Betriebssysteme

Sommersemester 2004

406 334 V0 2.0

Do 14:30–17:30, HS 23

4.3.2004 – 6.5.2004

Hans P. Zima

Institut für Softwarewissenschaft
Universität Wien, Liechtensteinstr. 22
A-1090 Wien
Tel. 4277-38801
E-Mail: zima@par.univie.ac.at

INHALT

Literatur:

- Andrew S. Tanenbaum: Modern Operating Systems. Second Edition. *Prentice Hall, 2001*
- Andrew S. Tanenbaum: Moderne Betriebsysteme.2.,überarbeitete Auflage. *Prentice Hall, Pearson Studium, 2002*
- Abraham Silberschatz, Peter Galvin, and Greg Gagne: Operating Systems Concepts. Sixth Edition. *John Wiley and Sons, 2003*
- Hans Zima: Betriebssysteme–Parallele Prozesse. Dritte Auflage. *Reihe Informatik Band 20, Bibliographisches Institut Mannheim, 1986 (vergriffen)*

Kapitel 1: Betriebssysteme

Aufgabenstellung und Struktur

Moderne Rechensysteme sind von hoher Komplexität und enthalten eine Vielzahl heterogener Komponenten, wie zum Beispiel Prozessoren, Cache, Hauptspeicher, Plattenspeicher, Magnetbandspeicher, Monitore, Netzwerkschnittstellen und viele andere Ein/Ausgabegeräte. Die *direkte* Kontrolle dieser Komponenten in Applikationsprogrammen ist weder praktikabel noch sicher: daher legt man eine abstrakte Ebene zwischen Hardware und System- sowie Anwendungsprogramme – dies ist das Betriebssystem. **Das Betriebssystem ist die Gesamtheit der Programme eines Rechensystems, welche die Betriebssteuerung und Ressourcenzuteilung erledigen und für die darüberliegenden Softwareschichten eine wohldefinierte Schnittstelle erzeugen.** Man sagt in diesem Zusammenhang, daß das Betriebssystem eine **virtuelle (abstrakte) Maschine** definiert. Die virtuelle Maschine setzt auf der gegebenen Hardware auf und bietet eine Reihe von Dienstleistungen für System- und Anwenderprogramme, die in ihrer Gesamtheit eine einfachere Schnittstelle als die ursprüngliche Hardware darstellt.

Die Untersuchung von Betriebssystemen steht im Mittelpunkt dieser Vorlesung. Bevor wir auf die Struktur von Betriebssystemen genauer eingehen, geben wir einen Überblick über die Komponenten von Rechensystemen (Fig. 1). Die Hardwarestruktur eines einfachen PC ist in Fig. 2 skizziert.

Komponenten eines Computersystems I

- **Physische Maschine**
  Dies ist die unterste Ebene – die selbst wieder tief strukturiert ist. Hierher gehören alle physischen Komponenten des Rechners, wie etwa VLSI Schaltkreise, Drähte, Datenbusse, Uhren und die Stromversorgung.

- **Mikroarchitektur**
  Auf der Ebene der Mikroarchitektur werden die Komponenten der physischen Maschine zu Funktionseinheiten zusammengefaßt, die von Instruktionen angesprochen werden können. Beispiele sind Register oder arithmetisch/logische Einheiten.

- **Maschinensprache**
  Durch die Maschinensprache (**Instruction Set Architecture –ISA**) ist, zusammen mit den durch sie ansprechbaren Hardwarekomponenten, die für den Assemblerprogrammierer sichtbare Hardware-Ebene definiert. Auf dieser Ebene werden über **Instruktionen** Speicherzellen direkt angesprochen, Transfers zwischen Speicher und Registern initiiert, arithmetisch/logische Befehle ausgeführt, und Ein/Ausgabegeräte über Zugriffe zu speziellen Geräteregistern gesteuert. Die Kontrolle eines Plattenspeichers zum Beispiel erfordert die Ausführung von Operationen mit einer Vielzahl von Parametern und komplexe Synchronisation wie etwa das Positionieren der Plattenköpfe und das Warten auf den Transfer von Daten.

Komponenten eines Computersystems II

- **Betriebssystem**
  Das Betriebssystem vereinfacht die Schnittstelle zur Hardware, indem es eine (erweiterte) abstrakte Maschine definiert. Zum Beispiel kann es für das Lesen von der Platte eine Anweisung wie **read block** (f,...) zur Verfügung stellen (f ist eine Datei), ohne daß sich der Benutzer um die die Details der Steuerung der Plattenköpfe kümmern muß.

  Zu den Aufgaben des Betriebssystems gehört auch die Verwaltung von **Betriebsmitteln (Ressourcen)**, das heißt die kontrollierte Zuweisung von Prozessoren, Speichern, Ein/Ausgabegeräten, etc. an konkurrierende Programme in **Zeit** und **Raum** (zum Beispiel Multiplexbetrieb von zentralen Prozessoren bzw. Partitionierung von Speichern).

- **Systemprogramme**
  Systemprogramme sind z.B. Kommando-Interpreter (*shell*), Compiler, Sprach-Interpreter, Editoren und andere Standardprogramme ein, die in Applikationen gebraucht werden. Systemprogramme (ebenso wie Applikationsprogramme) laufen im **Benutzermodus (user mode)**, während die kritischen Komponenten des Betriebssystems in einem hardwaregeschützten **System-Modus (kernel mode, supervisor mode)** ausgeführt werden.

- **Applikationsprogramme**
  Applikationsprogramme sind die von Benutzern zur Lösung eines Anwendungsproblems entwickelten (oder erworbenen) Programme, wie etwa Flugsteuerungssysteme, Crash Simulationen, oder Datenbanksysteme.

Betriebssystemklassen I

Struktur und Aufgaben von Betriebssystemen hängen zu einem wesentlichen Grad von der zugrundeliegenden Hardware ab. Folgende Klassen lassen sich unterscheiden:

- **Großrechner-Betriebssysteme**
  Dies sind Betriebssysteme für Rechenanlagen (zum Beispiel in industriellen Datencentern oder als Webserver), die mehrere CPUs, Tausende von Platten und Terabytes von Speicher enthalten können. Ein Beispiel ist das IBM OS/390 System.

  Solche Systeme bearbeiten normalerweise mehrere Jobs gleichzeitig und sind stark Ein/Ausgabe-fokussiert. Typischerweise werden drei Klassen von Dienstleistungen angeboten:

  - **Stapelbetrieb (batch processing)**
    Dies sind Routineanwendungen ohne Benutzer-Interaktion (z.B. Bearbeitung von Versicherungsfällen oder Berechnung von Verkaufsstatistiken für eine Handelskette).
  - **Transaktionsverarbeitung**
    Dies sind relativ kleine Aufgaben, die in sehr großer Zahl auftreten können (z.B. Reservierungssysteme, on-line Systeme für Banken).
  - **Timesharing**
    Timesharing erlaubt simultanen Zugriff auf einen Rechner durch mehrere Benutzer von entfernten Terminals.

Betriebssystemklassen II

- **Server-Betriebssysteme**
  Dies sind Betriebssysteme für Workstations oder große PCs, die einem Netzwerk von Benutzern den gemeinsamen Zugriff auf Hardware- und Software-Ressourcen ermöglichen (zum Beispiel in einem lokalen Netzwerk). Dazu gehört die Verwaltung von Plattenspeicher, Druckern und Webzugriffen.

  Beispiele sind Unix, Windows 2000, und Linux Systeme.

- **Multiprozessor-Betriebssysteme**
  Dies sind Betriebssysteme für parallele Multiprozessorsysteme oder Cluster von Workstations bzw. PCs (z.B. *Beowulf*). Diese Systeme sind oft Varianten von Server-Betriebssystemen, mit spezieller Unterstützung für Interprozessor-Kommunikation.

- **PC Betriebssysteme**
  Dies sind Systeme, die einem einzelnen Benutzer Zugriff zu seinem PC ermöglichen, z.B. Windows Systeme, Macintosh, oder Linux-Varianten.

Betriebssystemklassen III

- **Realzeit-Betriebssysteme (Real Time Operating Systems)**
  Realzeit-Betriebssysteme unterscheiden sich von anderen Systemen dadurch, daß die *Zeit* ein wesentlicher Parameter ist. Solche Systeme müssen in der Lage sein, gewisse Anforderungen garantiert innerhalb bestimmter Zeitschranken zu erledigen. Beispiele sind Systeme zur Kontrolle von industriellen Prozessen (Hochöfen, Auto-Assemblierung), Kraftwerken, Flugsteuerung, etc. Je nach der Dringlichkeit der Einhaltung von Zeitbedingungen und der Konsequenzen ihrer Nichteinhaltung unterscheidet man **harte** und **weiche** Realzeitsysteme. Ein Kollisions-Vermeidungssystem in einem Flugzeug oder ein Bremskraftregelungssystem in einem Auto ist ein hartes Realzeitsystem, während etwa digitale Audiosysteme zu weichen Realzeitsystemen zählen.

  Ein Beispiel ist das VxWorks System.

- **Eingebettete Betriebssysteme (Embedded Operating Systems)**
  Dies sind Betriebssysteme für "kleine" Geräte wie etwa *Personal Digital Assistants (PDAs)*, Mikrowellenherde, Fernsehgeräte, Mobiltelefone, oder Smart Cards. Diese Systeme haben mit Realzeitsystemen gewisse Eigenschaften gemein, sind aber speziellen Einschränkungen (insbesondere beim Stromverbrauch) unterworfen.

  Ein Beispiel ist PalmOS.

Hardware: Zentrale Prozessoren I

Der **zentrale Processor (CPU)** eines modernen speicherprogrammierten Rechensystems ist eine selbständige Funktionseinheit, welche die zur Instruktionsausführung sowie zur Initiierung bzw. Steuerung von Ein/Ausgabeoperationen erforderliche Funktionalität besitzt.

Die CPU transferiert Instruktionen aus dem Speicher in ein Register, decodiert sie, und steuert die durch die Instruktion vorgegebene Aktion. Dazu gehört oft das Laden von Operanden aus dem Speicher in allgemeine Register und die Ausführung von Operationen in einer Sub-Funktionseinheit, wie z.B. für arithmetisch/logische Operationen.

Die CPU enthält eine Reihe spezieller Register. Dazu gehören:

- **Instruktionsregister (program counter)**
  Dieses Register enthält die Speicheradresse der nächsten auszuführenden Instruktion. Nach dem Laden dieser Instruktion wird das Instruktionsregister inkrementiert.

- **Stackregister**
  Das Stackregister enthält einen Zeiger zum obersten Element des **Stapels (Stack)**. Der Stack enthält einen Datensatz (*frame*) für jeden offenen Funktions- oder Prozeduraufruf.

- **Programmstatuswort-Register (program status word, PSW)**
  Das PSW enthält Kontrollinformation zur aktuellen Programmausführung. Dazu gehören der Ausführungsmodus (Benutzer- oder System-Modus), Priorität, und Bedingungsbits.

Das Betriebssystem hat die Aufgabe, bei einem Wechsel zu einem anderen Programm im Multiplexbetrieb die aktuellen Registerinhalte zur späteren Weiterverwendung abzuspeichern.

Zentrale Prozessoren II: Arbeitsmodi

CPUs können in zwei Modi arbeiten:

- **System-Modus**
  In diesem Modus kann jede beliebige Instruktion ausgeführt und jede Hardware-Funktionalität angesprochen werden. Insbesondere sind in diesem Modus **privilegierte Instruktionen** ausführbar, welche direkt in die Betriebssteuerung, Ein/Ausgabe, die Kommunikation von Prozessen, oder Schutzmechanismen im Speicher eingreifen.

  Nur das Betriebssystem kann im System-Modus arbeiten – aber nicht alle Abläufe des Betriebssystems müssen in diesem Modus abgewickelt werden.

- **Benutzermodus**
  Im **Benutzermodus** können privilegierte Befehle nicht ausgeführt werden. Die damit verbundene Funktionalität läßt sich nur indirekt, über **Systemaufrufe (supervisor call)**, aktivieren. Systemaufrufe werden durch spezielle Instruktionen erzeugt, die einen Wechsel vom Benutzermodus zum System-Modus und damit verbunden die Aktivierung einer Betriebssystemfunktion bewirken. Nach Abschluß der Bearbeitung dieser Funktion wird wieder auf Benutzermodus geschaltet und die Kontrolle an das Benutzerprogramm zurückgegeben.

Zentrale Prozessoren III: Unterbrechungen

Die Arbeit einer CPU kann zwischen der Ausführung zweier aufeinanderfolgender Instruktionen unterbrochen werden. Eine **(Programm-)Unterbrechung (Interrupt)** resultiert in einem Sprung auf eine feste, der Unterbrechung zugeordnete Stelle. Dies ist ein Mechanismus, der die CPU zwingt, auf gewisse, interne oder externe, Ereignisse zu reagieren.

Man unterscheidet synchrone und asynchrone Unterbrechungen.

- **Synchrone Unterbrechungen** werden durch das Auftreten von speziellen Ereignissen während der Instruktionsausführung bewirkt. Beispiel dafür sind arithmetischer Überlauf, Division durch 0, illegaler Instruktionscode, illegale Adresse, Systemaufruf.
- **Asynchrone Unterbrechungen** werden durch Ereignisse außerhalb der CPU bewirkt. Sie werden häufig dazu benutzt, die Arbeit der CPU mit der von externen Funktionseinheiten zu koordinieren. Beispiele sind: Ende einer Ein/Ausgabe-Operation, Anforderung Eingabe (z.B. von einer Tastatur), oder Uhr-Alarm.

Bei der Bearbeitung von Unterbrechungen ist es nötig, den Zustand der CPU zum Zeitpunkt der Unterbrechung zu retten, um einen eventuell aufgetretenen Fehler identifizieren zu können bzw. nach Bearbeitung der Unterbrechung die unterbrochene Programmausführung fortsetzen zu können.

Hardware: Speicher I

Eine moderne Rechenanlage besitzt eine **Speicherhierarchie**, die von extrem schnellen, kleinen, und teuren Speichern an der Spitze (Register) bis zu relativ langsamen, großen und billigen Speichern auf den unteren Ebenen (Magnetbänder) reicht. Im einzelnen zählen dazu die folgenden Komponenten:

- **Register**
  Register werden explizit im Programm verwaltet. Zugriffszeiten sind durch die Geschwindigkeit der CPU vorgegeben und liegen in der Größenordnung von Nano- bzw. Picosekunden. Der gesamte über Register verfügbare Speicher liegt im KB Bereich.

- **Cache Speicher**
  Cache Speicher sind schnelle Zwischenspeicher, die entweder für die Speicherung von Instruktionen (**Instruktions-Cache**) oder für Daten (**Daten-Cache**) benutzt werden. Der Daten-Cache wird von Berechnungen mit *temporärer Lokalität* zum Puffern von häufig gebrauchten Operanden benutzt. Die Verwaltung von Cache Speichern ist weitgehend automatisiert; in aktuellen Rechnern werden bis zu drei Ebenen solcher Speicher benutzt.

  Zugriffszeiten liegen bei einigen Nanosekunden, Speichergrößen im MB Bereich.

Speicher II

- **Hauptspeicher (main memory)**
  Im Hauptspeicher ist der Großteil der von einem Programm verarbeiteten Daten abgelegt. Zugriffszeiten liegen bei einigen -zig Nanosekunden, Speichergrößen erreichen bereits GB. Der Hauptspeicher ist als **Random Access Memory (RAM)** ausgelegt.

- **Magnetplatten**
  Magnetplattenspeicher sind zylindrische Systeme von rotierenden Platten. Die Zugriffszeiten liegen im Millisekundenbereich, die Speichergröße übertrifft den Hauptspeicher um 2 Größenordnungen, der Preis liegt um 2 Größenordnungen niedriger.

- **Magnetbänder**
  Magnetbänder werden noch für extrem große Datenmengen oder als Hintergrundspeicher für Plattenspeicher benutzt. Sie sind billiger und größer als Plattenspeicher, jedoch durch extrem lange Zugriffszeiten (zig Sekunden) charakterisiert.

Neben den genannten Speicherklassen gibt es unter anderem noch vorprogrammierten **Read Only Memory (ROM)**, der bei Abschaltung der Stromzufuhr seinen Inhalt behält, jedoch nicht modifiziert werden kann. ROM wird unter anderem für die Steuerung von Gerätefunktionen auf niedriger Ebene benutzt.

Hardware: Ein/Ausgabegeräte

Die für die Ein/Ausgabe verantwortlichen Komponenten eines Rechensystems haben die Aufgabe, den Datentransfer zwischen Hauptspeicher und Geräten zu organisieren. Dazu gehört die Initiierung von Zugriffen zum Hauptspeicher, die Koordinierung der Geschwindigkeiten von Hauptspeicher und Geräten, und die Überprüfung der übertragenen Daten auf Korrektheit.

Ein/Ausgabegeräte sind im allgemeinen mit einer separaten **Steuerungseinheit (Controller)** verbunden, die ihrerseits die Schnittstelle zum Betriebssystem darstellt und dem Betriebssystem damit einfacheren Zugriff zum Gerät ermöglicht.

Controller für verschiedene Geräte sind gerätespezifisch verschieden; demnach unterscheiden sich auch die für den Zugriff zu Controllern erforderlichen Softwarekomponenten. Diese heißen **Gerätetreiber (device driver)**. Gerätetreiber sind aber auch betriebssystemspezifisch: so muß für jedes Betriebssystem (z.B. Windows 2000 oder Unix) ein system-spezifischer Treiber zur Verfügung gestellt werden. Gerätetreiber werden gewöhnlich im System-Modus ausgeführt.

Ein/Ausgabe kann in drei verschiedenen Modi ausgeführt werden, je nachdem wie die Kommunikation zwischen Betriebssystem und Gerät formuliert ist: (1) Aktives Warten, (2) interruptgesteuert, und (3) DMA-gesteuert.

Ein/Ausgabe-Modi

- Bei **aktivem Warten (busy waiting)** wird der Systemaufruf zur Aktivierung eines Geräts im Betriebssystem in einen Aufruf einer Routine für den entsprechenden Treiber übersetzt. Der Treiber startet den Ein/Ausgabeprozeß und überwacht ihn kontinuierlich (**polling**). Nach dessen Abschluß wird die Kontrolle wieder an das Benutzerprogramm übergeben.

  Der Nachteil dieser Methode besteht darin, daß die CPU ständig mit der Kontrolle des Ein/Ausgabeprozesses befaßt ist und nicht für andere Aufgaben zur Verfügung steht.

- Bei der **interrupt-gesteuerten Methode** startet der Treiber das Gerät und gibt an dieser Stelle die Kontrolle wieder an das Betriebssystem zurück. Das Betriebssystem blockiert den Benutzerprozeß, wenn nötig, und führt beliebige andere Aufgaben aus. Bei Beendigung des Ein/Ausgabeprozesses wird ein **Unterbrechungssignal (interrupt signal)** vom Controller ausgesandt, das vom Betriebssystem verarbeitet wird und in der Regel zur Entblockierung des ursprünglichen Benutzerprozesses führt.

- Bei der dritten Methode wird eine spezielle **Direct Memory Access (DMA)** Hardwarekomponente benutzt. Die CPU wird nur dazu benutzt, die DMA zu aktivieren und das abschließende Unterbrechungssignal zu verarbeiten: die DMA steuert den Datenfluß zwischen Controller und Speicher ohne Involvierung der CPU autonom.

Mehrprozessorsysteme

Der ursprüngliche Prototyp eines speicherprogrammierten Rechensystems geht auf **John von Neumann** zurück, in dem *ein monolithischer zentraler Prozessor* sämtliche mit der Ausführung von Instruktionen und Ein/Ausgabe notwendigen Aktionen sequentiell durchführt. Man spricht in diesem Zusammenhang vom von-Neumann'schen Rechnerkonzept (1948). Heutige CPUs bestehen dagegen aus vielen, unabhängig und simultan arbeitenden Funktionseinheiten, die autonom spezifische Teilfunktionen erfüllen. Zur Steigerung der Leistung von Rechensystemen über den von einem einzelnen Prozessor erzielbaren Wert hinaus wurden **Multiprozessorsysteme** entwickelt: solche Systeme enthalten mehrere CPUs und Speichermodule, die durch ein **Kommunikationsnetzwerk** verbunden sind. Man unterscheidet zwei Klassen:

- **Systeme mit Gemeinsamem Speicher (shared memory systems)**
  Jede CPU kann jeden im System verfügbaren Speicherplatz direkt adressieren. Es existiert also ein globaler Adreßraum. Man differenziert zwischen Systemen mit **uniformem Speicherzugriff (uniform memory access, UMA)** bzw. **nicht-uniformem Speicherzugriff (non-uniform memory access, NUMA)**.

- **Systeme mit Verteiltem Speicher (distributed memory systems)**
  Hier ist der Gesamtspeicher so partitioniert, daß jede CPU einen eigenen lokalen Speicher besitzt. Der Zugriff zum lokalen Speicher ist dann direkt, während nichtlokale Speicher nur über **Nachrichtenaustausch (message passing)** mit anderen CPUs zugegriffen werden können. Systeme mit verteiltem Speicher sind also inhärent nicht-uniform.

Funktionseinheiten und Aufträge

Moderne Rechensysteme sind (in Software wie Hardware) hierarchisch modular aufgebaut. Die einzeln identifizierbaren Komponenten in dieser Struktur bezeichnen wir als **Funktionseinheiten (functional units)**. Wir sprechen immer dann von einer Funktionseinheit, wenn wir ein durch Ausgabe oder Wirkung abgrenzbares Gebilde erkennen können. Je nach der betrachteten Abstraktionsebene kann man ein ganzes Rechensystem oder dessen Komponenten (CPUs, Speicher, Geräte, etc.) als Funktionseinheiten auffassen. Verschiedene Funktionseinheiten können *im Prinzip* (das heißt, wenn keine expliziten Beschränkungen existieren) unabhängig voneinander simultan arbeiten.

