagt aber, wenn andere Ereignisse während der Bearbeitung auftreten und die Daten nicht gepuffert werden.

- *Interruptgesteuerte Systeme*

  Der Prozessor führt als *task* eine Warteschleife (*idle loop*) aus. Treffen neue Daten ein, so wird jeweils ein Interrupt von dem Gerät ausgelöst und die *Interrupt Service Routine* ISR bearbeitet die neuen Daten. Werden den ISR Prioritäten zugeordnet, so findet damit automatisch durch die Unterbrechungslogik der Interruptbehandlung ein Prioritätsscheduling statt. Problematisch ist diese Strategie, wenn sich die Ereignisse häufen: Interrupts geringerer Priorität unterbrechen nicht und können so überschrieben werden, bevor sie ausgewertet werden können.

- *Minimal Deadline First*

  Der Prozeß wird zuerst abgearbeitet, der die kleinste, also nächste Zeitschranke (*deadline*) $T_D$ besitzt. Diese Strategie wird zwar oft bei Anwendungen benutzt (z. B. bei der Abwicklung von Softwareprojekten), hat aber einige Nachteile. So ist sie beispielsweise nutzlos, wenn alle Prozesse die gleiche Zeitschranke besitzen (Beispiel: Taskmenge = Motorsteuerungen mit dem Ziel „Haltet den Roboter an, bevor er gegen die Wand rast“).

- *Minimal Processing Time First*

  Derjenige Prozeß wird ausgewählt, der die minimale restliche Bedienzeit $T_C$ besitzt. Dies entspricht der SJF-Strategie und kann bedeuten, daß der kurze Job mit geringer Priorität dem langen mit hoher Priorität vorgezogen wird.

- *Rate-Monotonic Scheduling (RMS)*

  Haben wir ein festes Prioritätensystem mit festen Ausführungsraten der beteiligten *tasks* (s. oben das Beispiel des Flugzeugs), ist es optimal, wenn wir die hohen Prioritäten den hohen Ausführungsraten zuordnen und niedrige Prioritäten den niedrigen Raten (*rate-monotonic scheduling*). Falls kein *rate-monotonic*-Schedul für eine Menge von *tasks* gefunden werden kann, so ist bewiesen (Liu u. Layland 1973), daß dann auch kein anderer Schedul mit festen Prioritäten existiert, der erfolgreich ist. Hat die CPU eine Auslastung kleiner als 70%, so werden mit dem RMS alle Zeitschranken garantiert eingehalten. Allerdings kann dazu nötig sein, daß die geringe Priorität eines wichtigen Tasks mit geringer Ausführungsfrequenz angehoben werden muß (*Prioritätsinversion*).

- *Foreground-Background Scheduling*

  Auch in Echtzeitsystemen gibt es eine Menge Tasks, die nützlich, aber nicht unbedingt notwendig sind. Diese können im Hintergrund abgewickelt werden, sobald der Prozessor frei ist und nicht anderweitig benötigt wird. Jeder notwendige Task kann sie unterbrechen. Beispiele dafür sind:
  - *Selbsttests*, um defekte Teile zu entdecken.
  - *RAM scabbing:* Auslesen und zurückschreiben vom RAM-Inhalt. Dies ist in Systemen nützlich, die einen fehlerkorrigierenden Datenbus haben und so

die durch Höhenstrahlung (space shuttle!) entstandenen Bitfehler im RAM korrigieren.

- *Auslastungsmonitore,* um Fehler zu frühzeitig zu entdecken, z. B. durch Zeitüberwachung (*watch dog timer*) mittels „Totmannschalter", also Bits (Flags), die periodisch zurückgesetzt werden müssen, um einen Alarm zu verhindern.

Ein Echtzeitbetriebssystem besteht nicht nur aus kritischen Tasks, deren Zeitschranken unbedingt eingehalten werden müssen, sondern auch aus nichtkritischen, aber notwendigen Tasks, und einem Rest, der ebenfalls abgearbeitet werden sollte, falls noch Zeit dafür ist. Nehmen wir an, daß die kritischen, notwendigen Tasks nach einem festen RMS abgewickelt werden und die Tasks der dritten Kategorie im Hintergrund sind, so bleibt noch die große Menge der Tasks der zweiten Kategorie.

Für diese Art von Systemen entwickelten die Designer des „spring kernel" zusätzliche Strategien (Stankovic u. Ramamritham 1991). Sie kombinierten dazu die Variablen *Wichtigkeit*, *Zeitschranke* $T_D$ und *benötigte Betriebsmittel* (z. B. restliche Bedienzeit $T_C$) zu neuen Strategien

- *Minimum Earliest Start*

  Derjenige Prozeß wird gewählt, der die kleinste Zeit $T_S$ besitzt, zu der alle seine notwendigen Betriebsmittel frei werden.

- *Minimum Laxity First*

  Der Prozeß mit der kleinsten freien Zeit $T_D-(T_S+T_C)$ wird gewählt.

- *Kombiniertes Kriterium 1*

  Der Prozeß mit der kleinsten freien Zeit $T_D + wT_C$ wird gewählt.

- *Kombiniertes Kriterium 2*

  Der Prozeß mit der kleinsten freien Zeit $T_D + wT_S$ wird gewählt.

Simulationen ergaben, daß alle Schedule, die nur $T_D$ allein verwenden, schlechte Resultate für Single- und Multiprozessorsysteme erbrachten. Gut dagegen schnitten die kombinierten Kriterien ab, wobei Kriterium 2 den besten Schedul lieferte, wohl weil es auch die Betriebsmittel mitberücksichtigte.

### 2.2.6 Scheduling in Multiprozessorsystemen

Im allgemeinen Fall existiert für jedes Betriebsmittel, und damit für jeden Prozessor, eine eigene Warteschlange und ein eigener Scheduler. Allerdings sind die Übergänge zwischen den Warteschlangen nicht beliebig, sondern es ist bei mehreren, zusammenarbeitenden und parallel ablaufenden Prozessen eine Koordination notwendig. Diese muß berücksichtigen, daß zwischen den einzelnen Prozessen Abhängigkeiten bestehen in der Reihenfolge der Abarbeitung.

Bezeichnen wir die Betriebsmittel mit A, B, C und die Anforderungen an sie mit $A_i$, $B_j$, $C_k$, so läßt sich dies durch eine *Präzedenzrelation* „>“ ausdrücken. Dabei soll $A_i > B_j$ bedeuten, daß zuerst $A_i$ und dann (irgendwann) $B_j$ ausgeführt werden muß. Ist $A_i$ die direkte, letzte Aktion vor $B_j$, so sei die *direkte* Präzedenzrelation mit „>>“ notiert.

**Beispiel**

$A_1 >> B_1 >> C_1 >> A_5 >> B_3 >> A_6$
$B_1 >> B_4 >> C_3 >> B_3$
$A_2 >> A_3 >> B_4$
$A_3 >> C_2 >> B_2 >> B_3$
$A_4 >> C_2$

Eine solche Menge von Relationen läßt sich durch einen gerichteten, gewichteten Graphen visualisieren, bei dem ein Knoten die Anforderung und die Kante „→“ die Relation „>>“ darstellt. Die Dauer $t_i$ der Betriebsmittelanforderung ist jeweils mit einer Zahl (Gewichtung) neben der Anforderung (Knoten) notiert.

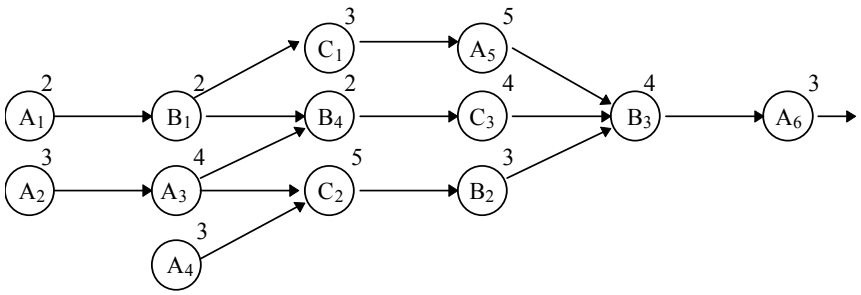



**Abb. 2.11** *Präzedenzgraph*

Für unser Beispiel sei dies durch Abb. 2.11 gegeben. Eine Reihenfolge ist genau dann nicht notwendig, wenn es sich um **unabhängige**, disjunkte Prozesse handelt, die keine gemeinsamen Daten haben, die verändert werden. Benötigen sie dagegen gemeinsame Betriebsmittel, so handelt es sich um **konkurrente** Prozesse.

Ein mögliches Schedulingschema läßt sich durch ein Balkendiagramm (**Gantt-Diagramm**) visualisieren, in dem jedem Betriebsmittel ein Balken zugeordnet ist. Für unser Beispiel ist dies in Abb. 2.12 zu sehen.

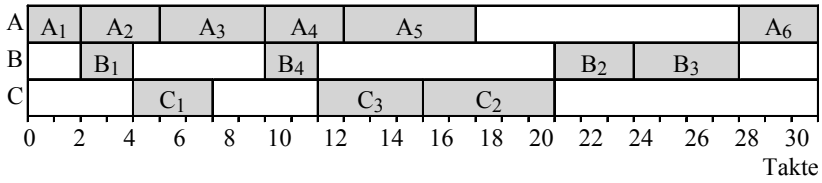



**Abb. 2.12** *Gantt-Diagramm der Betriebsmittel*

Was können wir bei einer guten Schedulingstrategie an Gesamtlaufzeit T erwarten? Es ist klar, daß bei *n* unabhängigen Prozessen mit den Gewichten $t_1$, ..., $t_n$ und *m* Prozessoren die optimale Gesamtausführungszeit $T_{opt}$ für ein preemptives Scheduling entweder von der Ausführungszeit aller Prozesse, verteilt auf die Prozessoren, oder aber von der Ausführungszeit des längsten Prozesses bestimmt wird:

$$T_{opt} = \max \left\{ \max_{1 \le i \le n} t_i, \frac{1}{m} \sum_{i=1}^{n} t_i \right\}$$

Dies ist der günstigste Fall, den wir erwarten können. Für unser obiges Beispiel in Abb. 2.11 mit n=13 Prozessen und m=3 Prozessoren ist günstigstenfalls T=43/3≤15 im Unterschied zu T=30 für den Schedul in Abb. 2.12. Führen wir nämlich zusätzliche Nebenbedingungen ein, wie beispielsweise die Bedingung, daß ein Prozeß vom Typ $A_i$ nur auf Prozessor A, ein Prozeß $B_j$ nur auf Prozessor B und ein Prozeß $C_k$ nur auf Prozessor C ablaufen darf, so verlängert sich die Ausführungszeit T.

Mit den oben eingeführten Hilfsmitteln wollen wir nun einige Schedulingstrategien für die parallele Ausführung von Prozessen auf multiplen Prozessoren näher betrachten.

### *Deterministisches Parallelscheduling*

Angenommen, die Betriebsmittelanforderung (das Knotengewicht) ist jeweils das Multiple einer festen Zeiteinheit Δt: Die Zeitabschnitte sind „*mutually commensurable*“. Die Menge aller Knoten des Graphen läßt sich nun derart in Untermengen zerteilen, daß alle Knoten einer Untermenge unabhängig voneinander sind, also keine Präzedenzrelation zwischen ihnen besteht.

**Beispiel**

Die Knotenmenge des Graphen aus Abb. 2.11 läßt sich beispielsweise in die unabhängigen Knotenmengen {$A_1$,$A_2$}, {$B_1$,$A_3$,$A_4$}, {$C_1$,$B_4$, $C_2$}, {$A_5$, $C_3$, $B_2$}, {$B_3$}, {$A_6$} unterteilen. Dabei ist zwar $B_4$ von $A_3$ und $B_1$ und damit auch {$C_1$,$B_4$, $C_2$} von {$B_1$,$A_3$,$A_4$} abhängig, aber nicht die Knoten innerhalb einer Untermenge.

Eine solche Einteilung kann auf verschiedene Weise vorgenommen werden; die Unterteilung ist nicht eindeutig festgelegt. Die N Untermengen bezeichnen wir als N *Stufen* (*levels*), wobei dem Eingangsknoten die Stufe 1, der davon abhängigen Knotenmenge die Stufe 2 usw. und dem letzten Knoten (*terminal node*) die N-te Stufe zugewiesen wird (**Präzedenzpartition**).

**Beispiel**

Die obige Zerteilung hat N=6 Stufen, wobei die Menge $\{A_1,A_2\}$ der Stufe 1 und dem Terminalknoten $A_6$ die Stufe 6 zugewiesen wird.

Man beachte, daß zwar das allgemeine Problem, einen optimalen Schedul zu finden, NP-vollständig ist, aber für „normale“ Prozesse unter „normalen“ Bedingungen gibt es durchaus Algorithmen, die einen brauchbaren Schedul entwerfen.
Es existieren nun drei klassische Strategien (Muntz u. Coffman 1969), um Prozesse preemptiv zu planen:

- *Earliest Scheduling*

  Ein Prozeß wird bearbeitet, sobald ein Prozessor frei wird. Dazu beginnen wir mit der ersten Stufe und besetzen die Prozessoren mit den Prozessen der Knotenmenge. Alle noch freien Prozessoren werden dann mit den Prozessen der zweiten Stufe (sofern möglich) belegt usw. Frei werdende Prozessoren erhalten darauf in der gleichen Weise die Prozesse der gleichen bzw. nächsten Stufe, bis alle Stufen abgearbeitet sind.

  **Beispiel**: Das Gantt-Diagramm der Präzedenzpartition des obigen Beispiels aus Abb. 2.11, interpretiert als Schedul für die Prozessoren $P_1$, $P_2$ und $P_3$, nimmt mit der *earliest scheduling*-Strategie folgende Gestalt an:

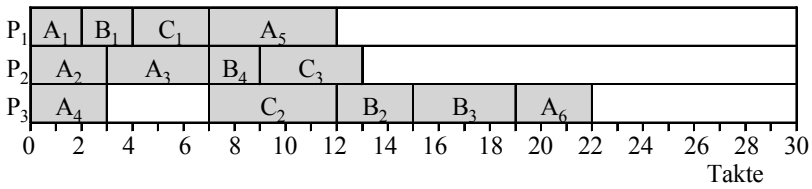



**Abb. 2.13** *Gantt-Diagramm bei der* earliest scheduling-*Strategie*

Mit diesem Schedul erreichen wir für das Beispiel eine Länge T=22.

- *Latest Scheduling*

  Ein Prozeß wird zum spätesten Zeitpunkt ausgeführt, an dem dies noch möglich ist. Dazu ändern wir die Reihenfolge der Stufenbezeichnungen in der Präzedenzpartition um; die N-te Stufe wird nun zur ersten und die erste Stufe wird

zur N-ten. Nun weisen wir wieder den Prozessoren zuerst die Prozesse der Stufe 1 zu, dann diejenigen der Stufe 2 usw.

**Beispiel**: Mit der *latest-scheduling*-Strategie ergibt sich für obiges Beispiel aus Abb. 2.11 das Gantt-Diagramm

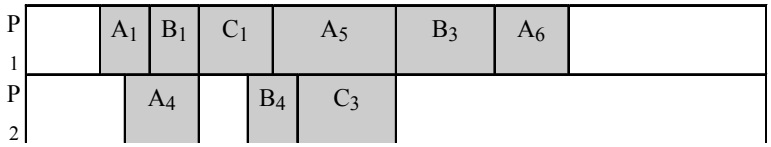



**Abb. 2.14** *Gantt-Diagramm bei* latest scheduling-*Strategie*

Mit diesem deterministischen Schedul erreichen wir für das gleiche Beispiel nur eine Länge T=22.

- *List Scheduling*

  Alle Prozesse werden mit Prioritäten versehen in eine zentrale Liste aufgenommen. Wird ein Prozessor frei, so nimmt er aus der Liste von allen Prozessen, deren Vorgänger ausgeführt wurden und die deshalb lauffähig sind, denjenigen mit der höchsten Priorität und führt ihn aus. Wird die Priorität bereits beim Einsortieren in die *bereit*-Liste beachtet und werden Prozesse, die auf Vorgänger warten, in die *blockiert*-Warteschlange einsortiert, so erhalten wir eine zentrale Multiprozessor-Prozeßwarteschlange ähnlich der Prozeßwarteschlange in unserem Einprozessorsystem. Im Unterschied dazu wird sie aber von allen Prozessoren parallel abgearbeitet. In Abschn. 2.3.5 ist als Beispiel das Warteschlangenmanagement beim Ultracomputer der NYU beschrieben.

Man sieht, daß man mit den Algorithmen zwar keinen optimalen, aber durchaus einen funktionsfähigen Schedul finden kann. Für m=2 Prozessoren und Prozesse einheitlicher Länge ($t_i=t_K$) können wir sogar einen optimalen, preemptiven Schedul der Prozeß-Gesamtmenge erreichen, indem wir für jede Untermenge (Stufe) einen optimalen preemptiven Schedul finden (Muntz u. Coffman 1969). Diesen Gedanken können wir auf den Fall $t_i \neq t_K$ erweitern, indem wir jeden Prozeß der Länge $s \cdot \Delta t$ virtuell aus einer Folge von s Einheitsprozessen bestehen lassen. Wir erhalten so aus unserem Präzedenzgraphen einen neuen Präzedenzgraphen, der nur Prozesse gleicher Länge enthält und damit für m=2 Prozessoren ein optimales preemptives Scheduling erlaubt. Allerdings ist es bei einer ungeraden Anzahl von Prozessen nötig, bei drei Prozessen $C_1$, $C_2$, $C_3$ aus einer der Stufen einen der Prozesse (z.B. $C_2$) zu teilen und beide Hälften auf die Prozessoren zu verteilen, s. Abb. 2.15. Dieser ist wieder optimal, so daß der Gesamtschedul auch optimal wird.

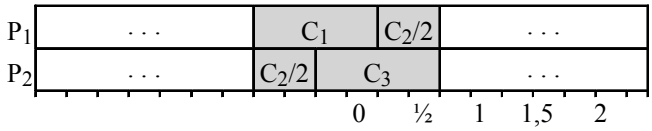



**Abb. 2.15** *Optimaler Schedul von drei Prozessen auf zwei Prozessoren*

Die Betrachtungen können auch ausgedehnt werden auf Präzedenzgraphen mit vielen parallelen, einfachen Jobketten, also Graphen mit einem Startknoten, einem Endknoten und (bis auf den Startknoten) nur jeweils maximal einem Nachfolgeknoten pro Knoten.

### *Die minimale Ausführungszeit bei Multiprozessorscheduling*

Angenommen, wir benutzen die prioritäts- und präzedenzorientierte, zentrale Warteschlange, was ist dann die kleinste Ausführungszeit $T_{prio}$, die wir für eine Liste von Jobs, versehen mit Prioritäten und Präzedenzen, erwarten können?

Für die Beantwortung dieser Frage führen wir eine Rechnung durch, die typisch für eine solche Problemstellung ist, siehe (Liu und Liu 1978). Angenommen, wir notieren mit $T_{prio}$ die Ausführungszeit für einen Schedul, der einem freiwerdenden Prozessor den ausführbaren Job mit der höchsten Priorität aus der Liste zuweist. Dann läßt sich $T_{prio}$ für jeden Prozessor *j* in unterschiedliche Zeitanteile für die Ausführungszeit $t_j$ der Jobs und die unnütze Leerlaufzeit $\phi_j$ unterteilen. Wir können anstelle eines Index *j*, der über alle Prozessoren läuft, die Jobanteile auch mit einem Doppelindex versehen: *i* für die Jobnummer (auf einem Prozessortyp) und *k* für den Prozessortyp. Mit diesen Bezeichnungen ist

$$T_{prio} = t_j + \phi_j = t_{ik} + \phi_{ik}$$

Die Ausführungszeit der parallel ausgeführten Jobs bei insgesamt *m* Prozessoren aus *r* verschiedenen Prozessortypen ist

$$T_{prio} = \frac{1}{m} m T_{prio} = \frac{1}{m}\left(\sum_{j=1}^{m} T_{prio}\right) = \frac{1}{m}\left(\sum_{j=1}^{m} t_j + \phi_j\right) = \frac{1}{m}\left(\sum_{k=1}^{r}\sum_{i=1}^{r_k} t_{ik} + \phi_{ik}\right)$$

Fassen wir jeweils die $r_k$ Ausführungs- und Leerlaufzeiten auf Prozessoren gleichen Typs *k* zusammen,

$$T_k = \sum_{i=1}^{r_k} t_{ik}\,,\quad \Phi_k = \sum_{i=1}^{r_k} \phi_{ik}\,,\quad \Phi = \sum_{k=1}^{r} \Phi_k$$

so ergibt sich die Summe zu

$$T_{prio} = \frac{1}{m}\left(\sum_{k=1}^{r} T_k + \Phi_k\right) = \frac{1}{m}\left(\Phi + \sum_{k=1}^{r} T_k\right) \tag{2.1}$$

Wie groß ist nun eine gute obere Schranke für die Leerlaufzeit Φ ? Die Beantwortung dieser Frage entscheidet über die Güte des Scheduls.