Ein **Auftrag (task)** ist eine Nachricht, die an eine Funktionseinheit übermittelt wird, um sie zu einer Bearbeitung zu veranlassen. Bestandteile von Aufträgen sind Operatoren und Operanden. Operatoren spezifizieren eine Funktion, Operanden die zu manipulierenden Daten.

Sei $\mathbf{A}$, mit $\mid \mathbf{A} \mid \geq 2$, eine Menge von Aufträgen für eine Funktionseinheit F und seien A und A' mit $A \neq A'$ beliebig aus $\mathbf{A}$ gewählt. Dann heißt A' **abhängig** von A genau dann, wenn (*gdw*) eine logische Beschränkung existiert, so daß A vor A' ausgeführt werden muß. Ein Beispiel für eine solche Beschränkung liegt dann vor, wenn A' Resultate benötigt, die von A produziert werden. Sind dagegen A und A' unabhängig, dann können sie in beliebiger Reihenfolge, überlappt, oder parallel ausgeführt werden.

Auftragsbearbeitung

Sei $\mathbf{A} = \{A_1, \ldots, A_n\}$ eine Menge von paarweise unabhängigen Aufträgen für eine Funktionseinheit F. F kann die Aufträge in $\mathbf{A}$ mit verschiedenen Methoden abarbeiten:

- Bei der **sequentiellen (seriellen) Verarbeitung** werden die Aufträge in irgendeiner Reihenfolge $A_{i_1}, \ldots, A_{i_n}$ so abgearbeitet, daß für alle $j, 1 \leq j < n$, $A_{i_{j+1}}$ erst begonnen wird, wenn A_{i_j} abgeschlossen ist.

- Bei der **konkurrenten (nebenläufigen) Verarbeitung** können sich zu einem beliebigen Zeitpunkt mehrere Aufträge in unterschiedlichen Phasen ihrer Bearbeitung befinden. Wir sprechen von **paralleler Verarbeitung**, wenn verschiedene Aufträge in Unter-Funktionseinheiten simultan bearbeitet werden können, und von **pseudo-paralleler Verarbeitung**, wenn Parallelität über **Multiplexbetrieb** simuliert wird. Beim Multiplexbetrieb werden Aufträge einzeln, in Zeitabschnitten verzahnt, von der Funktionseinheit bearbeitet; ein gerade bearbeiteter Auftrag kann *verdrängt*, d.h. zugunsten eines anderen Auftrags(teils) unterbrochen und zurückgestellt werden (*preemption*). Bei diesem Betrieb wird also die Zeit in Intervalle beliebiger Länge unterteilt, die nacheinander verschiedenen (Teil-)Aufträgen zugeordnet werden. Damit läßt sich im allgemeinen eine bessere Ausnutzung einer Funktionseinheit als beim seriellen Betrieb realisieren, da die Funktionseinheit nicht still-liegen muß, wenn bei der Bearbeitung eines Auftrags Wartezeiten auftreten.

  Ein Beispiel mit einem Vergleich von serieller und Multiplexbearbeitung findet sich in Fig. 3.

Kapitel 2: Prozesse und Threads

Der Begriff **Prozeß** ist von zentraler Bedeutung:
die gesamte Aktivität in einem Rechensystem läßt sich als ein System von sequentiellen Prozessen interpretieren.
Mit dem traditionellen Prozeßbegriff sind zwei unabhängige Konzepte verbunden:

- Zuordnung von Betriebsmitteln
- Ausführung von Programmen

Ein Prozeß dient als Fokus für die **Zuordnung einer Menge von Betriebsmitteln**, die in ihrer Gesamtheit für die Lösung einer Aufgabe gebraucht werden. Dazu gehört ein prozeßspezifischer Adreßraum, Speicherbereiche für Code und Daten, Hintergrundspeicher, Peripheriegeräte, und Dateien.

Ein Prozeß ist auch mit der **Ausführung von Programmen** verbunden. In traditionellen Systemen ist dies ein sequentieller Vorgang – man spricht daher auch von **sequentiellen Prozessen**. Der Prozeß besitzt dann einen einzigen Befehlzähler und arbeitet die Anweisungen bzw. Instruktionen des Programms sequentiell ab.

Von diesem Modell werden wir zu Beginn ausgehen. Später wird das Modell durch die Einführung von *Threads* erweitert: dadurch wird prozeßinterne Parallelität modellierbar.

Verschiedene Prozesse können unter Berücksichtigung ihrer Abhängigkeiten konkurrent ausgeführt werden.

2.1 Betriebsmittel

Alle Objekte, die von Prozessen zu ihrer Durchführung gebraucht werden, nennt man **Betriebsmittel (Ressourcen; resources)**. Betriebsmittel lassen sich untergliedern in Hardware-Betriebsmittel und Software-Betriebsmittel.
Hardware-Betriebsmittel sind alle hardwaremäßig existierenden Funktionseinheiten, die für die Ausführung von Programmen benötigt werden, zum Beispiel:

- Zentrale Prozessoren
- Hauptspeicher-Module
- Ein/Ausgabe-Prozessoren
- Geräte: Peripheriegeräte, Plattenspeicher, Magnetbandspeicher,...

Software-Betriebsmittel sind alle Software-Artifakte, die in Programmen angesprochen werden können, zum Beispiel:

- Konstanten und Variablen
- Klassen und Objekte in objektorientierten Sprachen
- Dateien
- Prozeduren

Betriebsmittel II

Ein Betriebsmittel ist entweder wiederverwendbar oder verbrauchbar.

Ein **wiederverwendbares (reusable)** Betriebsmittel ist ein Betriebsmittel, das nach seiner Benutzung in einem Prozeß noch existiert und in der Folge von anderen Prozessen benutzt werden kann.

Alle Hardware-Betriebsmittel sowie Software-Betriebsmittel wie Variablen, Dateien, und Prozeduren sind wiederverwendbar.

Ein **verbrauchbares (consumable)** Betriebsmittel existiert nach der Benutzung durch einen Prozeß nicht mehr. Dazu zählen zum Beispiel Nachrichten oder Aktivierungssätze von Prozeduraufrufen.

In der Folge werden nur wiederverwendbare Betriebsmittel diskutiert, solange nicht explizit etwas anderes gesagt wird. Betriebsmittel gleicher Art werden zu **Betriebsmitteltypen** zusammengefaßt; von jedem Betriebsmitteltyp kann es unterschiedliche **Einheiten** geben. Verschiedene Einheiten eines Typs sind für Prozesse ununterscheidbar.

Betriebsmittel III

Ein Betriebsmittel, das zu einem Zeitpunkt von einem oder mehreren Prozessen benutzt wird, heißt **belegt (busy)**, andernfalls **frei** oder **verfügbar**. Die in einem Betriebsmittel enthaltene Information wird als dessen **Zustand** bezeichnet: zum Beispiel ist der Zustand eines zentralen Prozessors durch die Inhalte der Register, des Befehlszählers, Programmstatusworts, und anderer interner Register gegeben. Der Zustand eines Moduls des Hauptspeichers ist durch die Inhalte der zugehörigen Speicherworte und interner Register charakterisiert.

Der Zustand eines Betriebsmittels kann bei der Benutzung durch einen Prozeß gleichbleiben oder sich verändern. So ändert sich der Zustand einer Variablen oder Datei bei lesendem Zugriff nicht, während er sich bei schreibendem Zugriff ändert. Der Zustand einer Prozedur im Programmcode ändert sich bei Benutzung (d.h. Aufruf) nie. Der Zustand eines zentralen Prozessors ändert sich bei der Benutzung.

Der Zugriff zu einem Betriebsmittel kann exklusiv oder nichtexklusiv sein. Ein Prozeß greift **exklusiv** auf ein Betriebsmittel zu, wenn kein anderer Prozeß das Betriebsmittel gleichzeitig benutzen darf. Exklusiver Zugriff ist erforderlich, wenn sich der Zustand des Betriebsmittels bei Benutzung ändert.

Bei **nichtexklusivem Zugriff (shared access)** können mehrere Prozesse gleichzeitig auf ein Betriebsmittel zugreifen. Dies ist zum Beispiel bei lesendem Zugriff mehrer Prozesse auf eine Datei der Fall.

Betriebsmittel IV

Ein typisches Schema für die Verwaltung von Betriebsmitteln orientiert sich an den beiden Aktionen Anforderung und Freigabe:

- wenn ein Prozeß ein Betriebsmittel benötigt, stellt er an das Betriebssystem eine **Anforderung**;
- nach Erfüllung der Anforderung besitzt der Prozeß **Zugriff** auf das Betriebsmittel;
- nach Beendigung der Benutzung initiiert der Prozeß die **Freigabe**.

Dieses Schema hat die Schwäche, daß die Dauer der Benutzung und die Initiierung der Freigabe in der Verantwortung des Prozesses liegt. Unter bestimmten Bedingungen ist es für das Betriebssystem notwendig, eine Trennung zwischen Betriebsmittel und Prozeß *ohne Mitwirkung des Prozesses* zu bewerkstelligen: man spricht dann von **Entzug (preemption)**. Bei einem Entzug wird der Prozeß durch das Betriebssystem unterbrochen und der Zustand gerettet, um eine spätere Fortsetzung des Prozesses zu ermöglichen.

Ob ein Entzug eines Betriebsmittels überhaupt möglich ist oder nicht hängt von den Charakteristiken des Betriebsmittels ab. Ist ein Entzug möglich, dann heißt das Betriebsmittel **entziehbar (preemptible)**. Typische Beispiel für entziehbare Betriebsmittel sind zentrale Prozessoren und Hauptspeicherbereiche.

Beispiele für nichtentziehbare Betriebsmittel sind Geräte mit mechanischer Positionierung wie etwa Magnetbänder. Häufig ist die Trennlinie zwischen entziehbaren und nichtentziehbaren Betriebsmitteln lediglich durch die Höhe der mit einem Entzug verbundenen Kosten bestimmt.

2.2 Prozesse

Die Gesamtheit der in einem Rechensystem existierenden Abläufe läßt sich als ein **System von Prozessen** verstehen. Die Ausführung eines Prozeßystems im Multiplexbetrieb nennt man **Mehrprogrammbetrieb (multiprogramming)**. Verschiedene Prozesse können im Prinzip konkurrent ablaufen. Prozesse können voneinander unabhängig sein oder (implizit oder explizit) kooperieren.

Die **Generierung** von Prozessen ist auf verschiedene Arten möglich:

1. Explizite Generierung
2. Systeminitialisierung
3. Benutzeranforderung
4. Initiierung einer Anwendung im Stapelbetrieb

Ein aktiver Prozeß kann einen neuen Prozeß durch einen entsprechenden Systemaufruf **explizit erzeugen**. Dies ist vor allem dann von Nutzen, wenn sich die Arbeit des generierenden Prozesses in unabhängige Teilaktivitäten strukturieren läßt. Ein Beispiel ist das Einlesen und Bearbeiten einer verteilten Datenmenge über ein Netzwerk – dies ist eine Instanz des Erzeuger/Verbraucher Problems. Ein anderes Beispiel ist die parallele Bearbeitung einer Menge von unabhängigen Aufträgen in einem Mehrprozessorsystem.

Generierung von Prozessen I

Wenn ein Prozeß p_1 einen Prozeß p_2 generiert, so nennen wir p_1 den **Vater** von p_2, und p_2 ein **Kind** von p_1.

Ein **Nachfahre (descendant)** von p_1 ist entweder ein Kind von p_1 oder ein Nachfahre eines Kindes von p_1. Umgekehrt ist ein **Vorfahre (ancestor)** von p_2 der Vater von p_2 oder ein Vorfahre des Vaters. Die Menge aller Nachfahren eines Prozesses heißt die **Prozeßgruppe**. Die Prozeßgruppe spielt in manchen Systemen eine Rolle. Wenn zum Beispiel in UNIX ein Benutzer ein Signal über die Tastatur sendet, wird dieses Signal an alle Prozesse weitergeleitet, die gegenwärtig in der zur Tastatur (zugehörig zum aktuellen Fenster) gehörenden Prozeßgruppe sind. Windows kennt ein analoges Konzept nicht.

Die Prozeßgruppe von p_1 läßt sich als ein Baum mit Wurzel p_1 darstellen, deren unmittelbare Nachfolger die Kinder von p_1 sind. Die Blätter dieses Baums sind kinderlose Prozesse. Das gesamte, zu einem Zeitpunkt existierende Prozeßsystem läßt sich als ein Baum mit einer Wurzel, p_0, darstellen, die im Betriebssystem ihren Ursprung hat.

Jeder Prozeß besitzt also immer einen eindeutig bestimmten Vater (außer p_0) und 0 oder mehr Kinder. Wegen der dynamischen Generierung und Terminierung von Prozessen sind alle diese Begriffe abhängig von der Zeit zu verstehen.

Im folgenden diskutieren wir die unterschiedlichen Klassen der Generierung von Prozessen. Alle diese Fälle lassen sich als spezielle Instanzen von expliziter Prozeßgenerierung verstehen.

Generierung von Prozessen II

Bei der **Systeminitialisierung (booting)** wird eine Menge von Prozessen erzeugt, die man als Vordergrundprozesse oder Hintergrundprozesse klassifiziert. *Vordergrundprozesse (foreground processes)* sind dadurch charakterisiert, daß sie mit Benutzern kommunizieren und Aufträge von Benutzern übernehmen. *Hintergrundprozesse (background processes)* sind nicht speziellen Benutzern zugeordnet, sondern führen allgemeine Funktionen aus. Hierzu gehört zum Beispiel ein Prozeß, der ankommende Email Nachrichten liest, eventuell filtert, und aufbereitet; oder ein Prozeß, der Anforderungen von außen für lokale Webseiten bearbeitet. Prozesse für derartige Aufgaben, die etwa auch die Aufbereitung von Daten für den Druck bewirken, werden auch als **daemons** bezeichnet.

In interaktiven Systemen lassen sich Prozesse durch **Benutzeranforderungen** generieren. Dies kann zum Beispiel durch die Ausführung eines Kommandos oder das Anklicken eines Icons bewirkt werden. Bei einer genaueren Analyse dieses Falls erkennt man, daß er auf Fall 1 zurückgeführt werden kann: beim Anklicken eines Icons wird ein damit verbundener Systemprozeß zur Ausführung eines Systemaufrufs gebracht, der den gewünschen Prozeß erzeugt.

Der letzte Fall ist die **Initiierung eines Auftrags im Stapelbetrieb** durch Benutzer von (möglicherweise entfernten) Terminals aus. Der Auftrag wird in eine Warteschlange eingereiht und zu einem gegebenen Zeitpunkt (der von der Zielfunktion des Betriebssystems und der Verfügbarkeit der Betriebsmittel abhängt) vom Betriebssystem als Prozeß aktiviert. Auch dieser Fall läßt sich auf den Fall 1 der expliziten Generierung zurückführen.

Terminierung von Prozessen

Bei der **Terminierung von Prozessen** werden folgende Fälle unterschieden:

1. Normale Terminierung
2. Fehler: freiwillige Terminierung
3. Fehler: erzwungene Terminierung
4. Terminierung von außen

Im ersten Fall, der **normalen Terminierung**, beendet ein Prozeß seine Ausführung zu dem Zeitpunkt, zu dem seine Aufgabe abgeschlossen ist.

Der Fall der **freiwilligen Terminierung** ist dadurch charakterisiert, daß der Prozeß einen Fehler findet, der eine Fortsetzung unmöglich macht. Ein Beispiel ist der Aufruf eines Compilers mit einer leeren Quelldatei. (Dies gilt nur bei Stapelverarbeitung: in einem interaktiven System würde dieser Fall nicht zu einer Beendigung des Prozesses führen, sondern zu einer Rückfrage an den Benutzer.)

Eine **unfreiwillige Terminierung** wird zum Beispiel als Resultat eines Fehlers im Prozeß bewirkt: illegale Befehlsausführung, Division durch 0, oder undefinierte Speicherzugriffe sind Beispiele. In manchen Systemen kann der Prozeß einen derartigen Fehler selbst behandeln, damit also eine Terminierung vermeiden.

Bei der **Terminierung von außen** führt ein Prozeß, p_1, einen Systemaufruf durch, der einen anderen Prozeß, p_2, terminiert (**kill**). Dies ist nur möglich, wenn p_1 zu dieser Aktion in bezug auf p_2 explizit autorisiert ist (Beispiel: Baumsuche).

Status von Prozessen

Wir betrachten die Ausführung eines Prozeßsystems im Mehrprogrammbetrieb auf einem zentralen Prozessor. Zentrale Prozessoren sind exklusiv zugreifbare, entziehbare Betriebsmittel, die eine Sonderstellung einnehmen, da sie für die Ausführung jeder Aktion eines Prozesses benötigt werden. Jeder Prozeß fordert implizit vor Durchführung seiner ersten Aktion einen zentralen Prozessor an und gibt diesen in der Regel erst nach seiner letzten Aktion wieder frei. Eine Trennung zwischen Prozeß und Prozessor kann vor der Beendigung eines Prozesses jedoch durch Entzug bewirkt werden.

Unter einer *logischen Blockierung* eines Prozesses verstehen wir im folgenden eine Situation, in der eine bestimmte Bedingung in der Umgebung des Prozesses, von der sein Fortschreiten abhängt, nicht erfüllt ist. Ein typisches Beispiel ist das Warten auf einen Eingabewert.

Der **Status** eines Prozesses kann drei verschiedene Werte annehmen (Fig. 4): aktiv, rechenwillig, und blockiert:
Ein Prozeß ist im Status **aktiv**, wenn er nicht logisch blockiert ist und ihm der zentrale Prozessor zugeteilt ist. Der Prozeß kann dann Aktionen ausführen.
Ein Prozeß ist im Status **rechenwillig (ready)**, wenn er nicht logisch blockiert ist und ihm der zentrale Prozessor nicht zugeteilt ist. Ein rechenwilliger Prozeß kann keine Aktionen ausführen.
Ein Prozeß ist im Status **blockiert**, wenn er logisch blockiert ist. Ein blockierter Prozeß kann keine Aktionen ausführen.

Statusübergänge

1. **aktiv → blockiert**: Ein Prozeß geht vom aktiven in den blockierten Status über, wenn er eine Stelle erreicht, an der er logisch blockiert ist. Dieser Übergang kann durch einen expliziten Systemaufruf oder, in speziellen Fällen, automatisch durch das System bewirkt werden. Die umgekehrte Transition – ein Übergang von blockiert nach aktiv – ist *nicht* vorgesehen. Ein Prozeß, für den der Grund seiner Blockierung entfallen ist, muß vor der Zuteilung eines Prozessors immer zuerst in den Zustand rechenwillig übergehen.

2. **aktiv → rechenwillig**: Diese Transition stellt den Entzug des Prozessors für einen logisch nicht blockierten Prozeß durch den *Scheduler* dar. Der Grund für den Entzug kann zum Beispiel im Ablauf eines für den Prozeß reservierten Zeitintervalls oder in der Ankunft eines Prozesses mit höherer Priorität in der Menge der rechenwilligen Prozesse liegen.

3. **rechenwillig → aktiv**: Diese Transition bewirkt die Zuteilung des Prozessors für einen rechenwilligen Prozeß. Wie Transition 2 wird sie durch den Scheduler initiiert.

4. **blockiert → rechenwillig**: Diese Transition wird durchgeführt, wenn die Bedingung, auf die ein logisch blockierter Prozeß wartet, durch Aktionen in seiner Umgebung erfüllt wird.

Die Transitionen 2 und 3 sind Aktionen, von denen der Prozeß "nichts weiß", die er also nicht in seinem Code berücksichtigen muß. Sie werden vom Scheduler durchgeführt und sind von dessen Zielfunktion und dem Systemzustand abhängig.

Implementierung von Prozessen I

Das Betriebssystem verwaltet eine **Prozeßtabelle**, in der jedem existierenden Prozeß genau ein **Prozeß-Kontrollblock** zugeordnet ist. Der Prozeß-Kontrollblock enthält alle Information, die für die Verwaltung eines Prozesses und seiner Betriebsmittel erforderlich ist: dies ist die Gesamtheit aller prozeßspezifischen Daten, die beim Übergang von aktiv nach blockiert oder rechenwillig gespeichert werden müssen, um später eine nahtlose Wiederaufnahme des Prozesses zu ermöglichen.

Wichtige Komponenten in dieser Struktur können in drei Klassen gruppiert werden:

- Prozeßverwaltung
- Speicherverwaltung
- Dateiverwaltung

Im folgenden skizzieren wir typische Komponenten dieser Klassen.

Implementierung von Prozessen II

- **Prozeßverwaltung**
  - Register
  - Programmzähler, Programmstatuswort
  - Status
  - Stackzeiger
  - Priorität
  - Schedulingparameter
  - Vaterprozeß, Prozeßgruppe
  - Verbrauchte CPU-Zeit des Prozesses und seiner Kinder
- **Speicherverwaltung**
  - Zeiger auf das Codesegment
  - Zeiger auf das Datensegment
  - Zeiger auf das Stacksegment
- **Dateiverwaltung**
  - Wurzelverzeichnis (*root directory*)
  - Arbeitsverzeichnis (*working directory*)
  - Dateideskriptoren

2.3 Threads

In traditionellen Systemen ist jeder Prozeß (1) mit einer Menge von Betriebsmitteln (insbesondere einem eigenen Adressenraum) und (2) mit einem sequentiellen Ablauf (einem einzigen Befehlszähler) verbunden. Diese beiden Aspekte lassen sich bis zu einem gewissen Grade trennen. Wir führen hier *Threads* ein, um prozeßinterne Parallelität zu ermöglichen.
Ein **Thread** ist ein sequentieller Ablauf *innerhalb* eines Prozesses. Zu jedem Thread gehören thread-spezifisch die folgenden Komponenten:

- Befehlszähler
- Register
- Stack
- Status

Ein Thread arbeitet im Adreßraum seines Prozesses, und kann im Regelfall auf alle Betriebsmittel des Prozesses zugreifen. Ein Prozeß kann einen oder mehrere Threads enthalten: Threads erlauben also die Programmierung konkurrenter Abläufe *innerhalb* von Prozessen, und damit mehrstufige Parallelität. Statt von Threads spricht man auch von *leichtgewichtigen Prozessen (lightweight processes)*. Das ursprüngliche Prozeßmodell ergibt sich als Spezialfall dieses Modells mit genau einem Thread pro Prozeß.