Bezeichnen wir mit H die Summe der Prozessorleerlaufzeiten, bei der mindestens ein Prozessor jeden Typs leer läuft, und mit $K_j$ die Summe der Leerlaufzeiten, bei denen kein Prozessor des Typs *j* leer läuft, so gilt

$$\Phi \le H + \sum_{j=1}^{r} K_j \tag{2.2}$$

Die Jobabarbeitung in der Leerlaufzeit H wird dadurch charakterisiert, daß Zwangsbedingungen (Präzedenzen) erfüllt werden müssen, die ein paralleles Abarbeiten gleicher Jobtypen verhindern; die Jobs können nur sequentiell abgearbeitet werden, unabhängig vom Schedul. Es gibt also keinen optimalen Schedul, dessen Gesamtausführungszeit $T_o$ bei mindestens einem freien Prozessor kleiner ist als die Leerlaufzeit pro Prozessor. Also muß die Ausführungszeit des optimalen Scheduls $T_o$ größer sein:

$$T_o \ge \frac{H}{m-1} \quad \text{wobei natürlich } T_o \ge \frac{1}{m}\sum_{k=1}^{r} T_k \text{ gilt (warum?).} \tag{2.3}$$

Für die Leerlaufzeiten $K_j$ mit mindestens einem arbeitenden Prozessor des Typs *j* gilt folgendes: Bezeichnen wir mit $S_j$ den Anteil der Ausführungszeit aller Jobs auf Prozessortyp *j*, bei dem mindestens ein Prozessor vom Typ *j* leer läuft, so ist $T_j – S_j$ die Zeit, bei der kein Prozessor vom Typ *j* leer läuft: Alle sind beschäftigt. Dieser Zeitabschnitt an der Scheduldauer ist also $(T_j – S_j)/m_j$. Der Zeitabschnitt, in dem die anderen $(m – m_j)$ Prozessoren leer laufen und zusätzlich alle Prozessoren vom Typ *j* arbeiten, ist sicher kleiner oder gleich dem Zeitabschnitt, in dem alle $m_j$ Prozessoren arbeiten:

$$\frac{K_j}{m - m_j} \le \frac{1}{m_j}\left(T_j - S_j\right)$$

Benutzen wir dies und setzen es in Gl.(2.2) ein, so erhalten wir als Abschätzung für die Leerlaufzeit

$$\Phi \le H + \sum_{j=1}^{r} \frac{m-m_j}{m_j}(T_j - S_j) = H + \sum_{j=1}^{r} \frac{m-m_j}{m_j} T_j - \sum_{j=1}^{r} \frac{m-m_j}{m_j} S_j$$

$$= H + m\sum_{j=1}^{r}\frac{1}{m_j}T_j - \sum_{j=1}^{r}T_j - m\sum_{j=1}^{r}\frac{1}{m_j}S_j + \sum_{j=1}^{r}S_j$$

Die vorletzte Summe können wir noch etwas verkleinern, indem wir nur den kleinsten Koeffizienten verwenden und als gemeinsamen Faktor aus der Summe ziehen:

$$\Phi \le H + m\sum_{j=1}^{r}\frac{1}{m_j}T_j - \sum_{j=1}^{r}T_j - m\left(\min_{1\le j\le r}\frac{1}{m_j}\right)\sum_{j=1}^{r}S_j + \sum_{j=1}^{r}S_j \qquad (2.4)$$

Nun müssen wir die Summe der Leerlaufzeiten $S_j$ abschätzen. Würde jeweils immer nur ein Prozessor leer laufen, so wäre H die Summe aller solchen Zeitanteile bei m–1 Prozessoren

$$(m-1)\sum_{j}S_j = H$$

Im allgemeinen Fall überlappen sich aber die Zeiten $S_j$, so daß die Summe größer wird als H/(m–1), und es gilt

$$\sum_{j}S_j \ge \frac{H}{m-1} \qquad (2.5)$$

Berücksichtigen wir noch die Beziehung (s. Aufgabe 2.2-4 )

$$m\sum_{j=1}^{r}\frac{1}{m_j}T_j - \sum_{j=1}^{r}T_j \;\le\; mT_o\,(r-1)$$

und setzen dies in Gl.(2.4) ein, so erhalten wir als Abschätzung für die Leerzeitsumme $\Phi$

$$\Phi \le H + mT_o\,(r-1) - \left(\min_{1\le j\le r}\frac{m}{m_j} - 1\right)\sum_{j=1}^{r}S_j \qquad (2.6)$$

und mit der Beziehung (2.5) bei positivem Klammerinhalt

$$\Phi \le \frac{m-1}{m-1}H + mT_o(r-1) - \frac{H}{m-1}\left(\min_{1\le j\le r}\frac{m}{m_j} - 1\right)$$

$$\le\; mT_o(r-1) + \frac{H}{m-1}\left(m - \min_{1\le j\le r}\frac{m}{m_j}\right)$$

Da die Klammer positiv ist, können wir nun die Beziehung (2.3) anwenden:

$$\Phi \le mT_o(r-1) + T_o(m - \min_{1\le j\le r} \frac{m}{m_j}) \;=\; mrT_o - mT_o\left(\min_{1\le j\le r} \frac{1}{m_j}\right)$$

$$= mT_o\left(r - \min_{1\le j\le r} \frac{1}{m_j}\right) \tag{2.7}$$

Nun können wir direkt das Verhältnis der Ausführungszeit des Prioritätenscheduls $T_{prio}$ zu der Ausführungszeit $T_o$ eines beliebigen Scheduls angeben. Mit der obigen Abschätzung (2.7), eingesetzt in (2.1), erhalten wir unter Berücksichtigung der zweiten Beziehung in (2.3) die Ungleichung

$$T_{prio} = \frac{1}{m}\left(\Phi + \sum_{k=1}^{r} T_k\right) \le T_o\left(r - \min_{1\le j\le r} \frac{1}{m_j}\right) + T_o$$

$$\frac{T_{prio}}{T_o} \le 1 + r - \min_{1\le j\le r} \frac{1}{m_j} \tag{2.8}$$

Diese Formel reduziert sich für gleichartige Prozessortypen (r=1) und m Prozessoren auf

$$\frac{T_{prio}}{T_o} \le 2 - \frac{1}{m} \tag{2.9}$$

Wie man aus (2.9) sieht, ist bei gleichartigen Prozessoren ein prioritätsgeführter Schedul im schlechtesten Fall doppelt so lang wie ein optimaler Schedul; im besten Fall gleich lang. Bei nur einem Prozessor sind sie natürlich identisch, da der eine Prozessor immer die gesamte Arbeit machen muß, egal, in welcher Reihenfolge die Jobs abgearbeitet werden.

Würde man allgemein mehr Prozessoren (Betriebsmittel) einführen, so würde zwar durch die zusätzlichen, parallel wirkenden Bearbeitungsmöglichkeiten meistens die Gesamtausführungszeit verkleinert werden. Da mit diesen zusätzlichen Betriebsmitteln aber die Auslastung sinkt, wäre dann natürlich auch die mittlere Leerlaufzeit der Betriebsmittel höher.

Interessanterweise führt das Lockern von Restriktionen, wie z. B.

- das Aufheben einiger Präzedenzbedingungen
- das Verkleinern einiger Ausführungszeiten ($t_i' \le t_i$)
- die Erhöhung der Prozessoranzahl ($m' \ge m$)

nicht automatisch zu einer kürzeren Gesamtausführungsdauer T, sondern kann auch zu ungünstigeren Schedulen führen (**Multiprozessoranomalien**). In diesem Zusammenhang sind die Studien von Graham (1972) interessant. Er fand heraus, daß in einem System mit m identischen Prozessoren, in dem die Prozesse völlig willkürlich den Prozessoren zugeordnet werden, die Gesamtausführungsdauer T der Prozeßmenge nicht mehr als das Doppelte derjenigen eines optimalen Scheduls

werden kann: Für die Gesamtdauer T´ eines veränderten Scheduls mit gelockerten Restriktionen gilt die Relation

$$\frac{T'}{T} \leq 1 + \frac{m-1}{m'}$$

so daß für m=m' die obere Schranke durch $2 - {}^{1}/_{m}$ gegeben ist.

Eine gute Methode, um erfolgreiche Schedule zu konstruieren, besteht in der Zusammenfassung mehrerer Prozesse (meist von einem gemeinsamen Elternprozeß erzeugt) zu einer **Prozeßgruppe**, die gemeinsam auf einem Multiprozessorsystem ausgeführt wird (**Gruppenscheduling**). Da es meist Abhängigkeiten in Form von Kommunikationsmechanismen unter den Prozessen einer Gruppe gibt, die über gemeinsamen Speicher (*shared memory*, s. Abschn. 3.3) laufen können, verhindert man so ein Ein- und Ausladen der Speicherseiten und des Prozeßkontextes der beteiligten Prozesse.

Eine wichtige Voraussetzung für die preemptiven Schedule ist ein geringer Kommunikationsaufwand, um einen Prozeß mit seinem Kontext auf einem Prozessor starten bzw. fortsetzen zu können. Dies ist in verteilten Systemen nicht immer gegeben, sondern meist nur in eng gekoppelten Multiprozessorsystemen oder bei Nutzung von Prozeßgruppen. Benutzt man non-preemptive Schedule, so wird dieser Nachteil vermieden; allerdings ist das Scheduling auch schwieriger. Zwar gelten die obigen preemptiven Schedulingstrategien auch für non-preemptives Scheduling von Prozessen der Einheitslänge, aber allgemeines non-preemptives Scheduling muß extra berücksichtigt werden.

Weitere deterministische Schedulingalgorithmen sind in Gonzalez (1977) zu finden.

### *Multiprozessorlastverteilung*

Die maximal erreichbare Beschleunigung (**speedup** = das Verhältnis AlteZeit zu NeueZeit) der Programmausführung durch zusätzliche parallele Prozessoren, I/O-Kanäle usw. ist sehr begrenzt. Maximal können wir die Last auf *m* Betriebsmittel (z. B. Prozessoren) verteilen, also NeueZeit=AlteZeit/m, was einen linearen *speedup* von *m* bewirkt. In der Praxis ist aber eine andere Angabe noch viel entscheidender: der Anteil von sequentiellem, nicht-parallelisierbarem Code. Seien die sequentiell ausgeführten Zeiten von sequentiellem Code mit $T_{seq}$ und von parallelisierbarem Code mit $T_{par}$ notiert, so ist mit $m \to \infty$ Prozessoren mit $T_{par} \to 0$ der *speedup* bei sehr starker paralleler Ausführung

$$speedup = \frac{\text{AlteZeit}}{\text{NeueZeit}} \to \frac{T_{seq} + T_{par}}{T_{seq} + 0} = 1 + \frac{T_{par}}{T_{seq}}$$

fast ausschließlich vom Verhältnis des parallelisierbaren zum sequentiellen Code bestimmt.

**Beispiel**

Angenommen, wir haben ein Programm, das zu 90% der Laufzeit parallelisierbar ist und nur für 10% Laufzeit sequentiellen Code enthält. Auch mit dem besten Schedulingalgorithmus und der schnellsten Parallelhardware können wir nur den Code für 90% der Laufzeit beschleunigen; die restlichen 10% müssen hintereinander ausgeführt werden und erlauben damit nur einen maximalen *speedup* von (90%+10%)/10% =10.

Da die meisten Programme weniger rechenintensiv sind und einen hohen I/O-Anteil (20-80%) besitzen, bedeutet dies, daß auch die Software des Betriebssystems in das parallele Scheduling einbezogen werden sollte, um die tatsächlichen Ausführungszeiten deutlich zu verringern. In Abb. 2.16 sind die zwei Möglichkeiten dafür gezeigt.

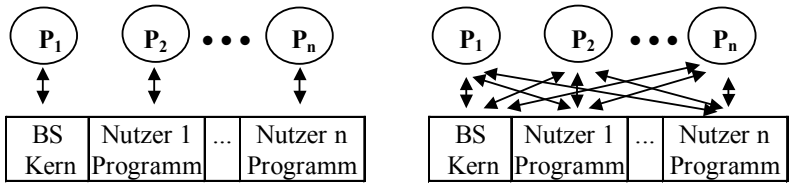



**Abb. 2.16** *Asymmetrisches und symmetrisches Multiprocessing*

Links ist die Situation aus Abb. 1.12 vereinfacht dargestellt: Jeder Prozessor bearbeitet streng getrennt einen der parallel existierenden Prozesse, wobei das Betriebssystem als sequentieller Code nur einem Prozessor zugeordnet wird. Diese Konfiguration wird als **asymmetrisches Multiprocessing** bezeichnet.

Im Gegensatz dazu kann man das Betriebssystem – wie auch die Anwenderprogramme – in parallel ausführbare Codestücke aufteilen, die von jedem Prozessor ausgeführt werden können. So können auch die Betriebssystemteile parallel ausgeführt werden, was den Durchsatz deutlich erhöht. Diese Konfiguration bezeichnet man als **symmetrisches Multiprocessing**.

Für das Scheduling bei Multiprozessorsystemen wird üblicherweise eine *globale Warteschlange* benutzt, in der alle bereiten Prozesse eingehängt werden. Jeder Prozessor, der seinen Job beendet hat, entnimmt den nächsten Job der einen Liste. Dies hat nicht nur den Vorteil, daß alle Prozessoren gleichmäßig ausgelastet werden, sondern führt bei den Prozessorfehlern, die sich nicht in einzelnen Fehlfunktionen, sondern in einem völligen Ausfall des Prozessors äußern (*fail-save* Verhalten), dazu, daß bis auf den betroffenen Job alle anderen unbehelligt weiter ausgeführt werden und der Rechenbetrieb so weiterläuft (*Ausfalltoleranz*).

Das allgemeine Problem, *n* Tasks auf *m* Prozessoren zu verteilen, ist bisher ungelöst; es ist ein typisches Rucksackproblem und damit NP-vollständig. Die verschiedenen Strategien dafür sind stark durch die Annahmen über die Taskeigenschaften geprägt. Da die Abhängigkeiten der Tasks untereinander sowie der benötigten Betriebsmittel im Ablauf bei unregelmäßigen Jobs nur Schätzungen sein können, sind die Schedulingalgorithmen meist nur mehr oder weniger stark heuristisch geprägte Approximationen. Bei stark variierenden Anforderungen ist ihr Nutzen sehr begrenzt.

### 2.2.7 Stochastische Schedulingmodelle

Aus diesem Grund gibt es zwei prinzipielle Ansätze, gute Schedulingalgorithmen auf der Basis von (statistischen) Beobachtungen der Taskeigenschaften zu konstruieren, die sich nicht ausschließen, sondern ergänzen. Zum einen kann man mit plausiblen Annahmen über die Jobs und ihre Bearbeitung ein mathematisches Modell erstellen. Dies ist die Aufgabe des *analytischen* Ansatzes, der meist die **Warteschlangentheorie** (s. z. B. Brinch-Hansen, 1973) benutzt.

**Beispiel**

Die Wahrscheinlichkeit $P_J$, mit der ein neuer Job bei einer Warteschlange in der Zeitspanne $\Delta t$ eintrifft, sei proportional zur (kleinen) Zeitspanne und unabhängig von der Vorgeschichte (Annahme einer Poisson-Verteilung)

$$P_J \sim \Delta t \quad \text{oder} \quad P_J = \lambda \Delta t$$

Die Wahrscheinlichkeit $P_N$, daß kein Job in $\Delta t$ ankommt, ist also

$$P_N(\Delta t) = 1 - P_J = 1 - \lambda \Delta t$$

Die Wahrscheinlichkeit, daß kein Job im Intervall $t+\Delta t$ ankommt, ist also

$$P_N(t+\Delta t) = P(\text{Kein Job in } t \cap \text{kein Job in } \Delta t) = P_N(t)\, P_N(\Delta t) = P_N(t)\,(1-\lambda \Delta t)$$

und somit

$$\frac{P_N(t+\Delta t) - P_N(t)}{\Delta t} = -\lambda P_N(t)$$

Im Grenzübergang für $\Delta t \to 0$ bedeutet dies

$$\frac{dP_N(t)}{dt} = -\lambda P_N(t)$$

so daß mit $P_N(0)=1$ die Integration $P_N(t) = e^{-\lambda t}$ ergibt und somit

$$P_J(t) = 1 - P_N(t) = 1 - e^{-\lambda t} \qquad \textit{(Exponentialverteilung)}$$

Damit haben wir eine wichtige statistische Annahme erhalten, die so nicht nur für die Ankunftszeit, sondern auch für die Bearbeitungszeit von Jobs ge-

macht wird. Die Wahrscheinlichkeitsdichte p(t) für die Ankunft bzw. Bearbeitung der Jobs ist mit

$$P_J(t) = 1 - e^{-\lambda t} = \int_0^t \lambda e^{-\lambda s}\, ds = \int_0^t p_J(s)\, ds \quad \Leftrightarrow p(t) = \lambda\, e^{-\lambda t}$$

Die mittlere oder erwartete Ankunftszeit $T_j$ ist somit

$$T_J = \int_0^\infty p(t)\, t\, dt = \int_0^\infty t\lambda e^{-\lambda t} dt = \underbrace{-te^{-\lambda t}\Big|_0^\infty}_{0} - \int_0^\infty -e^{-\lambda t} dt = -\frac{1}{\lambda} e^{-\lambda t}\Big|_0^\infty = \frac{1}{\lambda}$$

Die Umkehrung $1/T = \lambda$ ist eine Frequenz oder Rate, die **Ankunftsrate** $\lambda$. Analog dazu erhalten wir die **Bedienrate** $\mu$.

Ist $\lambda<\mu$, so ist die Warteschlange im Gleichgewichtszustand. Für diesen Fall gilt die Littlesche Formel über die mittlere Länge L der Warteschlange und die mittlere Wartezeit $T_W$ eines Jobs:

$$L = \lambda T_W \qquad \textit{(Littlesche Formel)}$$

was auch intuitiv einleuchtet.

Ein Ansatz für stochastisches Multiprozessorscheduling ist beispielsweise in Robinson (1979) zu finden. Leider treffen die Annahmen, die solche mathematischen Behandlungen ermöglichen, in dieser Form meist nicht zu. Statt dessen müssen komplexere Annahmen gemacht werden, die aber durch ihre Vielfalt die mathematische Behandlung erschweren und teilweise unmöglich machen. Deshalb sind die Ergebnisse einer mathematischen Modellierung meist nur eingeschränkt anwendbar; sie sind nur im Idealfall gültig und stellen eine Extremsituation dar.

Aus diesem Grund ist es sehr sinnvoll, als zweiten Ansatz auch die *Simulation* des untersuchten Systems durchzuführen. Es gibt dafür bereits verschiedene Softwarepakete, die dies erleichtern. Bei der Simulation werden alle parallelen Betriebsmittel wie Prozessoren, I/O-Kanäle usw. mit jeweils einer Warteschlange, Auftrittsrate und Bedienrate (*Bedienstation*) modelliert. Das reale Computersystem wird auf ein Netz aus Bedienstationen abgebildet, so daß man die einzelnen Parameter gut an die Realität anpassen kann. Verklemmungen und Leistungsengpässe („Flaschenhälse“) lassen sich so simulativ erfassen, geeignet interpretieren und beheben.

### 2.2.8 Beispiel UNIX: Scheduling

In UNIX kommt meist das Round-Robin-Verfahren zur Anwendung, das durch Prioritäten geeignet modifiziert wird. Jeder Task bekommt eine initiale Priorität zugeteilt, die sich beim Warten noch mit der Zeit erhöht. Dadurch sind einerseits

die Systemprozesse bevorteilt, zum anderen wird aber sichergestellt, daß auch langwartende Prozesse einmal an die Reihe kommen.

In UNIX entspricht jeder Priorität eine ganze Zahl. Frühere Versionen hatten Prioritäten von –127 bis +127, wobei kleine Zahlen hohe Prioritäten bedeuten. Dabei bedeuten negative Zahlen Systemprozesse, die normalerweise nicht unterbrochen werden können. Ein Kommando „`nice`“ erlaubt es Benutzern, die eigene Standardpriorität herabzusetzen, um bei unkritischen Jobs nett zu den anderen Benutzern zu sein. Typischerweise wird dies meist nicht benutzt.

Da bei ganzen Zahlen mehrere Jobs mit gleicher Priorität existieren, gibt es für alle Jobs gleicher Priorität jeweils eine eigene FCFS-Warteschlange. Sind alle Jobs einer Warteschlange abgearbeitet, so wird die nächste Warteschlange mit geringerer Priorität bearbeitet. Zusätzlich werden Jobs, deren Priorität sich durch Warten erhöht, in andere Schlangen umgehängt (*multi-level feedback scheduling*).

In neuen UNIX-Varianten erstreckt sich die Priorität von 0 bis 255 und ist zusätzlich unterteilt. In Abb. 2.17 sind die Prioritätswarteschlangen dieser *multi-level queue* für HP-UX (Hewlett-Packard 1991) gezeigt. Die Warteschlangen bilden eine verkettete Liste, deren inhaltliche Zeiger auf die Prozeßkontrollblöcke PCB der Prozeßtafel zeigen. Alle Jobs der Systemprioritäten (128-177) und der *User* (178-255) werden nach dem Zeitscheibenverfahren zugeteilt; die Jobs, die mit einem speziellen Systemaufruf `rtprio()` gestartet werden, werden extra behandelt. Ihre Priorität (0-127) ist initial höher und wird nicht mehr verändert.

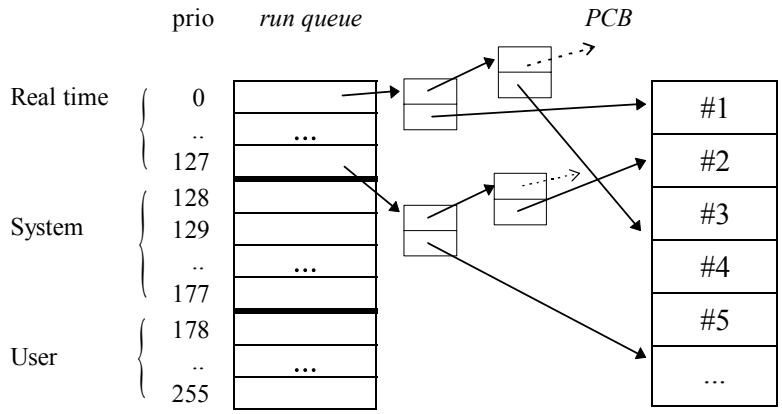



**Abb. 2.17** *Multi-level-Warteschlangen in UNIX*

Diese und andere zusätzliche, früher in UNIX nicht existierende Eigenschaften wie das preemptive Scheduling von Prozessen, die sich gerade im *kernel mode* befinden und deshalb normalerweise ununterbrechbar sind (es existiert nur ein

einziger *kernel stack*, der überlaufen könnte), fest reservierte Hauptspeicherbelegung (*memory lock*) zur Vermeidung des *swapping* sowie Dateien mit konsekutiven Blöcken ermöglichen auch ein Echtzeitverhalten in Real-Time-UNIX, siehe Furht, Borko, Dan Grostick, D. Gluch, G. Rabbat, J. Parker, M. McRoberts 1991.

### 2.2.9 Beispiel: Scheduling in Windows NT

Auch in Windows NT gibt es ein Multi-level-Scheduling, das Echtzeitjobs zuläßt. Die Prioritäten gehen dabei von Priorität 0 als der geringsten Priorität (für den *system-idle*-Prozeß) zur höchsten Priorität 31 für Echtzeitprozesse. In Abb. 2.18 ist eine Übersicht gegeben. Allerdings werden nicht Jobs verwaltet, sondern Leichtgewichtsprozesse (*threads*). Das Scheduling ist dabei vom Dispatching getrennt. Der Dispatcher unterhält eine eigene Datenbasis, die den Status der *threads* und der CPUs festhält und die Entscheidungsgrundlage für den Scheduler bildet. Windows NT unterstützt symmetrisches Multiprocessing. Ist kein *thread* abzuarbeiten, so führen die CPUs einen besonderen *thread*, den *idle thread*, aus.

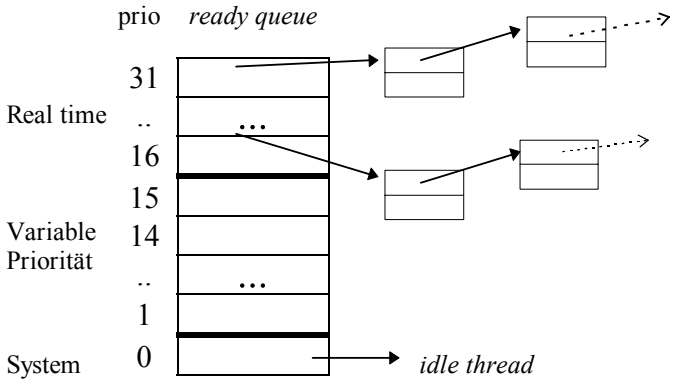



**Abb. 2.18** *Dispatcher ready queue in Windows NT*

Die *threads* werden nach dem preemptiven Zeitscheibenverfahren zugeteilt und bearbeitet. Nach jeder Zeitscheibe (*timer interrupt*) wird die Priorität des *threads* (Prio 1-15) etwas vermindert bis minimal auf die Basispriorität, Prio 2-6. Danach wird dann zum einen entschieden, welcher der *threads* in der Warteschlange die höchste Priorität hat, und zum anderen, ob der *thread* auf dem Prozessor ausgeführt werden kann. Dafür gibt es eine zusätzliche (veränderbare) Eigenschaft, die *Prozessoraffinität*.

Wird ein *thread* aus einer Warteschlange in die *bereit*-Liste umgehängt, so erhält er zusätzliche Priorität, die von der Art der Warteliste abhängt: Terminal-I/O erhält mehr Zusatzpriorität als Platten-I/O.

Wird ein *thread* bereit, der eine höhere Realzeit-Priorität (Prio 16-31) als einer der gerade ausgeführten *threads* auf einem der Prozessoren hat, so wird der *thread* in die Warteschlange eingereiht, und der betreffende Prozessor erhält einen Interrupt. Darauf schiebt er den *thread* zurück in die Warteschlange und wechselt zu dem mit höherer Priorität.