Threads II

Threads können unter anderem auf die folgenden Komponenten eines Prozesses gemeinsam zugreifen:

- Adreßraum
- Globale Variablen
- Dateien
- Kindprozesse
- Offene Alarme
- Signale
- Buchhaltungsinformation

Alle zu einem Prozeß gehörigen Threads arbeiten im Adreßraum des Prozesses und haben damit im Prinzip zu allen von allen Threads angesprochenen Adressen Zugang. Während es zwischen verschiedenen Prozessen eines Systems Schutzmechanismen gibt, die von der Hardware und dem Betriebssystem unterstützt werden, existiert ein solcher Schutz für die Threads innerhalb eines Prozesses nicht.

Wenn in einem Thread eines Prozesses eine Datei geöffnet wird, dann können im Prinzip alle anderen Threads diese Datei schreiben und lesen. Ähnliches gilt für Alarme, Signale, etc.

Motivation für Threads

- **Vereinfachung des Programmiermodells**
  In vielen Applikationen existieren innerhalb von Prozessen konkurrente Abläufe, die auch voneinander abhängen und blockieren können. Es ist einfacher, die Abläufe in einem solchen Prozeß durch eine Menge sequentieller Threads zu modellieren als durch einen einzigen Thread.

- **Einfachere Verwaltung**
  Da Threads mit keinen Betriebsmitteln verbunden sind, ist ihre Generierung und Terminierung sowie Prozessorzuordnung und Entzug einfacher als bei Prozessen. Dies ist insbesondere von Vorteil, wenn es eine große Zahl von Threads gibt und das Threadsystem raschen dynamischen Veränderungen unterworfen ist.

- **Höhere Prozessorausnutzung**
  Bei intensiver Ein/Ausgabeaktivität läßt sich ein Multiplexbetrieb *innerhalb* eines Prozesses organisieren: es ist nicht nötig, einen Prozeß, der in einem Thread auf eine Eingabe wartet, zu blockieren, wenn es einen anderen Thread gibt, der den Prozessor während der Wartezeit nutzen kann.

- **Multiprozessorbetrieb**

Thread Implementierung I

Die Implementierung von Threads kann entweder vollständig im Benutzermodus – ohne Einbeziehung des Betriebssystems – oder durch das Betriebssystem realisiert werden. Beide Ansätze haben Vor- und Nachteile; sie sind in Fig. 5 schematisch gegenübergestellt.

Es muß zumindest die folgende Funktionalität zur Verfügung gestellt werden:

- *thread_create(tid,f)*
  Generierung eines neuen Threads zur Ausführung der Funktion f. Die Identifikation des Thread wird über *tid* an den generierenden Thread zurückgegeben.
- *thread_exit()*
  Beendigung eines Threads. Nach Abschluß dieser Aktion ist der Thread nicht mehr existent.
- *thread_wait(tid)*
  Warten auf die Beendigung des durch *tid* identifizierten Threads.
- *thread_yield()*
  Freiwillige Aufgabe des Prozessors durch einen aktiven Thread.

Weitere Funktionalität wird in der Regel für wechselseitigen Ausschluß und Zustandssynchronisation sowie Scheduling angeboten. Darauf gehen wir an dieser Stelle nicht ein.

Thread Implementierung II: Benutzermodus

Die Implementierung von Threads im Benutzermodus erfordert keinerlei spezifische Funktionalität des Betriebssystems: Threads werden auf der Basis eines **Laufzeitsystems (run time system)** realisiert, das eine Bibliothek von Routinen zur Verfügung stellt, welche die erforderliche Funktionalität bereitstellen. Alle diese Aufrufe werden im Benutzermodus ausgeführt.

Zu jedem Prozeß gehört eine spezifische **Threadtabelle**, die vom Laufzeitsystem verwaltet wird. Sie entspricht der Prozeßtabelle des Betriebssystems, mit den thread-spezifischen Modifikationen. Das Laufzeitsystem ordnet den zentralen Prozessor solange lokale Threads zu, bis es entweder keinen rechenwilligen Thread mehr gibt oder das Betriebssystem dem Prozeß den Prozessor entzieht.

Die **Vorteile** einer solchen Implementierung können wie folgt zusammengefaßt werden:

- **Keine Betriebssystem-Funktionalität erforderlich**.
- **Hohe Effizienz**: Threadmanipulationen werden über normale Prozeduraufrufe abgewickelt; auch das Thread-Scheduling wird ohne Eingriff des Betriebssystems abgehandelt.
- **Flexibilität**: Jeder Prozeß kann im Prinzip seinen eigenen Schedulingalgorithmus entwickeln.

Thread Implementierung III: Benutzermodus

Die **Nachteile** dieses Schemas sind:

- **Blockierungen**
  Die Implementierung von Systemaufrufen zur Blockierung eines Prozesses stößt auf Schwierigkeiten: angenommen, ein Thread eines Prozesses muß auf eine Eingabe warten. Wie kann man vermeiden, daß auch die anderen, arbeitsfähigen Threads des Prozesses blockiert werden?

- **Seitenfehler**
  Das Problem beim Auftretens eines Seitenfehlers in einem Thread ist ähnlich dem Blockierungsproblem: wie kann man vermeiden, daß das Betriebssystem in diesem Fall nicht nur den Thread, sondern den ganzen Prozeß blockiert?

- **Unkooperative Threads**
  Es gibt keine Mechanismen, die Benutzung des zentralen Prozessors durch einen Thread zu begrenzen, wenn Threads dies nicht freiwillig, durch Aufruf von *thread_yield*, tun.

Thread Implementierung IV: Systemmodus

Bei der Implementierung von Threads im Systemmodus verwaltet das Betriebssystem die Menge aller in Prozessen generierten Threads: es enthält also zusätzlich zur Prozeßtabelle auch eine (globale) Threadtabelle.

Alle Aufrufe zur Threadmanipulation werden dann über das Betriebssystem abgewickelt. Das bedeutet jedoch nicht notwendig, daß die Blockierung eines Threads die Blockierung des zugehörigen Prozesses zur Folge hätte: das Betriebssystem entscheidet, ob es den Prozessor einem anderen Thread des gleichen Prozesses zuordnet oder einem anderen rechenwilligen Prozeß den Vorzug gibt.

Bei dieser Art der Implementierung ist es nicht erforderlich, neue, nichtblockierende Systemaufrufe einzuführen. Sowohl die Behandlung der Blockierung eines Threads als auch Seitenfehler können relativ einfach flexibel abgehandelt werden.

Der wichtigste Nachteil dieses Ansatzes liegt in den höheren Kosten für die Threadmanipulation. Es gibt mehrere Versuche, dieses Problem zu beheben: einer davon realisiert einen *hybriden Ansatz*: hier wird zwischen zwei Klassen von Threads differenziert. *System-Threads* werden wie Threads im Systemmodus behandelt. Daneben wird die Möglichkeit geschaffen, über jeden System-Thread eine Menge von Benutzer-Threads im Multiplexbetrieb zu generieren.

Kapitel 3: Berechnungsschemata

3.1 Struktur von Berechnungsschemata

Definition: Ein **Berechnungsschema**, $\mathcal{B}$, ist ein Tupel

$$\mathcal{B}= (P, V, \Omega, \sigma_0, \Sigma_{end}, F)$$

wobei

- P: Menge der **Akteure**. P ist nichtleer und endlich.
- V: Menge der **Zustandsvariablen**. V ist nichtleer und endlich.
- Ω: **Wertebereich** für die Variablen in V.
- σ_0: **Anfangszustand**.
- Σ_{end}: Menge der **Endzustände**.
- F: Menge der **Aktionsfunktionen**. □

Komponenten von Berechnungsschemata

3.1.1 Akteure stellen die aktiven Komponenten eines Schemas dar. Sie entsprechen also – je nach modellierter Abstraktionsebene – zum Beispiel den Prozessen eines Prozeßsystems, oder den Threads in einem Prozeß.

3.1.2 Zustandsvariablen sind entweder *lokal* – mit einem spezifischen Akteur assoziiert – oder *global*. Globale Variablen können von allen Akteuren gelesen und geschrieben werden, während die lokalen Variablen eines Akteurs nur durch diesen Akteur zugreifbar sind. Bei der Modellierung von Threads auf der Ebene der Maschineninstruktionen, zum Beispiel, ist der Befehlszähler eines Threads eine lokale Variable des Threads. Die Menge der Zustandsvariablen läßt sich demnach wie folgt strukturieren:

$$V = \bigcup_{p \in P} L(p) \cup G$$

wobei

- G: Menge der **globalen Variablen**
- $L(p)$: Menge der **lokalen Variablen von Akteur** $p \in P$

Schließlich definieren wir die **Menge der von Akteur** p **zugreifbaren Variablen** mit $Z(p) := G \cup L(p)$.

3.1.3 Wertebereich und Zustände

Wir treffen keine Einschränkungen über Werte, die von Variablen angenommen werden können. Insbesondere nehmen wir an, daß es eine *universelle Wertmenge*, Ω, gibt, in der alle relevanten Werte enthalten sind. Auf dieser Basis definieren wir einen **Zustand** von V als eine Funktion $\sigma : V \to \Omega$. Die **Menge aller Zustände** bezeichnen wir mit Σ. Zustände sind *partielle* Funktionen: falls für eine Variable, $v \in V$, und einen Zustand, $\sigma \in \Sigma$, gilt $v \in DEF(\sigma)$, dann ist v in σ *definiert* und besitzt den *Wert* $\sigma(v)$; andernfalls ist v in σ *nicht definiert*, und wir schreiben in diesem Fall auch $\sigma(v) = nil$. Der **Anfangszustand** des Schemas ist mit σ_0 bezeichnet.

3.1.4 Endzustände

Ein Akteur, $p \in P$, kann entweder endliche Abläufe generieren oder zyklisch definiert sein. Im ersten Fall existiert eine nichtleere Teilmenge von **Endzuständen** des Akteurs, $\Sigma^p_{end} \subseteq \Sigma$; im zweiten Fall ist $\Sigma^p_{end} = \phi$.

Σ_{end} ist die **Menge aller Endzustände**, und es gilt

$$\Sigma_{end} = \bigcup_{p \in P} \Sigma^p_{end}$$

3.1.5 Aktionsfunktionen

Die Menge F enthält genau eine **Aktionsfunktion** für jeden Akteur $p \in P$. Aktionsfunktionen beschreiben Mengen von Zustandstransformationen, $f : \Sigma \to \Sigma$, für die von Akteuren bewirkten Aktionen. Diese Formulierung läßt sich noch wie folgt präzisieren:

Die Aktionsfunktion eines Akteurs p darf nur von den Variablen in $Z(p)$ abhängen, und auch nur diese Variablen verändern.

Aktionsfunktionen sind inhärent partiell. Ist ein Zustand $\sigma \in \Sigma$ gegeben, und ist f die Aktionsfunktion für den Akteur $p \in P$, dann definieren wir:

$$f \text{ bzw. } p \text{ ist } \textbf{anwendbar in } \sigma \text{ gdw } \sigma \in DEF(f)$$

Falls $\sigma \notin DEF(f)$, dann sind zwei Fälle zu unterscheiden:

- $\sigma \in \Sigma^p_{end}$: in diesem Fall ist p in σ **beendet**. Ist ein Akteur in einem Zustand beendet, dann ist er weder in diesem Zustand noch in irgendeinem seiner Nachfolgezustände anwendbar.
- $\sigma \in \Sigma - \Sigma^p_{end}$: in diesem Fall nennen wir p bzw. die Aktionsfunktion in σ **blockiert**.

3.2 Aktionen und Aktionsfolgen

Sei $\sigma \in \Sigma$ beliebig gewählt.
Die **Transitionsmenge**, $\Gamma(\sigma)$, ist die Menge der in σ anwendbaren Akteure.

- Ist $\Gamma(\sigma) = \phi$, dann ist in σ kein Akteur anwendbar.
- Ist $\Gamma(\sigma) \neq \phi$, dann sind einer oder mehrere Akteure anwendbar. Für die nächste auszuführende Aktion wird **genau ein Akteur** ausgewählt. Sei $p \in P$ mit Aktionsfunktion $f \in F$ beliebig gewählt. Dann ist durch σ und f eine **Aktion** definiert, die wir in der Form $\sigma \xrightarrow{p} \sigma'$ notieren. Der durch diese Aktion bewirkte **Zustandsübergang** ist durch die Festlegung $\sigma' := f(\sigma)$ spezifiziert. Aktionen sind **atomar**.

Ist die Transitionsmenge des neuen Zustands, σ', wieder nichtleer, so läßt sich dieses Verfahren fortsetzen und es entsteht eine Aktionsfolge, die eine entsprechende Zustandsfolge generiert. Erreicht man einen Zustand mit leerer Transitionsmenge, dann bricht das Verfahren ab.

Wir präzisieren nun in der Folge den Begriff der Aktionsfolge und damit zusammenhängende Konzepte und führen eine formale Notation ein.

Sei $\sigma \in \Sigma$ beliebig gewählt. Ein Wort über P, $\alpha \in P^*$, ist eine **Aktionsfolge in** σ, die bei **Anwendung auf** σ **den Zustand** σ' **erzeugt** gdw die folgenden Bedingungen erfüllt sind:

1. $\alpha = \epsilon$ und $\sigma' = \sigma$, oder
2. $\alpha = \alpha_1 p$ mit $\alpha_1 \in P^*$ und $p \in P$ mit Aktionsfunktion f, wobei gelten muß:
   (a) α_1 ist eine Aktionsfolge in σ,
   (b) σ_1 sei der durch Anwendung von α_1 auf σ erzeugte Zustand,
   (c) $p \in \Gamma(\sigma_1)$, und
   (d) $\sigma' = f(\sigma_1)$.□

Wenn α eine Aktionsfolge in σ ist, dann schreiben wir für den durch die Anwendung von α auf σ bewirkten Zustandsübergang

$$\sigma \xrightarrow{\alpha} \sigma'$$

und verwenden in Folge alle Varianten dieser Notation, die aus der Theorie der Syntax bekannt sind. Die Menge der in einer Aktionsfolge, α, auftretenden Akteure bezeichnen wir mit $P(\alpha)$.

Ein Zustand, σ', ist von einem Zustand, σ, **erreichbar**, wenn es ein Aktionsfolge, α, in σ gibt, so daß $\sigma \xrightarrow{\alpha} \sigma'$. Die Menge aller von σ aus **erreichbaren Zustände** bezeichnen wir mit $E(\sigma)$. Alle Zustände, die durch eine Aktionsfolge der Länge 1 erreichbar sind, heißen **unmittelbare Folgezustände** von σ.

Eine Aktionsfolge im Anfangszustand σ_0 heißt eine **Berechnung** des Schemas $\mathcal{B}$. Eine Berechnung ist **vollständig**, wenn sie nicht fortsetzbar ist.

Man kann das Schema $\mathcal{B}= (P, V, \Omega, \sigma_0, \Sigma_{end}, F)$ als Spezifikation einer idealisierten virtuellen Maschine, $M^{\mathcal{B}}$, auffassen, die sich wie folgt charakterisieren läßt. Für diese Diskussion nehmen wir an $P = \{p_1, \ldots, p_n\}$:

1. $M^{\mathcal{B}}$ besitzt n **virtuelle Prozessoren**, die den Akteuren $p_1, \ldots, p_n$ entsprechen. Diese Prozessoren können als "Instruktionen" Aktionen ausführen, deren Eigenschaften durch die Aktionsfunktionen spezifiziert sind.

2. $M^{\mathcal{B}}$ besitzt einen **Speicher**, der durch die Menge, V, der Zustandsvariablen und den Wertebereich, Ω, charakterisiert ist. Durch σ_0 ist der Anfangszustand des Speichers gegeben.

3. Das **Verhalten** von $M^{\mathcal{B}}$ läßt sich als eine lineare Folge von Aktionen in der Zeit beschreiben. Die Aktionen treten nur zu *diskreten Zeitpunkten* auf, zwischen denen nichts geschieht: sei $\sigma_1 \ldots \sigma_k \ldots$ eine Zustandsfolge, die durch eine Aktionsfolge, α, in σ_1 erzeugt wird, und $t_1 \ldots t_k \ldots$ eine Folge von Zeitpunkten des Zeitrasters, das zur gewählten Abstraktionsebene gehört. Dann kann man von der *Aktion zum Zeitpunkt* t_j sprechen (nämlich $\alpha(j)$) bzw. vom *Zustand zum Zeitpunkt* t_j (nämlich σ_j), aber **nicht** von einer Aktion oder einem Zustand zu irgendeinem Zeitpunkt t, mit $t_{j-1} < t < t_j$ für ein j.

Aktionen sind atomar, und die Aktionen jedes virtuellen Prozessors und der Maschine selbst laufen sequentiell ab. Konkurrente Abläufe können dadurch modelliert werden, daß aufeinanderfolgende Aktionen von unterschiedlichen virtuellen Prozessoren ausgeführt werden. □

Betrachten wir nun die Aktivität eines einzelnen virtuellen Prozessors, $p \in P$, in einer Aktionsfolge, α. Diese Aktivität ist durch die Teilfolge von α charakterisiert, in der nur p auftritt: sie wird als **der dem Akteur p in α zugeordnete Prozeß** des Berechungsschemas bezeichnet. Entsprechend bezeichnen wir die Aktivität aller Prozesse bei der Ausführung von α als das durch α erzeugte **Prozeßsystem**.

Wir sprechen manchmal von der **Geschwindigkeit** eines Prozesses in einem Prozeßsystem. Darunter verstehen wir die relative Häufigkeit des Auftretens des Prozesses in der Aktionsfolge. **Eine Lösung für ein Synchronisationsproblem ist nur dann korrekt, wenn sie nicht von der Geschwindigkeit der beteiligten Prozesse anhängt.**
In der Folge werden wir die Begriffe "Akteur" und "Prozeß" in der Regel synonym verwenden. Aus dem Kontext geht dann jeweils hervor, ob die *statische* Bedeutung (im Sinne der Beschreibung einer Aktionsfunktion), oder die *dynamische* Interpretation im Sinne eines Ablaufs gemeint ist.

3.3 Programmierte Schemata

In den meisten Fällen spezifizieren wir Berechnungsschemata durch Programme in Pseudocode. Wir sprechen dann von **programmierten Schemata**. Hierzu müssen einige Konventionen vereinbart werden, die wir in diesem Abschnitt diskutieren.

In vielen Fällen benutzen wir Berechnungsschemata, um die Aktionen von Ein- oder Mehrprozessorsystemen auf der Ebene der Instruktionen von zentralen Prozessoren zu modellieren. Die globalen Variablenvereinbarungen definieren dann die Menge der Zustandsvariablen, zu denen noch lokale Variablen der Akteure hinzukommen, die entweder explizit deklariert sind oder interne Register wie die *Kontrollvariablen* (die den Befehlzählern entsprechen) darstellen. Die Werte der Kontrollvariablen sind explizite Marken von Anweisungen bzw. eine spezielle Endemarke (L_end) für endliche Akteure.

Ohne spezielle Voraussetzungen über die Unteilbarkeit zusammengesetzter Operationen (wie z.B. Semaphoroperationen oder **fetch_and_add**) können als Aktionen dann nur solche Operationen definiert werden, in denen höchstens eine globale Variable gelesen oder geschrieben wird (nicht beides). Dies bedeutet, daß die Inkrementierung einer globalen Variablen, etwa $x = x+1$, als **Folge von zwei Aktionen – also nicht als eine einzige Aktion – dargestellt werden muss**:

$$\mathrm{h} = \mathrm{x};\ \mathrm{x} = \mathrm{h} + 1$$

Hier ist h eine lokale Variable (zum Beispiel ein Register), deren Inkrementierung durch eine Aktion lokaler Natur dargestellt werden kann.

Beispiel:

```
begin schema
var x=0: integer; -- globale Variable
parbegin
actor p1= {
  var h1: integer; -- lokale Variable von Akteur 1
 L1_1: h1=x;
 L1_2: x=h1+1
};
actor p2={
  var h2: integer; -- lokale Variable von Akteur 2
 L2_1: h2=x;
 L2_2: x=h2+1
}
parend
end schema
```

Durch dieses Programm ist folgendes Berechnungsschema spezifiziert: $\mathcal{B}= (P, V, \Omega, \sigma_0, \Sigma_{end}, F)$, mit:

- $P = \{p1, p2\}$
- $V = (x, h1, \beta_1, h2, \beta_2)$
  x ist die einzige globale Variable; alle anderen Variablen sind lokal, wobei β_1 bzw. β_2 die Kontrollvariablen von $p1$ bzw. $p2$ bezeichnet. Damit ist $G = \{x\}$ und die von den Akteuren zugreifbaren Variablen sind wie folgt gegeben:

- $Z(p1) = \{x, h1, \beta_1\}$
- $Z(p2) = \{x, h2, \beta_2\}$

- $\Omega = \{0, 1, 2, L1\_1, L1\_2, L1\_end, L2\_1, L2\_2, L2\_end\}$

-
  - $\sigma_0(x) = 0$
  - $\sigma_0(\beta_1) = L1\_1$
  - $\sigma_0(\beta_2) = L2\_1$
  - $\sigma_0(v) = nil$ für $v \in V - \{x, \beta_1, \beta_2\}$

- $F = \{f1, f2\}$, mit
  - $f1(\beta_1 = L1\_1) = \{h1 = x; \beta_1 = L1\_2\}$
    $f1(\beta_1 = L1\_2) = \{x = h1 + 1; \beta_1 = L1\_end\}$
  - $f2(\beta_2 = L2\_1) = \{h2 = x; \beta_2 = L2\_2\}$
    $f2(\beta_2 = L2\_2) = \{x = h2 + 1; \beta_2 = L2\_end\}$

Die Notation für die Spezifikation der Aktionsfunktionen bedarf noch einer Erläuterung. Aktionsfunktionen sind Abbildungen $f : \Sigma \to \Sigma$, wobei nur diejenigen Zustände in Betracht kommen, die zu globalen Variablen und den lokalen Variablen des Akteurs gehören. Die Zeile

$$f1(\beta_1 = L1\_1) = \{h1 = x; \beta_1 = L1\_2\}$$

ist eine komprimierte Version der folgenden Spezifikation:

$$f1(x, h1, \beta_1 = L1\_1) = (x, h1 = x, \beta_1 = L1\_2)$$

welche besagt, daß Akteur 1, wenn seine Kontrollvariable auf die erste Anweisung ($L1\_1$) zeigt, erstens $h1$ durch Zuweisung des Werts von x definiert, und zweitens die Kontrollvariable auf $L1\_2$ setzt, um die Ausführung der nächsten Aktion zu ermöglichen. **Diese Aktion hängt ausschließlich von β_1 ab und ändert ausschließlich die Variablen β_1 und $h1$.**

In der Folge werden wir die Notation noch dahingehend vereinfachen, daß lokale Variablen, die aus dem Kontext als solche erkannt werden können, in einem programmierten Schema nicht indiziert werden. Wo notwendig, wird die Indizierung bei Spezifikation der Aktionsfunktion hinzugefügt. Außerdem fassen wir Klassen von "ähnlichen" Akteuren in einer Spezifikation zusammen. Das obige Schema läßt sich dann äquivalent wie folgt darstellen.

```
begin schema
var x=0: integer; -- globale Variable
parbegin
actor pj (j ∈ {1,2}) = {
  integer h;
  L1: h=x;
  L2: x=h+1
};
parend
end schema
```

In dem betrachteten Schema gibt es 6 vollständige Berechnungen. Diese sind, zusammen mit den zugehörigen Wertfolgen für die globale Variable x, unten aufgelistet (wir spezifizieren den Akteur lediglich durch seinen Index, also 1 oder 2):

1. $\alpha_1 = 1122$ $x = 0, 0, 1, 1, 2$
2. $\alpha_2 = 1212$ $x = 0, 0, 0, 1, 1$
3. $\alpha_3 = 1221$ $x = 0, 0, 0, 1, 1$
4. $\alpha_4 = 2112$ $x = 0, 0, 0, 1, 1$
5. $\alpha_5 = 2121$ $x = 0, 0, 0, 1, 1$
6. $\alpha_6 = 2211$ $x = 0, 0, 1, 1, 2$

Unter der Annahme, daß *beide* Akteure den Wert von x um 1 erhöhen möchten, erzielen also nur α_1 und α_6 das gewünschte Resultat, während in den anderen Aktionsfolgen aufgrund der Überlappung der Aktionen eine der Inkrementierungen verloren geht. Der Grund für das Problem liegt darin, daß die Kombination von Lesen und Inkrementieren einer Variablen keine unteilbare Aktion auf der Basis von gewöhnlichen Lese- und Schreiboperationen auf Speicherzellen darstellt.

Damit ist das obige Schema *keine* korrekte Implementierung dieser Funktion. Ein Ansatz zur Lösung des Problems wird im Kontext der Diskussion des wechselseitigen Ausschlusses beschrieben.

Kapitel 4: Synchronisation

Synchronisation wird in Systemen mit gemeinsamem Speicher benötigt, wenn die Aktionen konkurrenter Prozesse in eine gewisse Reihenfolge gebracht werden müssen, um Konflikte bei Zugriff auf Betriebsmittel zu vermeiden. Man unterscheidet zwischen wechselseitigem Ausschluß und Zustandssynchronisation.

Beim **wechselseitigen Ausschluß (mutual exclusion)** werden die exklusiven Zugriffe von Prozessen zu einem Betriebsmittel koordiniert, ohne daß die Reihenfolge dieser Zugriffe interessiert, solange gewährleistet ist, daß jeder Prozeß nach endlicher Zeit Zugriff erhält.

Bei der **Zustandssynchronisation (synchronization)** macht ein Prozeß sein Fortschreiten von der Gültigkeit einer Bedingung in seiner Umgebung abhängig. Im Gegensatz zum wechselseitigen Ausschluß kann durch Zustandssynchronisation die Reihenfolge der Aktionen von Prozessen genau festgelegt werden.

4.1 Wechselseitiger Ausschluss

4.1.1 Problemstellung

Wir betrachten ein Berechnungsschema, in dem $n \geq 2$ sonst unabhängige Akteure auf ein gemeinsam benutztes wiederverwendbares und nicht-entziehbares Betriebsmittel, B, exklusiv in einem **kritischen Abschnitt** zugreifen. Die Koordination der Zugriffe soll *dezentral* erfolgen, mit Hilfe einer von globalen *Testvariablen*. Die Reihenfolge, in der Prozessen das Betriebsmittel zugeteilt wird, ist irrelevant; jedoch muß ein korrektes Schema garantieren, daß jeder Prozeß, der einen Zugriff anfordert, diesen auch nach endlicher Zeit realisieren kann. Des weiteren darf sich ein Prozeß, der Zugriff zu B erhält, nur endlich lange in seinem kritischen Abschnitt aufhalten.

Die Programmstruktur ist für alle Akteure identisch und in Fig. 6 dargestellt. Akteure sind zyklisch und durchlaufen sukzessive vier Abschnitte:

Im **Testabschnitt** prüft ein Prozeß durch Abfrage der Werte von Testvariablen, ob er den kritischen Abschnitt betreten darf oder warten muß.

Im **kritischen Abschnitt (critical section)** kann auf B zugegriffen werden. Ein Prozeß darf nur endlich lange im kritischen Abschnitt bleiben.

Der **Endabschnitt** dient dazu, den anderen Prozessen durch entsprechendes Setzen der Testvariablen die Beendigung des Zugriffs auf B anzuzeigen.

Bezüglich des **Restabschnitts** wird lediglich vereinbart, daß dort weder Zugriffe auf B noch auf die Testvariablen vorgenommen werden.

Folgende Bedingungen müssen für die Korrektheit einer Lösung des Problems gelten:

1. **Sicherheit (safety)**: Zu keinem Zeitpunkt darf sich mehr als ein Prozeß in seinem kritischen Abschnitt befinden.

   Es darf also keine Berechnung geben, die zu einem Zustand führt, in der sich zwei Prozesse im kritischen Abschnitt befinden. Ein solcher Zustand wird ein **verbotener Zustand** genannt.

2. **Behinderungsfreiheit (fairness)**: Jeder Prozeß, der im Testabschnitt wartet, erhält nach endlicher Zeit Zugriff zum kritischen Abschnitt.

   Dies bedeutet, daß **jede** Fortsetzung einer Berechnung, die einen Prozeß in seinen Testabschnitt gebracht hat, zu einem Zustand führen muß, in dem dieser Prozeß sich im kritischen Abschnitt befindet.

Einige der einfachen Beispiele, die früher diskutiert wurden (Inkrementierung einer globalen Variablen) deuten darauf hin, daß ein naiver Ansatz zur Lösung des Problems, etwa nach folgendem Muster, fehlschlägt (warum?):

```
begin schema
var frei=true: boolean  -- globale Testvariable
parbegin
  actor pj (j ∈ 1..n) = {
    while true {
      T1: if ¬ frei then goto T1; -- Testabschnitt
      T2: frei=false;             -- Testabschnitt
      K:  -- kritischer Abschnitt --
      A:  frei=true;              -- Endabschnitt
      R:  -- Restabschnitt --
    }
  }
parend
end schema
```

Dies bedeutet jedoch nicht, daß das Problem des wechselseitigen Ausschlusses auf der Basis der Unteilbarkeit von Lese- und Schreiboperationen nicht lösbar ist: tatsächlich haben Dekker, Dijkstra, und Knuth eine korrekte Lösung vorgeschlagen. Diese ist jedoch nur von theoretischem Interesse, da sie **aktives Warten** der Prozesse in ihrem Testabschnitt erfordert: ähnlich wie bei dem (falschen) Ansatz oben muß ein Prozeß im allgemeinen in seinem Testabschnitt eine Schleife mit wiederholten Abfragen der Werte der Testvariable(n) durchlaufen.

Eine praktikable Lösung des Problems läßt sich mit Hilfe der von Dijkstra vorgeschlagenen **Semaphorvariablen** realisieren. Dies sind ganzzahlige Variablen, die nur über spezielle Methoden manipuliert werden können. Eine Besonderheit einer Methode (**wait**) ist, daß sie einen Test mit einer Änderung des Variablenwerts zu einer unteilbaren Operation verbindet.

4.1.2 Semaphore

Eine Variable, s, eines Berechnungsschemas ist eine **Semaphorvariable** gdw die folgenden Bedingungen erfüllt sind:

1. Der Wertebereich von s ist $\mathbf{N}_0$ (die Menge der natürlichen Zahlen unter Einschluß von 0)
2. Auf s lassen sich drei Operationen ausführen: Initialisierung, **wait** , und **signal** .
   (a) Vor der ersten Anwendung von **wait** oder **signal** muß s mit einem Wert aus $\mathbf{N}_0$ **initialisiert** werden.
   (b) Sei α eine Berechnung, und $\sigma_0 \xrightarrow{\alpha} \sigma$. Weiters sei p ein Akteur, dessen nächste auszuführende Aktion in σ **wait** (s) ist. Dann gilt:
      i. $p \in \Gamma(\sigma) \Leftrightarrow \sigma(s) \geq 1$
      ii. Sei $\sigma(s) \geq 1$, p für die Ausführung der nächsten Aktion ausgewählt, und $\sigma \xrightarrow{p} \sigma'$. Dann ist $\sigma'(s) = \sigma(s) - 1$.
   (c) Seien α und σ wie oben gegeben, und sei p ein Akteur, dessen nächste auszuführende Aktion in σ **signal** (s) ist. Dann gilt:
      i. $p \in \Gamma(\sigma)$
      ii. Sei p für die Ausführung der nächsten Aktion ausgewählt, und $\sigma \xrightarrow{\alpha} \sigma'$. Dann ist $\sigma'(s) = \sigma(s) + 1$.

(d) Die Strategie der Auswahl unter mehreren zu einem Zeitpunkt anwendbaren **wait** Operationen darf nicht zu Behinderungen führen. (Dies wird in der Regel bei der Implementierung von Semaphoroperationen dadurch erzwungen, daß bei der Freigabe blockierter Prozesse eine FIFO-Strategie angewandt wird.)

Mit Semaphoren läßt sich nun das Problem des wechselseitigen Ausschlusses auf einfache Weise lösen:

```
begin schema
var s=1: sema;                    -- globale Semaphorvariable
parbegin
  actor pj (j ∈ 1..n) = {
    while true {
      T: wait(s);                 -- Testabschnitt
      K: ...                      -- kritischer Abschnitt
      A: signal(s);               -- Endabschnitt
      R: ...                      -- Restabschnitt
    }
  };
parend
end schema
```

Bei genauerer Betrachtung stellt sich heraus, daß wir das Problem des wechselseitigen Ausschlusses durch die Einführung von Semaphorvariablen nicht wirklich gelöst, sondern lediglich auf eine niedrigere Abstraktionsebene transferiert haben: denn wie läßt sich die Verbindung von Test und Dekrementierung, die durch **wait** realisiert ist (dies ist *eine* Aktion) in der Praxis realisieren? Dazu wird Unterstützung der Hardware, konkreter des Hauptspeichers für eine entsprechende

Funktionalität benötigt. Zum Beispiel kann der Speicher zusätzlich zu den gewöhnlichen Lade- und Speichere-Operationen noch die Operationen **lock** m und **unlock** m bereitstellen, wobei m eine Speicherzelle ist:

- **lock** m: $L : [\mathbf{if}\, m = 1\, \mathbf{then}\, m = 0\, \mathbf{else\ goto}\, L]$
- **unlock** m: [m=1]

Der Einschluß einer Anweisungsfolge in eckige Klammern bedeutet hier, daß diese Folge als unteilbare Operation auszuführen ist.

Damit ist das Problem auf eine weitere, niedrigere Abstraktionsebene abgeschoben (Speicherlogik), mit der wir uns hier aber nicht weiter befassen wollen.

4.2 Zustandssynchronisation

Das Problem der **Zustandssynchronisation** tritt in Systemen kooperierender Prozesse auf. Zustandssynchronisation bedeutet, daß ein Prozeß sein Fortschreiten vom Vorliegen einer bestimmten Bedingung in seiner Umgebung, der **Synchronisationsbedingung**, abhängig macht. Da über die relativen Geschwindigkeiten der Prozesse keine Angaben gemacht werden, muß eine solche Abhängigkeit explizit in den Aktionen der beteiligten Akteure zum Ausdruck kommen. Zustandssynchronisation bewirkt, daß für bestimmte Teilfolgen von Aktionen eine Ordnung erzwungen wird, die – anders als beim wechselseitigen Ausschluß – genau festgelegt werden kann.

4.2.1 Zustandssynchronisation mit Semaphoren

Als **Erzeuger/Verbraucher Problem (producer/consumer problem)** bezeichnet man eine Klasse von Synchronisationsproblemen, die sich wie folgt charakterisieren lassen:

Gegeben sei ein Prozeßsystem mit zwei zyklischen Prozessen: einem **Erzeuger** und einem **Verbraucher**. In jedem Zyklus des Erzeugers wird eine Nachricht produziert; in jedem Zyklus des Verbrauchers wird eine Nachricht verarbeitet. Die Nachrichten werden vom Verbraucher in der gleichen Reihenfolge verarbeitet, in der sie vom Erzeuger produziert werden. Über einen langen Zeitraum gesehen, ist die Anzahl der in den beiden Prozessen durchlaufenen Zyklen annäherungsweise gleich; in kürzeren Zeitintervallen können die Geschwindigkeiten von Erzeuger und Verbraucher jedoch variieren: so kann es etwa sein, daß der Erzeuger eine Nachricht zu einem

Zeitpunkt produziert, zu dem der Verbraucher noch nicht in der Lage ist, sie zu verarbeiten. Es ist daher sinnvoll, Nachrichten nicht *direkt* von Erzeuger zum Verbraucher zu senden, sondern sie in einem **Puffer** zwischenzuspeichern (Fig. 7).

Der Puffer bestehe aus $q \geq 1$ **Pufferelementen** gleicher Größe, von denen jedes genau eine Nachricht aufnehmen kann. Ein Pufferelement heißt **voll** gdw in ihm vom Erzeuger eine Nachricht abgelegt wurde, die noch nicht vom Verbraucher aus dem Puffer entfernt wurde, sonst **leer**. Der Puffer heißt voll (leer), wenn *alle* Pufferelemente voll (leer) sind. Erzeuger und Verbraucher arbeiten konkurrent. Es gibt zwei Situationen, in denen eine Synchronisation stattfinden muß:

1. **EV1:** der Erzeuger kann nicht in den vollen Puffer schreiben
2. **EV2:** der Verbraucher kann nicht aus dem leeren Puffer lesen

Im Falle eines vollen Puffers muß das Schreiben des Erzeugers so lange verzögert werden, bis der Verbraucher die nächste Nachricht gelesen hat und dadurch ein leeres Pufferelement entsteht. Im Falle des leeren Puffers ist das Lesen des Verbrauchers so lange zu verzögern, bis der Erzeuger eine neue Nachricht produziert hat und dadurch ein volles Pufferelement entsteht.

Man beachte die Dualität der beiden Akteure: der Erzeuger "erzeugt" volle und "verbraucht" leere Pufferelemente; der Verbraucher "erzeugt" leere und "verbraucht" volle Pufferelemente. Dies führt zur Symmetrie in der in Fig. 8 angegebenen Lösung. Die Erzeugung von Nachrichten wird durch den Aufruf der Funktion *erzeuge*, der Verbrauch durch den Aufruf der Prozedur *verbrauche* dargestellt. Die einzige Voraussetzung hier ist die endlich lange Dauer dieser Aufrufe.

Wir überlegen uns nun, daß Fig. 8 eine korrekte Lösung des Erzeuger/Verbraucher Problems darstellt. Dazu ist zu zeigen, daß die Bedingungen (EV1) und (EV2) erfüllt sind, die allgemeinen Korrektheitsbedingungen Deadlockfreiheit und Behinderungsfreiheit erfüllt sind, und Erzeuger und Verbraucher nicht gleichzeitig zum selben Pufferelement zugreifen können:

Erzeuger und Verbraucher produzieren bzw. verbrauchen in jeder Iteration der **while**-Anweisung genau eine Nachricht. Die Werte der lokalen Variablen e bzw. v geben zu jedem Zeitpunkt die Anzahl der vollständig ausgeführten Iterationen des Erzeugers bzw. Verbrauchers an. Nun kann man leicht einsehen, daß in jedem erreichbaren Zustand σ die folgenden Bedingungen gelten (β^E, β^V sind die Kontrollvariablen des Erzeugers bzw. Verbrauchers):

1. $\sigma(v) \leq \sigma(e) \leq \sigma(v) + q$
2. $\sigma(v) < \sigma(e) \Rightarrow Verbraucher \in \Gamma(\sigma)$
3. $\sigma(e) < \sigma(v) + q \Rightarrow Erzeuger \in \Gamma(\sigma)$
4. $\sigma(v) = \sigma(e) \wedge \sigma(\beta^E) \in \{L1, L2, L3, L4\} \Rightarrow Verbraucher \notin \Gamma(\sigma)$
5. $\sigma(v) = \sigma(e) + q \wedge \sigma(\beta^V) \in \{L1, L2, L3\} \wedge \sigma(\beta^E) = L2 \Rightarrow Erzeuger \notin \Gamma(\sigma)$

Nun folgt aus (4) und (5) unmittelbar die Erfüllung der Synchronisationsbedingungen (EV1) und (EV2); aus (2) und (3) folgt die Deadlockfreiheit, und aus (1) die Behinderungsfreiheit. Es bleibt noch zu zeigen, daß Erzeuger und Verbraucher nicht gleichzeitig zum selben Pufferelement zugreifen können (man beachte, daß diese Zugriffe *nicht* in einem kritischen Abschnitt einge-

schlossen wurden). Wir überlegen uns dies indirekt: angenommen, es gäbe einen erreichbaren Zustand σ mit

$$\sigma(\beta^E) = L3 \wedge \sigma(\beta^V) = L2 \wedge (\sigma(e) \bmod q = \sigma(v) \bmod q)$$

Dann folgt aus (1)

$$(\sigma(e) = \sigma(v)) \vee (\sigma(e) = \sigma(v) + q)$$

Wäre $\sigma(e) = \sigma(v)$, dann gälte $\sigma(\mathit{volle\_puffer\_elemente}) = \sigma(e) - (\sigma(v) + 1) = -1$.

Wäre $\sigma(e) = \sigma(v) + p$, dann $\sigma(\mathit{leere\_puffer\_elemente}) = q + \sigma(v) - (\sigma(e) + 1) = -1$.

Wir erhalten also in beiden Fällen einen Widerspruch,qed.□

Das nächste Problem, das wir mit Semaphoren lösen, modelliert sowohl exklusiven als auch gemeinsamen Zugriff zu einem Betriebsmittel B (zum Beispiel eine Datei): im **Leser/Schreiber Problem (readers/writers problem)** werden zwei Klassen von Akteuren betrachtet: **Leser** greifen nur nichtexklusiv, **Schreiber** nur exklusiv auf B zu. Während B also zu jedem Zeitpunkt höchstens einem Schreiber zugeordnet sein kann, dürfen es mehrere Leser gleichzeitig benutzen. Fig. 9 ist ein Lösungsansatz für dieses Problem mit n_1 Lesern und n_2 Schreibern. Wir überlegen uns nun, daß dieses Programm bis auf Behinderungen eine korrekte Lösung des Problems ist.

Es ist leicht zu sehen, daß niemals ein Schreiber und ein Leser gleichzeitig auf B zugreifen können: die Semaphorvariable s wird hierzu ähnlich wie eine Semaphorvariable für den wechselseitigen Ausschluß benutzt, wobei jedoch nur der erste Leser in einem Strom von Lesern **wait** (s) ausführt, um Zugriff zu B zu erhalten, und nur der letzte Leser, und zwar durch **signal** (s), den Zugriff freigibt. Die anderen Leser betreten und verlassen den Abschnitt, in dem sie auf B zugreifen, ohne Synchronisationsoperationen auszuführen.

Schreiber können behindert werden: sobald sich ein Leser Zugriff zu B verschafft hat, können andere Leser ungehindert ihren Zyklus ausführen. Wenn die relativen Geschwindigkeiten der Leser so beschaffen sind, daß zu jedem Zeitpunkt mindestens ein Leser Zugriff zu B wünscht, dann haben die Schreiber keine Chance, jemals in die Phase "SCHREIBEN" zu gelangen.

4.2.2 Bedingte Kritische Regionen

Wir wenden uns nun einem Synchronisationsmechanismus auf höherer Ebene zu, als es Semaphore sind. Bedingte kritische Regionen erlauben die explizite Formulierung von Synchronisationsbedingungen in einer programmiersprachlichen Notation.