Die initialen Prioritäten werden nach der Art der Jobs zugeteilt: Interaktive Jobs sind wichtiger als allgemeine I/O-Jobs, die wiederum wichtiger sind als reine Rechenjobs.

### 2.2.10 Aufgaben

**Aufgabe 2.2-1 (Scheduling)**

a) Was ist der Unterschied zwischen der Ausführungszeit und der Bedienzeit eines Jobs?

b) Geben Sie den Unterschied an zwischen einem Algorithmus für ein *Foreground/background*-Scheduling, das RR für den Vordergrund und einen preemptiven Prioritätsschedul für den Hintergrund benutzt, und einem Algorithmus für *Multi-level-feedback*-Scheduling.

**Aufgabe 2.2-2 (Scheduling)**

Fünf Stapelaufträge treffen in einem Computer fast zur gleichen Zeit ein. Sie besitzen geschätzte Ausführungszeiten von 10, 6, 4, 2 und 8 Minuten und die Prioritäten 3, 5, 2, 1 und 4, wobei 5 die höchste Priorität ist. Geben sie für jeden der folgenden Schedulingalgorithmen die durchschnittliche Verweilzeit an. Vernachlässigen sie dabei die Kosten für einen Prozeßwechsel.

a) Round Robin
b) Priority Scheduling
c) First Come First Serve (Reihenfolge: 10, 6, 2, 4, 8)
d) Shortest Job First

Nehmen Sie für a) an, daß das System Mehrprogrammbetrieb verwendet und jeder Auftrag einen fairen Anteil an Prozessorzeit erhält. Nehmen sie für b)–d) an, daß die Aufträge nacheinander ausgeführt werden. Alle Aufträge sind reine Rechenaufträge.

**Aufgabe 2.2-3 (Adaptive Parameterschätzung)**

Beweisen Sie: Wird eine Größe a mit der Gleichung

$$a(t) = a(t-1) - 1/t\,(a(t-1) - b(t))$$

aktualisiert, so stellt a(t) in jedem Schritt t den arithmetischen Mittelwert der Größe b(t) über alle t dar.

**Aufgabe 2.2-4 (Paralleles Scheduling)**

a) In welchen Fällen wird aus der Gleichung in Gl.(2.2) eine Ungleichung ?
b) Warum gilt die rechte Ungleichung in Gl.(2.3) ?
c) Beweisen Sie: Für die Zeiten $T_j$ eines Scheduls der Gesamtzeit T bei *r* Prozessortypen, *m* Prozessoren und $m_j$ Prozessoren pro Typ *j* gilt

$$\sum_{j=1}^{r} \frac{1}{m_j} T_j \le \frac{1}{m} \sum_{j=1}^{r} T_j + (r-1)T_o$$

Hinweis: Es gelten $m = \sum_{k=1}^{r} m_k$ und $T_o \ge T_{max} = \max_i T_i$

**Aufgabe 2.2-5 (Paralleles Scheduling)**

a) Sei ein Programm gegeben mit 40% sequentiellem, nicht-parallelisierbarem Code. Wie groß ist der maximale *Speedup* ?
b) Angenommen, ein weiteres Betriebsmittel A' ist verfügbar. Wie kann man dann das Gantt-Diagramm in Abb. 2.12 so abändern, daß weniger Zeit verbraucht wird?

## 2.3 Prozeßsynchronisation

Die Einführung von Prozessen als unabhängige, quasi-parallel ausführbare Programmeinheiten bringt viel Flexibilität und Effizienz in die Rechnerorganisation; sie beschert uns aber neben dem Prozeßscheduling auch zusätzliche Probleme. Betrachten wir dazu zunächst folgende Situation.

### 2.3.1 *Race conditions* und kritische Abschnitte

Angenommen, in einer der vielen Warteschlangen des Prozeßsystems sind mehrere Prozesse eingehängt und warten auf eine Bearbeitung. Diese Situation tritt in allen Warteschlangen immer wieder auf und ist in Abb. 2.19 gezeigt.

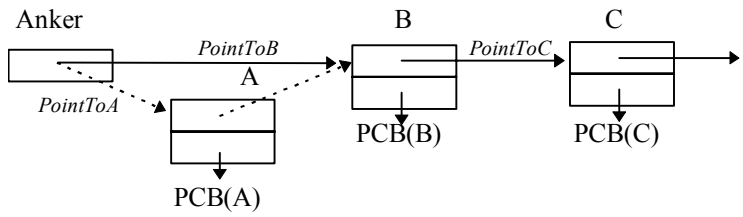



**Abb. 2.19** *Race conditions in einer Warteschlange*

Dabei können wir die Situationen unterscheiden, daß entweder ein Prozeß (hier Prozeß A) neu eingehängt wird, oder aber ein bestehender (hier Prozeß B) ausgehängt wird.

Für das Ein- bzw. Aushängen sollen folgende Schritte gelten:

| | *Einhängen* | *Aushängen* |
|---|---|---|
| (1) | Lesen des Ankers: PointToB | Lesen des Ankers: PointToB |
| (2) | Setzen des NextZeigers:=PointToB | Lesen des NextZeigers: PointToC |
| (3) | Setzen des Ankers:=PointToA | Setzen des Ankers:=PointToC |

Nach dem Aushängen wird evtl. der Speicherplatz des Listeneintrags (Doppelkästchen in der Zeichnung) freigegeben und mit dem Zeiger zum PCB eine Prozeßumschaltung vorgenommen.

Jede der Operationen geht gut, solange sie nur für sich an einem Stück durchgeführt wird. Haben wir aber ein Prozeßsystem, das ein beliebiges Umschalten nach einer Instruktion innerhalb einer Operation auf einen anderen Prozeß und damit das sofortige Umschalten auf eine andere Operation ermöglicht, so erhalten wir Probleme. Betrachten wir dazu die obigen Schritte bei der Operation *Aushängen*. Angenommen, Prozeß B soll ausgehängt werden und Schritte (1) und (2) wurden fürs Aushängen durchgeführt. Nun trete ein *timer*-Interrupt auf; die Zeitscheibe von B ist zu Ende. Prozeß A tritt auf, hängt sich in die Liste mittels der Operation *Einhängen* ein, verrichtet noch etwas Arbeit oder legt sich schlafen. Wenn nun B wieder die Kontrolle erhält, so arbeitet B fatalerweise mit den alten Daten weiter: In Schritt (3) wird der Anker auf C gesetzt, und damit ist der Prozeß A nicht mehr in der Liste; ist dies die *bereit*-Liste, so erhält er nie mehr die Kontrolle und arbeitet nicht mehr.

Das Heimtückische an diesem Fehler besteht darin, daß er nicht immer auftreten muß, sondern nur sporadisch, „unerklärlich“, nicht reproduzierbar und an Nebenbedingungen gebunden (wie große Systemlast etc.) auftreten kann. Da dies geschieht, wenn ein Prozeß den anderen „überholt“, wird dies auch als **race condition** bezeichnet; die Codeabschnitte, in denen der Fehler entstehen kann und die deshalb nicht unterbrochen werden dürfen, sind die **kritischen Abschnitte** (*critical sections*).

Diese Problematik ist typisch für alle Systeme, bei denen mehrere unabhängige Betriebsmittel auf ein und demselben gemeinsamen Datenbereich arbeiten. Beson-

ders bei Multiprozessorsystemen ist dies ein wichtiges Problem, das von den Systemdesignern erkannt und berücksichtigt werden muß.

Die Hauptproblematik besteht darin, für jeden kritischen Abschnitt im Benutzerprogramm oder Betriebssystem sicherzustellen, daß immer nur ein Prozeß oder Prozessor sich darin befindet. Das Betreten und Verlassen des Codes muß zwischen den Prozessen abgestimmt (**synchronisiert**) sein und einen **gegenseitigen Ausschluß** (*mutual exclusion*) in dem kritischen Abschnitt garantieren.

Dieses Problem wurde schon in den 60er Jahren erkannt und mit verschiedenen Algorithmen bearbeitet. An eine Lösung wurden folgende Anforderungen gestellt (Dijkstra 1965):

- Keine zwei Prozesse dürfen gleichzeitig in ihren kritischen Abschnitten sein (*mutual exclusion*).
- Jeder Prozeß, der am Eingang eines kritischen Abschnitts wartet, muß irgendwann den Abschnitt auch betreten dürfen: Ein ewiges Warten muß ausgeschlossen sein (*fairness condition*).
- Kein Prozeß darf außerhalb eines kritischen Abschnitts einen anderen Prozeß blockieren.
- Es dürfen keine Annahmen über die Abarbeitungsgeschwindigkeit oder Anzahl der Prozesse bzw. Prozessoren gemacht werden.

### 2.3.2 Signale, Semaphore und atomare Aktionen

Die einfachste Idee zur Einrichtung eines gegenseitigen Ausschlusses besteht darin, einen Prozeß beim Eintreten in einen kritischen Abschnitt so lange warten zu lassen, bis der Abschnitt wieder frei ist. Für zwei parallel ablaufende Prozesse ist der Code in Abb. 2.20 gezeigt. Die grau schraffierten Bereiche in der Abbildung bedeuten jeweils, daß diese Befehlssequenz nicht unterbrochen werden kann.

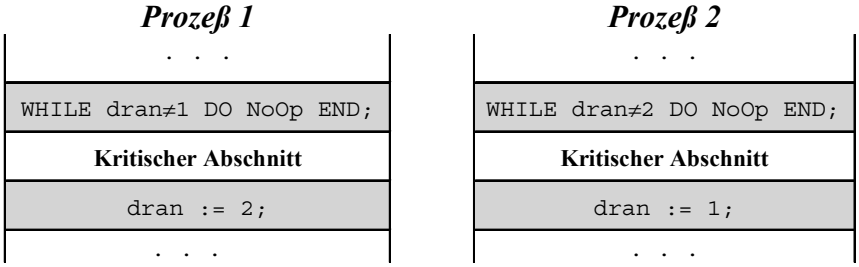



**Abb. 2.20** *Erster Versuch zur Prozeßsynchronisation*

Initialisieren wir die gemeinsame (globale) Variable `dran` beispielsweise mit 1, so erreicht der obige Code zwar den gegenseitigen Ausschluß, aber beide Prozesse können immer nur abwechselnd den kritischen Abschnitt durchlaufen. Dies widerspricht der Forderung nach Nicht-blockieren, da ein Prozeß sich selbst daran hindert, ein zweites Mal hintereinander den kritischen Abschnitt zu betreten, ohne dabei im kritischen Abschnitt zu sein. Stattdessen muß er solange warten, bis der andere Prozeß seinen kritischen Abschnitt durchlaufen hat und die gemeinsame Variable wieder zurückgesetzt hat. Geschieht dies nie, so wartet der eine Prozeß ewig: ein weiterer Verstoß, diesmal gegen die *fairness* –Bedingung.

Versuchen wir nun, die beiden Prozesse in beliebiger Reihenfolge zu synchronisieren, so bemerken wir, daß dies nicht so einfach ist. Beispielsweise hat die Konstruktion in Abb. 2.21 den Vorteil, daß ein Prozeß nicht mehr auf eine besondere Zuweisung warten, sondern nur noch darauf achten muß, daß nicht ein anderer Prozeß im kritischen Abschnitt ist.

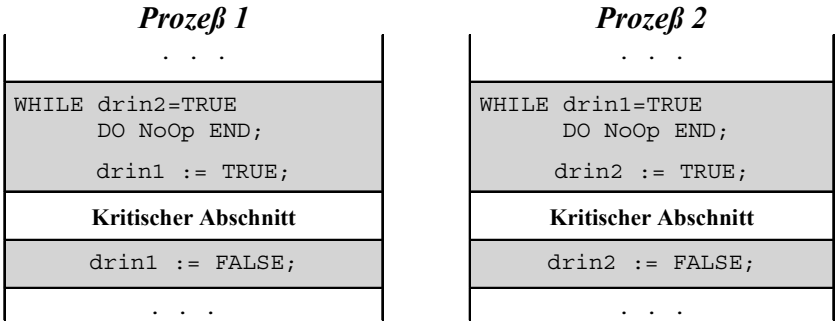



**Abb. 2.21** *Zweiter Versuch zur Prozeßsynchronisation*

Allerdings tritt hier ein anderer Fehler auf: Angenommen, die Synchronisationsvariablen seien mit `drin2:= drin1:= FALSE` initialisiert. Prozeß $P_1$ findet `drin2=FALSE` und Prozeß $P_2$ findet `drin1=FALSE`. Also setzen die Prozesse jeweils `drin1:=TRUE` bzw. `drin2:=TRUE`, und schon sind beide gleichzeitig im kritischen Abschnitt, im Widerspruch zu der Forderung nach gegenseitigem Ausschluß (1).

Auch ein Vertauschen der beiden Zeilen des ersten grau schraffierten Bereichs bringt keine Abhilfe. Erst der Vorschlag von T. Dekker, einem holländischen Mathematiker, zeigte einen Weg auf, das Problem für zwei Prozesse zu lösen, s. Dijkstra (1965). Eine einfachere Lösung stammt von G. Peterson (1981) und hat die in Abb. 2.22 gezeigte Form:

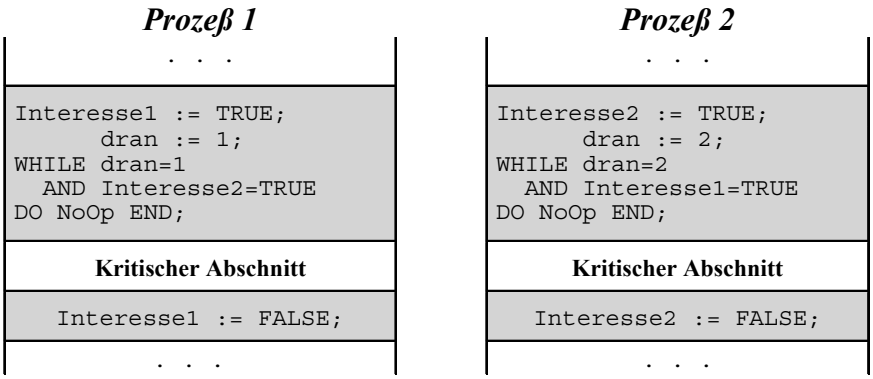



**Abb. 2.22** *Prozeßsynchronisation nach Peterson*

Dabei sind `Interesse1, Interesse2` und `dran` globale, gemeinsame Variablen. Die beiden Variablen `Interesse1` und `Interesse2` werden mit `FALSE` initialisiert. Angenommen, ein Prozeß arbeitet die grau schraffierte Befehlssequenz vor dem kritischen Abschnitt ab. Da der andere Prozeß kein Interesse gezeigt hat, ist die Bedingung der `WHILE`-Schleife nicht erfüllt, und der Prozeß betritt den kritischen Abschnitt. Erscheint der andere Prozeß etwas später, so muß er so lange warten, bis der andere Prozeß den kritischen Abschnitt verlassen hat und kein Interesse mehr bekundet.

Für die Situation, in der beide Prozesse (fast) gleichzeitig die grauen Bereiche betreten, ist die Abarbeitung der einen Instruktion entscheidend, mit der die gemeinsame Variable `dran` belegt wird. Derjenige Prozeß, der sie zuletzt belegt, erlöst den anderen Prozeß und muß warten; der andere kann den kritischen Abschnitt betreten.

Dieses Konzept kann man auch kompakter mit einem Sprachkonstrukt formulieren. Eine Idee dafür besteht darin, für jeden kritischen Abschnitt ein Signal zu vereinbaren. Die Prozesse synchronisieren sich dadurch, daß vor dem Betreten des kritischen Abschnitts das dafür vereinbarte Signal abgefragt wird. Es wird initial mit dem Wert „gesetzt“ belegt. Mit dem Befehl

```
waitFor(Signal)
```

blockiert sich der aufrufende Prozeß, falls kein Signal gesetzt war, ansonsten setzt er das Signal zurück und betritt den Abschnitt. Ist er fertig, so aktiviert er mit

```
send(Signal)
```

einen der Prozesse, die auf das Signal warten. Welcher davon aktiviert wird, ist Sache einer zusätzlich zu vereinbarenden Strategie.

Ein Konstrukt, das einen exklusiven Ausschluß für einen kritischen Abschnitt ermöglicht, wurde von Dijkstra (1965) vorgeschlagen. Er nannte die Signale **Se-**

**maphore**. Diese Signalbarken aus der Seefahrt lassen sich hier als eine Art Verkehrsampeln oder Zeichen auffassen. Sie werden von den folgenden zwei elementaren Operationen verwaltet:

- *Passieren P(s)*

  Beim Eintritt in den kritischen Abschnitt wird P(s) aufgerufen mit dem Semaphor s. Der aufrufende Prozeß wird in den Wartezustand versetzt, falls sich ein anderer Prozeß in dem dem Semaphor korrespondierenden Abschnitt befindet.

- *Verlassen V(s)*

  Beim Verlassen des kritischen Abschnitts ruft der Prozeß V(s) auf und bewirkt damit, daß einer der (evtl.) wartenden Prozesse aktiviert wird und den kritischen Abschnitt betreten darf.

Interessanterweise ist es dabei unwesentlich, ob für ein Signal oder ein Semaphor nur ein kritischer Abschnitt existiert, der von allen Prozessoren im Hauptspeicher durchlaufen wird (Multiprozessorsysteme); ob es mehrere Kopien davon gibt in den Prozessen (z. B. Mehrrechnersysteme), oder aber ob es Abschnitte verschiedenen Codes sind, die aber zueinander funktionell korrespondieren und mit einem gemeinsamen Signal abgesichert werden müssen, wie im obigen Beispiel das Ein- und Aushängen in einer Warteschlange

**Beispiel** *Benutzung der P(.)- und V(.)-Operationen*

Mehrere Prozesse sollen eine Zahl inkrementieren und ihren Wert ausgeben, so daß normal „1,2,3,..“ hochgezählt wird. Der kritische Abschnitt lautet also hier

```
...
z := z+1;
WriteInt(z,3)
...
```

Mit einer Semaphore als globaler Variable wird dies zu

```
...
P(s);      (* Abfragen: kritischer Abschnitt besetzt? *)
z := z+1;  (* kritischer Abschnitt *)
WriteInt(z,3)
V(s);      (* andere wartende Prozesse aufwecken *)
....
```

Man beachte, daß sowohl `send(Signal)` und `waitFor(Signal)` als auch `P(s)` und `V(s)` selbst wieder ununterbrechbare, kritische Abschnitte darstellen. Allerdings sind sie nur sehr kurz und lassen sich deshalb leichter als ununterbrechbare, sog. **atomare Aktionen** implementieren. Eine atomare Aktion wird immer entweder vollständig (alle Operationen des Abschnitts) oder gar nicht (keine einzige Operation) durchgeführt. Spezielle Puffer ermöglichen es, den Anfangszustand zu speichern. Muß die atomare Aktion abgebrochen werden, so wird der Ursprungszustand wieder hergestellt.

**Beispiel** *Atomare Aktionen*

Für eine Überweisung werden von Ihrem Konto 2000 € abgebucht. Leider tritt in diesem Augenblick ein Defekt im Computersystem auf, bevor der Betrag dem Zielkonto gutgeschrieben werden kann; das Computersystem hält an. Nach dem erneuten Systemstart haben Sie nur die Auswahl, entweder erneut 2000 € abgebucht zu bekommen und sie dem Zielkonto zu überweisen, oder aber nichts zu tun. Da das Geld nicht beim Empfänger ankam, bedeutet dies für Sie das gleiche: auf jeden Fall Geldverlust. Als Kunde werden Sie dabei sicher nicht von den Segnungen des Computerzeitalters überzeugt.

Diese Situation läßt sich entschärfen, wenn die Geldtransaktion als atomare Aktion konzipiert ist. Da die Aktion nicht abgeschlossen war, garantiert die Eigenschaft der Atomizität Ihnen, daß die erste Abbuchung nicht durchgeführt wurde und nur als temporäre Änderung bestand: Sie verlieren keine DM 2000. Die eigentliche Abbuchung wird erst nach dem erneuten Systemstart durchgeführt.

Dieses Zurücksetzen (*roll back*) ist aber nicht unproblematisch in verteilten Datenbanken. Aus diesem Grund bevorzugen Banken meist zentrale, von einem Großrechner (*Server*) verwaltete Datenbanken.

Die bisherigen Ideen zur Prozeßsynchronisation leiden an einem großen Problem: Der jeweilige Prozeß muß aktiv in einer Befehlsschleife warten (**busy waiting**), bis die Bedingung erfüllt ist und er in den kritischen Abschnitt eintreten kann. Diese Art von Warteoperationen heißen deshalb auch **spin locks**. Das Warten belastet aber nicht nur den Prozessor (und damit den Durchsatz) unnötig, sondern kann auch unter Umständen die *fairness*-Bedingung verletzen. Betrachten wir dazu die Situation, daß ein hochprioritärer Prozeß gerade dann die Kontrolle erhalten hat, wenn ein niedrigprioritärer Prozeß in seinem kritischen Abschnitt ist. In diesem Fall wird der eine Prozeß so lange im *spin lock* verbleiben, bis der andere Prozeß den kritischen Abschnitt verläßt. Da er aber geringere Priorität hat, wird dies bei statischen Prioritäten nie der Fall sein, und die *fairness*-Bedingung wird verletzt.

Das *busy waiting* läßt sich vermeiden, indem der wartende Prozeß sich „schlafen“ legt, also die Kontrolle beim Blockieren in `waitFor(Signal)` oder `P(s)` abgibt.

### *Software-Implementierung*

Eine der typischen Implementierungen von Semaphoren modelliert ein Semaphor als Zähler, der bei der Ankunft eines Prozesses dekrementiert wird. Im nachfolgenden ist eine *busy wait* Implementierung gezeigt, wie sie im Betriebssystemkern für sehr kurze Sperrungen verwendet werden kann. Dabei ist s mit 1 initialisiert.
Die Operationen P(s) und V(s) selbst sind als ganzes atomar, was sich etwa durch Sperren aller Interrupts während der gesamten Dauer der Operation (hohe Priorität der Prozeduren) erreichen läßt.

```
PROCEDURE P(VAR s:INTEGER)
BEGIN
  WHILE s<=0 DO NoOp END;
  s:=s-1;
END P;

PROCEDURE V(VAR s:INTEGER)
BEGIN
  s:=s+1;
END V;
```

Eine komfortablere Lösung für Benutzerprozesse und lange Sperrzeiten ist mit den Betriebssystemaufrufen `sleep` und `wakeup()` realisiert und betrachtet den Semaphor als zusammengesetzte Variable, die auch eine Liste der wartenden Prozesse enthält. Die Implementierung der `ProcessList` ist dabei hier nicht gezeigt. `s.value` ist mit 1 initialisiert.

```
TYPE Semaphor = RECORD
       value: INTEGER;
       list : ProcessList;
END;

PROCEDURE P(VAR s:Semaphor)
BEGIN
 s.value:=s.value-1;
 IF s.value < 0 THEN
    einhängen(MyID,s.list); sleep;
 END;
END P;

PROCEDURE V(VAR s:Semaphor)
VAR PID: ProcessId;
BEGIN
 IF s.value < 0 THEN
    PID:=aushängen(s.list); wakeup(PID);
 END;
 s.value:=s.value +1;
END V;
```

### *Hardware-Implementierung*

Eine wichtige Unterstützung für die Systemprogrammierung, gerade in einer Multiprozessorumgebung, stellt die Implementierung der atomaren Aktionen mittels Hardware dar. Es gibt dabei verschiedene Versionen, die kurz vorgestellt werden sollen:

- *Interrupts ausschalten*

  Eine der einfachsten Methoden für atomare Aktionen in Monoprozessorsystemen besteht darin, alle Unterbrechungen durch Setzen der entsprechenden Statusbits in der CPU bzw. im Interruptcontroller auszuschalten. Dies kann aber Nebeneffekte mit sich bringen, z. B. wenn der *timer*-Interrupt ausgeschaltet wird und die Zeitzählung durcheinander kommt, bei Stromausfall der *power failure*-Interrupt versagt und die Register nicht mehr gerettet werden können usw. Besser ist es, die Interrupts nicht zu sperren und statt dessen dem Prozeß im kritischen Abschnitt die höchstmögliche Priorität zu verleihen.