Eine **bedingte kritische Region (conditional critical region)** bezüglich eines Betriebsmittels B hat die Gestalt

$$\textbf{with } B \textbf{ when } S \textbf{ do } K$$

wobei

- B ist ein globales Objekt des Programms, das ein wiederverwendbares Betriebsmittel repräsentiert.
- S ist die **Synchronisationsbedingung**: dies ist ein Ausdruck, der nur von B und lokalen Variablen des aktiven Prozesses abhängt.
- K ist ein kritischer Abschnitt bezüglich B: die Ausführungen verschiedener kritischer Abschnitte bezüglich B schließen einander wechselseitig aus. In K dürfen nur B und lokale Variablen des aktiven Prozesses manipuliert werden.

Ein Prozeß, der eine bedingte kritische Region ausführt, wird so lange verzögert, bis der Wert von S wahr ist. Sobald S erfüllt ist, kann der kritische Abschnitt betreten werden: die Berechnung von S bildet dann zusammen mit der Ausführung von K eine unteilbare Aktion. Ist S

nicht erfüllt, dann wird der ausführende Prozeß blockiert.

Es gibt zwei wichtige Spezialfälle für bedingte kritische Regionen:

1. $S =$ **true**: in diesem Fall wird die bedingte kritische Region einfach für die Realisierung des wechselseitigen Ausschlusses benutzt.
2. K ist leer: hier wird lediglich auf die Erfüllung der Synchronisationsbedingung gewartet.

Bedingte kritische Regionen erlauben vielfach eine elegantere Formulierung von Synchronisationsproblemen als Semaphore. Ein Problem ist, daß in der Regel die Synchronisationsbedingung eines Prozesses mehrfach berechnet werden muß, bevor der Prozeß Zugang zum kritischen Abschnitt erhält. Dieses **aktive Warten** kann die Effizienz des Prozeßsystems wesentlich beeinflussen. Ein Beispiel ist in Fig.10 gegeben.

4.2.3 Monitore

Wir haben uns bisher ausschließlich mit der *dezentralen Koordination* von Prozessen befaßt: dabei wird der Zugriff zu einem Betriebsmittel durch Anweisungen im Code der beteiligten Akteure koordiniert.

Sei B ein Betriebsmittel des Rechensystems, zu dessen Verwaltung globale Variablen $b_1, \ldots, b_m$ benötigt werden, auf die in kritischen Abschnitten zugegriffen wird. In einem solchen System lassen sich für jeden Prozeß zwei Phasen unterscheiden, je nachdem ob sich der Prozeß innerhalb eines kritischen Abschnitts befindet oder nicht. Diese Unterscheidung ist in zweierlei Hinsicht von Bedeutung:

1. **Im Hinblick auf den Zugriff zu globalen Variablen:** Nur innerhalb eines kritischen Abschnitts dürfen $b_1, \ldots, b_m$ manipuliert werden.
2. **Im Hinblick auf die Zuteilung zentraler Prozessoren:** wird einem Prozeß innerhalb des kritischen Abschnitts der Prozessor entzogen, so führt dies zur Blockierung aller anderen Prozesse, die auf B zugreifen möchten – der gesamte Verwaltungsapparat für B wird dadurch lahmgelegt. Prozesse in kritischen Abschnitten müssen daher vom Betriebssystem bevorzugt behandelt werden.

Angesichts der besonderen Rolle der kritischen Abschnitte kann es von Vorteil sein, alle kritischen Abschnitte bezüglich eines Betriebsmittels, die in Akteuren auftreten, zentral zusammenzufassen. Auf diese Weise erhält man ein Objekt, das im wesentlichen die Variablen $b_1, \ldots, b_m$ und

eine Menge von Prozeduren enthält, die diese Variablen manipulieren. Dieser Gedankengang führt zum Konzept des **Monitors**, das auf Dijkstra, Brinch Hansen, und Hoare zurückgeht.

Ein Monitor kann als eine Klasse aufgefaßt werden, die folgende Komponenten enthält:

- **lokale Variablen:** dies sind die Variablen, die zur Verwaltung des Betriebsmittels erforderlich sind (die b_i in der obenstehenden Diskussion). Sie sind von außen nicht direkt zugänglich.
- **Prozeduren:** Prozeduren dürfen nur auf die lokalen Variablen des Monitors zugreifen. Ein Monitor kann sowohl lokale als auch von außen aufrufbare Prozeduren enthalten. Letztere können von allen Prozessen aufgerufen werden.

Wir sagen, daß sich ein Prozeß **innerhalb** eines Monitors befindet, wenn er eine Prozedur des Monitors aufgerufen, diese jedoch noch nicht verlassen hat. Von einem Prozeß, der sich innerhalb des Monitors befindet *und* aktiv ist, sagen wir, daß er den Monitor **besitzt**. Die Grundregel für Monitore ist:

Zu jedem Zeitpunkt darf höchstens ein Prozeß den Monitor besitzen

Durch diese Regel wird gewährleistet, daß Monitore den wechselseitigen Ausschluß für Zugriffe auf ihre Variablen garantieren. Monitore können jedoch nicht nur zur Realisierung des wechselseitigen Ausschlusses, sondern auch für die Zustandssynchronisation verwendet werden. Dies erreicht man durch Einführung von **Bedingungsvariablen** und darauf erklärten Operationen **waitcond** und **signalcond**.

Jede Bedingungsvariable, c, repräsentiert eine Synchronisationsbedingung, $S(c)$. Prozesse können bezüglich Bedingungsvariablen blockiert werden: daher wird eine Bedingungsvariable

durch eine *Warteschlange*, $Q(c)$, realisiert, die zu jedem Zeitpunkt alle bezüglich c blockierten Prozesse enthält. Bedingungsvariablen besitzen keine Werte im üblichen Sinn.

waitcond (c): die Ausführung dieser Operation in einer von Prozeß p aufgerufenen Monitorprozedur (1) blockiert p, (2) nimmt p in $Q(c)$ auf, und (3) gibt den Monitor frei.

signalcond (c): die Ausführung dieser Operation in einer von Prozeß p aufgerufenen Monitorprozedur hat für den Fall $Q(c) \neq \phi$ folgenden Effekt: (1) Elimination des am längsten in $Q(c)$ befindlichen Prozesses, etwa p', aus $Q(c)$; (2) Entblockierung von p', und (3) Blockierung von p. Der Besitz des Monitors geht damit von p nach p' über. Prozeß p wird zum frühestmöglichen Zeitpunkt, d.h. bei der Freigabe des Monitors durch einen Prozeß, wieder entblockiert.

Ist zum Zeitpunkt der Ausführung dieser Operation $Q(c) = \phi$, dann ist der Effekt der Operation leer.

Fig. 11 und 12 illustrieren die Anwendung des Monitorkonzepts auf das Leser/Schreiber Problem. Im Gegensatz zu der Formulierung des Problems mit Semaphorvariablen ist dieser Ansatz behinderungsfrei. Die Bedingungsvariable L repräsentiert hierbei die Synchronisationsbedingung "Leser darf lesen", d.h. $arbS = 0$ und kein Schreiber wartet; die Bedingungsvariable S steht für die Synchronisationsbedingung "Schreiber darf schreiben", d.h. $arbL = 0 \wedge arbS = 0$.

Kapitel 5: Verklemmungen

5.1 Grundlegende Begriffe

Eine *Verklemmung (Deadlock)* ist ein Systemzustand, in dem mindestens ein nicht beendeter Prozeß keine Aktionen mehr ausführen kann, unabhängig von der Art, wie die Aktionsfolge weitergeführt wird.
Wir bauen im folgenden auf der in Kapitel 3 eingeführten Notation für Berechnungsschemata auf:

$$\mathcal{B}= (P, V, \Omega, \sigma_0, \Sigma_{end}, F)$$

Die Menge der Zustandsvariablen ist strukturiert:

$$V = \bigcup_{p \in P} L(p) \cup G$$

wobei die Menge der von Prozeß $p \in P$ zugreifbaren Variablen durch $Z(p) = G \cup L(p)$ gegeben ist. Des weiteren haben wir die Menge der Endzustände des Prozesses p mit $\Sigma_{end}^{p} \subseteq \Sigma$ bezeichnet.

Definition: Sei σ ein erreichbarer Zustand und $p \in P$ beliebig gewählt.
p ist **verklemmt (deadlocked)** in $\sigma :\Leftrightarrow$ für alle $\sigma' \in E(\sigma)$ gilt: p ist blockiert in σ'.$\Box$

Ein Prozeß ist also in einem Zustand verklemmt, wenn er in diesem und allen erreichbaren Folgezuständen blockiert ist. Kein anderer Prozeß hat die Möglichkeit, die Blockierung eines verklemmten Prozesses zu beenden, unabhängig davon, welche Aktionen er ausführt:

$$p \text{ ist verklemmt in } \sigma \Leftrightarrow \forall \sigma' \in E(\sigma) : p \notin \Gamma(\sigma') \wedge \sigma' \notin \Sigma^p_{end}$$
$$p \text{ ist nicht verklemmt in } \sigma \Leftrightarrow (\sigma \notin \Sigma^p_{end} \Rightarrow \exists \sigma'' \in E(\sigma) : p \in \Gamma(\sigma''))$$

Wir gehen von den folgenden Zusicherungen über Schemata aus:

1. Die Blockierung eines Prozesses kann nur von globalen Variablen abhängen.
2. Die Entscheidung, ob ein Prozeß in einem Endzustand ist, kann nur von seinen lokalen Variablen abhängen.□

Definition:

1. Ein Zustand σ ist ein **Verklemmungszustand**:$\Leftrightarrow$
   es gibt einen Prozeß $p \in P$: p ist verklemmt in σ.
2. Ein Verklemmungszustand ist **total**:$\Leftrightarrow$ alle $p \in P$ sind verklemmt in σ. □

Aus dieser Definition folgt unmittelbar: wenn σ ein Verklemmungszustand ist, dann auch jeder Zustand in $E(\sigma)$. Falls σ ein totaler Verklemmungszustand ist, dann ist $\Gamma(\sigma) = \phi$.

5.2 Verklemmungen beim Zugriff auf wiederverwendbare Betriebsmittel

Notwendig für das Auftreten von Verklemmungen beim Zugriff auf wiederverwendbare Betriebsmittel ist:

1. die Betriebsmittel sind *nichtentziehbar*, und
2. der Zugriff ist *exklusiv*.

Wir nehmen an, daß die Prozesse des Schemas abgesehen vom Zugriff auf eine Menge globaler Betriebsmittel unabhängig voneinander sind. Folgende Grundregeln werden vorausgesetzt:

1. Die **Betriebsmittelkonstellation** – vorhandene Betriebsmitteltypen und die Zahl der vorhandenen Einheiten für jeden Typ – ist fest vorgegeben.
2. Die Betriebsmittelforderungen der Prozesse respektieren die Betriebsmittelkonstellation.
3. Im Anfangszustand belegt kein Prozeß Betriebsmittel.
4. Ein beendeter Prozeß belegt keine Betriebsmittel.
5. Ein Prozeß ist in einem Zustand genau dann blockiert, wenn seine nächste Aktion eine nicht erfüllbare Betriebsmittelforderung ist.

Wir treffen nun die folgenden Vereinbarungen:

- Die Prozesse werden durchnumeriert: $P = \{1, 2, \ldots, n\}, n \geq 2$.
- Seien $t_1, \ldots, t_m, m \geq 1$, die vorhandenen **Betriebsmitteltypen** und für jedes t_i die Zahl der vorhandenen **Einheiten** durch $g_i \geq 1$ gegeben.

  Der Vektor der vorhandenen Betriebsmitteleinheiten ist definiert als:

$$\mathbf{g} := (g_1, \ldots, g_m)^t$$

- In einem Zustand σ sei $b_i^j(\sigma)$ die von Prozeß j belegte Zahl von Betriebsmitteleinheiten des Typs t_i.
- Der Vektor $\mathbf{b}^j(\sigma) := (b_1^j(\sigma), \ldots, b_m^j(\sigma))^t$ heißt **Belegungsvektor** von Prozeß j.

  Die **Belegungsmatrix** des Systems im Zustand σ ist gegeben durch

$$B(\sigma) := (\mathbf{b}^1(\sigma), \ldots, \mathbf{b}^n(\sigma))$$

- Im Zustand σ sei $h_i(\sigma)$ die Zahl der **verfügbaren** Betriebsmitteleinheiten des Typs t_i. Für jedes i ergibt sich $h_i(\sigma)$ aus der Differenz von g_i und der insgesamt von Prozessen belegten Betriebsmittel des Typs t_i im gegebenen Zustand.

  Wir definieren weiter

$$\mathbf{h}(\sigma) := (h_1(\sigma), \ldots, h_m(\sigma))^t$$

- Eine Aktion zur **Anforderung** von Betriebsmitteln in Prozeß j schreiben wir in der Form

$$\mathbf{reserviere}\,(\,\mathbf{y}^{\,j})$$

wobei $\mathbf{y}^{\,j} := (y_1^j, \ldots, y_m^j)^t$, und y_i^j die Zahl der geforderten Betriebsmittel des Typs t_i ist.

In einem Zustand σ ist der **Forderungsvektor**, $\mathbf{y}^{\,j}(\sigma)$, von Prozeß j wie folgt definiert:

- ist die nächste Aktion von Prozeß j eine Betriebsmittelanforderung, dann ist $\mathbf{y}^{\,j}(\sigma)$ wie beschrieben definiert.
- ist die nächste Aktion von Prozeß j keine Betriebsmittelanforderung, dann ist $\mathbf{y}^{\,j}(\sigma) := \mathbf{o}$ (der Nullvektor).

Die **Forderungsmatrix** in einem Zustand σ ist gegeben durch

$$Y(\sigma) := (\mathbf{y}^{\,1}(\sigma), \ldots, \mathbf{y}^{\,n}(\sigma))$$

Sei eine Anforderung $\mathbf{reserviere}\,(\,\mathbf{y}^{\,j})$ im Zustand σ gegeben. Dann gilt

$$j \in \Gamma(\sigma) \Leftrightarrow \mathbf{y}^{\,j} \leq \mathbf{h}\,(\sigma)$$

Wir setzen nun voraus, daß $j \in \Gamma(\sigma)$ und daß die Anforderung $\mathbf{reserviere}\,(\,\mathbf{y}^{\,j})$ in Prozeß j als nächste Aktion ausgewählt wird, mit $\sigma \xrightarrow{j} \sigma'$. Dann gilt:

- $\mathbf{b}^{\,j}(\sigma') := \mathbf{b}^{\,j}(\sigma) + \mathbf{y}^{\,j}$
- $\forall j' \in ([1:n] - \{j\}) : \mathbf{b}^{\,j'}(\sigma') := \mathbf{b}^{\,j'}(\sigma) \wedge \mathbf{y}^{\,j'}(\sigma') := \mathbf{y}^{\,j'}(\sigma)$
- $\mathbf{h}\,(\sigma') := \mathbf{h}\,(\sigma) - \mathbf{y}^{\,j}$

- Eine Aktion zur **Freigabe** von Betriebsmitteln in Prozeß j schreiben wir in der Form

$$\mathbf{gibfrei}\,(\,\mathbf{w}^{\,j})$$

wobei $\mathbf{w}^{\,j} := (w_1^j, \ldots, w_m^j)^t$ mit w_i^j die Zahl der freigegebenen Betriebsmittel des Typs t_i ist. Ein Prozeß, der eine Freigabeaktion durchführen möchte, ist niemals blockiert.

Wir setzen nun voraus, daß die Freigabe $\mathbf{gibfrei}\,(\,\mathbf{w}^{\,j})$ in Prozeß j als nächste Aktion ausgewählt wird, mit $\sigma \xrightarrow{j} \sigma'$. Dann gilt:

- $\mathbf{b}^{\,j}(\sigma') := \mathbf{b}^{\,j}(\sigma) - \mathbf{w}^{\,j}$
- $\forall j' \in ([1:n] - \{j\}) : \mathbf{b}^{\,j'}(\sigma') := \mathbf{b}^{\,j'}(\sigma) \wedge \mathbf{y}^{\,j'}(\sigma') := \mathbf{y}^{\,j'}(\sigma)$
- $\mathbf{h}\,(\sigma') := \mathbf{h}\,(\sigma) + \mathbf{w}^{\,j}$

- Das Paar

$$(B(\sigma), Y(\sigma))$$

heißt die **Betriebsmittelsituation** im Zustand σ.

5.3 Der Betriebsmittelgraph

Der **Betriebsmittelgraph**, $BMG(\sigma)$, ist ein bichromatischer Graph, der zur Darstellung der Betriebsmittelsituation in einem Zustand dient. Er ist wie folgt definiert:

1. Jedem Prozeß und jedem Betriebsmitteltyp entspricht genau ein **Knoten** von $BMG(\sigma)$.
   Wir stellen Prozesse durch Rechtecke und Betriebsmitteltypen durch Kreise dar.
2. Die Menge aller **Kanten** von $BMG(\sigma)$ ergibt sich wie folgt:
   (a) Für alle i, j mit $b_i^j \neq 0$ existiert eine gerichtete Kante (**Belegungskante**) mit Anfangsknoten t_i und Endknoten j.
   (b) Für alle i, j mit $y_i^j \neq 0$ existiert eine gerichtete Kante (**Forderungskante**) mit Anfangsknoten j und Endknoten t_i.□

Beispiel: Wir betrachten ein System mit zwei Prozessen und zwei Betriebsmitteltypen, und zeigen das Entstehen einer Verklemmung. Es gelte also:

- $P = \{1, 2\}$ mit Aktionsfunktionen f^1, f^2.
- $m = 2$
- $\mathbf{g} = (1, 1)^t$

Ausgangspunkt sei ein Zustand, σ_1, in dem kein Prozeß Betriebsmittel belegt. Nun mögen die beiden Prozesse in diesem Zustand die folgenden Betriebsmittelanforderungen stellen:

- **Prozeß 1:** $L11:$ **reserviere** $(1,0)^t$; $L12:$ **reserviere** $(0,1)^t$
- **Prozeß 2:** $L21:$ **reserviere** $(0,1)^t$; $L22:$ **reserviere** $(1,0)^t$

Die Betriebsmittelsituation im Zustand σ_1 ist gegeben durch:

- $\mathbf{b}^1(\sigma_1) = (0,0)^t$
- $\mathbf{b}^2(\sigma_1) = (0,0)^t$
- $\mathbf{y}^1(\sigma_1) = (1,0)^t$
- $\mathbf{y}^2(\sigma_1) = (0,1)^t$
- $\mathbf{h}(\sigma_1) = \mathbf{g} = (1,1)^t$

und es gilt $\Gamma(\sigma_1) = \{1,2\}$.

Nun werde Prozeß 1 für die erste Aktion ausgewählt: dies ergibt einen Zustand $\sigma_2 := f^1(\sigma_1)$. Die neue Betriebsmittelsituation ist wie folgt gegeben:

- $\mathbf{b}^1(\sigma_2) = (1,0)^t$
- $\mathbf{b}^2(\sigma_2) = (0,0)^t$
- $\mathbf{y}^1(\sigma_2) = (0,1)^t$
- $\mathbf{y}^2(\sigma_2) = (0,1)^t$
- $\mathbf{h}(\sigma_2) = \mathbf{g} = (0,1)^t$

und es gilt $\Gamma(\sigma_2) = \{1, 2\}$. Nun werde Prozeß 2 für die nächste Aktion ausgewählt, resultierend im Zustand $\sigma_3 := f^2(\sigma_2)$, mit der Betriebsmittelsituation:

- $\mathbf{b}^1(\sigma_3) = (1, 0)^t$
- $\mathbf{b}^2(\sigma_3) = (0, 1)^t$
- $\mathbf{y}^1(\sigma_3) = (0, 1)^t$
- $\mathbf{y}^2(\sigma_3) = (1, 0)^t$
- $\mathbf{h}(\sigma_3) = \mathbf{g} = (0, 0)^t$

$\Gamma(\sigma_3) = \phi$ und σ_3 ist ein totaler Verklemmungszustand. Der zugehörige Betriebsmittelgraph ist in Fig.13 dargestellt. □
Es läßt sich zeigen, daß die im Beispiel beschriebene Situation typisch für den allgemeinen Fall ist. Ein Verklemmungszustand, σ, läßt sich demnach allgemein wie folgt charakterisieren:

V1: In σ sind mindestens zwei Prozesse verklemmt

V2: Mindestens zwei der verklemmten Prozesse belegen Betriebsmittel

V3: Jeder der verklemmten Prozesse fordert Betriebsmittel, die von einem anderen verklemmten Prozeß belegt werden: es existiert also eine zirkuläre Wartesituation.

Notwendig für das Auftreten einer Verklemmung unter den gegebenen Voraussetzungen (Exklusivität des Zugriffs und Nichtentziehbarkeit der Betriebsmittel) ist die Möglichkeit für einen Prozeß, der bereits Betriebsmittel besitzt, neue Betriebsmittel anzufordern.

Formale Charakterisierung von Verklemmungszuständen

Sei für $\sigma \in \Sigma - \Sigma_{end}$ die Betriebsmittelkonstellation durch den Vektor $\mathbf{g}$ gegeben, und die Betriebsmittelsituation durch das Paar $(B(\sigma), Y(\sigma))$. Sei ferner $U \subseteq [1:n]$ eine beliebige nichtleere Teilmenge von Prozessen. Wir definieren nun die folgende Bedingung bezogen auf σ:

$$\forall u \in U : \mathbf{y}^{u}(\sigma) \not\leq \mathbf{g} - \Sigma_{u' \in U} \mathbf{b}^{u'}(\sigma)$$

Dieser Ausdruck wird durch $VKL(U)$ abgekürzt.

Satz: Sei $K \subseteq [1:n]$ die Menge der in σ verklemmten Prozesse.

1. Es gilt: $VKL(K)$ ist erfüllt.
2. Für jede Wahl einer nichtleeren Teilmenge, $U \subseteq [1:n]$, von Prozessen gilt:
   Aus der Gültigkeit von $VKL(U)$ folgt $U \subseteq K$.$\square$

5.4 Verklemmungsstrategien

Verklemmungsstrategien lassen sich in drei Hauptgruppen unterteilen: Strategien zur Verhinderung, Erkennung, und Vermeidung von Verklemmungen.
Man spricht von **Verhinderung (deadlock prevention)**, wenn durch die Strukturierung des Systems Prozesse beim Zugriff auf Betriebsmittel Einschränkungen unterworfen werden, die Verklemmungen unmöglich machen. Im Gegensatz zu den Strategien zur Vermeidung (s.u.) ist hierzu keine Überprüfung der Betriebsmittelanforderungen zur Laufzeit erforderlich.

Bei einer Strategie zur **Erkennung von Verklemmungen (deadlock detection)** wird die Betriebsmittelsituation periodisch auf das Vorliegen einer Verklemmung überprüft. Wird eine Verklemmung erkannt, dann müssen ein oder mehrere betroffene Prozesse von ihren Betriebsmitteln getrennt werden. Dies ist wegen der Nichtentziehbarkeit der Betriebsmittel eine Nichtstandard-Aktion, die zum Verlust von Information führen kann.

Eine Strategie zur **Vermeidung von Verklemmungen (deadlock avoidance)** setzt voraus, daß Prozesse ihren maximalen Betriebsmittelbedarf spezifizieren. Diese Information kann dann zu einer Steuerung der Betriebsmittelvergabe benutzt werden, die Verklemmungen ausschließt.

5.4.1 Verhinderung von Verklemmungen

Die beiden wichtigsten in der Praxis angewandten Methoden zur Verhinderung von Verklemmungen sind die folgenden:

- **Statische Betriebsmittelzuteilung:** Prozesse, die bereits Betriebsmittel belegen, dürfen keine neuen Betriebsmittelanforderungen stellen.
- **Hierarchische Betriebsmittelzuteilung:** Prozesse dürfen Betriebsmittel nur in einer vorgegebenen Reihenfolge anfordern.

Die statische Betriebsmittelzuteilung negiert eine notwendige Voraussetzung für Verklemmungen (s.o.).

Bei der hierarchischen Betriebsmittelzuteilung wird eine lineare Ordnung in der Menge der Betriebsmitteltypen definiert. Betriebsmittelanforderungen können nur unter Respektierung dieser Ordnung gestellt werden: Prozesse, die bereits Betriebsmittel eines Typs belegen, dürfen nur Betriebsmittel eines höheren Typs anfordern. Dies verhindert das Entstehen einer zirkulären Wartesituation.

5.4.2 Erkennung von Verklemmungen

Es stellt sich die Aufgabe, aufgrund der in einem Zustand σ gegebenen Betriebsmittelsituation $(B(\sigma), Y(\sigma))$ zu bestimmen, ob (1) σ ein Verklemmungszustand ist oder nicht, und (2), falls σ ein Verklemmungszustand ist, alle in σ verklemmten Prozesse zu ermitteln. Wir nehmen o.B.d.A. an, daß kein Prozeß in σ beendet ist, d.h. $\sigma \in \Sigma - \Sigma_{end}$, und daß jeder Prozeß eine nichtleere Menge von Endzuständen besitzt: $\Sigma^p_{end} \neq \phi$ für alle $p \in P$. Information über Betriebsmittelanforderungen von Prozessen jenseits der gegebenen Betriebsmittelsituation wird nicht vorausgesetzt.

Unser Vorgehen basiert auf der Untersuchung der Möglichkeiten, die zu σ führenden Aktionsfolge, α, fortzusetzen. Eine solche Fortsetzung nennen wir **günstig**, wenn jeder Prozeß, der in der Fortsetzung Aktionen ausführt, ohne weitere Betriebsmittelforderungen zu Ende läuft und alle seine Betriebsmittel wieder freigibt. Offenbar genügt es zur Erkennung einer Verklemmung, nur günstige Fortsetzungen zu betrachten: denn ein Prozeß, der in σ blockiert ist und in keiner günstigen Fortsetzung wieder aktiv werden kann, ist offenbar verklemmt. Läßt sich umgekehrt eine günstige Fortsetzung angeben, in der ein Prozeß aktiv wird, dann ist er nach unseren Voraussetzungen nicht verklemmt. Wir beschränken uns auf die Betrachtung von Fortsetzungen, in denen die involvierten Prozesse sequentiell abgewickelt werden, bis sie ihren Endzustand erreichen. Solche Fortsetzungen nennen wir **Aktivierungssequenzen**.

Eine Aktivierungssequenz, A, hat also die Gestalt

$$A = \alpha^{p_1}\alpha^{p_2}\ldots\alpha^{p_r}$$

wobei gilt:

1. $p_1, p_2, \ldots, p_r$ ist eine Folge paarweise verschiedener Prozesse in P
2. Jede Teilfolge $\alpha^{p_k}, 1 \leq k \leq r$, enthält nur Aktionen von Prozeß p_k
3. Sei σ_k gegeben durch $\sigma \xrightarrow{\alpha_k} \sigma_k$, wobei $\alpha_k := \alpha^{p_1}\alpha^{p_2}\ldots\alpha^{p_k}$ $(1 \leq k \leq r)$. Dann gilt $\sigma_k \in \Sigma_{end}^{p_k}$.

Eine nicht fortsetzbare Aktivierungssequenz heißt **komplett**. Eine Aktivierungssequenz heißt **vollständig** genau dann, wenn sie alle Prozesse enthält: $r = n$.

Die nachstehenden Überlegungen basieren auf der Gültigkeit des folgenden Lemmas:

Lemma: Jede komplette günstige Aktivierungssequenz in einem Zustand σ enthält genau alle in σ nicht verklemmten Prozesse. □

Der Algorithmus in Fig.14 erkennt Verklemmungen. Er wird in einem Zustand σ mit einer gegebenen Betriebsmittelsituation angewandt und konstruiert eine komplette Aktivierungssequenz. Nach Ausführung des Algorithmus gibt der Wert der booleschen Variablen VKL an, ob ein Verklemmungszustand vorliegt oder nicht, und K enthält genau alle in σ verklemmten Prozesse. Der angegebene Algorithmus besitzt die Komplexität $\mathbf{O}(m.n^2)$. Er läßt sich relativ einfach durch einen Algorithmus der Komplexität $\mathbf{O}(m.n)$ ersetzen.

5.4.3 Vermeidung von Verklemmungen

Wir befassen uns hier mit dem Problem, durch vorausschauende Planung das Auftreten von Verklemmungen zu vermeiden. Hierzu nehmen wir an, daß jeder Prozeß seinen maximalen Betriebsmittelbedarf zu Beginn angibt. Es sei an dieser Stelle daran erinnert, daß die Prozesse als paarweise unabhängig vorausgesetzt wurden. Abgesehen vom Zugriff auf gemeinsam benutzte Betriebsmittel können also keine Konflikte auftreten.

Zwei Fragestellungen sind von Bedeutung:

1. Ist ein bestimmter Zustand sicher, d..h. gibt es von diesem Zustand aus zumindest eine Fortsetzung, in der keine Verklemmung auftritt, selbst wenn alle Prozesse ihre maximalen Betriebsmittelanforderungen stellen.

   Es zeigt sich, daß zur Lösung dieses Problems der oben angegebene Algorithmus herangezogen werden kann.

2. Gehen wir von einem sicheren Zustand aus. Das System kann nur durch die Erfüllung einer Betriebsmittelanforderung in einen unsicheren Zustand geraten. Es stellt sich also die Frage, welche Betriebsmittelanforderungen befriedigt werden können, ohne einen Übergang in einen unsicheren Zustand zu bewirken.

Wir erweitern nun das weiter oben eingeführte Betriebsmittelmodell durch die Spezifikation der maximalen Betriebsmittelforderung für Prozesse.

Für jeden Prozeß j sei x_i^j die maximal benötigte Zahl von Betriebsmitteleinheiten des Typs t_i. Der Vektor $\mathbf{x}^j := (x_1^j, \ldots, x_m^j)^t$ heißt **Vektor des maximalen Bedarfs** von Prozeß j. Die Matrix

$$X := (\mathbf{x}^1, \ldots, \mathbf{x}^n)$$

heißt **Matrix des maximalen Bedarfs**. Es muß offenbar gelten

$$\forall \sigma \in \Sigma : B(\sigma) \leq X$$

Sei σ ein Zustand, in dem kein Prozeß beendet ist. Eine Fortsetzung der zu σ führenden Aktionsfolge heißt **ungünstig**, wenn jeder Prozeß, der in der Fortsetzung seine Aktionen ausführt, die von ihm belegten Betriebsmittel erst freigibt, nachdem sein maximaler Bedarf befriedigt wurde.

Ein Zustand σ heißt **sicher** genau dann, wenn es in σ eine vollständige ungünstige Aktivierungssequenz gibt.

Ist ein Zustand sicher, dann gibt es eine Aktivierungssequenz, in der alle Prozesse ihren Endzustand erreichen, selbst wenn sie ihre maximalen Betriebsmittelforderungen stellen. In einem unsicheren Zustand existiert eine solche Fortsetzung nicht. Dies bedeutet jedoch nicht, daß ein unsicherer Zustand ein Verklemmungszustand ist – es kann zum Beispiel sein, daß die künftigen Forderungen der Prozesse unter der Maximalforderung bleiben.

Das Problem der Überprüfung der Sicherheit eines Zustands läßt sich aufgrund des nachfolgenden Satzes auf das Problem der Erkennung eines Verklemmungszustands zurückführen:

Satz: Sei σ mit der Betriebsmittelsituation $(B(\sigma), Y(\sigma))$ gegeben. Wir konstruieren den fiktiven Zustand σ' mit der Betriebsmittelsituation $(B(\sigma'), Y(\sigma'))$, wobei angenommen sei $B(\sigma') := B(\sigma)$ und $Y(\sigma') := X - B(\sigma)$. Dann gilt:

σ ist sicher genau dann, wenn σ' kein Verklemmungszustand ist. $\square$

Kapitel 6: Speicherverwaltung

Moderne Rechenanlagen besitzen eine **Speicherhierarchie**, die von extrem schnellen, kleinen, und teuren Speichern an der Spitze (Register) bis zu relativ langsamen, großen und billigen Speichern auf den unteren Ebenen (Magnetbänder) reicht. Die Hardware-Charakteristiken dieser Speichermedien wurden in Kapitel 1 besprochen.

Wir werden uns in diesem Kapitel auf die Verwaltung des Hauptspeichers, und hier insbesondere auf virtuelle Speicher konzentrieren. Davor diskutieren wir kurz *Swapping*.

Die Aufgaben der **Speicherverwaltung (memory management)** lassen sich wie folgt zusammenfassen:

- Erfassung freier Speicherbereiche
- Zuweisung von Speicher an Prozesse
- Freigabe von Speicher durch Prozesse
- Transfer von Daten zwischen Hauptspeicher und Hintergrundspeicher

6.1 Swapping

Swapping ist eine Strategie, die Speicherverwaltung für Mehrprogrammbetrieb in Rechnern ohne virtuellen Speicher unterstützt.

Bei der Zuweisung eines Prozessors an einen Prozeß wird der gesamte von diesem Prozeß benötigte Speicherplatz reserviert. Bei einer Unterbrechung des Prozesses werden seine Daten auf die Platte transferiert. Die Größe des einem Prozeß zugeordneten Speicherbereichs und dessen physikalische Lokalisierung kann über verschiedene Phasen des Prozesses variieren.

Zu jedem Zeitpunkt kann mehreren Prozessen Speicherplatz zugewiesen sein. Es entsteht dann das Problem, den freien Speicherplatz effizient zu nutzen. Eine der dafür benutzten Methoden ist **Kompaktifizierung (memory compaction)**: hier wird dafür gesorgt, daß die Speicherbereiche von Prozessen unmittelbar aneinander angrenzen, so daß der freie Speicher einen zusammenhängenden Bereich darstellt. Dies wird durch periodisches "Zusammenschieben" der Speicherbereiche von Prozessen erreicht. Dieses Verfahren wird wegen seines hohen Aufwands wenig benutzt.

Zur Verwaltung des Speichers im Betriebssystem gibt es zwei Methoden: **Bitmaps** und Listenverwaltung. Das auf Bitmaps basierende Verfahren läßt sich wie folgt charakterisieren:

- Der Speicher ist in **Zuweisungseinheiten (allocation units)** unterteilt.
- Alle Zuweisungseinheiten haben eine fest vorgegebene Länge, die von wenigen Worten bis zu mehreren KB reichen kann.
- Jeder Zuweisungseinheit ist genau ein Bit in einer Bitkette zugeordnet. Der Wert dieses Bits spezifiziert ob die entsprechende Zuweisungseinheit belegt ist (1) oder nicht (0).

Der Vorteil des Verfahrens liegt in dem relativ geringen Speicherbedarf, der für die Verwaltung benötigt wird. So werden zum Beispiel für einen Speicher der Größe 64MB und einer Zuweisungseinheit von 512B lediglich 16KB für die Bitmap gebraucht. Die Schwierigkeit liegt im Zeitaufwand für den Zuweisungsalgorithmus: werden für einen Prozeß k Zuweisungseinheiten gebraucht, dann muß der Algorithmus in der Bitmap einen Bereich mit k aufeinanderfolgenden Nullen finden.

Eine Alternative zur Bitmap ist die Verwaltung der belegten und freien Speicherbereiche unter Zuhilfenahme von getrennten Listen. Es gibt eine Reihe algorithmischer Strategien, die in diesem Zusammenhang angewandt werden:

- **First Fit:** finde den ersten passenden Bereich – Suche vom Beginn der Liste
- **Next Fit:** finde den ersten passenden Bereich – Suche vom letzten Endpunkt der Suche
- **Best Fit:** durchsuche die gesamte Liste und finde den kleinsten passenden Bereich
- **Quick Fit:** Hier werden mehrere Listen geführt, die zu typischen Speichergrößen gehören

6.2 Virtueller Speicher

In einer Maschine mit virtueller Adressierung wird zwischen dem **virtuellen (logischen) Adreßraum** eines Prozesses und dem **physischen Adreßraum** der Rechenanlage unterschieden. Jeder Prozeß besitzt einen eigenen virtuellen Adreßraum, der disjunkt von dem aller anderen Prozesse ist, sofern nicht Vereinbarungen zur gemeinsamen Benutzung von Daten oder Prozeduren getroffen werden.

Der virtuelle Adreßraum kann wesentlich größer sein als der physische Adreßraum. Dies ist kein Widerspruch – es reicht für die Ausführung eines Prozesses in der Regel aus, nur einen Teil des virtuellen Adreßraums auf den physischen Speicher abzubilden.

Sei A^p der virtuelle Adreßraum eines Prozesses p, und M der physische Adreßraum der betrachteten Maschine. Wir nehmen zunächst an, daß alle Adressen positive ganze Zahlen sind: $A^p = [0 : 2^{k_v} - 1]$ und $M = [0 : 2^{k_p} - 1]$.

Eine **Adreß-Abbildungsfunktion (AAF)** für Prozeß p ist definiert als eine partielle Funktion, $\mu : A^p \to M$. Von dieser Funktion setzen wir zunächst nur voraus, daß sie verschiedene virtuelle Adressen auf verschiedene physische Speicheradressen abbildet.

In einem virtuellen Speichersystem wird die AAF bei Zugriff auf eine Adresse mit Unterstützung der Hardware und des Betriebssystems berechnet. Wenn für ein $a \in A^p$ gilt $a \notin DEF(\mu)$, dann sprechen wir von einem **Adreßfehler**. Das Betriebssystem hat in diesem Fall die Aufgabe, für das mit dieser Adresse verbundene Objekt im Hauptspeicher Platz zuzuweisen.

6.2.1 Paging

Bei Paging ist der virtuelle Speicher in **Seiten (pages)** einer bestimmten, festen Größe unterteilt, und der physische Speicher entsprechend in **Blöcke (frames)**, welche die gleiche Größe wie Seiten haben. Sei die Seitenlänge durch 2^k gegeben. Eine virtuelle Adresse läßt sich dann darstellen als

$$a = s * 2^k + r$$

wobei $s \geq 0$ die **Seitennummer**, und r mit $0 \leq r \leq 2^k - 1$ die **Relativadresse** innerhalb der Seite s ist. Wir werden a in Zukunft in der Form $a = (s, r)$ schreiben.

Analog kann man physische Adressen in der Form

$$m = b * 2^k + r$$

darstellen, wobei $b \geq 0$ die **Blocknummer** ist und r die gleiche Bedeutung hat wie bei der virtuellen Adresse. Wir schreiben analog $m = (b, r)$.

Zur Bestimmung der AAF ist es bei Paging nur nötig, die Abbildung von Seiten auf Blöcke zu definieren. Sei ν eine Funktion, die das leistet: wir nennen eine solche Funktion **Seitenabbildungsfunktion (SAF)**. Dann kann man definieren:

$$\mu(s, r) := (\nu(s), r)$$

Ein Adreßfehler bei Paging wird als **Seitenfehler (page fault)** bezeichnet.

Im einfachsten Fall wird die SAF durch eine **Seitentabelle (page table)** dargestellt: eine Seitentabelle für einen Prozeß enthält 2^{k_v-k} Elemente; das Element in Position s spezifiziert $\nu(s)$.

In einem realen System enthält die Seitentabelle zugehörig zu jedem Element s zusätzliche Information, welche die Berechnung der hardware-realisierten Abbildung SAF unterstützt bzw. dem Betriebssystem Steuerungsinformation übermittelt. Hierzu zählen das Definiert-Bit, die Zugriffsberechtigung, Veränderungs- und Benutzungsinformation:

- D: **Definiert-Bit**.
  D gibt an ob $\nu(s)$ definiert ist oder nicht. Diese Information wird von der Hardware interpretiert. Ist D= **true** , dann ersetzt der Hardwaremechanismus zur Berechnung der SAF s durch $\nu(s)$ und baut danach die physische Adresse auf.

  Ist dagegen D= **false** , dann liegt ein Seitenfehler vor und das Betriebssystem wird über einen synchronen Interrupt aktiviert.

- Z=(R,W,X): **Zugriffsberechtigung**.
  Dies ist eine 3-bit Information, welche die Zugriffsberechtigung des ausführenden Prozesses auf die Seite s spezifiziert (W: Schreiben (write), R: Lesen (read), X: Ausführen (execute)). Die Zugriffsberechtigung wird bei jedem Zugriff eines Prozesses zu einer Seite hardwaremäßig kontrolliert. Wenn der Zugriff des Prozesses zur Seite s konsistent mit der in seiner Seitentabelle für s eingetragenen Berechtigung ist, geschieht nichts. Andernfalls wird das ausführende Programm durch einen (synchronen) Interrupt unterbrochen.

- MOD: **Veränderungsinformation (modification)**.
  MOD gibt an, ob die Seite durch Zugriffe des ausführenden Prozesses verändert wurde oder nicht. MOD wird automatisch per Hardware gesetzt, wenn der Prozeß in die Seite schreibt. Diese Information ist für das Betriebssystem zur Steuerung gewisser Algorithmen nützlich.

- REF: **Benutzungsinformation (reference)**.
  REF gibt an, ob der ausführende Prozeß auf die Seite (in irgendeiner Form) zugegriffen hat. REF wird automatisch per Hardware gesetzt, wenn der Prozeß die Seite anspricht. Häufig wird REF vom Betriebssystem periodisch auf **false** zurückgesetzt. Wie MOD kann auch diese Information vom Betriebssystem benutzt werden (s. Seitenersetzungsalgorithmen).

Bemerkung: Die Repräsentation der SAF durch eine Seitentabelle ist realistisch, solange die Zahl der Seiten relativ klein bleibt. Betrachten wir nun 64-bit virtuelle Adressierung und nehmen wir k=12 an, also eine Seitenlänge von 4096B. Dies bedeutet $2^{64-12} = 2^{52}$, also etwa $4 * 10^{15}$ Seiten! Eine lineare Tabellenstruktur steht in diesem Fall außer Frage. Eine mögliche Lösung ist die hierarchische Organisation der Seitentabelle: zum Beispiel wäre bei einer zweistufigen Seitentabelle die Wurzel des Baums eine Tabelle von Referenzen auf Seitentabellen der Art wie oben besprochen. Ein anderer Ansatz besteht in der Invertierung der Seitentabelle – d.h. der Organisation einer Blocktabelle für die im *physischen* Speicher benutzten Blöcke. □

Der Zugriff zu den Adressen eines Speichers ist in der Regel durch **Lokalität** charakterisiert. Man unterscheidet zwei Arten von Lokalität:

- **Temporale Lokalität (temporal locality)** drückt aus, daß eine zur Zeit t angesprochene Adresse innerhalb eines kleinen Zeitraums Δt mit überdurchschnittlich hoher Wahrscheinlichkeit wieder referiert wird.
- **Räumliche Lokalität (spatial locality)** bedeutet, daß ein Algorithmus, der zum Zeitpunkt t eine Adresse a anspricht, mit überdurchschnittlicher Wahrscheinlichkeit innerhalb eines kleinen Zeitraums eine Adresse "nahe bei" a ansprechen wird.

Unter Ausnutzung der Lokalität verwenden viele Architekturen einen Assoziativspeicher, **Translation Lookahead Buffer (TLB)**, in dem in der Regel eine kleine (4 bis etwa 64) Zahl von Paaren $(s, \nu(s))$, zusammen mit der benötigten Zusatzinformation gespeichert wird. Durch die Einbindung des Assoziativspeichers in die Hardware-Realisierung der Funktion SAF wird die Übersetzung wesentlich beschleunigt.