- *Atomare Instruktionsfolgen*

  In Multiprozessoranlagen stellt das Sperren der Interrupts sowieso kein geeignetes Mittel dar. Hier haben sich eher atomare Aktionen in Form *nichtunterbrechbarer Maschinenbefehle (atomic operation, atomic action)* bewährt, die einen komplexen Maschinenbefehl in einem einzigen, ununterbrechbaren Speicherzyklus durchführen. Beispiele dafür sind

  - *Test And Set*

    Die *test and set*-Instruktion (auch *test and set lock tsl* genannt) liest den Inhalt einer Speicherzelle aus und ersetzt ihn innerhalb desselben Speicherzyklus durch einen anderen Wert. Dies läßt sich mit einer Funktion spezifizieren:

    ```
    PROCEDURE TestAndSet(VAR target: BOOLEAN):BOOLEAN
    VAR tmp:BOOLEAN;
    BEGIN
      tmp:=target;   target:= TRUE;  RETURN tmp;
    END TestAndSet;
    ```

    Das Rücksetzen auf `FALSE` wird nur von einem Prozeß durchgeführt und kann daher normal ohne *atomic action* geschehen.

  - *Swap*

    Man kann auch die obige Operation verallgemeinern und den hineinzuschreibenden Wert spezifizieren. In diesem Fall tauscht swap die Werte der Variablen source und target miteinander aus.

    ```
    PROCEDURE swap(VAR source,target: BOOLEAN)
    VAR tmp:BOOLEAN;
    BEGIN
      tmp:=target; target:=source; source:=tmp;
    END swap;
    ```

  - *Fetch And Add*

    Diese Instruktion liest den Inhalt einer Speicherzelle aus und schreibt im selben ununterbrechbaren Speicherzyklus den neuen, um eine Zahl inkrementierten ausgelesenen Wert hinein.

```
PROCEDURE fetchAndAdd(VAR a, value:INTEGER): INTEGER;
VAR tmp:INTEGER;
BEGIN
    tmp:=a; a:=tmp+value;  RETURN tmp;
END fetchAndAdd;
```

Es ist klar, daß mit diesen atomaren HW-Befehlen leicht das *busy wait* eines *spin locks* für einen exklusiven Ausschluß implementiert werden kann. Verwenden wir dies, um die atomaren Operationen P(.) und V(.) des Semaphors zu implementieren, so lassen sich auch die Semaphoroperationen hardwaremäßig absichern, ohne komplexe Maschineninstruktionen für die gesamten Prozeduren P(.) und V(.) zu implementieren.

**Beispiel** *Multiprozessorsynchronisation*

In Multiprozessoranlagen mit gemeinsamem Speicher läßt sich eine HW-Synchronisation nur mit einer atomaren Instruktion durchführen, da gemeinsame Interrupts nicht existieren müssen. Die parallel ausgeführten atomaren Instruktionen der Einzelprozessoren wirken deshalb untrennbar, weil sie im engen Verbund mit dem Speicherzugriffssystem erfolgt, das für eine Speicherstelle nur einmal existiert und damit für kurze Zeit (einen Speicherzyklus) exklusiven Zugriff auf die globale Variable garantiert.

Die Implementierung von Multiprozessor-Semaphoroperationen mit einem atomaren Maschinenbefehl kann folgendermaßen aussehen:

```
VAR s: INTEGER;                (* initial = 1*)

PROCEDURE P(s);
BEGIN
  LOOP
    WHILE s<1 DO NoOp END;(a)
    IF (FetchAndAdd(s,-1) >= 1) THEN RETURN END;  (b)
    FetchAndAdd(s,1);      (c)
  END;
END P;

PROCEDURE V(s);
BEGIN
  FetchAndAdd(s,1)
END V;
```

Diese Implementierung (Gottlieb, Lubachevsky, Rudolph 1981) hat neben dem LOOP nicht nur eine Warteschleife in P(), sondern in Zeile (a) mit WHILE eine zweite – warum?

Dies ist durch die Multiprozessorumgebung bedingt. Entfernen wir die WHILE-Schleife, so reicht der Test mit FetchAndAdd (FAA) in Zeile (b) nicht mehr aus. Betrachten wir dazu den Fall, bei dem drei Prozessoren P(s) zur

gleichen Zeit die Prozedur durchlaufen, wobei initial `s=1` sei. Die drei Prozessoren sollen als Rückgabe für s die Werte 1,0 und –1 erhalten, wobei zum Schluß `s=-2` gesetzt wird. Der erste Prozessor betritt und verläßt den kritischen Abschnitt, die beiden anderen warten in der Schleife. Soweit, so gut.

Angenommen, Prozessor 2 hat nun mit der ersten FAA in Zeile (b) gerade erst den Semaphor `s` auf –1 erniedrigt, wenn der erste Prozessor seine `V()`-Operation durchführt und damit den Semaphor auf `s=0` erhöht. Würde nun Prozessor 2 die zweite FAA-Instruktion in (c) durchführen und so auf `s=1` erhöhen, so könnte der Prozessor, der danach die FAA in (b) als erster durchläuft, normal die `P()`-Operation verlassen. Diese Gelegenheit kann aber durch den 3. Prozessor verhindert werden, indem er die FAA in (b) durchläuft noch bevor der 2. Prozessor (c) durchlaufen hat und damit den Semaphor wieder auf –1 erniedrigt, so daß er anschließend vom 2. Prozessor nur auf 0 statt auf 1 erhöht wird.

So ist durch die Sequenz (*timing*) [$P_3$(b), $P_2$(c), $P_2$(b), $P_3$(c)], ... eine alternierende Sequenz für den Semaphor mit s=0,–1,0,–1, ... denkbar, in der der Semaphor nie den richtigen Wert hat, um einen der Prozessoren aus der Warteschleife zu entlassen: Die beiden Prozessoren bleiben blockiert, obwohl der kritische Abschnitt frei ist.

Dies wird nun durch die `WHILE`-Schleife verhindert, da hier `s` nur gelesen und nicht verändert wird, so daß alle „Verlierer"-Prozesse zunächst einfach warten, ohne den Weg für andere blockieren zu können.

### 2.3.3 Beispiel UNIX: Semaphore

In UNIX gibt es in einigen Versionen den Zugriff auf Semaphore als Systemaufruf. Beispielsweise existieren in HP-UX die folgenden Aufrufe:

| | |
|---|---|
| `lockf` | Semaphoroperation zum Dateizugriff |
| `msem_init` | Initialisierung eines Semaphors zum Speicherzugriff |
| `msem_lock` | Sperren eines Semaphors |
| `msem_unlock` | Entsperren eines Semaphors |
| `msem_remove` | Entfernen eines Semaphors zum Speicherzugriff |
| `semctl` | Allgemeine Semaphorkontrolloperationen |
| `semget` | Hole Semaphor |
| `semop` | Semaphoroperation |

Die Syntax und Semantik der Semaphoroperationen ist dabei von UNIX-Version zu -Version verschieden. Wichtiger noch als die expliziten Semaphore sind die impliziten atomaren Operationen, die immer standardmäßig angeboten werden und mit denen sich auch Semaphore implementieren lassen.

Beispielsweise ist das Erzeugen einer Datei (genauer: einer Dateiverwaltungsinformation mit Dateinamen etc.) eine atomare Operation: Entweder existiert eine

neue Datei mit dem angegeben Namen nach dem Systemaufruf, oder die Prozedur liefert eine Fehlermeldung zurück, beispielsweise „Name doppelt vorhanden“. Dies ist nötig, um zu verhindern, daß zwei Prozesse gleichzeitig zwei verschiedene Dateien mit demselben Namen erzeugen. Es wird deshalb dazu verwendet, um das nötige Zusammenspiel zwischen Datenerstellung und Ausdruckprozeß (*Erzeuger-Verbraucher*-Modell) für den Ausdruck von Dateien (*printer demon*) zu ermöglichen. Die dafür erzeugten Dateien bzw. Semaphore werden als Schlösser oder Schloßdateien (*lock files*) bezeichnet.

Atomare Aktionen werden in UNIX dadurch begünstigt, daß ein Prozeß, der im Kernmodus ist, nicht abgebrochen werden kann. Er behält so lange die Kontrolle, bis er in einer Kernprozedur schlafen gelegt wird, oder aber in den *user mode* zurückkehrt.

In den Multiprozessor-UNIX-Systemen werden Semaphore im Kern auch zur Koordination der Prozessoren eingesetzt. In der Hewlett-Packard UNIX-Version HP-UX beispielsweise werden alle kritische Abschnitte für die wichtigen Kerndaten und Tabellen durch *busy-wait*-Semaphore (*spin locks*) abgesichert.

### 2.3.4 Beispiel Windows NT: Semaphore

Es gibt verschiedene Konstrukte zur Synchronisation in Windows NT. Für die klassische Synchronisation von Prozessen und *threads* gibt es die `CreateSemaphore()`- und `OpenSemaphore()`-Betriebssystemaufrufe, die Semaphore im globalen Namensraum erzeugen (ähnlich wie eine Datei) bzw. den Zugriff darauf initialisieren. Den `P()`- und `V()`-Operationen entsprechen die Prozeduren `WaitForSingleObject(Sema,TimeOutValue)` und `ReleaseSemaphore()`.

Dabei kann man die Semaphore als Zähler mit einem maximalen endlichen Wert oder als wechselseitige ausschließende (binäre) Untervariable wählen. Durch den möglichen systemweiten Zugriff im Namensraum sind diese Semaphorkonstrukte mit einem erhöhten Aufwand ansprechbar. Deshalb sind für die Koordination der *threads* innerhalb eines Prozesses zusätzliche „Leichtgewichtssemaphoren“ geschaffen worden, die nur innerhalb eines Prozesses (innerhalb eines Programms) bekannt sind. Diese als „*critical section*“ benannte Semaphore werden vom Typ `CRITICAL_SECTION` deklariert und mit `InitializeCriticalSection(S)` initialisiert. Die `P()`- und `V()`-Operationen heißen hier `EnterCriticalSection(S)` und `LeaveCriticalSection(S)`.

Da Windows NT speziell für enggekoppelte Multiprozessorsysteme mit gemeinsamem Speicher gedacht ist, benötigt der Kern besonders Primitive zur Multiprozessorsynchronisation. Hier werden als Semaphore *spin locks* eingesetzt, so daß ein *thread* mit einem *spin lock* den Prozessor so lange blockiert, bis es den *spin lock* wieder verläßt. Deshalb werden die *spin lock*-Operationen nur für Dienste innerhalb der NT Executive zur Verfügung gestellt und unterliegen einigen Einschränkungen. Beispielsweise dürfen im kritischen Abschnitt keine Referenzen zu Speicherbereichen gemacht werden, die auf den Massenspeicher ausgelagert

wurden, man kann keine externe Prozeduren aufrufen (incl. Systemaufrufe) und kann keine Interrupts oder Ausnahmebehandlungen (*exceptions*) auslösen.

Für höhere Dienste, die auch vom Anwender genutzt werden können, gibt es deshalb als Primitive sog. *kernel objects*, die verschiedene Synchronisationseigenschaften haben und in das normale Schedulingsystem integriert sind. Beispiele für Kernobjekte sind die Semaphore, Ereignisse (*events*), Ereignispaare, *timer*, *threads*, sog. Mutanten (benutzerdefinierte *mutual exclusion*-Objekte) usw.

Die Objekte können in zwei Zuständen sein: *signalisiert* und *nicht-signalisiert*. Beispielsweise geht ein Semaphorobjekt in den „signalisiert"-Zustand über, wenn der Zähler auf null geht und bewirkt, daß alle darauf wartenden *threads* „befreit" werden. Ein *thread* kann in Windows NT auf andere *threads*, auf Ereignisse, Semaphore etc. warten und sich dadurch mit ihnen synchronisieren. Die Win32 API stellt dafür die Prozeduren `WaitForSingleObject()` und `WaitForMultipleObjects()` zur Verfügung. Beispielsweise kann ein *thread* in einem Spreadsheet-Programm ein anderes *thread* aufrufen, das ein Spreadsheet ausdruckt. Beendet der Benutzer das Programm, so wartet der Hauptprozeß (*main thread*) mit `WaitForSingleObject()` auf das Ende des Druckprozesses, bevor das Programm vollständig beendet wird und alle internen Daten verloren gehen.

In Windows NT gibt es noch ein weiteres Konzept, den Zugriff auf globale Ressourcen zu koordinieren: Die Gruppierung von *threads* und Objekten in sogenannte *Apartments*. Enthält ein Apartment nur einen *thread* (Single-Theaded Apartment STA), so ist ein Zugriff auf alle Objekte des Apartments unproblematisch und ohne besondere Sicherheitsvorkehrungen möglich. Versucht ein *thread* von außerhalb des Apartments auf ein Objekt zuzugreifen, so muß er dazu einen Systemaufruf durchführen; ein direkter Zugriff auf diese Objektinstanz ist nicht möglich. Dazu wird er in eine Warteschlange („Marshalling") für das gewünschte Objekt gehängt, die durch Semaphoren abgesichert ist. In Abb. 2.23 ist ein Beispiel mit verschiedenen Apartments gezeigt.

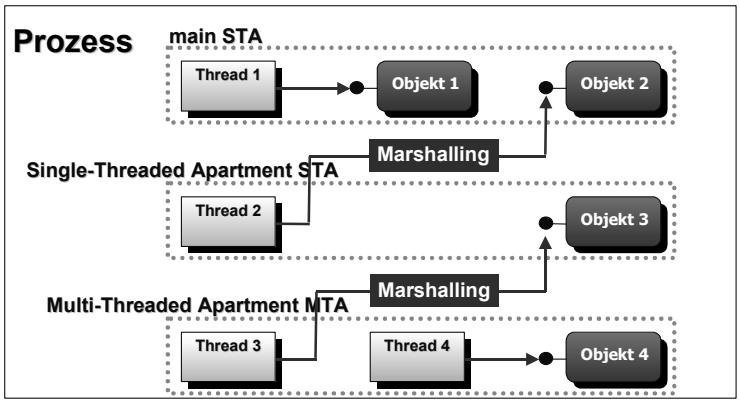



**Abb. 2.23** *Ein Beispiel für das* Apartment-*Modell in Windows NT*

Abgesehen von einem Apartment für den Haupt-Thread des Programms (main STA) gibt es noch Apartments, in denen mehrere *threads* nebeneinander existieren (Multi-Threaded Apartment MTA). Hier muß der Zugriff auf gemeinsame Objekte vom Programmierer selbst geregelt werden, um Probleme zu verhindern.

### 2.3.5 Anwendungen

Semaphore lassen sich für eine Vielzahl von unterschiedlichen Synchronisationsproblemen anwenden. Hier sollen einige wichtige und instruktive Beispiele aufgeführt werden.

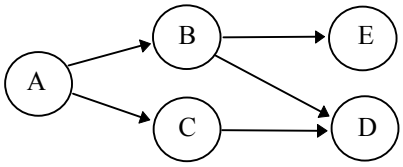
#### *Synchronisation von Prozessen*

Die `P()`-Operation für Semaphore blockiert einen Prozeß so lange, bis er durch die `V()`-Operation freigegeben wird. Dies können wir benutzen, um beispielsweise den gegenseitigen Abhängigkeiten in einem Prozeßsystem Rechnung zu tragen und die Reihenfolge der Prozeßaktivierung exakt festzulegen.

Dazu definieren wir uns für jede Abhängigkeit einen besonderen Semaphor, der am Anfang bzw. am Ende der Prozesse zum Warten bzw. Start benutzt wird.

**Beispiel** *Synchronisation mit Semaphoren*

Der Präzedenzgraph der Tasks A,B,C,D,E sei folgendermaßen gegeben:




Dann lassen sich die Prozesse einzeln so formulieren:

```
PROCESS A; BEGIN         TaskBodyA; V(b); V(c);     END A;
PROCESS B; BEGIN P(b);   TaskBodyB; V(d1);V(e);     END B;
PROCESS C; BEGIN P(c);   TaskBodyC; V(d2);          END C;
PROCESS D; BEGIN P(d1); P(d2);  TaskBodyD;          END D;
PROCESS E; BEGIN P(e);   TaskBodyE;                 END E;
```

Die Semaphore `b,c,d1,d2,e` müssen als globale Variablen definiert und mit 0 initialisiert sein. Dadurch warten alle Prozesse, auch wenn sie zu unterschiedlichen Zeiten loslaufen, an den Semaphoren auf ihre Aktivierung. Interessanterweise macht es nichts aus, ob hier ein Prozeß zu spät kommt: Das Wecksignal wird in dem Semaphor gespeichert, so daß es nicht verloren gehen kann.

### *Das Erzeuger-Verbraucher- (bounded buffer)-Problem*

Viele Aufgaben der Datenerstellung, -verwaltung oder -filterung lassen sich als ein System von zwei Einheiten oder Prozessen formulieren, von denen einer Daten produziert und der andere sie abnimmt. Ein praktisches Beispiel ist das Aneinanderheften, Formatieren und Ausdrucken von Text in UNIX, s. S. 18.

Bezeichnen wir den Erzeuger als *producer* und den Verbraucher als *consumer,* so besteht das *producer-consumer*-Problem darin, daß die Erzeugung und Abnahme der Daten jeweils mit unterschiedlichen Raten (Daten pro Zeiteinheit) erfolgt. Üblicherweise wird man einen Puffer zwischen die Prozesse schalten, der vom Erzeuger gefüllt und vom Verbraucher geleert wird. Da der Puffer nur endlich ist (**bounded buffer**), muß der Strom der Daten zwischen Erzeuger und Verbraucher durch eine Flußkontrolle synchronisiert werden. Beispielsweise legt sich der Erzeuger schlafen, wenn der Puffer voll ist und wird vom Verbraucher bei nichtvollem Puffer wieder aufgeweckt. Entsprechend läßt sich auch analog für den Verbraucher umgekehrt verfahren. In Abb. 2.24 ist zunächst der einfache, naive Ansatz gezeigt, wobei die Puffergröße mit `N` und die globale Variable `used` mit `0` initialisiert werden:

*Erzeuger*

```
LOOP
  produce(item)
  IF used=N THEN sleep();
  putInBuffer(item);
  used := used+1;
  IF used=1
   THEN wakeup(Verbraucher);
END
```

*Verbraucher*

```
LOOP
  IF used=0 THEN sleep();
  getFromBuffer(item);
  used := used-1;
  IF used=N-1
    THEN wakeup(Erzeuger);
  consume(item);
END
```

**Abb. 2.24** *Das Erzeuger-Verbraucher-Problem*

Diese Lösung hat ein Problem: Es kann leicht zu einer *race condition* kommen. Um dies zu sehen, betrachten wir den folgenden Fall. Angenommen, der Puffer ist leer und der Verbraucher hat gerade `used=0` gelesen, als auf den Erzeuger umgeschaltet wird. Dieser produziert ein `item`, schiebt es in den Puffer und inkrementiert `used` um 1. Da `used=1`, will er den Verbraucher wecken. Dieser muß aber nicht aufgeweckt werden; erst bei der nächsten Prozessorzuteilung legt er sich schlafen, da `used=0` war. Nachdem der Erzeuger den Puffer gefüllt hat, legt er sich ebenfalls schlafen. Beide schlafen dann ewig.

Diese Situation entstand, weil das *wakeup*-Signal zu früh kam und damit verlorenging. Würden wir es statt dessen speichern, beispielsweise mit einem *wakeup-*

Bit, hilft das kurzfristig bei diesem Beispiel, aber bei weiteren Prozessen müssen weitere Bits eingeführt werden, um das Problem zu verhindern. Haben wir eine unbekannte Anzahl von Prozessen, wie dies in normalen Systemen der Fall ist, so fällt diese Lösung aus.

Statt dessen führt die Einführung von Semaphoren zu einer Lösung. Da Semaphore inkrementiert und dekrementiert werden, sind alle *wakeup*-Signale gespeichert und entbinden uns von der Notwendigkeit spezieller Statusbits. Die in Abb. 2.25 gezeigte Lösung mit Semaphoren wird mit `belegt:=0`, `frei:=N` und `mutex:=1` initialisiert.

*Erzeuger*

```
LOOP
  produce(item)
  P(frei);
  P(mutex);
  putInBuffer(item);
  V(mutex);
  V(belegt);
END
```

*Verbraucher*

```
LOOP
  P(belegt);
  P(mutex);
  getFromBuffer(item);
  V(mutex);
  V(frei);
  consume(item);
END
```

**Abb. 2.25** *Lösung des Erzeuger-Verbraucher-Problems mit Semaphoren*

Der *mutual exclusion*-Zugang zum Puffer ist jeweils durch den gemeinsamen Semaphor `mutex` abgesichert. Mit dem Semaphor `belegt` wird die Zahl der belegten Pufferplätze gezählt und mit `frei` die Zahl der freien Plätze. Anstelle der Abfrage des Zählers und dem `sleep`- bzw. `wakeup`-Aufruf werden nun die Semaphoroperationen aufgerufen, wobei für die Schlafbedingungen `used=N` bzw. `used=0` jeweils ein eigener Semaphor verwendet wird. Der produzierende Prozeß dekrementiert nur die Zahl der freien Plätze; wenn sie null sind, wird er blockiert, bevor er den nächsten Platz belegen kann. Kann er den Platz belegen, so inkrementiert er die Zahl der belegten Plätze und weckt so den Verbraucherprozeß auf, der sich bei null belegten Plätzen schlafen gelegt hatte.

Man beachte, daß diese Lösung durch die Initialisierung vom `mutex` auf eins nur einen gegenseitigen Ausschluß im kritischen Abschnitt der Pufferverwaltung bewirkt. Bei der Erzeugung und Abnahme der `items` dagegen können auch mehrere Produzentenprozesse und Konsumentenprozesse mit dem gleichen Code existieren und den Puffer füllen bzw. leeren, ohne sich gegenseitig zu stören.

### *Das readers/writers-Problem*

Das Grundproblem der Synchronisation, Schreiben und Lesen von mehreren Prozessen auf gemeinsamen Speicherbereichen zu koordinieren, läßt sich auch beim Schreiben und Lesen von Dateien wiederfinden und wird dann als das **readers/writers-Problem** bezeichnet. Grundforderung ist, das gleichzeitige Lesen und Schreiben auf dem gemeinsamen Datenbereich auszuschließen, um Dateninkonsistenzen zu vermeiden. Dies läßt sich mit einem einzigen Semaphor regeln. Stellen wir allerdings zusätzliche Forderungen auf, so werden die Lösungen komplizierter. Folgende Versionen des Problems existieren:

- Ein Leser soll nur warten, falls ein Schreiber bereits Zugriffsrecht zum Schreiben bekommen hat (**erstes** *readers/writers*-Problem). Dies bedeutet, daß kein Leser auf einen anderen Leser warten muß, nur weil ein Schreiber wartet.

- Wenn ein Schreiber bereit ist, führt er das Schreiben so schnell wie möglich durch (**zweites** *readers/writers*-Problem). Wenn also ein Schreiber bereit ist, die Zugriffsrechte zu bekommen, dürfen keine neuen Leser mit Lesen beginnen.

Man beachte, daß eine Lösung des ersten Problems darin resultieren kann, daß ein Schreiber ewig wartet; eine Lösung des zweiten Problems kann dazu führen, daß ein Leser ewig wartet. In beiden Fällen *verhungern* die Prozesse, da das Warten nur von dem Strom der Leser bzw. Schreiber abhängt und keine prinzipielle Blokkade vorliegt.

Eine mögliche Lösung für das erste Problem führt einen gemeinsamen Zähler, der die Anzahl der Leser im kritischen Abschnitt registriert. Ist ein Leser in dem Abschnitt, wird der Schreiber ausgesperrt, sonst nicht. Dazu gibt es zwei Semaphore `ReadSem` und `RWSem`: Einer, der den Zähler fürs Lesen schützt, und einer, der den exklusiven Zugang Lesen/Schreiben regelt. Der Pseudocode fürs Lesen ist

```
P(ReadSem);
  readcount:=readcount+1;
  IF readcount=1 THEN P(RWSem) END;
V(ReadSem);
  . . .
Reading_Data();
  . . .
P(ReadSem);
  readcount:=readcount-1;
  IF readcount=0 THEN V(RWSem) END;
V(ReadSem);
```

und fürs Schreiben

```
P(RWSem);
    . . .
Writing_Data();
  . . .
V(RWSem);
```

Der erste Leser auf `RWSem` wird nur blockiert, falls ein Schreiber im kritischen Abschnitt ist. Alle anderen Leser warten auf `ReadSem`. Außerdem: Warten sowohl ein Leseprozeß als auch ein Schreibprozeß auf `RWSem`, so entscheidet der Scheduler, welcher von beiden bei der Operation `V(RWSem)` aufgeweckt wird.

#### *Multiprozessor-Warteschlangenmanagement beim NYU-Ultracomputer*

Eines der interessantesten Parallelprozessor-Systeme der 80er Jahre war der Ultracomputer am Courant Institute der New York University (Gottlieb et al. 1983). Er hat eine Multiprozessorarchitektur nach Abbildung 1.12. Für das Scheduling wurde ein spezieller Algorithmus implementiert, der die *fetch and add*-Operation zur Synchronisierung der Prozessoren auf dem *shared memory* benutzt und der hier kurz geschildert werden soll.

Die Warteschlange der bereiten Prozesse (*ready queue*) sollte auch bei parallelem Zugriff durch Multiprozessoren so funktionieren, daß eine Art FIFO-Eigenschaft eingehalten wird: Falls die Einfügeoperation für einen Task *p* abgeschlossen wird, bevor sie für einen Task *q* anfängt, dann darf es nicht möglich sein, daß die Aushängeoperation für *q* vor derjenigen für *p* endet.

Der folgende Algorithmus gilt für einen Ringpuffer der Länge N, s. Abb. 2.26.

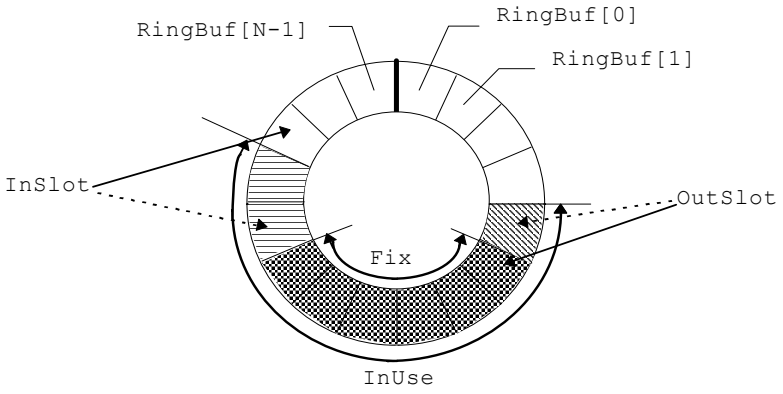



**Abb. 2.26** *Der Ringpuffer des NYU-Ultracomputers*

Dazu existieren die globalen Variablen `InSlot` und `OutSlot`, die den Platz (*Slot*) im Ringpuffer bezeichnen, wo ein neuer Task eingehängt bzw. ein alter entnommen werden kann. Der Puffer hat drei Zonen: eine Einfügezone, eine Aushängezone und eine fixe Zone, in der keine Aktivität herrscht.

Am Anfang ist der Puffer leer und deshalb `InSlot=OutSlot=0`. Beim fortlaufenden Betrieb verschiebt sich der fixe Abschnitt langsam in Uhrzeigerrichtung im Ringpuffer, wobei die `InSlot-` und `OutSlot`-Variablen mittels *fetch and add*-Primitiven geschützt werden. Da noch zusätzlich das Problem des Pufferüberlaufs `Full:=TRUE` bzw. Leerpuffers `Empty:=TRUE` berücksichtigt werden muß, wird das Warteschlangenmanagement schwieriger. Da beim Einhängen zuerst `InSlot` inkrementiert wird und beim Aushängen danach `OutSlot`, stimmt es leider nicht, daß `Full=(InSlot-OutSlot=N)` oder `Empty= (InSlot-OutSlot=0)` gilt, da es eine Reihe von Slots geben kann, die gerade gefüllt werden und noch nicht fertig sind, um geleert zu werden. Aus diesem Grund wird statt dessen für `Full` die anspruchsvollere Bedingung `(InUse=N)` und für `Empty` die Bedingung `(Fix=0)` abgefragt. Die Zahl der Prozessoren, die gerade auf der Schlange arbeiten, ist damit durch `InUse-Fix` gegeben.

***Einhängen***

```
IF InUse>=N
 THEN Full:=TRUE; RETURN
END;

IF FetchAndAdd(InUse,1)>=N
THEN
   FetchAndAdd(InUse,-1);
   Full:=TRUE; RETURN
END;

MyInSlot:=FetchAndAdd
           (InSlot,1)MOD N;

P(InSem[MyInSlot]);
RingBuf[MyInSlot]:= data;
V(OutSem[MyInSlot]);

FetchAndAdd(Fix,1);
```

***Aushängen***

```
IF Fix <=0
  THEN Empty:=TRUE; RETURN
END;

IF FetchAndAdd(Fix,-1)<=0
THEN
   FetchAndAdd(Fix,1);
   Empty:=TRUE; RETURN
END;

MyOutSlot:=FetchAndAdd
           (OutSlot,1)MOD N;

P(OutSem[MyOutSlot]);
data:= RingBuf[MyOutSlot];
V(InSem[MyOutSlot]);

FetchAndAdd(InUse,-1);
```

**Abb. 2.27** *Ein- und Aushängen bei der ready-queue*

Der Code in Abb. 2.27 hat verschiedene Feinheiten. Ohne die Semaphore entspricht dies im wesentlichen der Implementierung der Semaphoroperationen, wie sie im Beispiel der Multiprozessorsynchronisation auf Seite 62 für *fetch and add* gezeigt sind, um den kritischen Abschnitt, das Einhängen und Aushängen der Daten des Gesamtpuffers, zu schützen und einen kontrollierten Zugang bereitzustellen. Die zweite Abfrage (grauer Bereich in Abb. 2.27) vorher, die mit einer Feh-

lermeldung `Full` oder `Empty` zurückkehrt, ist aus den schon bei diesem Beispiel erwähnten Gründen nötig. Der aufrufende Prozeß kann dann selbst entscheiden, was in diesen Ausnahmefällen zu tun ist und muß nicht automatisch in einer Schleife warten.

Die Existenz der beiden Semaphoroperationen, die durch *fetch and add*-Operationen implementiert werden können und im Originalbeitrag (Gottlieb et al. 1983) es auch sind, haben einen anderen Sinn. Angenommen, von den beiden ersten Einträgen, die in die leere Liste gefüllt werden, wird der zweite Prozessor zuerst fertig. Ein nachfolgendes Aushängen würde nun sofort Eintrag 1 aushängen, ohne zu merken, daß dieser noch nicht aktuell vorhanden ist, und die ungültigen Daten weiterverwenden. Aus diesem Grund wird nicht nur die Zahl der Einträge und damit das *bounded buffer*-Problem geregelt, sondern auch jeder einzelne Slot mit einem Semaphor versehen, der initial auf eins gesetzt wird. Dies hat zur Folge, daß entweder ein Einhängen oder ein Aushängen auf dem Slot durchgeführt wird, aber nicht beides gleichzeitig. Die Verwendung von *zwei* Semaphoren, deren Aufrufe wechselseitig verschränkt sind, garantiert nun mit den Anfangsbedingungen `InSem[i]:=1`, `OutSem[i]:=0` zusätzlich, daß bei jedem Slot zuerst ein Einfügen stattfindet, dann ein Aushängen und so wechselseitig fort.

Man beachte, dass sowohl `InSlot` als auch `OutSlot` nur erhöht werden, also irgendwann überlaufen. Wird aber N als Zweierpotenz gewählt, so bedeutet die Modulo-Operation, dass der Überlauf nicht weiter bemerkt wird, da sowieso nur die unteren Bits ausgelesen werden.

### 2.3.6 Aufgaben

**Aufgabe 2.3-1**

Was wäre, wenn in Abschn. 2.3.1 zuerst Prozeß A die Kontrolle erhält und nach Schritt (2) die Umschaltung erfolgt? Gibt es noch weitere Möglichkeiten der Fehlererzeugung?

**Aufgabe 2.3-2 (mutual exclusion)**

Formulieren Sie die Prozeduren `betrete_Abschnitt(Prozess: INTEGER)` und `verlasse_Abschnitt(prozess:INTEGER)`, die die Lösung von Peterson für das *mutual exclusion*-Problem enthalten. Definieren Sie dazu die zwei globalen, gemeinsamen Variablen `Interesse[1..2]` und `dran`. Könnte eine Verallgemeinerung auf *n* Prozesse existieren und wenn ja, wie?

**Aufgabe 2.3-3 (Semaphore)**

a) Wie würden die Semaphoroperationen P und V formuliert, wenn s die Anzahl der jeweils wartenden Prozesse enthalten soll ?
b) Wie ändert sich die Lösung des Erzeuger-Verbraucher-Problems dadurch?
c) Wie muß man *s* verändern, wenn mehrere Betriebsmittel existieren?

**Aufgabe 2.3-4 (Prozeßsynchronisation)**

Wie lautet die Synchronisierung der Betriebsmittel aus Abb. 2.11 mit Hilfe von Semaphoren?

**Aufgabe 2.3-5 (Chinesische, dinierende Philosophen)**

*N* Philosophen sitzen um einen Eßtisch und wollen speisen. Auf dem Tisch befindet sich eine große Schale mit Reis und Gemüse. Jeder Philosoph meditiert und ißt abwechselnd.

Zum Essen benötigt er jeweils 2 Stäbchen, auf dem Tisch liegen aber nur *N* Stäbchen, so daß er sich ein Stäbchen mit seinem Nachbarn teilen muß. Also können nicht alle Philosophen zur selben Zeit essen; sie müssen sich koordinieren. Wenn ein Philosoph hungrig wird, versucht er in beliebiger Reihenfolge das Stäbchen links und rechts von ihm aufzunehmen, um mit dem Essen zu beginnen. Gelingt ihm das, so ißt er eine Weile, legt die Stäbchen nach seinem Mahl wieder zurück und meditiert. Dabei sind allerdings Situationen denkbar, in der alle gleichzeitig agieren und sich gegenseitig behindern.

Geben Sie eine Vorschrift für jeden Philosophen (ein Programm in einem MODULA oder C-Pseudocode) an, das dieses Problem löst!

### 2.3.7 Kritische Bereiche und Monitore

Obwohl die Einführung von Semaphoren die Synchronisation bei Prozessen stark erleichtert, ist der Umgang mit Ihnen nicht unproblematisch. Vertauschen wir beispielsweise die Reihenfolge der Operationen zu

`V(mutex); ..` krit. Abschnitt `..; P(mutex);`

so ist das Ergebnis gerade invers zum Gewünschten: Alle sind nur im kritischen Abschnitt zugelassen. Auch der Fehler

`P(mutex); ..` krit. Abschnitt `..; P(mutex);`

oder das einfache Weglassen einer der beiden Operationen führt zu starken Problemen. Dabei reicht schon eine subtile, in großen Programmen fast nicht bemerkbare Änderung aus, um die unerwünschten, nur sporadisch auftretenden Fehler zu erzeugen. Beispielsweise ist eine Vertauschung der Operationen in der Lösung des *Erzeuger-Verbraucher*-Problems

von `P(frei);P(mutex);putInBuffer(item);V(mutex);V(belegt);`

zu `P(mutex);P(frei);putInBuffer(item);V(mutex);V(belegt);`

äußerst problematisch (warum?).

Aus diesem Grund schlugen Hoare (1972) und Brinch-Hansen (1972) vor, die richtige Erzeugung und Anordnung der Semaphoroperationen dem Compiler zu überlassen und statt dessen ein neues Sprachkonstrukt, den **kritischen Bereich**

(**critical region**), einzuführen. Die Sprachsyntax für eine gemeinsame Variable (Semaphor) *s* eines beliebigen Typs ist

**region** s **do** <statement>

was der Folge entspricht `P(s); <statement>; V(s)`

Gibt es also mehrere Prozesse, die für sich einen kritischen Bereich mit derselben gemeinsamen Variablen *s* (deklariert mit **shared** *s*) betreten wollen, so garantieren die vom Compiler erzeugten Synchronisationsmechanismen dafür, daß immer nur ein Prozeß dies kann. Zwei parallel ablaufende Prozesse mit

**region** s **do** S1 **region** s **do** S2

erzeugen also Sequenzen der Form S1; S2; ... oder S2; S1; ..., wobei zwar die Reihenfolge der Befehle bzw. Befehlssequenzen S1 und S2 beliebig, die Befehlsausführungen aber nicht gemischt sein dürfen.

Führt man noch zusätzlich eine Bedingung B ein (Hoare 1972), die den Zugang zu einem solchen kritischen Abschnitt regelt, in der Form

**region** s **when** B **do** <statement>

so erhalten wir einen *bedingten* kritischen Bereich (*conditional critical region*). Die Semantik des bedingten kritischen Bereichs dieses Konstrukts sieht vor, daß ein Prozeß nach Belegen von s nur dann den kritischen Abschnitt betreten kann, wenn auch die Bedingung B erfüllt ist. Ist dies nicht der Fall, wird s wieder aufgegeben, und der Prozeß legt sich schlafen, bis die Bedingung erfüllt ist. Sodann versucht er wieder, *s* zu belegen.

Dieses Konzept wurde in objektorientierter Weise erweitert, um auch passive Daten mitzuschützen. Dazu schlugen Brinch Hansen (1973) und Hoare (1974) einen neuen abstrakten Datentyp (Klasse) vor, der die Aufgabe übernimmt, alle darin enthaltenen, von außen ansprechbaren Prozeduren (*Methoden,* abstrakte Datentypen ADT) und die lokalen, dazu benutzten Daten und Variablen durch Synchronisationsprimitive automatisch zu schützen und den wechselseitigen Ausschluß von Prozessen darin zu garantieren. Diese Klasse nannten sie einen **Monitor**, was aber nichts mit einem Bildschirm oder einem Softwaremonitor aus Abschn. 2.2.5 zu tun hat.

Das Monitorkonstrukt folgt syntaktisch dem Schema

```
MONITOR <name>
   (* lokale Daten und Prozeduren *)
   PUBLIC
   (*Prozeduren der Schnittstelle nach außen *)
   BEGIN (*Initialisierung *)
   ...
   ENDMONITOR
```

Der Aufruf der allgemein zugänglichen (also unter PUBLIC deklarierten), geschützten Prozeduren erfolgt durch Voranstellen des Prozedurnamens.

**Beispiel** *Erzeuger-Verbraucher*

```
MONITOR Buffer;
TYPE       Item = ARRAY[1..M] of BYTE;
VAR        RingBuf: ARRAY[1..N] of Item;
           InSlot,  (* 1. freier   Platz *)
           OutSlot, (* 1. belegter Platz *)
           used:  INTEGER;
PUBLIC PROCEDURE putInBuffer(item:Item);
            BEGIN  ...  END putInBuffer;
       PROCEDURE getFromBuffer(VAR item:Item);
            BEGIN  ...  END getFromBuffer;
BEGIN
 used:=0; InSlot:=1; OutSlot:=1;
END MONITOR;
```

Der Gebrauch des Monitors sieht dann wie in Abb. 2.28 aus:

*Erzeuger*

```
LOOP
  produce(item)
  Buffer.putInBuffer(item)
END
```

*Verbraucher*

```
LOOP
  Buffer.getFromBuffer(item)
  consume(item);
END
```

**Abb. 2.28** *Lösung des Erzeuger-Verbraucher-Problems mit einem Monitor*

Allerdings ist der Code noch nicht vollständig. Im Beispiel von Abb. 2.25 sahen wir, daß es nötig sein kann, zusätzliche Wartebedingungen (hier: zur Flußkontrolle des Pufferüberlaufs) innerhalb eines kritischen Abschnitts vorzusehen. Dies ist im Monitor prinzipiell durch spezielle Variablen vom Typ CONDITION vorgesehen. Die einzigen Operationen, die auf diesen Variablen zugelassen sind, sind die beiden folgenden Signaloperationen:

- *signal(s)*

  sendet ein Signal s innerhalb eines Monitors an alle, die darauf warten, unabhängig davon, ob überhaupt jemand wartet.

- *wait(s)*

  wartet so lange, bis ein Signal s gesendet wird, und führt danach die nächsten Instruktionen aus.

Wir erweitern deshalb unseren Monitor um die Bedingungsvariablen

```
VAR frei, belegt: CONDITION
```

die für die Flußkontrolle nötig sind. Die beiden vom Monitor geschützten Zugriffsoperationen sehen damit dann folgendermaßen aus:

```
PROCEDURE putInBuffer(item: Item);
BEGIN
  IF used=N THEN wait(frei) END;
  RingBuf[InSlot]:=item;
  used := used+1;
  InSlot := (InSlot+1) MOD N;
  signal (belegt);
END putInBuffer;

PROCEDURE getFromBuffer(VAR item: Item);
BEGIN
  IF used=0 THEN wait(belegt) END;
  item := RingBuf[OutSlot];
  used := used-1;
  OutSlot := (OutSlot+1) MOD N;
  signal(frei);
END getFromBuffer;
```

Diese Implementation erinnert uns an den naiven Ansatz in Abb. 2.24, S.67. Im Unterschied zu diesem Ansatz haben wir allerdings nun die Abfragen zur Flußsteuerung mit gegenseitigem Ausschluß geschützt, so daß sich keine *race conditions* bilden können.

Die Anwendung des Monitorkonstrukts in Betriebssystemen ist allerdings nicht unproblematisch (Keedy 1979) Die wesentlichsten Kritikpunkte sind die unerwartete und unnötige Sequentialisierung von Betriebssystemabläufen, die bei der (ungewollten) Schachtelung von Monitoraufrufen durch die Funktionsaufrufe an sich unabhängiger, separat geschriebener Betriebssystemteile auftritt (Lister 1977) sowie die besonderen Wartezustände der Prozesse in den Monitoren, die es schwierig machen, bei Zeitüberschreitungen, Abschalten des Gesamtsystems (*shutdown*) oder Prozeßabbrüchen die Wartezustände konsistent aufzulösen (Lampson u. Redell 1980). Mit den allgemeinen Konstrukten wie *Prozesse* und *Signale* kann man bei geschickter Programmierung die genannten Probleme vermeiden.

Ein weiteres praktisches Problem besteht darin, daß die Semaphoroperationen als Zusatzmodule (z. B. mit speziellen Maschinenbefehlen) einer Anwenderbibliothek geschrieben und damit leicht in fast allen Programmiersprachen zur Verfügung gestellt werden können, im Gegensatz dazu aber für kritische Bereiche und Monitore die Compiler erweitert werden müssen. Dies bringt zusätzliche Komplikationen mit sich und hat sich deshalb kaum durchgesetzt.

**Beispiel** *Java*

Eine wichtige objektorientierte Version von C bzw. C++ ist die Programmiersprache *Java*. Ihre Anwendungsmodule (**applets**) sind dazu gedacht, als Softwaremodule kleiner Systeme zu fungieren sowie als Bestandteil einer World-Wide-Web-Seite beim Benutzer spezielle Funktionen zu erfüllen, die der normale Web-Browser nicht erfüllen kann. Dazu wird der Sourcecode (oder der kompilierte, symbolische Code) über das Netz auf den Benutzerrechner geladen und dort von einem Java-Interpreter interpretiert, der Bestandteil des Browsers oder des Betriebssystems (z.B. wie in OS/2) sein kann.

Der Java-Code soll auf sehr unterschiedlichen Rechnersystemen gleiche Ergebnisse liefern, wobei auch mehrere *threads* parallel abgearbeitet werden können. Die Koordination mehrerer dieser Leichtgewichtsprozesse für den Zugriff auf ein gemeinsames Objekt wird mit Hilfe des Schlüsselwortes `synchronized` erreicht. Die Syntax ist mit

```
synchronized (<expression>) <statement>
```

sehr ähnlich einem kritischen Bereich. Ein *thread* kann nur dann die Instruktion (bzw. Instruktionsfolge) `<statement>` ausführen, wenn vorher das Objekt oder Feld `<expression>` (durch Semaphore geschützt) belegt wurde. Ist das Objekt schon belegt, so wird er so lange blockiert, bis der andere *thread* den kritischen Bereich wieder verläßt. Innerhalb eines `synchronized`-Bereichs kann man zusätzlich mit den Methoden `wait()` auf ein Signal warten, das mit `notify()` gesendet wird.

### 2.3.8 Verklemmungen

Auch in einem wohlsynchronisierten Betriebssystem mit guten Schedulingalgorithmen können Situationen auftreten, in denen Prozesse sich gegenseitig behindern und blockieren, so daß die Programme nicht ausgeführt werden können. Wie ist das möglich?

Betrachten wir dazu folgende Situation: Angenommen, Prozeß 1 sammelt Daten, aktualisiert dann Einträge in einer Datei und protokolliert dies auf einem Drucker. Nun beginnt ein Prozeß 2, der die gesamte Datei ausdrucken soll. Beide Betriebsmittel, der Drucker und die Datei, sind mit wechselseitigem Ausschluß geschützt, um einen Wirrwarr beim Schreiben und Lesen zu verhindern. Nun geschieht folgendes: Nach dem Datensammeln sperrt Prozeß 1 die Datei, aktualisiert einen Eintrag und will ihn vor der Aktualisierung des zweiten Eintrags ausdrucken. Währenddessen hat Prozeß 2 den Drucker gesperrt, die Überschrift ausgedruckt und beantragt die Datei. Da diese gerade benutzt wird, legt sich Prozeß 2 mit dem Semaphor schlafen. Aber auch Prozeß 1 legt sich beim Versuch, den Drucker zu bekommen, schlafen. Beide schlafen nun ewig – im Prozeßsystem ist eine **Verklemmung** (**deadlock**) aufgetreten.

Wir haben also die Situation, daß ein Prozeß Betriebsmittel A (die Datei) belegt und ein weiteres Betriebsmittel B (den Drucker) haben will, das seinerseits von einem anderen Prozeß belegt ist, der wiederum das bereits belegte Betriebsmittel A haben will.