Bei Vorhandensein eines TLB wird beim Zugriff auf eine Seite s zunächst von der Hardware geprüft, ob es ein Element für s im TLB gibt. Ist das der Fall, kann $\nu(s)$ unmittelbar bestimmt werden; andernfalls muß die Seitentabelle aufgesucht werden.

6.2.2 Seitenersetzungsalgorithmen

Wir betrachten den Fall, daß beim Zugriff zu der Seite s eines Prozesses p ein Seitenfehler auftritt und zu diesem Zeitpunkt kein Block des physischen Speichers für die Zuordnung der neuen Seite zur Verfügung steht. In diesem Fall sind vom Betriebssystem folgende Aktionen durchzuführen:

1. Bestimme die "unwichtigste" Seite, s', mit $D(s') = \mathbf{true}$.
2. Eliminiere s' aus dem Hauptspeicher. Sei $b' = \nu(s')$.
   (a) Falls $MOD(s') = \mathbf{true}$ (oder falls Modifikation nicht ausgeschlossen werden kann), muß die in b' abgelegte Information in den Hintergrundspeicher kopiert werden. Falls $MOD(s') = \mathbf{false}$, dann dann ist die Kopie von b' im Hintergrundspeicher gültig und der Transfer kann unterbleiben.
   (b) In der Seitentabelle von Prozeß p wird $D(s') := \mathbf{false}$ gesetzt. Dies bewirkt beim nächsten Zugriff auf s' einen Seitenfehler.
   (c) In einer internen Tabelle des Betriebssystems wird zugehörig zu p und s' ein Zeiger auf die Kopie von b' im Hintergrundspeicher eingesetzt.
3. Falls die Seite s bereits ein Bild im Hintergrundspeicher besitzt, wird die entsprechende Information in den Block b' des Hauptspeichers übertragen.
4. Setze $\nu(s) := b'$ und lege diese Information in Element s der Seitentabelle ab.
5. Die Steuerungsinformation in Element s der Seitentabelle von p wird wie folgt definiert:

(a) $D(s) := \mathbf{true}$

(b) $Z(s)$ wird entsprechend den im Betriebssystem notierten Zugriffsrechten gesetzt.

(c) $MOD(s) := \mathbf{false}$

(d) $REF(s) := \mathbf{true}$

Es stellt sich nun die Frage, wie man die "unwichtigste" Seite auswählt. Dieses Problem wird von **Seitenersetzungsalgorithmen (page replacement algorithms)** angesprochen. Es gibt hier viele Ansätze, von denen wir einige wenige im folgenden diskutieren. Generell ist es das Ziel, die Zahl der Seitenfehler zu minimieren. Die "unwichtigste" Seite ist in diesem Sinne die am wenigsten benutzte.

Optimaler Algorithmus. Dies ist ein fiktiver, nicht realisierbarer, Ansatz, der nichtsdestoweniger die Zielrichtung dieser Algorithmen klarstellt. Er läßt sich wie folgt skizzieren.

Seien zum Zeitpunkt des Auftretens eines Seitenfehlers die Seiten $s_0, \ldots, s_{q-1}$ von Prozeß p im Hauptspeicher. Wir nehmen an, daß für p kein weiterer Block zur Verfügung steht. Sei $r(s)$ die Zahl der Instruktionen bis zur nächsten Referenz auf s. Dann wird die zu eliminierende Seite aus denjenigen mit maximalem $r(s)$ ausgewählt.

Hierzu ist festzustellen, daß (1) das Betriebssystem keine Möglichkeit besitzt, die Werte $r(s)$ zu bestimmen, und (2) diese Werte nicht nur durch das Programm, sondern auch durch die spezifisch für die betrachtete Ausführung gewählten Eingabedaten bestimmt sind.

Durch Benutzung eines Simulators kann entsprechende Information für eine nachfolgende Ausführung des Programms gewonnen werden.

Not-Recently-Used (NRU) Algorithmus. NRU benutzt die Information MOD und REF, um vier Klassen von Seiten zu bestimmen:

1. C1: (MOD,REF)=(**false**, **false**)
2. C2: (MOD,REF)=(**true**, **false**)
3. C3: (MOD,REF)=(**false**, **true**)
4. C4: (MOD,REF)=(**true**, **true**)

NRU wählt aus der niedrigsten nichtleeren Klasse eine Seite beliebig zur Elimination aus. Man beachte, daß eine Seite, die nicht (in einem gewissen Zeitintervall) angesprochen, aber modifiziert wurde, bezüglich Elimination Vorrang besitzt vor einer Seite, die nicht modifiziert, aber referenziert wurde. Die Motivation für diese Entscheidung liegt in dem Bestreben, häufig angesprochene Seiten nicht zu eliminieren.

Least-Recently-Used (LRU) Algorithmus. Der LRU Algorithms benutzt die Lokalität von Speicherzugriffen für eine Approximation des Optimalen Algorithmus. Er eliminiert diejenige Seite, die am längsten nicht angesprochen wurde.

Ohne Hardware-Unterstützung ist LRU sehr aufwendig: es muß eine Liste aller Seiten geführt werden, die nach der Reihenfolge des letzten Ansprechens sortiert ist. Diese Liste muß bei jedem Speicherzugriff adaptiert werden.

Eine einfache Methode der Hardware-Unterstützung für eine Variante dieses Algorithmus (*least used*) ordnet jeder Seite einen **Zähler**, C, zu, der mit 0 initialisiert und bei jedem Ansprechen der Seite inkrementiert wird. Beim Auftreten eines Seitenfehlers wird eine Seite, s, mit minimalem $C(s)$ eliminiert.

6.2.3 Das Working Set Modell

Der **Working Set** eines Prozesses zum Zeitpunkt t, $w(t)$, ist die Menge aller Seiten, die im Zeitintervall $[t - \Delta : t]$ angesprochen wurden, wobei wir Δ als fest vorgegeben annehmen. (Eine Alternative zur Betrachtung eines Zeitintervalls ist die Analyse einer vorgegebenen Zahl von Speicherreferenzen).

Das Working Set Modell beruht auf der Annahme von Lokalität: bei gegebenem Working Set $w(t)$ zum Zeitpunkt t ist die Wahrscheinlichkeit groß, daß der nächste Speicherzugriff sich auf eine Seite in $w(t)$ bezieht. Man versucht also, die Seiten des Working Set im Speicher zu halten. Ein auf diesem Modell basierender Seitenersetzungsalgorithmus wird beim Auftreten eines Seitenfehlers versuchen, eine Seite zu eliminieren, die nicht in $w(t)$ liegt. Falls eine solche Seite nicht existiert, muß auf eine der früher besprochenen Methoden zurückgegriffen werden.

6.2.4 Abschließende Bemerkungen

Wir führen hier noch kurz einige Begriffe ein, die im Zusammenhang mit Paging eine wichtige Rolle spielen:

- **Demand Paging**
  Dies bezeichnet eine Strategie, die beim Laden eines Prozesses zur Ausführung *keiner seiner Seiten* Speicherplatz im Hauptspeicher zuweist. Erst beim Auftreten eines Seitenfehlers – für Daten oder Instruktionen – wird die betreffende Seite im Hauptspeicher abgelegt.

- **Prepaging**
  Dies ist eine gegensätzliche Strategie zu Demand Paging. Beim Laden eines Prozesses wird für eine Teilmenge seiner Seiten Platz im Hauptspeicher reserviert. Prepaging kann unter anderem im Rahmen des Working Set Modells benutzt werden.

- **Thrashing**
  Dies ist ein Effekt, der auftritt, wenn der Hauptspeicherbereich zu klein ist, um alle Seiten eines realistischen Working Sets aufzunehmen. Ein Programm kann in sehr kurzen Zeitabständen Seitenfehler verursachen.

- **Lokale und Globale Strategien**
  Wir haben bisher nur lokale Strategien zur Seitenersetzung diskutiert, bei denen einem Prozeß eine feste Anzahl von Blöcken des Hauptspeichers zugeordnet wird und die Strategie auf diesen Prozeß fixiert ist. Globale Strategien beziehen die Situation *aller* Prozesse ein, denen Platz im Hauptspeicher zugewiesen ist.

6.2.5 Segmentierung

Grundlage von Paging ist die feste Seitenlänge, welche die Speicherverwaltung im Betriebssystem (wie auch die Realisierung der SAF in der Hardware) vereinfacht. Allerdings reflektiert die feste Seitenlänge nicht die Realität von Algorithmen, in denen Bereiche unterschiedlicher Größe für Instruktionen und Daten (Stack, Heap, etc.) benötigt werden.

Sementierung realisiert virtuelle Adressierung auf der Basis von **Segmenten**. Jedes Segment repräsentiert einen linearen virtuellen Adreßraum, dessen Adressen von 0 bis zu einem Maximalwert numeriert werden. Unterschiedliche Segmente können verschiedene Länge besitzen, und die Länge von Segmenten kann während der Ausführung variieren. Aus logischer Sicht sind Segmente im Hinblick auf die Speicherzuweisung unabhängig voneinander.

Segmentierung kann mit Paging kombiniert werden. Dies ist im Intel Pentium realisiert, dessen Implementierung in vieler Hinsicht auf das in den 60-er Jahren entwickelte MULTICS System zurückgeht.

Kapitel 7: Ein/Ausgabe und Dateisysteme
7.1 Ein/Ausgabe-Software

Die Kontrolle der Ein/Ausgabe gehört zu den wichtigsten Aufgaben eines Betriebssystems. Dies schließt folgende Aktivitäten ein:

- Kontrolle der E/A-Geräte
- Bearbeitung von Programmunterbrechungen
- Fehlerbehandlung

Generell wird versucht, trotz der unterschiedlichen Charakteristika von Geräten eine einheitliche Schnittstelle zur Verfügung zu stellen.

Die Kommunikation zwischen Betriebssystem und den Steuerungseinheiten der Geräte wird über spezielle Register sowie Pufferspeicher abgewickelt, zu denen sowohl das Betriebssystem als auch das Gerät Zugriff haben. In manchen Systemen werden alle Register auf reservierte Speicherbereiche abgebildet (**memory-mapped I/O**).

Wir betrachten anhand eines Beispiels (Lesen von der Platte) die unterschiedlichen Ansätze bei interruptgesteuerter Eingabe und DMA-Eingabe.

Bei der **interruptgesteuerten** Version durchläuft man die folgenden Schritte:

1. Die Steuereinheit liest einen Datenblock bitweise seriell von der Platte in einen Puffer ein.
2. Die Steuereinheit überprüft die Korrektheit der eingelesenen Information (Bitsumme).
3. Die Steuereinheit generiert eine Programmunterbrechung.
4. Die Programmunterbrechung aktiviert eine zugehörige Betriebssystemfunktion.
5. Die aktivierte Betriebssystemroutine überträgt die Daten aus dem Pufferspeicher, über Benutzung spezifischer Register der Steuereinheit, in den Hauptspeicher.

Dieser Prozeß wird iterativ wiederholt, bis alle benötigten Daten gelesen sind.

Die Abwicklung des Transfers über DMA läßt sich wie folgt skizzieren:

1. Die CPU programmiert die DMA-Steuereinheit mit den Parametern für die Datenübertragung (Quelle, Senke, Länge des Bereichs) und aktiviert die Steuereinheit der Platte zur Initiierung der Datenübertragung in den Pufferspeicher.
2. DMA setzt ein, sobald der Puffer der Steuereinheit der Platte korrekte Daten enthält. Die DMA steuert von hier an den gesamten Eingabevorgang (in der Regel in mehreren Iterationen) und erzeugt erst am Ende eine Programmunterbrechung für die CPU. Zu diesem Zeitpunkt sind bereits alle eingelesenen Daten im Hauptspeicher.

Wir fassen hier die wichtigsten Kriterien eines modernen Softwaresystems für Ein/Ausgabe zusammen:

- **Geräteunabhängigkeit**
  Die Initiierung von E/A-Transfers in Programmen sollte in einer Form spezifizierbar sein, die unabhängig von den Eigenschaften spezifischer Geräte ist.

- **Einheitlicher Namensraum**
  Namen von Geräten sollten einheitlichen Kriterien unterliegen, unabhängig von der spezischen Art des Geräts.

- **Fehlerbehandlung**
  Fehler sollten im Rahmen eines hierarchischen Schemas behandelt werden. Fehler von Hardware-Komponenten sollte nach Möglichkeit lokal abgehandelt werden, ohne höhere Systemschichten einzubeziehen.

- **Synchronisation**
  Das Betriebssystem sollte Synchronisationsprobleme auf hardwarenaher Ebene lokal abhandeln und für das Programm ein vereinfachtes Synchronisationsschema auf hoher Abstraktionsebene bereitstellen.

- **Pufferung**
  Das Betriebssystem muß automatische Pufferungsmechanismen für temporäre Daten bereitstellen.

7.2 Dateisysteme

Dateisysteme sind die Antwort auf drei Problembereiche, die mit den bisher diskutierten Ansätzen nicht bewältigt werden können:

- Der einem Prozeß im Hauptspeicher zur Verfügung stehende Raum ist beschränkt, selbst bei Vorhandensein virtueller Adressierung.
- Viele Applikationen benötigen einen "permanenten" Langzeitspeicher, dessen Lebensdauer unabhängig von der Lebensdauer von Prozessen ist.
- Unabhängige Prozesse, deren Lebensdauer nicht aufeinander abgestimmt ist, müssen unabhängig und konkurrent auf gemeinsam benutzte Datenbereiche zugreifen können.

Dateisysteme realisieren einen Abstraktionsmechanismus: sie schaffen eine Möglichkeit, Information auf Hintergrundspeicher zu schreiben und später wieder zu lesen. Der Benutzer ist von den Details der Realisierung geschützt, muß also nicht darauf Rücksicht nehmen, welche Geräte zur Speicherung der Dateien benutzt werden und wie die Information auf den Speichermedien repräsentiert wird.

Dateien werden über einen **Dateinamen** angesprochen. Sie können von unterschiedlichem Typ sein – zum Beispiel *regulär* oder *Verzeichnis (directory)*, sowie Typen, welche gewisse Eigenschaften von Geräten abbilden (Zeichenkettendateien, Blockdateien).

Der **Zugriff** auf Dateien ist entweder sequentiell oder direct (random access). Hier sind einige der Grundoperationen, die auf Dateien angewandt werden können:

- Kreieren und Löschen
- Öffnen und Schließen
- Lesen und Schreiben
- Erweitern (ein spezieller Fall von Schreiben)
- Suchen
- Attribute bestimmen und setzen
- Umbenennen

Zu den Themen, die für die Implementierung eines Dateisystems eine Rolle spielen, gehören:

- Struktur der Verzeichnisse
- Operationen auf Verzeichnissen
- Konstruktion von Pfadnamen
- Organisation der Daten auf dem Speichermedium
- Organisation des gemeinsamen Zugriffs mehrerer Prozesse auf Dateien
- Fehlertoleranz
- Konsistenz
- Effizienz der Datei-Manipulationen

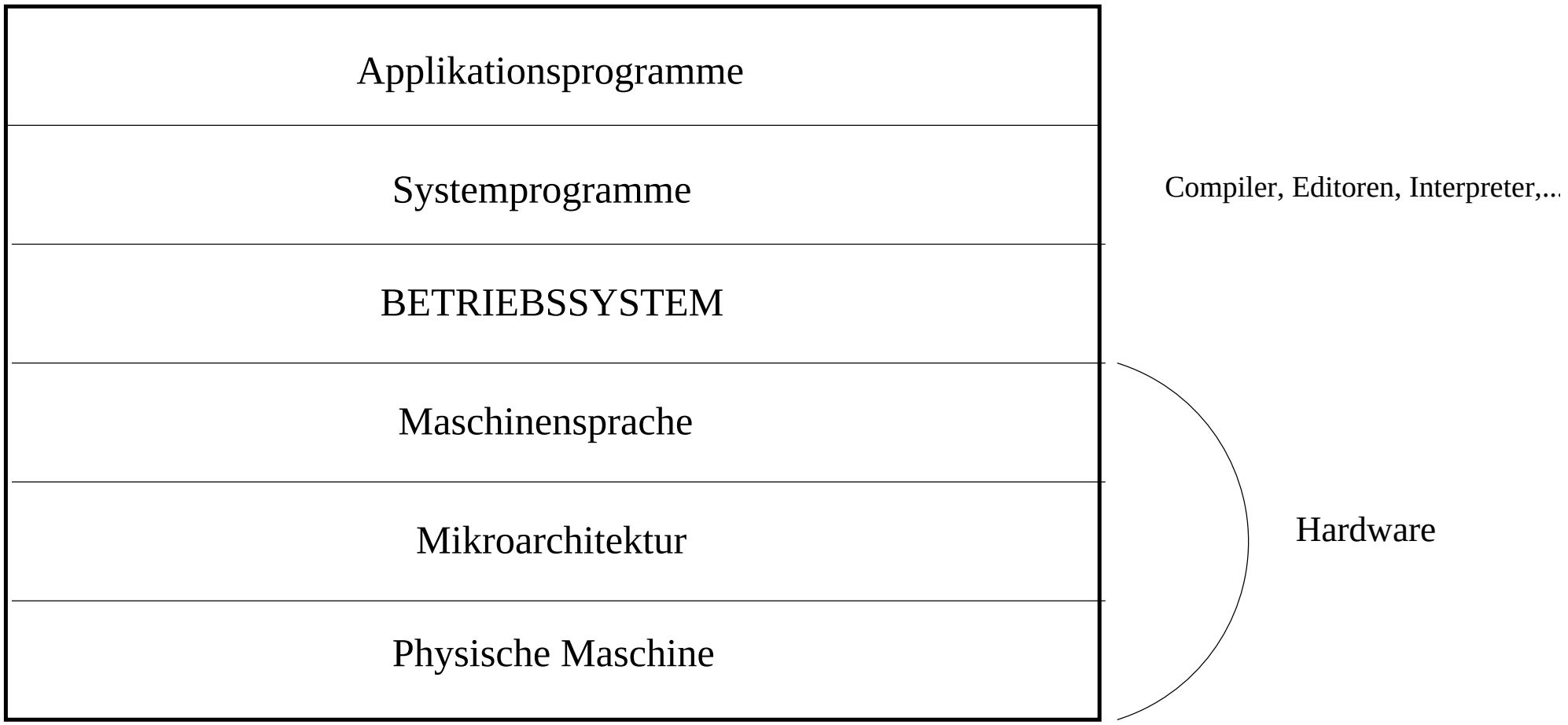


Abbildung 1: **Struktur eines Computersystems**

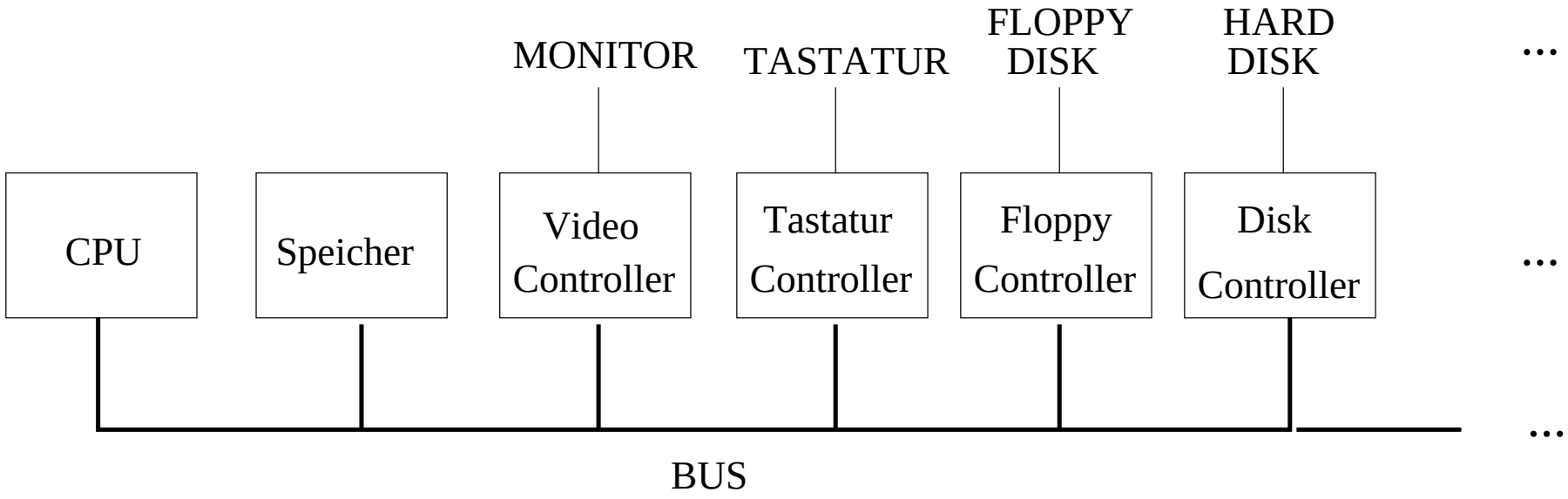
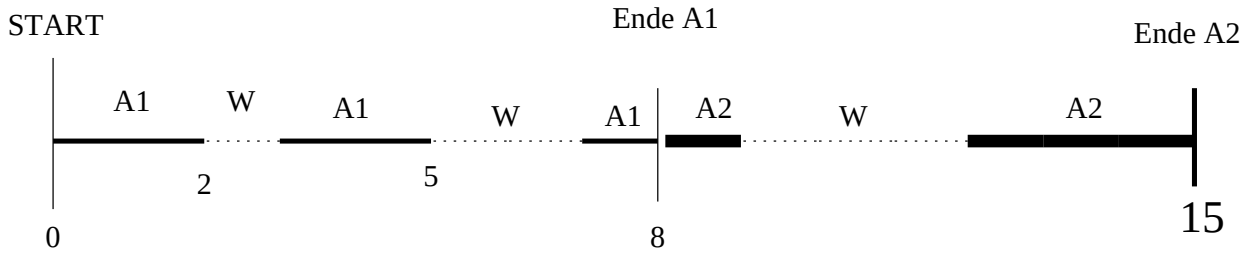