Aus unserem Beispiel lassen sich einige Eigenschaften abstrahieren, die nach Coffman et al. (1971) notwendige und hinreichende Bedingungen für diese Art von Störungen sind:

(1) *Beschränkte Belegung (mutual exclusion)*

Jedes involvierte Betriebsmittel ist entweder exklusiv belegt oder aber frei.

(2) *Zusätzliche Belegung (hold-and-wait)*

Die Prozesse haben bereits Betriebsmittel belegt, wollen zusätzlich weitere Betriebsmittel belegen und warten darauf, daß sie frei werden. Die insgesamt nötigen Betriebsmittel werden also nicht auf einmal angefordert.

(3) *Keine vorzeitige Rückgabe (no preemption)*

Die bereits belegten Betriebsmittel können den Prozessen nicht einfach wieder entzogen werden, ähnlich dem preemptiven Scheduling der CPU, sondern müssen von den Prozessen selbst explizit wieder zurückgegeben werden.

(4) *Gegenseitiges Warten (circular wait)*

Es muß eine geschlossene Kette von zwei oder mehr Prozessen existieren, bei denen jeweils einer Betriebsmittel vom nächsten haben will, die dieser belegt hat und die deshalb nicht mehr frei sind.

Eine solche Situation gibt es nicht nur bei solchen Betriebsmitteln wie Druckern, Bandgeräten oder Scannern, sondern auch bei logischen Einheiten wie Semaphoren. Betrachten wir beispielsweise das *Erzeuger-Verbraucher*-Beispiel in Abb. 2.25 auf Seite 68 und nehmen wir an, daß zusätzlich zu `belegt:=0` aus Versehen auch `frei:=0` gesetzt wurde. Nun haben wir die Situation

| *Erzeuger* | *Verbraucher* |
|---|---|
| `...` | `...` |
| `wait(frei)` | `wait(belegt)` |
| `...` | `...` |
| `send(belegt)` | `send(frei)` |

Beide warten auf die Freigabe des Semaphors, den der andere jeweils „besitzt“, aber nur freigeben kann, wenn seinen eigenen Wünschen Genüge getan wurde.

Für die Behandlungen von Verklemmungen gibt es folgende vier erfolgreiche Strategien:

- das Problem ignorieren,
- die Verklemmungen erkennen und beseitigen,

- die Verklemmungen vermeiden,
- die Verklemmungen unmöglich machen, indem man eine der obigen Bedingungen (1)–(4) verhindert.

Wir wollen im folgenden diese vier Strategien etwas näher betrachten.

### *Mögliche Verklemmungen ignorieren*

Auf den ersten Blick scheint die Strategie, Probleme zu ignorieren statt sie zu lösen, absolut inakzeptabel zu sein, besonders für Mathematiker. Berücksichtigen wir aber die Tatsache, daß es in großen Systemen noch sehr viele weitere potentielle Störungsmöglichkeiten gibt, die uns das Leben schwer machen können, angefangen bei HW-Problemen (schlechten Lötstellen, wackeligen Kabeln, partiellen Chipausfällen, Netzspannungsschwankungen, …) bis zu SW-Problemen (Fehlern im Betriebssystem, neue Compilerversionen, Umkonfiguration der verwendeten Software, falsche Dateneingaben etc.), so lassen sich die Fälle, bei denen zwei Prozesse sich verklemmen, in der täglichen Praxis als „unbedeutend“ klassifizieren, ähnlich wie der potentielle Überlauf der Prozeßtafeln oder das Überschreiten des Maximums der gleichzeitig offenen Dateien. Physiker betrachten dies eher als „Schmutzeffekt“, den man ignoriert, solange er nicht zu viel stört; EDV-Manager sehen in der Vermeidung von Verklemmungen eher eine „Ablenkung von wichtigeren Aufgaben“. Aus diesem Grund sind in UNIX keine Extramaßnahmen für Verklemmungen vorgesehen.

Unabhängig von der Häufigkeit des Auftretens wäre es aber trotzdem besser, auch Verklemmungen im Betriebssystem automatisch zu behandeln, um wenigstens den vorhersehbaren Ärger zu vermeiden. Leider aber ist dies, wie wir sehen werden, einerseits nicht einfach und andererseits nicht ohne Aufwand für den laufenden Betrieb. Aus diesem Grund muß man beim Einsatz der folgenden Maßnahmen immer Anlaß und Aufwand gegeneinander abwägen.

### *Verklemmungen erkennen und beseitigen*

Eine Strategie zur Behandlung von Verklemmungen besteht darin, die normalen Betriebssystemfunktionen zu überwachen und beim Auftreten von verdächtigen Symptomen einen Algorithmus zu starten, der systematisch die bestehende Situation untersucht und ermittelt, ob tatsächlich eine Verklemmung vorliegt.
Verdächtige Symptome können dabei sein,

- wenn sehr viele Prozesse warten und der Prozessor unbeschäftigt (*idle*-Task!) ist. Dies erfordert eine genaue Definition von „sehr viele“, die man z. B. über eine obere Grenze (Schwellwert) herstellen kann.
- wenn mindestens zwei Prozesse zu lange auf Betriebsmittel warten müssen. Auch hier muß eine obere Grenze für „zu lange“ definiert werden.

Die Situation der Prozesse und der Betriebsmittel läßt sich durch einen Betriebsmittelgraphen (*resource allocation graph*) visualisieren, s. Abb. 2.29.

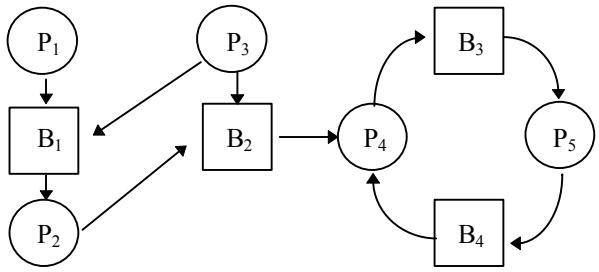



**Abb. 2.29** *Ein Betriebsmittelgraph*

Hierbei sei ein Prozeß als Kreis, ein Betriebsmittel (*resource*) als Viereck und die Belegung als Pfeil symbolisiert. Die Pfeilrichtung Prozeß → Betriebsmittel bedeutet, daß ein Prozeß das Betriebsmittel haben möchte; die Richtung Prozeß ← Betriebsmittel deutet an, daß der Prozeß das Betriebsmittel fest hat.

Die vier notwendigen und hinreichenden Verklemmungsbedingungen bedeuten für den Betriebsmittelgraphen, daß es bei Verklemmungen Zyklen in dem Graphen gibt, was wir im obigen Beispiel direkt bei den Knoten $P_4$, $B_3$, $P_5$, $B_4$ bemerken. Dies läßt sich bei größeren Graphen durch entsprechende Graphenalgorithmen nachprüfen. Einen Algorithmus für den Fall, daß es mehrere Betriebsmittel derselben Art gibt, wurde von Holt (1972) vorgestellt.

Ein anderer, einfacher Algorithmus (Habermann 1969) prüft nach, ob die Zahlen $A_S$ der noch verfügbaren Betriebsmittel des Typs *s* ausreichen, um einen verklemmungsfreien Ablauf zu ermöglichen. Dazu prüfen wir, ob ein Prozeß existiert, der mit den noch freien Betriebsmitteln auskommt. Ist ein solcher Prozeß gefunden, so wird angenommen, daß er nach seinem Abschluß die belegten Betriebsmittel wieder zurückgibt und die Menge $\{A_S\}$ der freien Betriebsmittel anwächst. Dazu wird er markiert und der nächste Prozeß gesucht. Dies wiederholen wir so lange, bis entweder alle Prozesse markiert sind und damit feststeht, daß keine Verklemmung existiert, oder aber mehrere unmarkierte Prozesse übrigbleiben, die sich gegenseitig mit ihren Anforderungen blockieren. Dabei gehen wir natürlich davon aus, daß bei vollständig unbelegten Betriebsmitteln jeder Prozeß für sich

genügend Betriebsmittel zur Verfügung hätte und so allein vollständig abgearbeitet werden könnte.

Mit der Notation

| | |
|---|---|
| $E_S$ | Zahl der existierenden Einheiten pro Betriebsmittel s,<br>z. B. $E_2$=5 für 5 existierende Einheiten vom Typ s=2 (Drucker) |
| $B_{KS}$ | Zahl der Einheiten vom Typ s, die der Prozeß k belegt hat |
| $C_{KS}$ | Zahl der Einheiten vom Typ s, die Prozeß k zusätzlich fordert |

können wir diesen Algorithmus genauer formulieren, wobei wir als Summe der belegten Betriebsmittel $\Sigma_k B_{ks}$, die Spaltensumme der Matrix ($B_{KS}$), erhalten. Die Anzahl $A_S$ der noch freien Betriebsmittel des Typs s ist somit

$$A_S = E_S - \sum_K B_{KS}$$

Die einzelnen Schritte des Algorithmus lauten also:

(0) Alle Prozesse sind unmarkiert.

(1) Suche einen unmarkierten Prozeß k, bei dem für alle Betriebsmittel s gilt $A_S \geq C_{KS}$ .

(2) Falls ein solcher Prozeß existiert, markiere ihn und bilde $A_S := A_S + B_{KS}$ .

(3) Gibt es keinen solchen Prozeß, STOP. Ansonsten durchlaufe erneut (1) und (2).

In Pseudocode läßt sich dies wie folgt formulieren:

```
FOR k:=1 TO N DO unmarkProcess(k) END; d:=0;        (*Step 0*)
REPEAT
  k:= satisfiedProcess();                           (*Step 1*)
  IF k#0 THEN  d:=d+1; markProcess(k);              (*Step 2*)
      FOR s:=1 TO M DO A[s]:= A[s]+B[k,s] END;
  END;
UNTIL k=0;                                          (*Step 3*)
IF d<N THEN Error('Deadlock exists!') END;
```

Geht der Algorithmus ohne Fehlermeldung zu Ende, so ist nur festgestellt, daß keine ausweglose Situation entstehen muß; wie sich die Situation tatsächlich nach dem Test weiterentwickeln wird, ist nicht gesagt. Erhalten wir dagegen eine Fehlermeldung, so läßt sich zeigen, daß tatsächlich eine Verklemmung vorhanden ist, egal, in welcher kombinatorischen Reihenfolge wir die Prozesse markiert haben. Der Grund für die Unabhängigkeit dieser Tatsache von der Markierungsreihenfolge ist das stetige Anwachsen der Betriebsmittel beim Markieren.

**Beispiel** *Verklemmungen*

Seien $(B_{ij}) =$

| | tapes A | plotters B | printers C | CD ROM D |
|---|---|---|---|---|
| $P_1$ | 3 | 0 | 1 | 1 |
| $P_2$ | 0 | 1 | 0 | 0 |
| $P_3$ | 1 | 1 | 1 | 0 |
| $P_4$ | 1 | 1 | 0 | 1 |
| $P_5$ | 0 | 0 | 0 | 0 |

*bestehende Belegung*

und $(C_{ij}) =$

| | tapes A | plotters B | printers C | CD ROM D |
|---|---|---|---|---|
| $P_1$ | 1 | 1 | 0 | 0 |
| $P_2$ | 0 | 1 | 1 | 2 |
| $P_3$ | 3 | 1 | 0 | 0 |
| $P_4$ | 0 | 0 | 1 | 0 |
| $P_5$ | 2 | 1 | 1 | 0 |

*zusätzlich gewünschte Belegung*

wobei E = (6 3 4 2)

Die Gesamtbelegung ist (5 3 2 2), so daß also noch **A**=(1 0 2 0) übrigbleibt. Von allen Prozessen kann also nur $P_4$ zusätzlich (0 0 1 0) belegen, seine Arbeit vollständig durchführen und seine gesamten Betriebsmittel (1 1 1 1) zurückgeben. Damit erhalten wir **A**=(2 1 2 1). Nun können $P_1$ und $P_5$ die Arbeit beenden, so daß wir **A**=(5 1 3 2) erhalten, und, zuletzt, wird mit $P_2$ und $P_3$ **A**=(6 3 4 2). Man beachte, daß die Reihenfolge $P_4$, $P_1$, $P_5$, $P_2$, $P_3$ nicht zwangsläufig ist; es hätte auch $P_4$, $P_5$, $P_1$, $P_2$, $P_3$ sein können.

Wie beseitigen wir nun eine aufgetretene Verklemmung? Aus dem obigen Algorithmus ist leicht ersichtlich, welche Betriebsmittel und Prozesse verklemmt sind. Nun können wir

- *Prozesse abbrechen*

  Die einfachste Methode besteht darin, entweder einen der verklemmten Prozesse oder einen unbeteiligten Prozeß, der aber das benötigte Betriebsmittel hat, einfach abzubrechen und zu hoffen, daß sich alles selbst wieder arrangiert. Welche Prozesse sinnvoll abgebrochen werden können, muß man selbst ermitteln. Einen Hinweis darauf gibt der obige Algorithmus, der mit der Folge der markierten Prozesse auch eine Schedulingfolge liefert, die verklemmungsfrei die Arbeit beendet.

- *Prozesse zurücksetzen*

  In Datenbanksystemen gibt es meist aus Sicherheitsgründen regelmäßige Zeitpunkte, an denen der gesamte Systemzustand abgespeichert wird. Tritt nun eine Verklemmung auf, so kann man einen oder mehrere Prozesse, anstatt sie brutal abzubrechen, auf einen Zustand zurücksetzen, in dem sie die Betriebsmittel noch nicht belegt hatten, und die Prozeßausführung so lange verzögern, bis die benötigten Betriebsmittel von den verklemmten Prozessen wieder freigegeben werden.

- *Betriebsmittel entziehen*

  In manchen Fällen ist es möglich, einem Prozeß das von anderen Prozessen benötigte Betriebsmittel zu entziehen und es zunächst den anderen Prozessen zur Verfügung zu stellen. Sobald es wieder frei wird, kann sich dann auch dieser Prozeß wieder darum bewerben und an der abgebrochenen Programmstelle weitermachen.

  Diese Strategie ist nicht immer möglich, wie z. B. beim Beschreiben einer Datei, oder führt zu manuellem Mehraufwand, beispielsweise bei einem Drukker, dessen bisheriger Papierausdruck per Hand weggeräumt und wieder im richtigen Augenblick hingelegt werden muß.

Die Nachteile dieser Methoden sind klar: Brechen wir Prozesse einfach ab, so geht alle weitere Arbeit verloren. Setzen wir die Prozesse zurück, so geht alle bisher geleistete Arbeit seit dem Sicherungszeitpunkt verloren. Außerdem gibt es dabei viele Zusatzprobleme: Sowohl der abgebrochene als auch der zurückgesetzte Prozeß können Daten geschrieben oder Nachrichten verschickt haben, die nicht zurückgenommen werden und unter Umständen zu inkonsistenten Dateien führen können. Die Auswahl des entsprechenden Prozesses sollte sich deshalb nach dem Kriterium des geringsten angerichteten Schadens richten.

### *Verklemmungen vermeiden*

Eine der einfachsten Strategien gegen Verklemmungen besteht darin, beim Auftreten eines Fehlers („Betriebsmittel bereits belegt“) dies dem Benutzer anzuzeigen und den Benutzer selbst die richtige Reaktion darauf vornehmen zu lassen. Das ist meistens bei Einbenutzersystemen (*Personal Computer* PC) der Fall, in denen der Benutzer alles überschauen kann, genügend Zugriffsrechte besitzt und so Verklemmungen vermeiden kann.

In Mehrbenutzersystemen ist dies aber eine unbrauchbare Strategie. Hier werden ausgefeiltere Techniken benötigt. Eine Idee besteht darin, zwischen **unsicheren**, **verklemmungsbedrohten Zuständen** und **sicheren Zuständen** zu unterscheiden und immer nur solche Belegungen zuzulassen, die das System in sicheren Zuständen lassen. Was sind unsichere Zustände? Betrachten wir *sichere* Zustände als solche, bei denen es immer einen Weg (eine Schedulingstrategie) für die vorhandenen Jobs gibt, um alle normal zu beenden, und *unsicherere* als solche, die nicht zwangsläufig zu einer Verklemmung führen müssen, aber können. Dann haben wir bereits einen Algorithmus kennengelernt, um dies zu testen: unseren Erkennungsalgorithmus für Verklemmungen.

Wir betrachten dazu die Betriebsmittelanforderungen der Prozesse im Algorithmus als Maximalforderungen, die alle auf einmal auftreten können, aber nicht müssen. Die Diagnose „Verklemmung“ des Testalgorithmus bedeutet nun, daß eine Verklemmung auftreten kann, aber nicht muß – es könnte ja sein, daß die restlichen Betriebsmittel von den anderen Prozessen einzeln nacheinander und nicht gleichzeitig gefordert und zurückgegeben werden. Im Unterschied zur Bele-

gung im vorigen Abschnitt, die auf einmal realisiert und zurückgegeben wird, muß hier keine Verklemmung auftreten. Immerhin ist damit klar: Der Zustand ist verklemmungsbedroht und damit abzulehnen. Beispielsweise kann $P_2$ im obigen Beispiel einen Drucker zusätzlich bekommen, ohne daß das System verklemmungsbedroht wäre und damit den sicheren Zustand verläßt. Geben wir aber $P_5$ den letzten Drucker, so kann $P_4$ nicht mehr vollständig abgearbeitet werden: Das System ist verklemmungsbedroht und damit in einen unsicheren Zustand übergegangen. Die letzte der beschriebenen Zuteilungen sollte also vermieden werden. Dabei ist das System nicht verklemmt, sondern nur bedroht: Gibt z. B. $P_1$ seinen Drucker vor dem Ende wieder zurück, so droht keine Gefahr mehr.

Der Testalgorithmus wurde zuerst von Dijkstra (1965) angegeben und wird als „**Banker-Algorithmus**“ bezeichnet, da er das Verhalten eines Bankiers in einer Kleinstadt widerspiegelt, der mit begrenzten Betriebsmitteln (Kapital) versucht, die Kreditwünsche seiner Kunden zu befriedigen. Seine Strategie besteht darin, bei jedem Kreditwunsch eines Kunden zu prüfen, ob dann noch eine Reihenfolge besteht, um die Wünsche eines anderen Kunden im vollen, maximalen Kreditrahmen zu berücksichtigen und mit der anschließenden Rückzahlung die weiteren Wünsche abzudecken.

Die Anwendung dieses Algorithmus, um Verklemmungen zu vermeiden, bereitet allerdings praktische Probleme:

- Bei den meisten Anwendungen ist die Zahl der maximal nötigen Betriebsmittel unbekannt.
- Die Anzahl der Prozesse im System ist dynamisch und wechselt ständig.
- Die Anzahl der verfügbaren Betriebsmittel ist ebenfalls veränderlich.
- Der Algorithmus ist laufzeit- und speicherintensiv.

Aus diesen Gründen wird der bekannte Algorithmus in der Praxis kaum eingesetzt.

### *Verklemmungen unmöglich machen*

Die Grundidee bei dieser Strategie, Verklemmungen zu behandeln, besteht darin, eine der vier auf Seite 78 genannten Bedingungen, die hinreichend und notwendig für eine Verklemmung sind, unmöglich zu machen. Dies läßt sich bei jeder Bedingung unterschiedlich durchführen.

(1) *Mutual Exclusion*
Die Konkurrenz für ein Betriebsmittel läßt sich abschaffen, indem man es beispielsweise einem einzigen, speziell dafür zuständigen Prozeß übergibt und alle Anfragen für dieses Betriebsmittel als Dienstleistung an den Prozeß umwandelt.

Ein Beispiel dafür ist der Drucker: Ein spezieller Prozeß, der **Druckerdämon** (*printer demon*), erhält dauerhaft den Drucker; alle anderen Prozesse übergeben die auszudruckenden Daten diesem Prozeß in eine Warteschlange.

Der Druckerdämon arbeitet stellvertretend für die ihn beauftragenden Prozesse die Warteschlange ab (*spooling*), so lange ein Auftrag vorliegt, und legt sich dann bei leerer Warteschlange schlafen. Damit wird zunächst jede Verklemmung vermieden.

Diese Strategie ist nicht unproblematisch. Zum einen ist dies nicht für alle Betriebsmittel möglich (z. B. für Semaphore, die ja gerade für den wechselseitigen Aufruf da sind), zum anderen wird das Problem nur vorverlagert. In unserem Beispiel geht der wechselseitige Ausschluß vom Drucker auf den Druckerpuffer im Massenspeicher über, der als endlicher Platz ebenfalls ein Betriebsmittel darstellt. Haben wir zwei Prozesse, deren Pufferbedarf jeweils die Hälfte des verfügbaren Platzes überschreitet, so stellt sich hier wieder das gleiche Problem wie vorher. Jeder Prozeß schreibt seine Daten in den Puffer, bis kein Platz mehr da ist. Da der Auftrag erst nach dem vollständigen Übermitteln der Daten in die Warteschlange eingehängt wird, ist die Auftragsschlange leer und beide Prozesse warten, bis wieder Platz verfügbar ist, um die Übermittlung abzuschließen – sie bleiben verklemmt. Aus diesem Grunde werden *spooling*-Systeme auch manchmal so programmiert, daß nur die Originaldatei zum Drucken verwendet wird und nicht vorher eine Kopie davon gemacht wird.

Allgemein ist aber der Übergang von der direkten Belegung auf ein Stellvertreter- und Auftragssystem (*client-server*) üblich.

(2) *Hold And Wait*

Können wir verhindern, daß die Prozesse zusätzliche Ressourcen anfordern, so verhindern wir Verklemmungen. Eine Möglichkeit dafür besteht darin, nur ein einziges Betriebsmittel pro Prozeß zuzulassen. Dies ist aber inakzeptabel: Ein Prozeß, der Daten von einem Band liest und sie ausdruckt, wäre damit verboten.

Eine andere Möglichkeit sieht vor, immer alle Betriebsmittel, die für den Job nötig sind, am Anfang des Jobs auf einmal anzufordern. Dies erfordert allerdings zusätzlichen Programmieraufwand, da dies anfangs weder dem Job noch dem Betriebssystem, sondern nur dem Programmautor bekannt sein kann. Der Nachteil dieser Methode besteht darin, daß immer alle benötigten Betriebsmittel für die gesamte Laufzeit des Jobs belegt sind und für keine anderen Jobs mehr zur Verfügung stehen. Ein Programm, das für fünf Stunden rechnet und dann einen kurzen Ausdruck macht, verhindert damit fünf Stunden lang jeglichen Ausdruck – ein inakzeptables Verhalten in einer Mehrbenutzerumgebung und außerdem eine schlechte Betriebsmittelauslastung.

Eine interessante Variante davon sieht vor, bei jeder zusätzlichen Betriebsmittelbelegung zunächst einmal alle bereits belegten Betriebsmittel zurückzugeben und dann in einer Anfrage alle für die nächste Phase benötigten Betriebsmittel anzufordern. Dies erhöht zwar den Belegungsaufwand, verhindert aber eine Verklemmung.

(3) *No Preemption*

Ein vorzeitiger Entzug eines Betriebsmittels ist, wie bereits angedeutet, nicht einfach so möglich, da bereits Veränderungen durchgeführt sein können, beispielsweise beim Schreiben von Daten oder beim Ausdrucken von Papier. Ein vorzeitiger Entzug der Betriebsmittel etwa durch eine Zeitüberwachung muß deshalb im Programm explizit mitberücksichtigt werden und kann nicht automatisch transparent für das Programm geschehen.

(4) *Circular Wait*

Eine Möglichkeit, die Zyklen zu durchbrechen, besteht darin, bereits bei der Anforderung den ersten Schritt zu einem Zyklus zu vermeiden und eine Ordnungsrelation für die Betriebsmittelanforderung zu definieren (*Linearisierung* der Betriebsmittelanforderung).

- Dazu werden alle Betriebsmittel durchnumeriert und einem Prozeß nur gestattet, zusätzliche Betriebsmittel einer höheren Nummer zu belegen als er bereits hat. Dadurch kann kein Prozeß auf ein früher durch einen anderen Prozeß bereits belegtes Betriebsmittel mit einer kleineren Nummer warten – es ist prinzipiell nicht verfügbar.

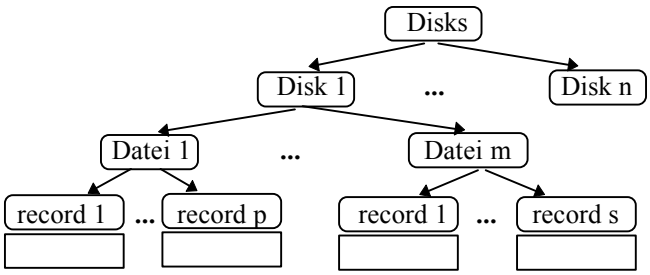
  Mit dieser Annahme betrachten wir nun einen Zyklus. Schreiben wir in unserem Betriebsmittelgraphen in jedes Kästchen die Nummer, so können wir die in einem Zyklus durchlaufenen Betriebsmittelnummern notieren. Bei der obigen Einschränkung ist es nicht möglich, von den gehaltenen Betriebsmitteln zu den angeforderten Mitteln immer nur höhere Nummern zu erreichen – irgendwann schließt sich der Kreis, und dann folgt die kleinste Nummer als Anforderung auf eine größere, die belegt wurde. Dies widerspricht aber der Voraussetzung. Also kann sich kein Zyklus ausbilden.

  Der Nachteil der Linearisierungsmethode liegt darin, daß so leicht keine Ordnung gefunden werden kann, die für alle Anwendungen brauchbar ist. Ist eine bestimmte gewünschte Anforderungsreihenfolge nicht möglich, so muß der Prozeß alle benötigten Ressourcen statt dessen auf einmal anfordern, was wieder die Auslastung der Ressourcen verschlechtert.

- Eine weitere Möglichkeit bietet eine Betriebsmittelzuteilung, bei der die Betriebsmittel in einer Hierarchie angeordnet sind und jeweils bestimmten Verwaltungsprozessen (Druckerdämon usw.) zugeordnet werden. Diese Prozesse erfüllen die Benutzeranforderungen und achten dabei auf die Verklemmungsfreiheit, wobei jeder Benutzerauftrag zeitlich begrenzt wird.

  Die Gesamtheit der Betriebsmittel (z. B. alle Dateien) kann aufgeteilt und ihre Verwaltung Unterprozessen übertragen werden. Da dies auch für den Unterprozeß gilt, entsteht so eine baumartige Hierarchie der Belegung, die verklemmungsfrei ist.

**Beispiel** *Dateiverwaltung*




Dabei wird jeder *record* nur von einem einzigen Prozeß alloziert. Die Aufträge (schreiben, lesen) werden von der Wurzel an die Unterprozesse weitergeleitet. Die Baumstruktur garantiert, daß die Reihenfolge der Bearbeitungen auch bei den Unteraufträgen überall gleich ist. Dadurch bleiben die Zustände der *records* konsistent.

### 2.3.9 Aufgaben

**Aufgabe 2.3-6 (Begriffe)**

a) Was ist der Unterschied zwischen Blockieren, Verklemmen und Verhungern von Prozessen?
b) Worin liegt der Unterschied zwischen aktivem (*busy wait*) und passivem Warten?
c) Geben Sie ein praktisches Beispiel für ein Verklemmung an.

**Aufgabe 2.3-7 (Verklemmung/Blockierung)**

Ein Student $S_1$ hat ein Buch A in der Bibliothek ausgeliehen. In Buch A findet er einen Literaturhinweis auf Buch B. Deshalb möchte er auch Buch B ausleihen. Dies ist momentan von Student $S_2$ ausgeliehen, der wiederum in Buch B einen Verweis auf Buch A findet. Also versucht er, Buch A auszuleihen. Stellt diese Situation eine Verklemmung, eine Blockierung oder keines von beiden dar? Bitte begründen Sie Ihre Antwort.

**Aufgabe 2.3-8 (Monitore)**

a) Implementieren Sie das Leser/Schreiber-Problem als Monitorlösung.
b) Was sind die Vor- und Nachteile einer Monitorlösung?

**Aufgabe 2.3-9 (Banker-Algorithmus)**

a) Welche der beiden Reihenfolgen wären im Beispiel auf Seite 82 für den Banker-Algorithmus für die fünf Prozesse auch möglich?

b) Angenommen, Prozeß $P_1$ bekommt ein Bandlaufwerk zusätzlich zugestanden. Ist das System dann verklemmungsbedroht?

**Aufgabe 2.3-10 (Sichere Systeme)**

a) Ein Computersystem habe sechs Laufwerke und *n* Prozesse, wobei ein Prozeß jeweils zwei Laufwerke benötigt. Wie groß darf *n* sein für ein sicheres System?
b) Für große Werte für *m* Betriebsmittel und *n* Prozesse ist die Anzahl der Operationen, um einen Zustand als „sicher" zu klassifizieren, proportional zu $m^a n^b$. Wie groß sind a und b?

**Aufgabe 2.3-11 (Verklemmungen unmöglich machen)**

In einem Transaktionssystem einer Bank gibt es für jede der vielen parallel stattfindenden Überweisungen von Konto S nach Konto E einen eigenen Prozeß. Die Lese-/Schreiboperationen pro Konto müssen deshalb vor parallelem Zugriff durch einen anderen Prozeß geschützt werden. Wird dazu ein Semaphor pro Konto verwendet, so kann es bei gleichzeitigen Rücküberweisungen leicht zu Verklemmungen kommen, da zuerst S und dann E von einem Prozeß und zuerst E und dann S vom anderen Prozeß belegt werden.

Entwerfen Sie ein Zugriffsschema, um eine Verklemmung unmöglich zu machen. Beachten Sie bitte, daß Lösungen, die zuerst das eine Konto sperren, es verändern sowie entsperren und dann das andere Konto belegen und verändern, gefährlich sind, da bei Computerstörungen der Überweisungsbetrag verschwinden kann. Diese Art von Lösungen scheidet also aus.

## 2.4 Prozeßkommunikation

Ein wichtiges Mittel, um in einem Betriebssystem die Aktionen mehrerer Prozesse zu koordinieren und ein gemeinsames Arbeitsziel zu erreichen, ist der Nachrichtenaustausch. In den klassischen Betriebssystemen wurde dies sehr vernachlässigt; erst in UNIX wurden Programme als Bausteine angesehen, deren koordiniertes Zusammenwirken ein neues Programm ergeben kann. Die Mittel dafür waren in älteren UNIX-Versionen aber sehr beschränkt. Erst der Zwang, auch über Rechnergrenzen hinweg in Netzen die Arbeit zu koordinieren, führte dazu, auch auf einem einzelnen Rechner die Interprozeßkommunikation als Betriebssystemdienst zu ermöglichen.

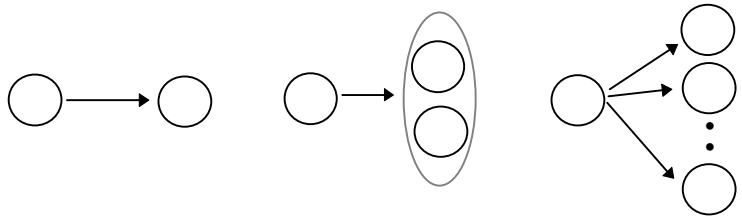
In diesem Abschnitt sind deshalb die grundsätzlichen Überlegungen präsentiert, wie sie sowohl für Mono- als auch für Multiprozessorsysteme und Netzwerke zutreffen. Die stärker speziellen für Netzwerke zutreffenden Aspekte für die Betriebssysteme sind gesondert in Kap. 6 beschrieben.

Der Nachrichtenaustausch zwischen Prozessen folgt dabei bestimmten Überlegungen.

### 2.4.1 Kommunikation mit Nachrichten

Bei dem Nachrichtenaustausch unterscheidet man drei verschiedene Arten der Verbindung: die Punkt-zu-Punkt-(**unicast**)-Verbindung von einem einzelnen Prozeß zu einem anderen, die **multicast-Verbindung** von einem Prozeß zu mehreren anderen und der Rundspruch (**broadcast-Verbindung**) von einem Prozeß zu allen anderen. In Abb. 2.30 sind diese Verbindungsarten visualisiert.

Dabei läßt sich jede Verbindungsart durch eine andere implementieren. Beispielsweise kann man einen *broadcast* und *multicast* durch mehrere *unicast*-Verbindungen realisieren oder durch eine Verbindung, bei der in der Adresse eine Liste aller Empfänger mitgeschickt wird und diese vom Empfänger benutzt wird, um die nächste Verbindung aufzubauen. Umgekehrt kann man eine Nachricht an alle Empfänger schicken, in der als Adressat nur ein einziger Empfänger angegeben ist; die anderen Empfänger sehen dies und ignorieren daraufhin die Nachricht.



(a) unicast (b) multicast (c) broadcast

**Abb. 2.30** *Verbindungsarten*

Die Kommunikation kann man dabei verschieden implementieren. Traditionellerweise wird eine Kommunikation, ähnlich einer Telefonverbindung, durch einen Verbindungsaufbau charakterisiert (**verbindungsorientierte Kommunikation**) wie dies in Abb. 2.31 gezeigt ist.

| | |
|---|---|
| `openConnection (Adresse)` | Feststellen, ob der Empfänger existiert und bereit ist: Aufbau der Verbindung; |
| `send(Message)/ receive(Message)` | Nachrichtenaustausch; |
| `closeConnection` | Leeren der Nachrichtenpuffer, Beenden der Verbindung. |

**Abb. 2.31** *Ablauf einer verbindungsorientierten Kommunikation*

Der Aufwand für den Aufbau und den Unterhalt einer derartigen Verbindung (*Kanal*) ist allerdings ziemlich groß. Aus diesem Grund wurde eine andere Art von Nachrichtenaustausch modern: die **verbindungslose Kommunikation**. Im Unterschied zu dem **synchronen** Verbindungsaufbau, bei dem Sender und Empfänger ihre Bereitschaft für die Verbindung bekunden müssen, werden hierbei die Nachrichten **asynchron** verschickt:

```
send (Adresse, Message) / receive(Adresse, Message)
```

Da nun nicht mehr sichergestellt ist, daß die Nachricht auch tatsächlich ankommt (Übermittlung und Empfänger können gestört sein), wird normalerweise jede Nachricht quittiert. Trifft innerhalb einer Zeitspanne keine Quittungsnachricht ein, so sendet der Sender seine Nachricht erneut sooft, bis er eine Quittung erhält. Erst dann sendet er eine neue Nachricht. Ist die Quittungsnachricht verloren gegangen, so erhält der Empfänger mehrere Kopien der gleichen Nachricht. Um neue von alten Nachrichten unterscheiden zu können, sind die Nachrichten in diesem Fall mit einer fortlaufenden Nummer versehen. Eine Nachricht kann somit aus folgender Grundstruktur bestehen:

```
TYPE tMessage= RECORD
                  EmpfängerAdresse:   STRING;
                  AbsenderAdresse:    STRING;
                  NachrichtenTyp:     tMsgTyp;
                  SequenzNummer INTEGER;
                  Laenge:             CARDINAL;
                  Data:               POINTER TO tBlock;
               END;
```

Dies wird auch als *Nachrichtenkopf* (*message header*) bezeichnet und entspricht einem Umschlag, ähnlich wie bei einem Brief. Dabei ist nur ein Minimum an Angaben gezeigt; reale Nachrichten zwischen Prozessen haben meist noch mehr Einträge. Man beachte, daß zwischen Sender und Empfänger Übereinstimmung herrschen muß, wie die Nachrichtenfelder zu interpretieren sind; sowohl der Datentyp (wie z. B. bei `tMsgTyp`) als auch die Reihenfolge der Felder und die Interpretation der Zahlen (kommt bei einem `INTEGER` zuerst das höchstwertige (*high order*) Byte oder zuerst das niedrigstwertige?). Aus diesem Grund werden beim Datenaustausch zwischen Rechnern verschiedenen Typs zuerst die Datenbeschreibungen und dann die Daten selbst geschickt, vgl. Abschn. 6.2.3.

Die interne Verwaltung von Nachrichten verschiedener Länge ist dabei nicht so einfach. Zwar ist der Nachrichtenkopf immer gleich lang, aber die unterschiedliche Länge der Daten gestattet keine feste Längenangabe im `Data` Feld. Beim Empfänger muß deshalb zunächst der Nachrichtenkopf gelesen und dann erst der Speicherplatz für den Datenblock alloziert werden.

Das Senden bzw. Empfangen kann dabei in verschiedener Weise vorgenommen werden. Nach dem Senden wird entweder der Sender so lange verzögert, bis eine Rückmeldung erfolgt ist (blockierendes oder **synchrones** Senden), oder das Sy-

stem puffert die Nachricht (als Kopie oder direkt), kümmert sich um die korrekte Ablieferung und läßt den Sender weitermachen (nicht-blockierendes oder **asynchrones** Senden). Allerdings muß auch beim nicht-blockierenden Senden eine Fehlermeldung zurückgegeben werden, wenn der systeminterne Puffer überläuft, oder aber der Sender muß in diesem Fall verzögert werden.

Auch für das Empfangen und Lesen der Nachrichten gibt es blockierende und nicht-blockierende Versionen: Entweder wird der Empfänger verzögert, bis eine Nachricht vorliegt, oder aber der Empfänger erhält beim nicht-blockierenden Lesen eine entsprechende Rückmeldung, wenn keine Nachricht vorliegt.

### *Adressierung*

Die Adressierung der Empfänger ist sehr unterschiedlich. Bei einer Punkt-zu-Punkt-Verbindung besteht die Adresse zunächst aus einer Prozeßnummer. Ist der Prozeß auf einem anderen Rechner, so kommt noch die Nummer oder der Name des anderen Rechners hinzu, eventuell auch die Firma, die Region und das Land, die man in der Zeichenkette des Namens durch Punkte voneinander trennen kann:

Adresse = Prozeß-ID.RechnerName.Firma.Land
z.B. 5024.hera.uni-oldenburg.de

Für eine *multicast*-Verbindung benötigt man dann nur eine Liste aller Prozesse bzw. Rechner, die mit der Nachricht gleich mitgeschickt werden kann.

Allerdings ist eine solche Art von Adressierung sehr ungünstig, wenn die Kommunikationspartner sich nicht direkt kennen und ihre Prozeß-ID wissen. Möchte beispielsweise ein Formatierungsprogramm einem Drucker die fertigen Daten zum Ausdrucken schicken, so kennt es den PID des Druckprozesses nicht, aber es weiß, daß ein Drucker existiert. Es ist deshalb besser, einen feststehenden logischen Empfängernamen „Drucker“ im System zu definieren, dessen Zuordnung zu einem aktuellen Prozeß-ID je nach Systemzustand wechselt und in einer Zuordnungstabelle des Betriebssystemkerns beim Empfänger festgehalten wird. Alle Aufrufe zu „Drucker“ werden damit transparent für den sendenden Prozeß an die richtige Adresse geleitet. Dieses Prinzip der logischen und nicht physikalischen Namensgebung kann man auch auf die allgemeine Kommunikation über Rechnernetze ausdehnen. Da die komplette Information über alle Prozesse, Rechner und Institutionen sinnvollerweise nicht auf jedem Rechner eines Netzes gehalten und aktualisiert werden sollte, gibt es für die Adressenzuordnung (Adressauflösung, *address resolution*) bei weiten Entfernungen besondere Rechner (**name server**), die dies durchführen, siehe Abschn. 6.2.

Das bisher Gesagte gilt natürlich nicht nur für die *unicast*-Verbindung, sondern auch für die *multicast*-Verbindung, bei der mehrere Prozesse und Rechner in Gruppen zusammengefaßt und unter ihrem Gruppennamen adressiert werden können.

Eine besondere Variante stellt die bedingte Adressierung durch eine **Prädikatsadresse** dar. Hier gibt man ein logisches Prädikat an, das von logischen Bedingungen abhängt, beispielsweise vom Maschinen- oder CPU-Typ, freien RAM-Speicher, vorhandenen Peripheriegeräten usw. Der Empfänger evaluiert dies wie eine IF-Abfrage; ist das Ergebnis WAHR, so fühlt er sich angesprochen; ist es FALSCH, so kommt er als Empfänger nicht in Frage, und die Nachricht wird ignoriert. Dies ermöglicht es besser, dynamisch Arbeit zu verteilen und freie Peripherie zu nutzen.

### *Mailboxen*

Beim asynchronen Nachrichtenaustausch müssen die Nachrichten, da der Empfänger sie nicht sofort liest, in einem Puffer zwischengelagert werden. Diese Nachrichtenpuffer können auch mit einem Namen versehen werden, so daß man den Prozeß nicht mehr zu kennen braucht, dem man die Nachricht schickt. Auch die Systemverwaltung ändert sich; anstelle der Zuordnung von logischen Prozeßnamen zu physikalischen Prozeß-IDs wird eine Zuordnung von logischen Puffernamen zu physikalischen Pufferadressen durchgeführt.

Ein solcher Nachrichtenpuffer (**Mailbox**) kann auch als Warteschlange strukturiert werden, in die Nachrichten eingehängt und vom Empfänger ausgelesen werden. Sehen wir noch eine weitere Warteschlange für die Empfangsprozesse einer Prozeßgruppe vor, falls keine Nachrichten verfügbar sind, sowie eine Warteschlange für die Sendeprozesse, falls die Nachrichtenschlange voll ist, so erhalten wir eine Mailbox mit der Struktur

```
TYPE Mailbox = RECORD
                  SenderQueue :   tList;
                  EmpfängerQueue :    tList;
                  MsgQueue :          tList;
                  MsgZahl :           INTEGER;
               END
```

Der Zugriff auf die Warteschlangen (`einhängen(Msg)` und `aushängen(Msg)`) muß durch Semaphoroperationen geschützt werden, so daß pro Mailbox ein Semaphor vorgesehen werden muß. Die Variable „`MsgZahl`“ dient dabei als Zähler zur Flußkontrolle innerhalb der kritischen Abschnitte `einhängen(Msg)` und `aushängen(Msg)`. Ist mit `MsgZahl=N` die maximale Kapazität der Mailbox erreicht, so wird der Sendeprozeß eingehängt und schlafen gelegt; umgekehrt ist dies bei `msgZahl=0` für den Empfänger der Fall.

Wird also bei `MsgZahl=N` gelesen, so muß der Empfänger noch zusätzlich ein `wakeup(SenderQueue)` durchführen, um evtl. vorhandene Senderprozesse aufzuwecken. Entsprechend muß bei `MsgZahl=0` der Sendeprozeß eventuell wartende Empfänger aufwecken.

Für die Koordination der Sende- und Empfangsprozesse ist es sinnvoll, die Prozeduren `einhängen(Msg)` und `aushängen(Msg)` ebenfalls in die Mailboxdekla-

ration aufzunehmen und den Zugang zur Mailbox nur über diese kontrollierten Operationen zu gestatten. Die gesamte Datenstruktur wird damit zu einem abstrakten Datentyp. In objektorientierten Sprachen gestattet die PUBLIC-Deklaration ein Verbergen der Datenstrukturen und Kapselung, so daß wir eine Datenstruktur erhalten, die ähnlich einem `MONITOR`-Konstrukt ist.

### 2.4.2 Beispiel UNIX: Interprozeßkommunikation mit *pipes*

Die Interprozeßkommunikation benutzt seit den ältesten Versionen von UNIX einen speziellen, gepufferten Kanal: eine **pipe**. Im einleitenden Beispiel von Abschn. 2 sahen wir, daß mehrere UNIX-Programme, in diesem Fall Prozesse, durch eine Anweisung

Programm1 | Programm2 | .. | Programm N

zusammenwirken können. Hier initiiert der senkrechte Strich | einen Übergabemechanismus der Daten von einem Programm zum nächsten, der jeweils durch eine solche *pipe* realisiert wird. Ein solcher Kommunikationskanal funktioniert folgendermaßen:

Mit dem Systemaufruf `pipe()` wird ein Nachrichtenkanal eröffnet, auf den mit den normalen Lese- und Schreiboperationen `read(fileId,buffer)` und `write(fileId, buffer)` zugegriffen werden kann. Für jede Kommunikationsverbindung eröffnet das Hauptprogramm (hier: die Benutzer-*shell*) eine eigene *pipe* und vererbt sie mit dem Prozeßkontext an die Kindsprozesse weiter; die Sende-Kindsprozesse benutzen nur den `fileId` zum Schreiben und die Empfangsprozesse nur den zum Lesen. Da die Dateikennungen `fileId` Nummern von internen Tabellen sind, die beim `fork()`-Aufruf dem Kindsprozeß vererbt werden, kann in UNIX durch *pipes* nur eine Interprozeßkommunikation (IPC) zwischen Eltern- und Kindsprozessen eingerichtet werden; sie gehören derselben **Prozeßgruppe** an. In Abb. 2.32 sind zwei Prozesse gezeigt, die durch eine anonyme *pipe*, die sie von dem Elternprozeß im Prozeßkontext geerbt haben, miteinander kommunizieren. Die *pipe* ist dabei in der Schnittmenge der Prozeßkontexte (graue Linien), dem noch gemeinsamen Teil des kopierten Prozeßkontextes, enthalten.

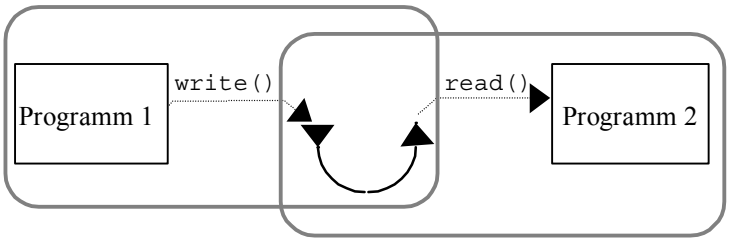



**Abb. 2.32** *Interprozeßkommunikation mit* pipes *in UNIX*

Eine *pipe* ist in UNIX nur unidirektional; für einen Nachrichtenaustausch zwischen zwei Prozessen werden immer zwei *pipes* benötigt.

Im Normalmodus blockiert sich der Sendeprozeß nur, wenn die *pipe* voll ist; der Leseprozeß nur, wenn sie leer ist. Im nicht-blockierenden Modus erhält der Leseprozeß eine Null zurück, wenn keine Nachricht vorlag; ansonsten die Anzahl der gelesenen Bytes, die in der Puffervariablen gespeichert wurden.

Eine IPC zwischen beliebigen Prozessen und über Rechnergrenzen hinweg ist erst bei neueren UNIX-Versionen über globale Namen möglich, die für eine *pipe* vergeben werden (**named pipes**), sowie über die sog. **Socket**-Konstrukte, die in Abschn. 6.2 näher erläutert werden.

### 2.4.3 Beispiel Windows NT: Interprozeßkommunikation mit *pipes*

Auch in Windows NT gibt es IPC mit *pipes*. Sie wird durch den Aufruf `CreatePipe()` eingerichtet. Danach kann man auch durch die normalen Lese- und Schreiboperationen `WriteFile()` und `ReadFile()` darauf zugreifen. Mit einem `CloseHandle()`-Aufruf wird sie abgeschlossen.

Da auch hier bei der namenslosen *anonymous pipe* der Zugriff auf diese Kanäle durch eine äußere Kennung nicht existiert, ist sie ebenfalls auf die Eltern/Kind-Prozeßgruppe beschränkt.