Abbildung 2: **Hardware Komponenten eines einfachen PC**

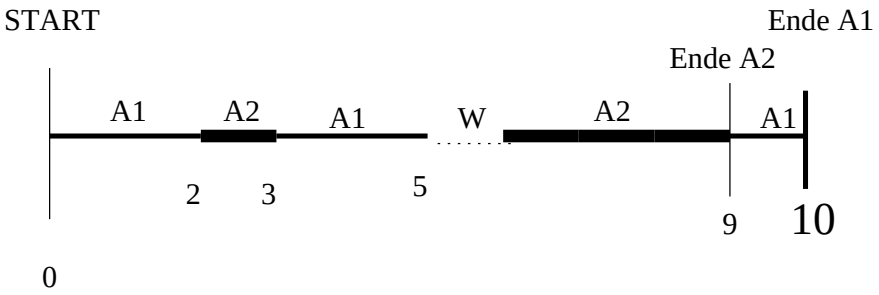
F1



serielle Bearbeitung

von A1 und A2

F2



Abarbeitung von

A1 und A2 im

Multiplexbetrieb

Abbildung 3: **Serieller Betrieb versus Multiplexbetrieb**

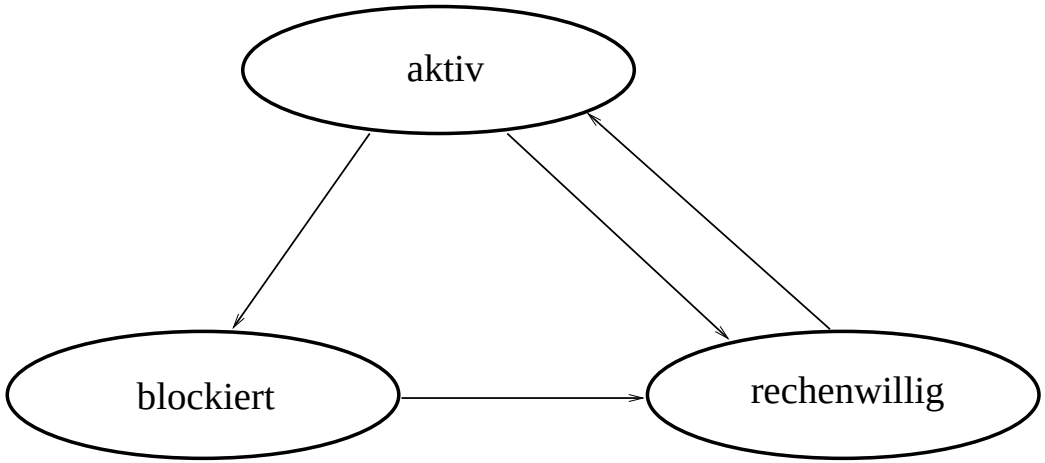


Abbildung 4: **Status von Prozessen**

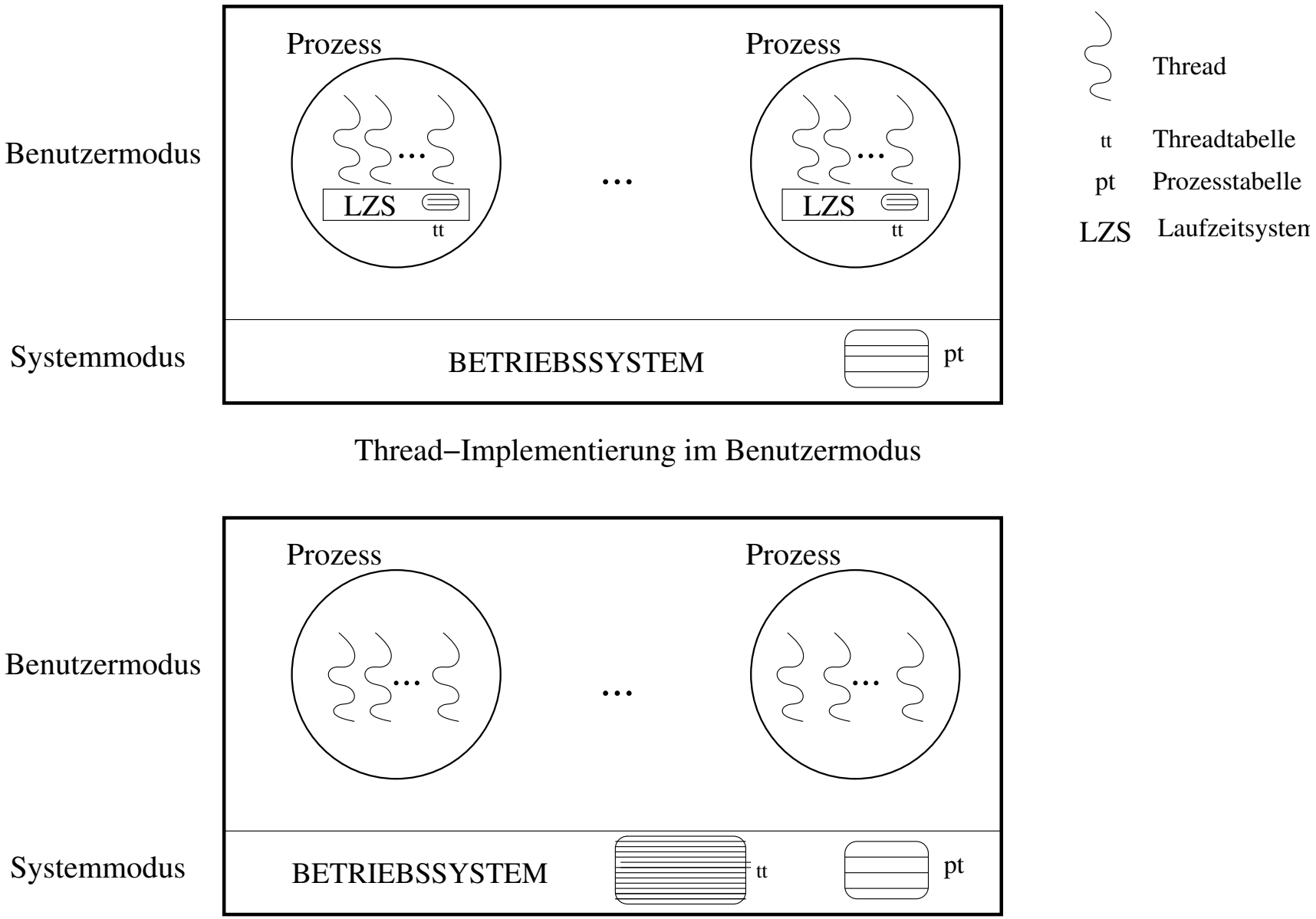


Abbildung 5: **Implementierung von Threads**

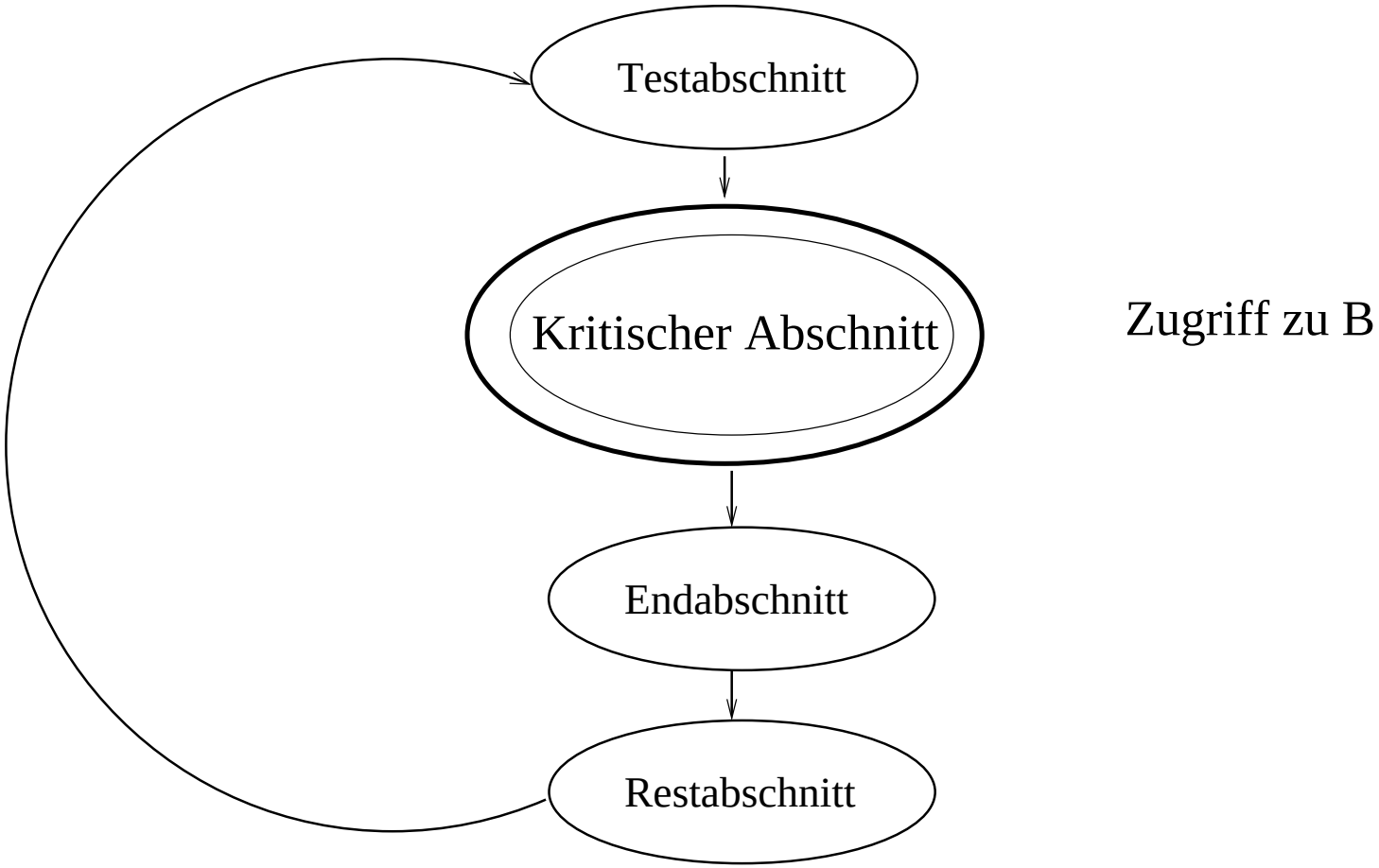


Abbildung 6: **Programmstruktur für wechselseitigen Ausschluß**

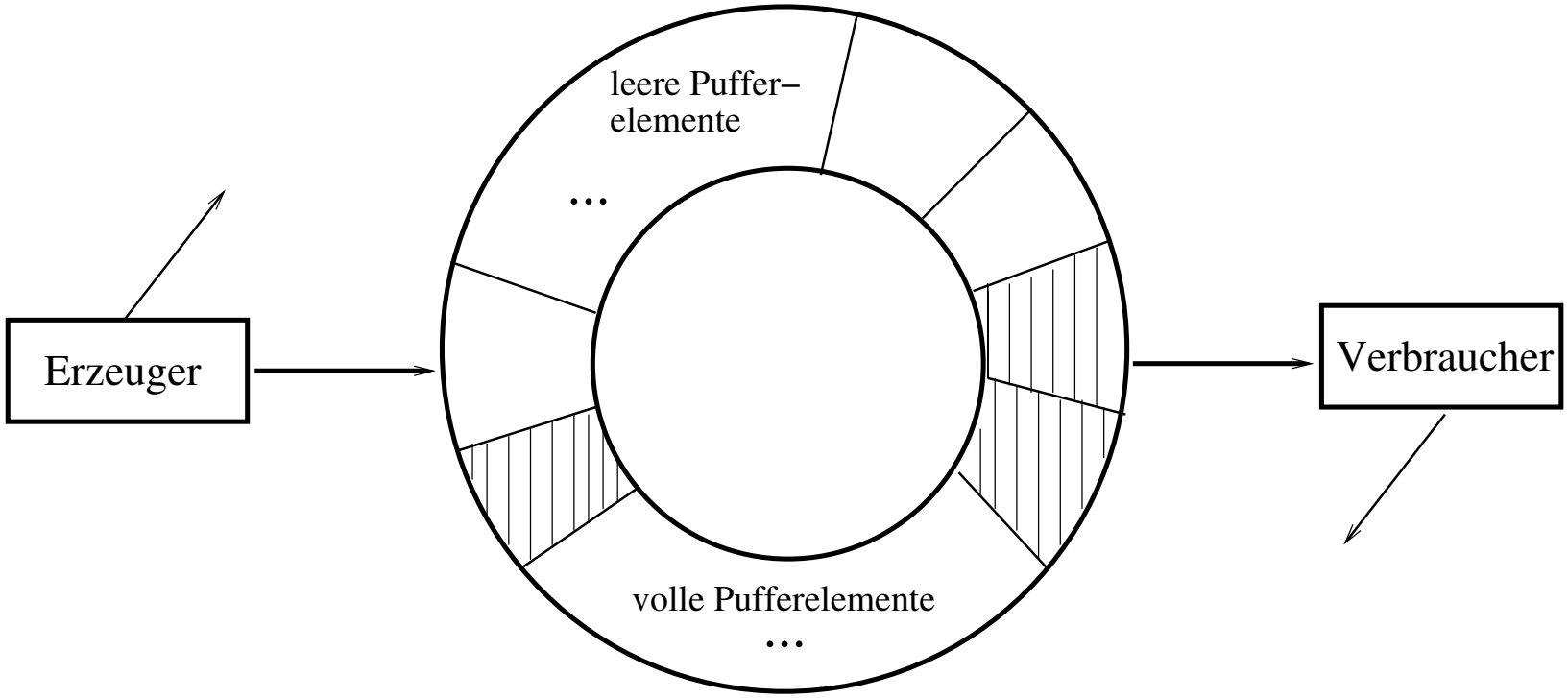


Abbildung 7: **Das Erzeuger/Verbraucher Problem**

```
begin schema
var Puffer: array (0..q-1) of integer;
var leere_puffer_elemente=q, volle_puffer_elemente=0 :sema;
integer function erzeuge() { ... };
function verbrauche(z: integer ) { ... };
parbegin
  actor Erzeuger {
     var e=0,x :integer      – – lokale Variablen
     while true {
       L1: x=erzeuge();
       L2: wait (leere_puffer_elemente);
       L3: Puffer(e mod q)=x;
       L4: signal (volle_puffer_elemente);
           e=e+1
    };
  actor Verbraucher {
     var v=0,y :integer      – – lokale Variablen
     while true {
       L1: wait (volle_puffer_elemente);
       L2: y=Puffer(v mod p);
       L3: signal (leere_puffer_elemente);
           verbrauche(y);
           v=v+1
    }
  };
parend
end schema
```

Abbildung 8: **Erzeuger/Verbraucher Problem**

```
begin schema
var Leserzaehler=0 : integer  – – Zahl der Leser, die sich um Zugriff zu B bewerben oder bereits lesen
var s=1, w=1 : sema;
parbegin
  actor Leser_j (1 ≤ j ≤ n_1) {
    while true {
      wait (w);
      Leserzaehler=Leserzaehler+1;
      if Leserzaehler=1 then wait (s); – – erster Prozeß in einem Strom von Lesern: Synchronisation mit Schreibern
      signal (w);
      – – LESEN – –
      wait (w);
      Leserzaehler=Leserzaehler-1;
      if Leserzaehler=0 then signal (s); – – letzter Prozeß in einem Strom von Lesern: Freigabe von B
      signal (w)
      – – RESTABSCHNITT – –
    }
  };
  actor Schreiber_j (1 ≤ j ≤ n_2) {
    while true {
      wait (s);
      – – SCHREIBEN – –
      signal (s)
      – – RESTABSCHNITT – –
    }
  };
parend
end schema
```

Abbildung 9: **Leser/Schreiber Problem mit Leserpriorität**

```
begin schema
var Puffer: array (0..q-1) of integer;
var ex=0, vx=0, volle=0 : integer;
integer function erzeuge() { ... };
function verbrauche(z: integer) { ... };
parbegin
  actor Erzeuger {
     var x :integer;
     while true {
       x = erzeuge();
        with Puffer when volle < q {
         Puffer(ex) = x;
         ex = (ex+1) mod q;
         volle = volle + 1
        }
    };
  actor Verbraucher {
     var y :integer;
     while true {
       with Puffer when volle > 0 {
         y = Puffer(vx);
         vx = (vx+1) mod q;
         volle = volle -1
       };
      verbrauche(y)
    }
  };
parend
end schema
```

Abbildung 10: **Erzeuger/Verbraucher Problem mit bedingten kritischen Regionen**

```
begin monitor schema
var arbL=0, arbS=0 : integer  -- Zahl der Leser bzw. Schreiber, die zu einem Zeitpunkt Zugriff zu B besitzen.
var L, S: condition  -- Vereinbarung der Bedingungsvariablen.

-- Es folgen die Vereinbarungen der Monitorprozeduren
boolean function leer(c: condition) { ... }; -- ein Aufruf dieser Funktion liefert true gdw Q(c) = φ
public function beginne_Lesen() {
   if arbS>0 ∨ ¬ leer(S) then waitcond(L);
   arbL = arbL + 1;
   signalcond(L)  -- hierdurch werden alle blockierten Leser freigegeben
   };

public function beende_Lesen() {
   arbL = arbL - 1;
   if arbL=0 then signalcond(S)
   };

public function beginne_Schreiben() {
   if arbL>0 ∨ arbS > 0 then waitcond(S);
   arbS = 1
   };

public function beende_Schreiben() {
   arbS=0;
   if ¬ leer(L) then signalcond(L)
                else signalcond(S)
   };
```

Abbildung 11: **Leser/Schreiber Problem ohne Behinderungen: Teil 1**

```
parbegin
  actor Leser_j (1 ≤ j ≤ n_1) {
     while true {
       beginne_Lesen()
       – – LESEN – –
       beende_Lesen()
       – – RESTABSCHNITT – –
     }
  };

  actor Schreiber_j (1 ≤ j ≤ n_2) {
     while true {
       beginne_Schreiben()
       – – SCHREIBEN – –
       beende_Schreiben()
       – – RESTABSCHNITT – –
     }
  }

parend
end monitor schema
```

Abbildung 12: **Leser/Schreiber Problem ohne Behinderungen: Teil 2**

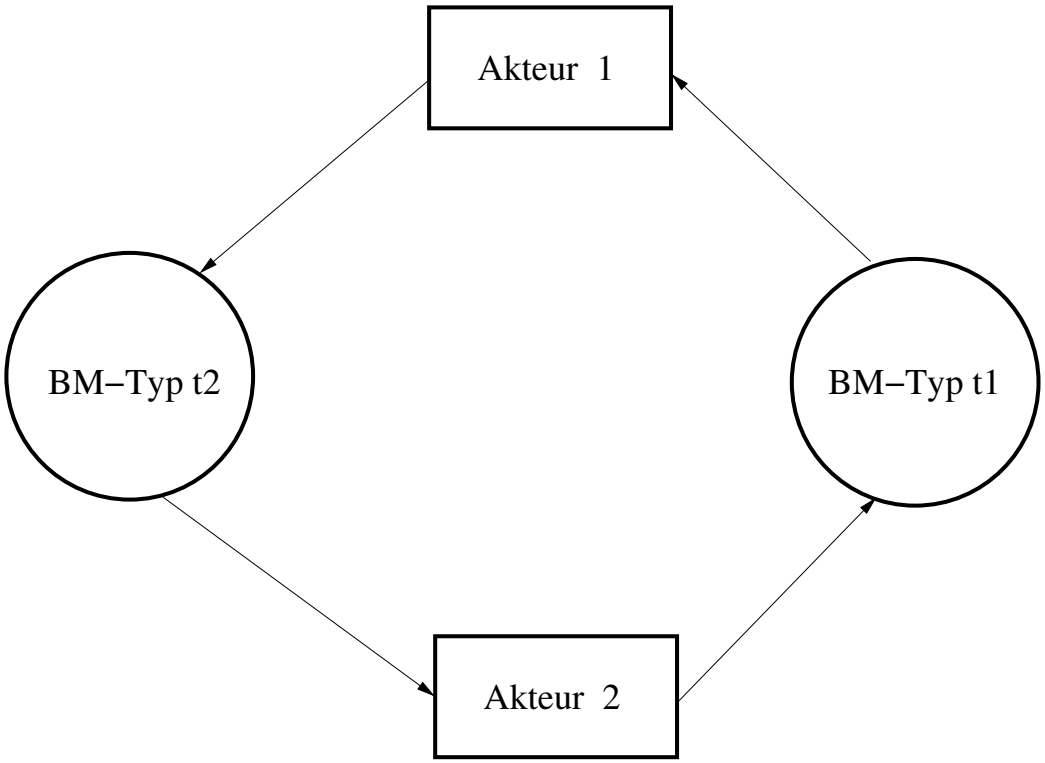


Abbildung 13: **Betriebsmittelgraph für einen Verklemmungszustandzustand**

```
begin
var H:= h(σ) : vector;
var U:= [1:n]; KK:=φ; K: set;
var u: integer;
var VKL: boolean;

while U ≠ φ {
 u:= select_and_remove(U);
 if y^u ≤ H
    then — — Prozeß u ist nicht verklemmt und wird in die Aktivierungssequenz aufgenommen
         KK:= KK ∪ {u};
         U := [1:n] - KK;
         H := H + b^u(σ); — — Freigabe der Betriebsmittel bei Beendigung
 end if
    };
K := [1:n] - KK;
VKL := (K ≠ φ)
end
```

Abbildung 14: **Erkennung von Verklemmungen**