Zusätzlich zu möglichen Statusänderungen *blocking/non-blocking* gibt es auch in Windows NT (bidirektionale) *named pipes* sowie Socket-Konstrukte, die eine Interprozeßkommunikation über Rechnergrenzen hinweg ermöglichen.

### 2.4.4 Prozeßsynchronisation durch Kommunikation

Im vorigen Abschnitt sahen wir, daß Semaphore als Synchronisationsprimitive zu elementar sind und Monitore meist nicht verfügbar. Beides ist in einer Mono- und Multiprozessorumgebung sinnvoll, nicht aber in einem Rechnernetz. Wir benötigen deshalb Synchronisationsprimitive, die auch in verteilten Systemen vorliegen. Dies läßt sich vor allem durch Nachrichtenaustausch mit sehr kurzen Nachrichten (**Signale**) und durch Warten auf das Senden einer Nachricht erreichen. Genau genommen besteht ja das Empfangen einer Nachricht aus zwei Teilen: dem Warten, d. h. Synchronisieren mit der Nachricht, und dem Lesen der Nachricht. Bei Nachrichten der Länge null bleibt also noch die Synchronisation dabei übrig. Die Funktionen `send(Message)` und `receive(Message)` sind mit den Operationen `send(Signal)` und `waitFor(Signal)` für solche Nachrichten identisch. In beiden Fällen gehen wir davon aus, daß die übermittelten Nachrichten zwischengespeichert werden und vor dem Lesen nicht verloren gehen. Eine reine Synchronisation kann also leicht zur Kommunikation mit Nachrichten erweitert und umgekehrt eine allgemeine Kommunikation zur reinen Synchronisation benutzt werden. Betrachten wir zunächst die Synchronisation durch Signale.

### *Synchronisation durch Signale*

Einer der wichtigsten Gründe, um Nachrichten zu empfangen, ist das Auftreten von Ereignissen. Dies sind Alarme über auftretende Fehler im System (z. B. Datenfehler, unbekannte Speicheradressen etc.), Ausnahmebehandlungen (*exceptions,* z. B. Division durch Null) und Signale anderer Prozesse. Dabei kann der empfangende Prozeß entweder *synchron* auf das Ereignis warten, bis es eintrifft, oder aber nur die Verarbeitung der Ereignismeldung einrichten (initialisieren) und die Bearbeitung der *asynchron* eintreffenden Nachricht der angemeldeten Verarbeitungsprozedur überlassen.

Der zweite Fall ist besonders in Ausnahmebehandlungen üblich, die prozeßspezifisch behandelt werden müssen wie Division-durch-Null, Stacküberlauf, Verletzung von Feldgrenzen usw. Um der Vielzahl der unterschiedlichen Ausnahmebehandlungen gerecht zu werden, wird deshalb sinnvollerweise eine einzige Schnittstelle zum Betriebssystem definiert, die programmintern für jedes Modul extra initialisiert und aufgesetzt wird.

**Beispiel UNIX** *Signale*

In UNIX gibt es ein Signalsystem, mit dem die Existenz eines Ereignisses einem Prozeß mitgeteilt werden kann. Dazu sind folgende, in Abb. 2.33 übersichtsweise aufgelisteten Signale definiert (s. *signal.h*):

| | |
|---|---|
| SIGABRT | *abort process*: Aufforderung zum sofortigen Prozeßabbruch |
| SIGTERM | *terminate*: geordnetes Beenden des Prozesses erwünscht |
| SIGQUIT | Aufforderung zum Prozeßabbruch mit Speicherabzug (*core dump*) |
| SIG FPE | *floating point error* |
| SIGALRM | *alarm*-Signal: Zeituhr abgelaufen |
| SIGHUP | *hang up*: Telephonverbindung aufgelegt |
| SIGKILL | *kill*-Signal: bricht den Prozeß auf jeden Fall ab |
| SIGILL | *illegal instruction*: nichtexistenter Maschinenbefehl |
| SIGPIPE | Es existiert kein Empfänger für *pipe*-Daten. |
| SIGSEGV | *segmentation violation*: nicht verfügbare Speicheradresse |
| SIGINT | *interrupt*-Signal (CTRL C) |
| SIGUSR1 | anwendungsspezifisch |
| SIGUSR2 | anwendungsspezifisch |

**Abb. 2.33** *POSIX- Signale*

Standardmäßig gibt es in UNIX 16 Signale, was durch die frühere Wortbreite des Signalregisters im PCB von 16 Bit bedingt war. Der allgemeine Betriebssystemdienst `send(Signal)` ist in UNIX aus dem Senden des Signals SIGKILL entstanden, um einen Prozeß abzubrechen, und heißt deshalb immer noch `kill()`. Die Verwendung der obigen Signale ist zwar, wie an ihrer Namensgebung zu bemerken, für bestimmte Dinge gedacht, aber mit allen Signalen läßt sich eine selbst definierte Prozedur mittels des Systemaufrufs `sigaction()` verknüpfen, die beim Auftreten eines Signals im Prozeß (*software interrupt*) angesprungen wird, so daß sich sowohl beim Senden als auch beim Empfangen beliebige Signale zur Kommunikation verwenden lassen.

Bei Signalen unterscheidet man noch zusätzlich, ob nur auf ein bestimmtes aus einer Menge gewartet werden soll, oder aber darauf gewartet wird, daß alle Ereignisse einer Menge stattgefunden haben. Die verschiedenen Betriebssysteme haben dazu unterschiedliche Dienste, bei denen mit UND oder ODER Bedingungen für die Prozeßaktivierung durch Ereignisse bzw. Signale gesetzt werden können.

**Beispiel**

Die Ereignisse {MausDoppelClick, LinkeMausTaste, RechteMausTaste, Ascii-Taste, MenuAuswahl, FensterKontrolle} erfordern sehr unterschiedliche Reaktionen. Üblicherweise erwarten die interaktiven Programme, beispielsweise die Manager der Fenstersysteme, eines der Ereignisse als Eingabe. Dazu setzen sie einen Aufruf (Warten auf Multi-Event) ab, bei dem angegeben wird, auf welche Ereignisse sie warten wollen. Warten sie auf einen Mausclick ODER einen ASCII-Tastendruck, so wird eine entsprechende Maske gesetzt. Aber auch eine UND-Maske ist möglich, wenn z. B. auf die Tastenkombination SHIFT-MausDoppelClick reagiert werden soll.

Die Signale können aber auch dazu benutzt werden, eine Prozeßsynchronisation einzurichten. Beispielsweise läßt sich mit Hilfe von `send(Signal)` und `waitFor(Signal)` leicht die Semaphoroperation implementieren. In MODULA-2 ist dies auf einem Rechner durch die Typdefinition

```
TYPE       Semaphor  = POINTER to tSemaphor
           tSemaphor = RECORD
                          besetzt : BOOLEAN;
                          frei    : SIGNAL;
                       END;
```

und die Operationen

```
PROCEDURE P(VAR S:Semaphor);
BEGIN
    IF S^.besetzt THEN waitFor(S^.frei) END;
    S^.besetzt:= TRUE;
END P;
```

```
PROCEDURE V(VAR S:Semaphor);
BEGIN
    S^.besetzt:= FALSE;
    send(S^.frei)
END V;
```

möglich. Das gesamte Semaphormodul muß allerdings eine höhere Priorität (Interrupt!) als die übrigen Prozesse aufweisen, um die `P()`-und `V()`-Operation atomar werden zu lassen.

Dies läßt sich entweder in MODULA-2 durch eine Prioritätsangabe bei der Moduldeklaration erreichen oder in einer anderen Sprache durch einen Aufruf `setPrio(high)` jeweils direkt nach `BEGIN`, ergänzt durch `setPrio(low)` jeweils direkt vor `END`.

Die Semaphore müssen anfangs erzeugt und initialisiert werden, was durch einen Aufruf der Prozedur

```
PROCEDURE createSemaphor(VAR S:Semaphor);
BEGIN
      ALLOCATE(S,TSIZE(tSemaphor));
      S^.besetzt:=FALSE
      initSignal(S^.frei);
END createSemaphor.
```

geschehen kann.

Das damit errichtete Semaphorsystem hat allerdings den Nachteil, daß es nur zwischen Prozessen auf demselben Prozessor funktioniert, wo die Erhöhung der Priorität eine Unterbrechung ausschließen kann. Sowohl in Multiprozessorsystemen als auch in Mehrrechnersystemen müssen deshalb andere Mechanismen angewendet werden, um eine Synchronisation zu erreichen.

#### *Synchronisation durch atomic broadcast*

Ein wichtiger Aspekt der Synchronisierung durch Nachrichten ist die Frage, ob eine Nachricht überhaupt beim Empfänger angekommen ist oder nicht. Haben einige Empfänger sie erhalten, andere aber nicht, so ist es für den Sender schwierig, die Kommunikation richtig zu verwalten und mit seinen Nachrichten eine einheitliche, konsistente Datenbasis bei den Empfängern zu garantieren. Um diesen Zusatzaufwand zu verringern und gerade bei Transaktionen in verteilten Datenbanken den vollständigen Abschluß einer Transaktion zu gewährleisten, ist es deshalb sinnvoll, die *broadcast*-Nachrichtenübermittlung als *atomare Aktion* zu konzipieren. Ein **atomarer Rundspruch** (**atomic broadcast**) definiert sich durch folgende Forderungen (Cristian et al. 1985):

- Die Übertragungszeit der Nachrichten ist endlich.
- Entweder erhalten alle Empfänger die Nachrichten oder keiner.
- Die Reihenfolge der Nachrichten ist bei allen Empfängern gleich.

Die gleiche Reihenfolge der Nachrichten bei allen Empfängern unabhängig von bestehenden Kausalitäten („Totale Ordnung“) läßt sich als nicht-blockierender Grundmechanismus einer konsistenten Datenhaltung verwenden. Sind nämlich überall Reihenfolge und Inhalt der Nachrichten gleich, so resultieren bei gleichem Anfangszustand und gleichen Änderungen bei allen beteiligten Prozessen im Gesamtsystem die gleichen Zustände der gemeinsam geführten Daten, also der verteilt aktualisierten globalen Variablen und Dateien, s. Dal Cin, Brause, Lutz, Dilger, Risse (1987). In Abb. 2.34 ist dies visualisiert.

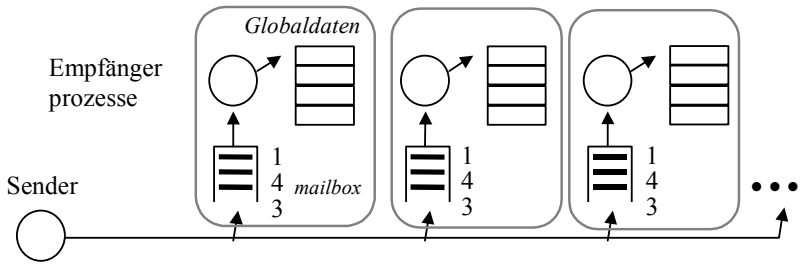



**Abb. 2.34** *Datenkonsistenz mit atomic broadcast-Nachrichten*

Man beachte, daß die Reihenfolge der Nachrichten 1, 4, 3 den Zustand der globalen Variablen festlegt, unabhängig von der Nummer der Nachricht, die ja von der Zählung des jeweiligen Senders abhängt. Ist der Sender auch Mitglied der Gruppe, so muß natürlich zur Datenkonsistenz innerhalb der Gesamtgruppe der Sender die Nachricht ebenfalls an sich senden und darf erst dann, beim Empfang der eigenen Nachricht, die globale Variable ändern. Tut er das vorher, so kann es sein, daß seine Nachricht bei den anderen erst nach einer bestimmten Änderungsmeldung ankommt, bei sich aber vorher und damit die Wirkung bei ihm anders ist als bei allen anderen Prozessoren: Eine Inkonsistenz ist da. Dies gilt beispielsweise für die Belegungswünsche verschiedener Prozesse für ein Betriebsmittel, die überall streng in der gleichen Reihenfolge abgearbeitet werden müssen, um überall den gleichen Zustand für die Belegungsvariable zu erzeugen.

Das obige Verfahren benötigt für die konsistente Reihenfolge der Nachrichten keine explizite globale Numerierung der Nachrichten. Es vermeidet damit Probleme beim Nachrichtenaustausch zwischen wechselnden lokalen Gruppen. Man beachte allerdings, daß damit nicht unbedingt die Reihenfolge kausal aufeinanderfolgender Nachrichten garantiert wird („Kausale Ordnung“). Diese kausale Se-

quentialisierung des Nachrichtenverkehrs läßt sich nur durch kausale Zusatzbedingungen erreichen.

Die Frage der Konsistenz von Daten durch atomaren broadcast oder multicast spielt allgemein bei verteilten Systemen eine wichtige Rolle. Man denke dabei z.B. an verteilte Prozesse (Reihenfolge der Aktivierung), verteilten Speicher (Wer schreibt welche Daten in welcher Reihenfolge in den Speicher?) und dezentrale Synchronität (Jeder Rechner hat seine eigene,lokale Zeit. Was bedeutet „gleichzeitig“?). Mehr zu diesem Thema ist beispielsweise im Buch von F. Mattern (1989) zu finden.

### *Synchronisation von Programmen*

Für das Senden und Empfangen von Signalen kann man auch eine ungepufferte Kommunikation verwenden. In diesem Fall wird das empfangende Programm so lange verzögert, bis das sendende Programm die Nachricht sendet (*synchrones bzw. blockierendes* Empfangen). Dies führt zu einer Synchronisierung zwischen Sender und Empfänger, die in manchen Prozeßsystemen gewollt ist und unter dem Namen **Rendez-vous-Konzept** bekannt ist. Die Programmiersprache ADA hat ein solches Konzept bereits im Sprachumfang enthalten und ermöglicht damit den Ablauf eines Programms aus kommunizierenden Prozessen auf allen Systemen, auf denen ein ADA-Compiler installiert ist. In Abb. 2.35 ist der Synchronisationsverlauf gezeigt.

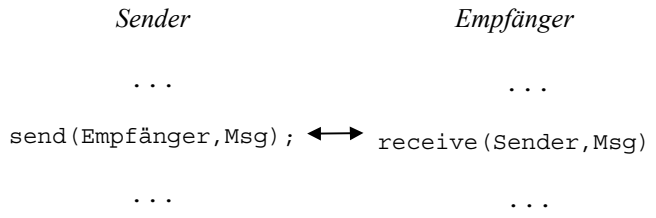



**Abb. 2.35** *Das Rendez-vous-Konzept*

Ein anderes wichtiges Konzept ist die parallele Programmausführung durch kommunizierende Prozesse. Viele Programme können einfacher, erweiterbar und wartungsfreundlicher gestaltet werden, wenn man den Sachverhalt als eine Aufgabe formuliert, die von vielen kleinen, unabhängigen Spezialeinheiten erledigt wird, die miteinander kommunizieren.

Zerteilen wir also ein Programm in kurze Codestücke, z. B. *threads*, so benötigen diese Programmstücke auch eine effiziente Kommunikation, um die gemeinsame Aufgabe des Gesamtprogramms zu erfüllen. Das Kommunikationsmodell innerhalb eines Programms sollte dabei so einfach und klar wie möglich für den Programmierer sein, um Fehler zu vermeiden und eine effiziente Programmierung zu ermöglichen.

Zu diesem Zweck konzipierte Hoare (1978) sein sprachliches Modell der kommunizierenden sequentiellen Prozesse **CSP**. In CSP gibt es die sprachlichen Konstrukte

| | | | |
|---|---|---|---|
| | `Empfänger!data1` | für die Prozedur | `send(Empfänger,data1)` |
| und | `Sender?data2` | für | `receive(Sender,data2)` |

Beide Kommunikationspartner, Sender und Empfänger, müssen zum Nachrichtenaustausch aufeinander warten. Dies entspricht dem Rendez-vous-Konzept, einer Kommunikation ohne Pufferung. Nach der Synchronisation wird die Datenzuordnung `data2:=data1` durchgeführt, wobei `data1` und `data2` vom gleichen Datentyp sein müssen, beispielsweise INTEGER oder REAL. `Empfänger` und `Sender` sind Namen, die innerhalb des Programms deklariert wurden und bekannt sind.

Die parallele Programmausführung wird mit Hilfe der folgenden Notation für eine Befehlsauswahl bewirkt:

$$\begin{aligned} & B_1 \rightarrow S_1 \\ [] \ & B_2 \rightarrow S_2 \\ & \ldots \\ [] \ & B_n \rightarrow S_n \end{aligned}$$

Alle Booleschen Bedingungen $B_1$, …, $B_n$ werden geprüft. Ist eine der Bedingungen $B_i$ erfüllt, so wird der entsprechende Befehl $S_i$ ausgeführt. Ist mehr als eine erfüllt, so wird eine davon zufällig ausgewählt. Eine Klammer der Form *[…] um diese alternativen Sequenzen bewirkt das wiederholte Durchlaufen der Befehlsauswahl so lange, bis keine der Bedingungen mehr erfüllt ist. Dann geht die Befehlsausführung auf das weitere Programm über.

Diese von Dijkstra erdachte Konstruktion der bewachten Befehle (*guarded commands*) wurde nur teilweise in die parallele Programmiersprache OCCAM (Inmos 1988) übernommen, die für die kommunizierenden Prozessoren (*Transputer*) der Fa. Inmos geschaffen wurde. Die kommunizierenden Leichtgewichtsprozesse können dabei nur aus wenigen Anweisungen einer Zeile bestehen. Beispielsweise läßt sich das *Erzeuger-Verbraucher* in OCCAM als ein Senden an einen Pufferprozeß darstellen, von dem der *Verbraucher* wiederum empfängt. Dies ist in Abb. 2.36 gezeigt.

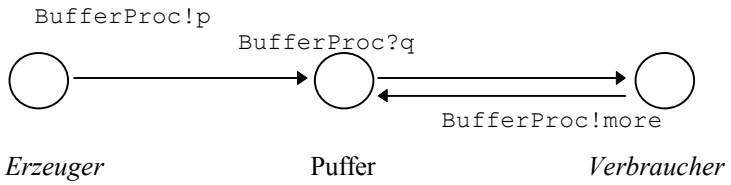



**Abb. 2.36** *Erzeuger-Verbraucher-Modell mit kommunizierenden Prozessen*

Der Puffer wird als Ringpuffer mit 10 Elementen gekapselt in dem Prozeß `BufferProc` mit dem Code

```
CHAN OF items producer, consumer :
INT in, out :
SEQ
    in :=0
    out:=0
    WHILE TRUE
       ALT
           IF (in<out+10)AND(producer?buffer(in REM 10))
               in:=in+1
           IF (out<in) AND (consumer?more)
               consumer!buffer(out REM 10)
               out:=out+1
```

Dabei ist der korrekte Index des Ringpuffers für die Ein- und Ausgabe jeweils mittels der Modulo-Operation REM (*remainder*) gegeben. Die Notation ALT gibt eine beliebige und SEQ eine sequentielle Reihenfolge an; nicht gezeigt wurde PAR, das eine parallele Ausführung aller Befehlszeilen spezifiziert. Da nur eine Eingabe in der Bedingung abgefragt werden kann, ist ein zusätzliches Signal `more` des Verbrauchers nötig, um das nächste Datum aus dem Puffer abzurufen.

### 2.4.5 Implizite und explizite Kommunikation

In unserem Beispiel des *Erzeuger-Verbraucher*-Modells aus Abschn. 2.3.5 auf Seite 67 wurden Daten von einem Prozeß über einen gemeinsamen Speicherbereich an einen anderen Prozeß übergeben, der sie verarbeitet. Diese Informationsübergabe läßt sich als implizite Kommunikation auffassen, die man auch explizit formulieren kann: Anstatt `putInBuffer(item)` und `getFromBuffer(item)` verwendet man dann `send(consumer,item)` und `receive(producer,item)`

*Erzeuger*

```
LOOP
  produce(item);
  send(consumer,item);
END
```

*Verbraucher*

```
LOOP
  receive(producer,item);
  consume(item);
END
```

mit

- `send(consumer,item)` übergibt `item` an einen systeminternen Puffer, der mittels einer `P()`- und `V()`-Operation gefüllt wird. Ist der Puffer voll, so wird der Prozeß verzögert, bis wieder Platz für das `item` vorhanden ist.
- `receive(producer,item)` liest ein `item`, geschützt durch `P()`- und `V()`-Operationen, aus dem Puffer. Ist kein `item` verfügbar, wird der Prozeß so lange verzögert, bis eines eintrifft.

Diese Art des *Erzeuger-Verbraucher*-Modells hat den Vorteil, daß der Synchronisationsmechanismus des *Erzeuger-Verbraucher*-Modells mit dem Synchronisationsmechanismus des Botschaftenaustauschs implementiert werden kann und auch über Rechnergrenzen hinweg funktioniert.

Wir sparen uns also die Semaphoroperationen in unserem Beispiel, indem wir entsprechende Funktionalitäten der Nachrichtenprozeduren `send(Msg)` und `receive(Msg)` voraussetzen. Konkret läßt sich dies beispielsweise auf einem Monoprozessorsystem durch lokale Semaphoroperationen implementieren. Der gemeinsame Pufferbereich ist in diesem Fall eine Mailbox, in die die Nachrichten eingehängt werden.

Allgemein gilt, daß fast alle Kommunikationsmechanismen, die zwischen Prozessen mit Hilfe gemeinsamer Speicherbereiche funktionieren (*shared memory,* s. Abschn. 3.3.3), besser durch expliziten Nachrichtenaustausch formuliert werden sollten. Der geringfügige Zusatzaufwand zahlt sich aus, wenn die darauf aufbauende Software nicht nur auf einem Monoprozessor, sondern auch in verteilten Systemen ablaufen soll. Hier kann man leicht von der Interprozeßkommunikation zur Interprozessorkommunikation überwechseln, indem bei gleicher Schnittstelle eine andere, auf Netzkommunikation basierende Implementierung der `send()`- und `receive()`-Funktionen als Bibliothek verwendet wird.

### 2.4.6 Aufgaben zur Prozeßkommunikation

**Aufgabe 2.4-1 (Prozeßkommunikation)**

a) Beschreiben Sie detailliert, was folgende Kommandozeile in UNIX bewirkt:

```
grep deb xyz | wc -l
```

b) In UNIX ist die Kommunikation durch die Signale auf eine Eltern/Kind-Prozeßgruppe beschränkt. Warum könnten die Implementatoren diese Einschränkung getroffen haben?

c) Formulieren Sie den Übergang zwischen Prozeßzuständen aus Abschn. 2.1 mit Hilfe von Nachrichten und Mailboxen. Wer schickt an wen die Nachrichten?

**Aufgabe 2.4-2 (Pipes)**

Betrachten Sie ein Prozeßsystem, das nur mit UNIX-*pipes* kommuniziert, deren Puffer aus einem gemeinsamen Speicherplatz alloziert werden. Jede *pipe* hat genau einen Sende- und Empfangsprozeß.

a) Unter welchen Umständen wird ein Prozeß gezwungen, zu warten?
b) Skizzieren Sie einen geeigneten Mechanismus, um für dieses System Verklemmungen festzustellen oder zu vermeiden.
c) Gibt es eine Regel in diesem System, um den „Opferprozeß“ auszuwählen, wenn eine Verklemmung existiert? Wenn ja, begründen Sie die Regel.

http://www.springer.com/978-3-540-00900-9

Betriebssysteme
Grundlagen und Konzepte
Brause, R.
2004, XI, 400 S. 170 Abb., Softcover
ISBN: 978-3-540-00900-9